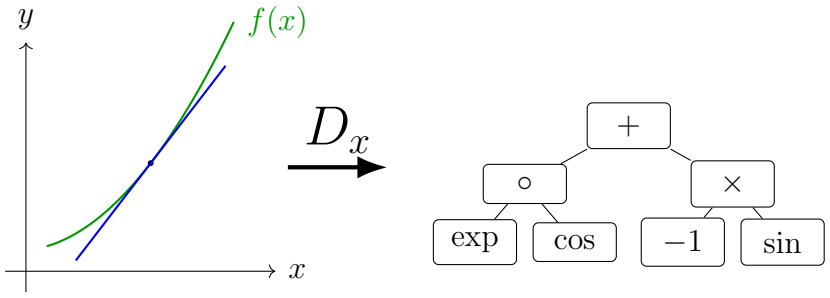
Calculus For Computer Scientists



Jim Newton

September 22, 2026

Contents

Chapter 0

Introduction

0.1 Course Objectives

- Understand why calculus is relevant to computer scientists: optimization in machine learning, numerical simulation, and the analysis of systems that evolve over time.
- Recognize the CS-first framing of this course: derivatives are introduced as symbolic reduction rules before their limiting definition, mirroring the pattern-matching on ASTs you already know.
- Identify the two central tools—the derivative and the integral—and their connection through the Fundamental Theorem of Calculus.
- Know the role Python, NumPy, and SciPy play throughout the course: both as computational tools and as a medium for building intuition.

0.2 About This Course

Calculus is one of the great intellectual achievements of human history. Developed independently by Newton and Leibniz in the second half of the 17th century, it provided for the first time a precise mathematical language for describing change and accumulation—concepts that had

resisted rigorous treatment since antiquity. The tools they introduced, the derivative and the integral, turned out to be two faces of the same coin, a profound and surprising connection known as the Fundamental Theorem of Calculus. For nearly three centuries, calculus has been the mathematical foundation of physics, engineering, and the natural sciences. More recently it has become equally central to computer science, driving the optimization algorithms that underpin machine learning, the numerical methods used in simulation, and the analysis of systems that evolve over time.

This course is an introduction to calculus tailored specifically for computer scientists. Rather than following the traditional curriculum developed for physicists and engineers, we organize the material around ideas and tools that are directly relevant to your work as a programmer and system designer. We begin with the derivative, introduced not through the usual limiting arguments but as a set of symbolic reduction rules—a calculus in the original sense of the word, meaning a system of mechanical calculation. This framing will feel familiar: the reduction rules map naturally onto the abstract syntax trees and pattern-matching techniques you have already encountered. Only after you are fluent with derivatives as a computational tool do we revisit the question of what a derivative actually is—its definition as a limit, its geometric meaning as the slope of a tangent line, and why the limiting process is more subtle than it first appears. The second half of the course covers integration, beginning with Riemann sums as a concrete computational procedure, progressing through the Fundamental Theorem which connects integration back to differentiation, and developing the standard techniques needed to evaluate integrals in closed form. Throughout the course we use Python—specifically NumPy and SciPy—both as a tool for computation and as a medium for building intuition, for instance discovering the antiderivatives of inverse trigonometric functions empirically before deriving them analytically.

Figure 1 contrasts this ordering with the traditional calculus cur-

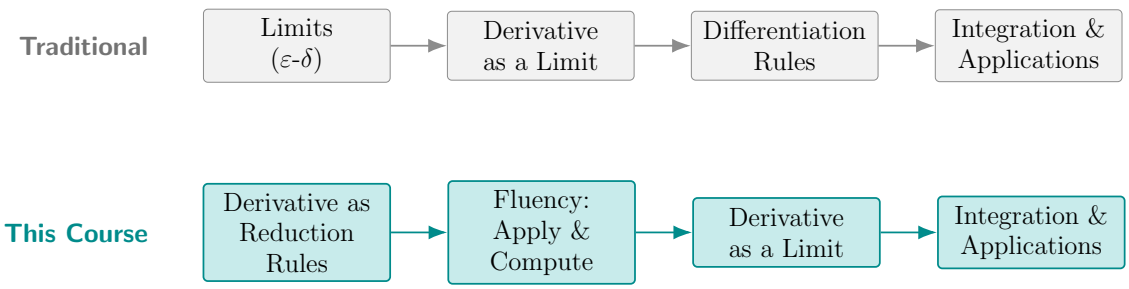


Figure 1: Traditional calculus courses (top row) begin with ε-δ limits and derive the rules from that foundation before reaching applications. This course (bottom row) begins instead with the derivative as symbolic reduction rules—already familiar from AST pattern-matching—builds fluency by computing with them, and only afterward revisits the limiting definition, before turning to integration.

riculum. The two rows share the same four milestones—rigorous limits, the derivative, differentiation technique, and integration—but this course reaches them in a different sequence: computation and fluency come first, and the limiting definition is revisited only once you already know how to use a derivative.

A word about mathematical rigor. This course does not aim to make you a mathematician, and we will not develop epsilon/delta proofs or prove every theorem we use. What we do aim for is genuine understanding: you should finish the course knowing not just how to apply the rules but why they work and when they apply. The historical development of calculus is itself instructive in this regard—Newton and Leibniz used derivatives and integrals correctly and productively for decades before anyone could give a rigorous foundation for the limiting process they relied upon. Useful mathematics often precedes its own justification. We follow the same path deliberately: computation first, foundations second, and always with an eye toward the problems in computer science that make these tools worth learning.

0.3 Types of Sessions

This course is organized into five distinct types of activity.

CM — Cours Magistral (lecture). CM sessions are two-hour lectures in which new material is presented. Content alternates between mathematically rigorous development and CS-centric applications and implementations. There are eleven CM sessions for L2 students and ten for L3 (L3 skips CM-6, the Definition of the Derivative, taught only to L2).

TP — Travaux Pratiques (practical session). TP sessions are hands-on computing sessions. Rather than presenting new theory, they guide you through exploration and experimentation with numerical tools—building intuition empirically before or alongside the formal treatment. There are two TP sessions in this course, covering NumPy and SciPy respectively.

HW — Written homework. HW assignments are completed outside of class, individually or in a small group of classmates. They consist of pen-and-paper problems designed to consolidate the material from recent CM sessions. These assignments are **not graded** and do not contribute directly to your course grade; however, the final exam is constructed largely from these problems. If you are stuck, you are encouraged to consult your preferred LLM or to bring your question to the next CM session.

Group Project — the PointFree derivative. Working in a group of two or three, you implement the symbolic differentiation engine (`derivative.py`). Grading is by *live oral exam*: each team member is questioned individually and must be able to explain and defend any

part of the submitted code. No test file is provided for `derivative.py` — designing your own test suite is itself part of the assignment. A hidden test suite also runs automatically on submission and contributes to your grade; only a pass/fail summary is shown, not which cases failed. Chapter 6 gives the full instructions.

Intra — automatically graded programming assignments. Intra assignments are submitted to Forge/Intra and graded automatically, the same way as in your previous courses: the tests you are graded against are given to you, so there are no surprises. Before anything else, submit `hello.py` — a trivial one-line warm-up with no calculus content at all, confirming that your `git`, `ssh`, and Forge/Intra setup actually works before you depend on it for a real assignment. It still counts toward your Intra average like every file below — essentially free credit, so there is no reason to put it off. `secant.py` opens at the same time as `hello.py` and closes well before the rest — it only needs Section 1.1.1's secant-line approximation, not the `PointFree`/AST machinery of Chapter 5, so there is no reason to wait. Six files in total are graded this way, individually:

- Intra-0, `hello.py` — print a one-line greeting; confirms your submission pipeline actually works.
- Intra-1, `secant.py` — approximating the derivative from a sequence of secant-line slopes: `secant_slope`, `converging_x_values`, and `approximate_derivative`.
- Intra-4, `riemann_sum.py` — numerical integration routines.
- Intra-3, `checks.py` — numerical comparison tools (an infinity-norm distance between two functions), built on `riemann_sum.py`.

- Intra-5, `inner_product.py` — an L^2 distance between two functions, via the inner product of Section 9.8, also built on `riemann_sum.py`.
- Intra-2, `pattern_match.py` — counting the nodes (`count_nodes`), measuring the depth (`max_depth`), and rewriting (`distribute`) a `PointFree` tree by pattern matching.

`fast_power.py`, a recursive exponentiation-by-squaring algorithm over `PointFree` trees, exists in the repository as an optional, ungraded exercise — it is not covered in lecture and does not factor into your Oral Defense. It is a genuinely interesting rabbit hole if you have the time: exponentiation by squaring promises $O(\log n)$ work, but it does not actually deliver that here, and the handout chapter on the ADT works through why.

`toPython.py` is provided complete in the repository — it serializes an expression AST to a Python expression string, reusing the same `match` / `case` skeleton as `evaluate`. It is not graded and not covered in lecture, but it is worth reading: the `Compose` case has to thread a growing expression string through the recursion rather than a number, which is the trickiest pattern-matching moment in the whole chapter.

`checks.py` and `pattern_match.py` are intended as practice alongside `derivative.py`: easier problems over the same ADT, and `checks.py` in particular produces the numerical-comparison tools you will want for testing `derivative.py` yourself. Working through these with your group while you develop `derivative.py` builds the same pattern-matching muscle before your oral defense.

Due Dates and Late Submissions. HW and Intra assignments each have an initial due date. As in previous semesters, late submissions are still accepted, but incur a percentage *malus* — the mirror

image of a bonus. Quizzes (Section 0.4) are written assuming you have already worked through the corresponding assignment on time, so submitting late costs you twice: the malus itself, and walking into the quiz without having seen the material it covers.

Final Exam. A single written exam at the end of the course covering all material. The exam is drawn in large part from the HW problems, so working through the HW carefully is the most effective preparation.

0.4 Grading

The percentages below are **tentative**. If they change, the change will be clearly announced in class and in writing.

Component	Description	Weight
Final Exam	Written, covers entire course	40%
Oral Defense	Group project (`derivative.py`), individual oral exam	30%
Intra	Auto-graded programming assignments (`hello.py`, `secant.py`, `riemann_sum.py`, `checks.py`, `inner_product.py`, `pattern_match.py`)	15%
Quizzes	Unannounced; given at the start of certain lectures	10%
Summary	Write one page summary of the course.	5%
		100%
Bonus	Reporting errata (see **Bonus: Reporting Errata** below)	**+5%**

Quizzes. A number of short quizzes will be given without advance notice at the beginning of certain CM sessions. Students who are absent or arrive late may not be permitted to sit the missed quiz. Quizzes are written assuming you have already submitted the corresponding HW or Intra assignment on time — submitting on time is therefore its own reward, independent of the late-submission malus, since you will walk into the quiz having already seen the material. The best preparation is simply to keep up with the HW and Intra assignments and to attend every session on time.

Summary. Write a single page summary of this course, submitted as a text file (not hand-written), due on the day of the final exam (the exam date itself has not yet been set). What are the good parts and the parts

that need to be improved. It is impossible to cover all of Calculus in this short course: which parts covered in this treatment should be omitted and possibly replaced with what? Does the student have enough time to absorb the important topics? Does the treatment make the student curious to explore more? Is the connection to Informatics (Compute Science) strong enough or does it need to be improved?

Bonus: Reporting Errata. Up to 5 bonus points are available for finding and reporting errors — in this handout's prose (grammatical, spelling, factual, mathematical, or otherwise) or in the homework or project Python code. An error report is not the same thing as a suggestion for improving the course: an error is something that is *wrong*; an enhancement idea, however good, is not bonus-eligible here and belongs instead in the Summary you submit at the end of the course (see **Summary** above). Report an error by opening a merge request against the student repository, `https://gitlab.cri.epita.fr/jim.newton/bcs-calculus-student` (new to merge requests? see GitLab's own guide, `https://docs.gitlab.com/ee/user/project/merge_requests/creating_merge_requests.html`). A report with no proposed fix is entirely welcome — you do not need to know how to fix an error to get credit for finding it — though a report that also includes a correct fix earns more credit than one without. When the same error is reported more than once, the first merge request to arrive normally gets the credit; if several arrive at nearly the same time, the most thorough one gets it instead, regardless of order. The bonus cannot push your final grade above 100%: a score is capped at 20/20, however large the bonus.

0.5 Setting Up Your Grader Workarea

If you have already taken other courses that use this same Grader-workarea system, you have done this before and the process below will

be familiar.

At the end of the first class you will be given a QR code (Appendix A.2) linking to a Python setup script, `checkout-workarea-calculus.py` (`https://gitlab.cri.epita.fr/jim.newton/bcs-calculus-student/-/blob/main/bin/checkout-workarea-calculus.py`). Running it creates and populates a personal git repository — your Grader workarea — on your Desktop. You are allowed, and encouraged, to move this repository afterward to a location that is safe and easy to find — for example, alongside the workareas you already keep for other courses.

Each student is required to work through this process individually, either alone or with the help of the class representative. If the script fails, or something about it is confusing, paste the code into your favorite LLM and ask for help debugging your specific situation. If you find a genuine bug in the script itself, you may report it as errata using the merge-request mechanism described in Section 0.4 — bug reports here are bonus-eligible too.

We will not spend class time debugging individual workarea setup problems. Get help from the class representative first; contact the professor outside of class only as a last resort.

0.6 How to Read This Document

Throughout this document, margin annotations and colored boxes serve as navigation aids.

Margin annotations. Icons appearing in the margin signal the nature of the adjacent content.

- **Essential** — a core concept you must understand and be able to apply.
- **Recall** — a result or technique introduced elsewhere; look it up if needed.
- **Warning** — a common mistake or a subtle point requiring care.
- **Enrichment** — optional deeper material for the curious reader.

Colored boxes. Different types of mathematical content are distinguished by background color.

Chapter Objectives — the skills and understanding you should have by the end of the chapter.

Activity — a hands-on task to run or observe, followed by a discussion question for the class.

Open Questions — unresolved or deliberately out-of-scope questions raised by the chapter, some answered later in the handout, some left for independent exploration.

Theorem — a mathematical statement that has been proven true.

Definition — a precise meaning assigned to a mathematical term.

Example — a worked illustration of a concept or technique.

Proposition — a useful mathematical claim, with short proof.

Corollary — a result following directly from the preceding theorem.

Lemma — intermediate result used in the proof of a larger theorem.

Proof — a logical argument establishing a claim.

Listing — a code extract for reference or study.

Pattern — a reusable strategy or recipe to recognise and apply; the margin icon marks later uses.

0.7 Syllabus

0.7.1 Lectures, Practicals, and Exam

Session	Chapter	Topics
CM-1	1	Derivative as reduction rules: constant, identity, power, sum, product, quotient, linearity
CM-2	2	Composition and the Chain Rule, implicit differentiation, derivative of arctan
CM-3	3	Shape of a function: critical points, concavity, extrema
CM-4	5	ADT/AST for differentiable functions, pattern matching, serialization
CM-5	6	Testing the derivative: numerical testing, `functions_agree`
CM-6 (L2 only)	7	Definition of the Derivative: ε-δ limits, non-differentiable functions, first-principles proofs
CM-7	8	Riemann Sums: left/right, midpoint, trapezoid, Simpson's rule
TP-1	9	NumPy: inverse trig functions, inner product, norm, distance
CM-8	10	FTC Parts 1 and 2, integrals as linear combinations
TP-2	12	SciPy: `quad`, `optimize`, preview of `solve_ivp`
CM-9	13	Integration by Substitution: u-substitution, trigonometric substitution
CM-10	14	Standard Techniques: partial fractions, repeated linear factors
CM-11	16	Debriefing: symbolic integration, the Risch algorithm; exam review
~~Exam~~	~~Final Exam~~	

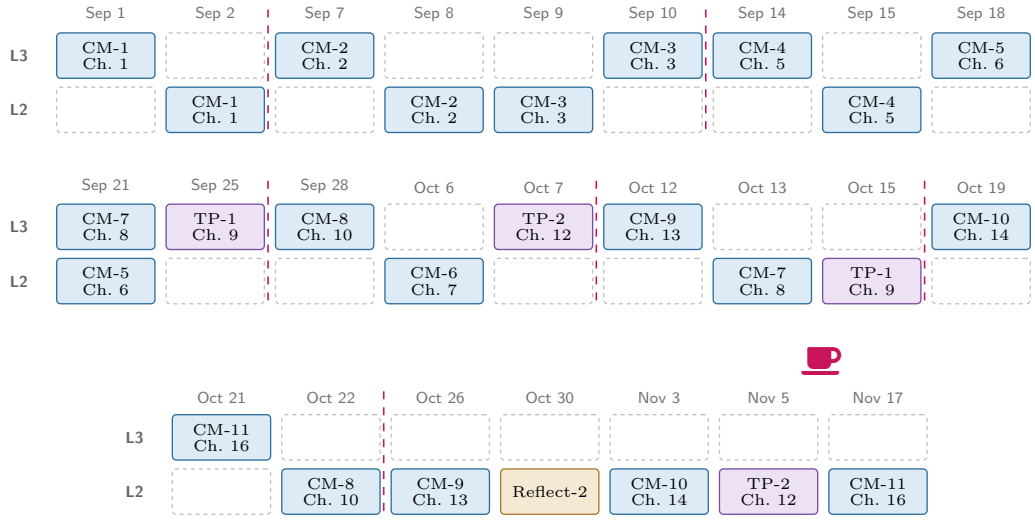
0.7.2 Homework, Group Project, and Intra Assignments

Session	Chapter	Topics
Intra-0	0	`hello.py`: submission pipeline warm-up
Intra-1	1	`secant.py`: `secant_slope`, `converging_x_values`, `approximate_derivative`
HW-1	4	Derivatives by hand, Taylor series, skip list optimization
Group-Project	6	Oral Defense: implement `derivative.py`
Intra-2	6	`pattern_match.py`: `count_nodes`, `max_depth`, `distribute`
Intra-3	6	`checks.py`: `inf_norm`, `distance`, `functions_agree`
Intra-4	8	`riemann_sum.py`: left, right, midpoint, trapezoid, Simpson sums
Intra-5	9	`inner_product.py`: `inner_product`, `norm`, `distance`
HW-2	11	Inner product, antiderivatives by table lookup, Taylor series integration
HW-3	15	U-substitution, partial fractions, variable data-transfer rate

0.7.3 Preliminary Time Plan

L2 and L3 meet on different days and cover the syllabus above at different paces (L2 pauses for one reflection/catch-up session, takes a longer break in October, and additionally covers Chapter 7, Definition of the Derivative, as CM-6 – a session L3 does not have). The dia-

gram below lines the two groups' actual class days up side by side – one column per teaching day, not per calendar day – so you can see at a glance which sessions the two groups cover together and where they drift apart. Dates are provisional and may shift if the school's timetable changes.



☕ marks a week with no class for either group – the week of Nov 9 is the only one, sitting between L2's Nov 5 and Nov 17 sessions.

Chapter 1

The Derivative

Chapter Objectives

- Understand the derivative intuitively as the slope of a tangent line, and why a two-point (secant) approximation to it always carries some measurement error — an error that shrinks as the two points move closer together.
- Recall the derivatives built up in this chapter for constants, the identity, x^n (Power Rule), e^x, $\sin$, $\cos$, and $\tan$, and apply them by hand; the full table, including $\ln$ and the inverse trig functions, is assembled in Chapter 2 once the Chain Rule is available.
- Use linearity, the product rule, and the quotient rule to differentiate compound expressions step by step. (The chain rule, for differentiating a *composition* of functions, is introduced in Chapter 2.)
- Recognize that extending term-by-term differentiation from a finite sum to an *infinite* power series, as with e^x, is a tempting but unjustified leap — it needs its own theorem, not just linearity, plus a condition that makes it valid.
- Identify an anti-derivative from the small table built in this chapter (including the Power Rule's inverse, valid when $n + 1 \neq 0$, and $\ln|x|$ as the antiderivative of $1/x$), and explain why finding anti-derivatives is harder than finding derivatives.

In this chapter we will discuss how to *compute* the derivative of certain functions. We will not define exactly what that means. The precise definition will be given in Chapter 7.

How can we work with things we don't understand? In computer programming it is common that we are tasked with implementing algorithms which we do not completely understand. Often we work in teams, and some other team member understands the process but is not the programming expert. The team together develops the deliverable.

Nevertheless, a brief intuition of what a derivative is may be useful.

1.1 Intuition of the Derivative

We will use tangent lines freely in the next several sessions. You have a geometric intuition for them, and that intuition is more or less reliable for the functions we will work with. We look at some common misconceptions in Section 1.1.2. In Chapter 7 we will return to this question and make it precise — at which point you may find that your intuition was right for the right reasons, or right for the wrong reasons.

Figure 1.1 shows the graph of a function $y = f(x)$, and the tangent line of the curve at a point. For the function shown — and for most functions we work with in this course — every point along the curve has a uniquely determined tangent line. This is not automatic: it may not be obvious that a curve should have a unique tangent line at every point, and indeed not every function does. Having a uniquely determined tangent line at every point of its domain is precisely what it means for a function to be *differentiable*.

Some functions are differentiable everywhere, some are differentiable only on part of their domain, and some are differentiable nowhere at all. *I.e.*, some functions are *differentiable* and some are not.

A natural objection from a CS perspective: most quantities we

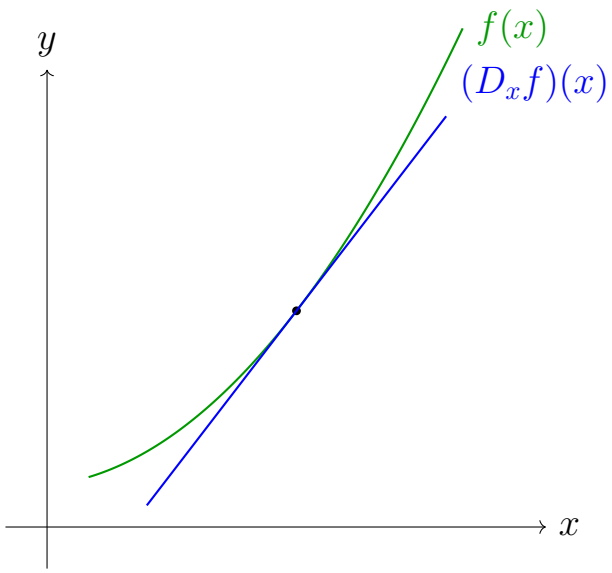


Figure 1.1: A function and its tangent line, illustrating the value of the derivative at a point.

actually measure — network throughput, CPU usage, packet counts, bit-flow rate — are discrete and discontinuous. How can calculus apply? The answer is that we *model* the discrete reality with a smooth continuous function, and then apply calculus to the model. The model is an approximation; the calculus gives exact results for the model; and the total error has two parts: the numerical error of the computation, and the modeling error from using a smooth function to represent something inherently jagged. This modeling step is standard practice in engineering and applied science, and it is the reason a course in calculus is relevant to computer scientists.

Figure 1.2 makes this concrete: the jagged line is what a sensor actually reports; the smooth curve is the model calculus operates on.

The tangent line of a curve at the point (x, y) has a slope, unless the tangent line is vertical at that point. But in the cases where the tangent line is not vertical, the slope, m, (which is a number) is the

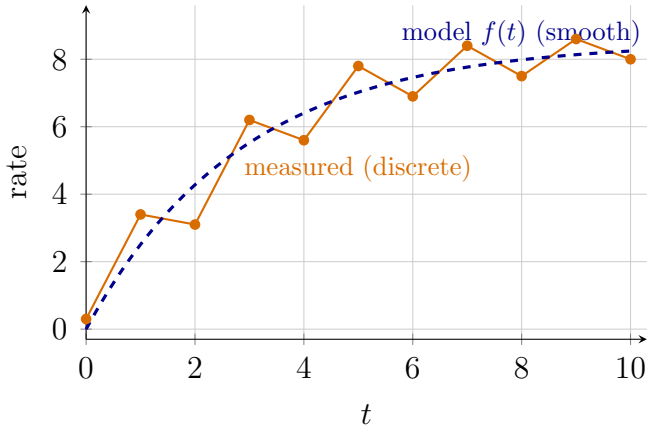


Figure 1.2: A discrete, jagged measurement (e.g. a bit-flow rate sampled once per second) alongside the smooth continuous function $f(t)$ (dashed) used to model it. Calculus applies to the smooth model, not directly to the jagged measurements.

value of the derivative of the function f at the input value x. We write $\mathcal{D}_x f(x) = m$.

1.1.1 Approximating the Derivative

If f represents some physical, measurable, process, we can approximate the value of the derivative at any point by approximating the slope. To approximate the slope we simply measure the function at two x-values close to each other, x_1 and x_2 (or equivalently x and $x+h$) and compute:

$$m_{approx} = \frac{f(x_2) - f(x_1)}{x_2 - x_1} = \frac{f(x+h) - f(x)}{h},$$

where x_1 and x_2 are chosen close to each other (a distance h apart) and $f(x_1)$ and $f(x_2)$ are experimentally determined by some measurement device.

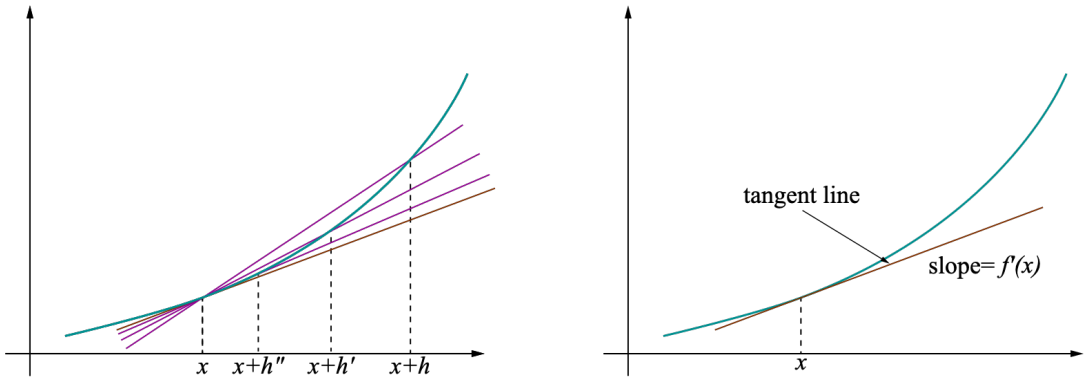


Figure 1.3: We can approximate the tangent to a curve at a point by successive approximations of secant lines. The slope of the tangent line at a point is equal to the derivative evaluated at that x-value.

Whenever we choose x_1 and x_2, two points on the curve are identified $(x_1, f(x_1))$ and $(x_2, f(x_2))$, equivalently $(x, f(x))$ and $(x+h, f(x+h))$. As shown in Figure 1.3, these two points determine a secant line. As the sample points are chosen closer together, the secant line better approximates the tangent line.

Example 1.1.1 (*Approximating Acceleration*)

As an example imagine you are a passenger in a car, and the driver presses the accelerator. The car starts changing speed and after a short time settles on a new speed. At each moment during this period you feel a force pushing you backward into the seat. By this backward force and the physical law, $M = ma$, (force equal mass times acceleration) we know that at each point of time, the car (and everything in it) was undergoing an acceleration.

Suppose at time t_1 you look at the speedometer and you approximate that the car speed is 50 km/h. Then at a time, half a second later, t_2 you approximate the speed reading on the speedometer to be 60 km/h. You can approximate the acceler-

ation (which is the derivative of the speed) to be

$$\begin{aligned} a &\approx \frac{s_2 - s_1}{t_2 - t_1} \\ &= \frac{60\frac{km}{h} - 50\frac{km}{h}}{\frac{1}{2}sec} \\ &= \left(\frac{10\ km}{h}\right)\left(\frac{2}{sec}\right)\underbrace{\left(\frac{1000\ m}{km}\right)}_{=1}\underbrace{\left(\frac{1\ h}{60\ min}\right)}_{=1}\underbrace{\left(\frac{1\ min}{60\ sec}\right)}_{=1} \qquad (1.1) \\ &\approx 5.556\frac{m}{sec^2}\,. \end{aligned}$$

Equation (1.1) includes a technique which is very common in scientific computation. The first two terms $\left(\frac{10\,km}{h}\right)\left(\frac{2}{sec}\right)$ compute a quantity whose units are $\frac{km}{h\,sec}$ which is not incorrect but is very unusual. Since acceleration is more understandable in $\frac{m}{sec^2}$, we have to perform some units conversion. To convert between units, we multiply by 1, but 1 in a special form: multiplying by $\frac{1000\,m}{km}$ converts from km to m; multiplying by $\frac{1\,h}{60\,min}$ converts from $\frac{1}{h}$ to $\frac{1}{min}$; and finally multiplying by $\frac{1\,min}{60\,sec}$ converts from $\frac{1}{min}$ to $\frac{1}{sec}$.

We note that the passenger in Example 1.1.1 *measured* the speed at two time-points half a second apart. If the passenger had instead measured at two points a quarter of a second apart, or 0.6 seconds apart, a slightly different approximate acceleration would have been the result.

Notice, though, that changing which two points we measure changes the answer: measuring at t_1 and t_2 gives one value for a_{approx}; measuring at t_1 and a different point t_3 closer to t_1 gives a different value, close to the first but not exactly equal to it. We can think of this disagreement as *measurement error*: both approximations are estimates of the same

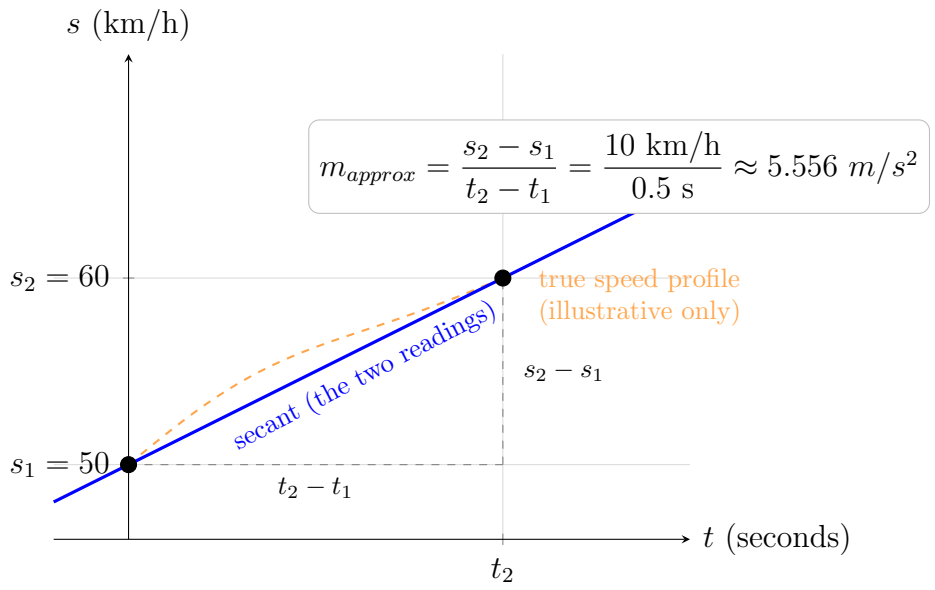


Figure 1.4: The two speedometer readings from Example 1.1.1, (t_1, s_1) and (t_2, s_2), determine a secant line whose slope is the approximate acceleration m_{approx}. The thin, dashed, orange curve is not measured — it is one plausible true speed profile the car might follow while settling into $60\,km/h$, drawn only to suggest that the secant's slope is an *average* over the interval, generally close to but not exactly equal to the true, instantaneous acceleration at either endpoint.

underlying quantity, and they differ only because each is, after all, an estimate. Moving the two time-points closer together tends to shrink that error — a shorter interval usually lands closer to the value a still-shorter interval would give.

But that last sentence only makes sense if there is something for the error to be measured *against*. "The error gets smaller" is not a free-standing statement about two approximate numbers; it silently assumes a *third* number — the correct one — that every approximation is in error relative to. The value of the derivative *is* that correct number.

This is not just a mathematical convenience — you already have

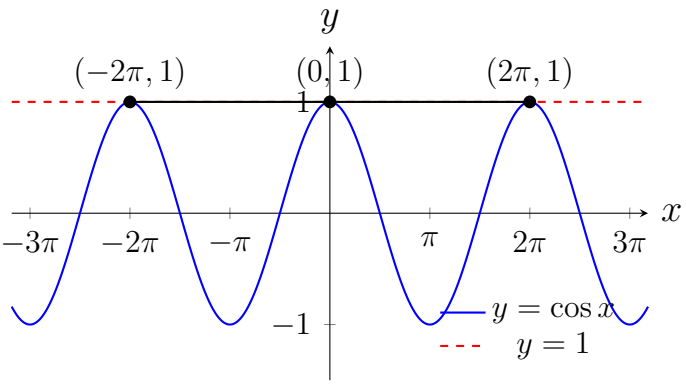


Figure 1.5: The line $y = 1$ is tangent to the curve $y = cosx$ at the point (0,1), but intersects the curve infinitely many times.

a physical intuition for why such a correct value should exist: the backward force you felt in Example 1.1.1 was a single, unambiguous force at each instant, not something that depended on whether the speedometer readings you happened to take were half a second or a quarter second apart. The two-point measurement is only *our* way of estimating that instantaneous quantity; the force itself — and the acceleration it causes — does not care how, or how coarsely, we choose to measure it.

Making this fully precise — saying exactly what "the correct answer" means, and proving that shrinking the interval really does drive the error to zero — is the subject of the optional Chapter 7, which builds on the formal definition of limit from Algebra-1. For the rest of this course, the geometric picture above (Figure 1.1: the derivative is the slope of the tangent line) together with the measurement-error argument just given is enough to work with.

1.1.2 Tangent Line Misconceptions

In the previous section we talked about *tangent lines* without attempting to give an intuition of what they are. If you have an intuition of

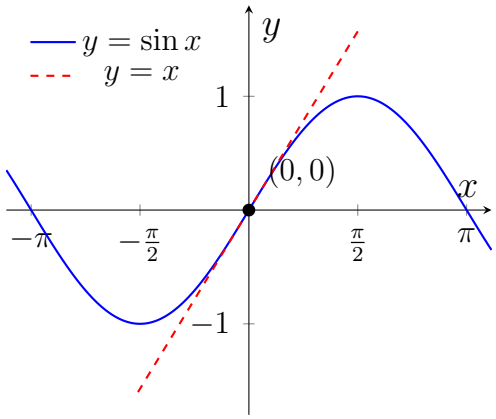


Figure 1.6: The line $y = x$ is tangent to the curve $y = sinx$ at the origin, and crosses the curve at that point.

tangent line from geometry, your intuition may be flawed. A tangent line is not defined the same way as in geometry. In geometry, a tangent to a circle is a line that touches it at exactly one point. For general curves this definition breaks down: the tangent to $y = \cos x$ at $x = 0$ (Figure 1.5) is the line $y = 1$, which touches the curve infinitely many times, once per period. Furthermore, the tangent line of $y = \sin x$ at $x = 0$ (Figure 1.6) is the line $y = x$, crossing the curve, whereas the tangent of a circle does not cross the circle. Instead, the tangent line at a point is approximated by a secant line, as shown in Figure 1.3—a line drawn through two nearby points on the curve—and that approximation's error shrinks as the two points are brought closer together. We will make this precise in Chapter 7.

1.2 Differential Rules

In Chapter 7 we will discuss how to compute the derivative in general, but for the moment we present the derivative as a computation process with some axiomatic rules.

In each of the following cases we will assume that $f, g, h : \mathbb{R} \to \mathbb{R}$ are *differentiable* functions. Again, we acknowledge that we have not satisfactorially defined exactly what that means. We use two notations, distinguished by whether the input variable is named.

Point-free form. The notation $\mathcal{D}\,f$ denotes the derivative of f as a function, without naming its input variable: $\mathcal{D}\,f : \mathbb{R} \to \mathbb{R}$. For example the derivative of sine is $\mathcal{D}\sin$. Writing functions without ever mentioning the input point is called *point-free* form; Chapter 5 explains why this is the natural choice for the symbolic-differentiation code you will implement.

Pointwise form. When the input variable is explicit we write $\mathcal{D}_x f(x)$ and call this the *pointwise* form. For example the derivative of $f(x) = ax^2 - bx + c$ is written $\mathcal{D}_x(ax^2 - bx + c)$, emphasizing that x is the differentiation variable and a, b, c are constants.

Once every rule in this chapter and the Chain Rule (Chapter 2) are in hand, we collect them all in one derivative table (Figure 2.1, in Chapter 2) listing both forms side by side.

1.2.1 Constant

The first rule is how to compute the derivative of a constant function.

Theorem 1.2.1 (*Derivative of a Constant Function*)

> If $f(x) = c$, for some constant $c \in \mathbb{R}$, then $\mathcal{D}\,f = 0$. *I.e.*, $\mathcal{D}\,f$ is the function which has value 0 everywhere.

1.2.2 Identity

The next function we look at is the identity function: the function whose output value is the same as its input value. We can denote this function id. The derivative of the id function is the constant function 1.

Theorem 1.2.2 (*Derivative of the Identity function*)

$$\mathcal{D}\left(id\right) = 1$$

We often (most often) don't explicitly use the name of the identity function; rather, we use the expression $f(x) = x$, in which case we denote $\mathcal{D}_x f(x) = 1$.

1.2.3 Power

The next set of functions we look at is the set $\{f_n \mid f_n(x) = x^n\}$ where $n \in \mathbb{Z}$.

Theorem 1.2.3 (*Power Rule*)

If $n \in \mathbb{Z}$, a function of the form, $f(x) = x^n$, then $\mathcal{D}_x x^n = nx^{n-1}$.

Example 1.2.4 (*Derivative of a Power of* x)

Let's compute the derivative of $f(x) = x^6$. We simply multiply by the current exponent, 6, and reduce the exponent from 6 to 5.

$$\begin{aligned} \mathcal{D}_x x^6 &= 6x^{6-1} \\ &= 6x^5 \end{aligned}$$

Example 1.2.5 (*Derivative of a negative Power of* x)

Let's compute the derivative of $f(x) = x^{-6}$. We simply multiply by the current exponent, -6, and reduce the exponent from -6

to -7.

$$\begin{aligned} \mathcal{D}_x x^{-6} &= -6x^{-6-1} \\ &= -6x^{-7} \end{aligned}$$

Theorem 1.2.3 can also be extended to exponent values between 0 and 1, but to do this we must restrict $x > 0$. For example for $x > 0$, if $f(x) = \sqrt{x} = x^{1/2}$ then $\mathcal{D}_x x^{1/2} = \frac{1}{2}x^{-1/2} = \frac{1}{2\sqrt{x}}$.

By restricting the domain to $x > 0$ we completely avoid the question of what is the derivative of the square root at $x = 0$, 0 is not even in the domain of the function in question.

If $x > 0$ then we can extend the power rule to rational exponents such as $x^{a/b}$ where $a, b \in \mathbb{Z}$, so that

$$\mathcal{D}_x x^{a/b} = \frac{a}{b} x^{\frac{a}{b}-1}.$$

However, to show this is true, we will need the Chain Rule, which is given later in Theorem 2.2.1.

As a special case of the Power Rule, when $n = 0$ and $x \neq 0$, we have $\mathcal{D}\,1 = \mathcal{D}_x x^0 = 0x^{-1} = 0$. This agrees with Theorem 1.2.1 that the derivative of 1 (a constant) is 0.

1.2.4 Sum of Differentiable Functions

If $f = h + g$ is the sum of two differentiable functions, then we can easily compute its derivative by summing the derivatives of the functions h and g.

Theorem 1.2.6 (*Summation Rule (Point-Free Form)*)

If $f = h + g$, then $\mathcal{D}\,f = \mathcal{D}\,h + \mathcal{D}\,g$.

Theorem 1.2.7 (*Summation Rule (Point Form)*)

If $f(x) = h(x) + g(x)$, then

$$\begin{aligned} \mathcal{D}_x f(x) &= \mathcal{D}_x(h(x) + g(x)) \\ &= \mathcal{D}_x h(x) + \mathcal{D}_x g(x). \end{aligned}$$

1.2.5 Product

If $f = h \cdot g$, is the product of two differentiable functions, then we can compute its derivative by the following formula. Unlike the Quotient Rule, there is no elementary proof of the Product Rule available at this stage — every route we could take (logarithmic differentiation, differentiating e^{hg}) either silently requires $h, g \neq 0$ (which the theorem itself does not) or turns out to be circular. The genuine proof, directly from the limit definition, is Theorem 7.4.6 in Chapter 7 — which also includes the logarithmic-differentiation argument as a second, illuminating derivation, explicitly caveated to the case $h, g \neq 0$.

Theorem 1.2.8 (*Product Rule*)

If $f(x) = h(x)g(x)$ then

$$\mathcal{D}\,(h\ g) = (h)(\mathcal{D}\,g) + (g)(\mathcal{D}\,h)\,.$$

A special case of the Product Rule is the Scalar rule; *i.e.*, if $h(x)$ is a constant.

Corollary 1.2.9 (*Scalar Rule*)

If $f(x) = c\ g(x)$ then

$$\mathcal{D}\,(c\ g) = c\mathcal{D}\,g\,.$$

Proof. Let $h(x) = c$, and compute the derivative of $h(x)g(x)$. Note that $\mathcal{D}_x h(x) = \mathcal{D}_x c = 0$

$$\begin{aligned}\mathcal{D}_x f(x) &= \mathcal{D}_x h(x)g(x) \\ &= \underbrace{h(x)}_{=c}(\mathcal{D}\, g(x)) + g(x)\,\underbrace{(\mathcal{D}\, h(x))}_{=0} \\ &= c\mathcal{D}\, g(x)\end{aligned}$$

1.2.6 Difference

Similar to the Summartion rule, we can also differentiate the difference of two differentiable functions.

Corollary 1.2.10 (*Difference Rule*)

If $f = h - g$, then $\mathcal{D}\, f = \mathcal{D}\, h - \mathcal{D}\, g$.

Proof. Let $f(x) = h(x) - g(x)$.

$$\begin{aligned}\mathcal{D}_x f(x) &= \mathcal{D}_x(h(x) - g(x)) \\ &= \underbrace{\mathcal{D}_x(h(x) + (-1)g(x))}_{\text{use Summation Rule}} \\ &= \mathcal{D}_x h(x) + \underbrace{\mathcal{D}_x((-1)\, g(x))}_{\text{use Scalar Rule}} \\ &= \mathcal{D}_x h(x) + (-1)\mathcal{D}_x g(x) \\ &= \mathcal{D}_x h(x) - \mathcal{D}_x g(x)\end{aligned}$$

1.2.7 Linearity of the Derivative

The Scalar Rule (Corollary 1.2.9) and the Summation Rule (Theorem 1.2.6) together express a single, fundamental fact: differentiation is a *linear operator*. For any differentiable functions f and g and any constants $\alpha, \beta \in \mathbb{R}$,

$$\mathcal{D}\,(\alpha f + \beta g) = \alpha\,\mathcal{D}\,f + \beta\,\mathcal{D}\,g\,.$$

Why does this matter? Because in calculus you will constantly encounter *linear combinations* — expressions of the form $\alpha f + \beta g$, or longer, but still *finite*, sums $a_0 + a_1 f_1 + a_2 f_2 + \cdots + a_n f_n$. Linearity says you can *distribute* the derivative across any such finite sum and pull constants out, handling each term independently. This is not just a convenience: it is the reason differentiating a polynomial reduces to differentiating one term at a time. *Infinite* sums — Taylor series and Fourier series among them — reduce the same way too, but linearity alone does not license that step; Section 1.2.8 states the theorem that does, and the extra condition it needs. The same finite-sum structure reappears for the definite integral in Chapter 10 — the integral is also a linear operator, and the two facts are worth keeping side by side as instances of the same idea.

At this point, we know how to compute the derivative of any polynomial. First, we apply linearity to differentiate each term independently. Second, each term is either a constant or a power ax^n, which we differentiate using the Power Rule (Theorem 1.2.3).

Example 1.2.11 (*Derivative of a Polynomial*)

Compute the derivative of $f(x) = 3x^4 - 5x^2 + x - 1$.

$$\begin{aligned}\mathcal{D}_x f(x) &= \mathcal{D}_x 3x^4 - 5x^2 + x - 1 \\ &= \mathcal{D}_x 3x^4 - \mathcal{D}_x 5x^2 + \mathcal{D}\, x - \mathcal{D}\, 1 \\ &= (3)(4)x^{4-1} - (5)(2)x^{2-1} + 1 - 0 \\ &= 12x^3 - 10x + 1\end{aligned}$$

We can state the principle from Example 1.2.11 as a generalized corollary.

Corollary 1.2.12 (*Derivative of a Polynomial*)

If a polynomial is of the form

$$\begin{aligned}f(x) &= \sum_{k=0}^{n} a_k x^k \\ &= a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \cdots + a_n x^n\,,\end{aligned}$$

then

$$\begin{aligned}\mathcal{D}_x f(x) &= \sum_{k=1}^{n} a_k\, k\, x^{k-1} \\ &= a_1 + a_2 x + a_3 x^2 + \cdots + a_n x^{n-1}\,.\end{aligned}$$

Proof.

$$
\begin{aligned}
\mathcal{D}_x f(x) &= \mathcal{D}_x \left(\sum_{k=0}^{n} a_k x^k \right) \\
&= \sum_{k=0}^{n} \mathcal{D}_x \left(a_k x^k \right) && \text{by Summation Rule} \\
&= \sum_{k=0}^{n} a_k \mathcal{D}_x \left(x^k \right) && \text{by Scalar Rule} \\
&= \sum_{k=0}^{n} (a_k)(k) x^{k-1} && \text{by Power Rule} \\
&= 0 + \sum_{k=1}^{n} (a_k)(k) x^{k-1} \\
&= \sum_{k=1}^{n} (a_k)(k) x^{k-1}
\end{aligned}
$$

1.2.8 Differentiating a Power Series

Corollary 1.2.12 differentiates a polynomial term by term, but a polynomial is a *finite* sum: the Summation Rule it rests on (Theorem 1.2.6) only ever combines two functions at a time, extended to n terms by repeated application. An infinite sum,

$$f(x) = \sum_{n=0}^{\infty} a_n (x - c)^n \,,$$

called a *power series*, is a genuinely different kind of object: it is not a finite chain of additions but a *limit* of one — $f(x)$ *is*, by definition,

$$f(x) = \lim_{N\to\infty} \sum_{n=0}^{N} a_n (x-c)^n \,.$$

The question is whether and when, writing $f_n(x) = a_n(x-c)^n$ for the n-th term,

$$\mathcal{D}_x \sum_{n=0}^{\infty} f_n(x) = \sum_{n=0}^{\infty} \mathcal{D}_x f_n(x) \,?$$

The notation makes it very tempting to simply swap the $\mathcal{D}_x$ and the $\sum_{n=0}^{\infty}$ symbols, treating the infinite sum exactly like the finite ones the Summation Rule was built for. But extending a rule proved for finitely many terms to infinitely many terms all at once is a substantially stronger claim, not an automatic consequence — nothing proven so far justifies it. Doing so silently swaps two different limiting processes (the one that defines the series, and the one that defines the derivative), and swapping limits is not automatically valid in general. Once the precise definition of the derivative is available (Chapter 7), Section 7.5.1 spells out exactly what this swap involves and why it needs its own justification.

A power series may diverge for some values of x. There is a number $R \in [0,\infty]$, the series' *radius of convergence*, such that the series converges (absolutely) for every x with $|x-c| < R$ and diverges for every x with $|x-c| > R$ — $R = 0$ means the series converges only at $x = c$ itself, and $R = \infty$ means it converges for every real x. This is exactly the condition that makes term-by-term differentiation valid.

Theorem 1.2.13 (*Differentiation of a Power Series*)

Suppose $f(x) = \sum_{n=0}^{\infty} a_n (x-c)^n$ has radius of convergence $R > 0$ (possibly $R = \infty$). Then for every x with $|x-c| < R$, f is differ-

entiable at x, and

$$\mathcal{D}_x f(x) = \sum_{n=1}^{\infty} n\, a_n (x-c)^{n-1},$$

the series obtained by differentiating term by term. Moreover, the differentiated series has the *same* radius of convergence R.

The "same radius of convergence" half of this theorem is not hard to see: R is determined by $\limsup_{n\to\infty} |a_n|^{1/n}$, and differentiating multiplies the n-th coefficient by n; since $\lim_{n\to\infty} n^{1/n} = 1$, multiplying every coefficient by n does not change that lim sup at all. The harder half — that the differentiated series' sum is *actually equal* to $f'(x)$, and not merely *some* number — is where the swap of limits genuinely needs justifying, and a rigorous justification requires *uniform convergence*, a topic from real analysis this course does not cover. We use the theorem freely from here on, exactly as we already use the Product Rule (Theorem 1.2.8) without an elementary proof at this stage, deferring the genuine proof to Theorem 7.4.6 in the optional Chapter 7.

Every power series used in this course — for e^x, e^{-x}, $\sin x$, and $\cos x$ — has radius of convergence $R = \infty$: they converge, and may be differentiated term by term, for *every* real x. The condition in the theorem above is never actually a restriction in this course, but it is not vacuous in general — the geometric series $\sum_{n=0}^{\infty} x^n$, for instance, has $R = 1$ and says nothing at all about $x = 2$.

1.2.9 Example: Differentiating a Power Series

Several important functions in this course — e^x, $\sin x$, $\cos x$ among them — are given not by a finite formula but by an *infinite* power series (a Taylor series). Theorem 1.2.13 justifies differentiating each of them term by term. We work three examples using only the Power

Rule and the Sum/Difference Rule — no new machinery required.

Recall $e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$. Differentiating term by term:

$$\begin{aligned}
\mathcal{D}_x e^x &= \mathcal{D}_x 1 + \mathcal{D}_x x + \mathcal{D}_x \frac{x^2}{2!} + \mathcal{D}_x \frac{x^3}{3!} + \mathcal{D}_x \frac{x^4}{4!} + \cdots \\
&= 0 + 1 + \frac{2x}{2!} + \frac{3x^2}{3!} + \frac{4x^3}{4!} + \cdots \\
&= 0 + 1 + \frac{2x}{2} + \frac{3x^2}{3 \times 2!} + \frac{4x^3}{4 \times 3!} + \cdots \\
&= \qquad 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots = e^x .
\end{aligned}$$

More explicitly,

$$\begin{aligned}
\mathcal{D}_x e^x &= \mathcal{D}_x \left(\sum_{n=0}^{\infty} \frac{x^n}{n!} \right) = \mathcal{D}_x \left(\frac{x^0}{0!} + \sum_{n=1}^{\infty} \frac{x^n}{n!} \right) \\
&= \mathcal{D}_x \left(1 + \sum_{n=1}^{\infty} \frac{x^n}{n!} \right) = 0 + \sum_{n=1}^{\infty} \mathcal{D}_x \left(\frac{x^n}{n!} \right) \\
&= \sum_{n=1}^{\infty} n \frac{x^{n-1}}{n!} = \sum_{n=1}^{\infty} \frac{x^{n-1}}{(n-1)!} \\
&= \sum_{n=0}^{\infty} \frac{x^n}{n!} = e^x
\end{aligned}$$

The series reproduces itself: differentiating e^x gives back e^x.

Now substitute $-x$ for x *in the series itself*, before differentiating anything:

$$e^{-x} = \sum_{n=0}^{\infty} \frac{(-x)^n}{n!} = \sum_{n=0}^{\infty} (-1)^n \frac{x^n}{n!} = 1 - x + \frac{x^2}{2!} - \frac{x^3}{3!} + \cdots .$$

Differentiating this term by term:

$$\mathcal{D}_x e^{-x} = -1 + x - \frac{x^2}{2!} + \cdots = -\left(1 - x + \frac{x^2}{2!} - \cdots\right) = -e^{-x}\,.$$

More explicitly,

$$\begin{aligned}
\mathcal{D}_x e^{-x} &= \mathcal{D}_x\left(\sum_{n=0}^{\infty} \frac{(-1)^n x^n}{n!}\right) = \mathcal{D}_x\left(\frac{x^0}{0!} + \sum_{n=1}^{\infty} \frac{(-1)^n x^n}{n!}\right) \\
&= \mathcal{D}_x\left(1 + \sum_{n=1}^{\infty} \frac{(-1)^n x^n}{n!}\right) = 0 + \sum_{n=1}^{\infty} \mathcal{D}_x\left(\frac{(-1)^n x^n}{n!}\right) \\
&= \sum_{n=1}^{\infty} (-1)^n \mathcal{D}_x\left(\frac{x^n}{n!}\right) = \sum_{n=1}^{\infty} (-1)^n n \frac{x^{n-1}}{n!} \\
&= \sum_{n=1}^{\infty} (-1)^n \frac{x^{n-1}}{(n-1)!} = \sum_{n=0}^{\infty} (-1)^{n+1} \frac{x^n}{n!} \\
&= \sum_{n=0}^{\infty} (-1)(-1)^n \frac{x^n}{n!} = -\sum_{n=0}^{\infty} (-1)^n \frac{x^n}{n!} = -e^{-x}
\end{aligned}$$

Once you have the Chain Rule from Chapter 2, you can reach the same answer a second way: $\mathcal{D}_x e^{-x} = e^{-x} \cdot \mathcal{D}_x(-x) = -e^{-x}$ — worth revisiting then, to see exactly where the extra minus sign comes from.

Finally, let $g(x) = e^x + e^{-x}$ and $h(x) = e^x - e^{-x}$. No new series work is needed here either — just the Sum and Difference Rules applied to the two derivatives just computed:

$$\begin{aligned}
\mathcal{D}_x g(x) &= \mathcal{D}_x e^x + \mathcal{D}_x e^{-x} = e^x - e^{-x} = h(x)\,, \\
\mathcal{D}_x h(x) &= \mathcal{D}_x e^x - \mathcal{D}_x e^{-x} = e^x + e^{-x} = g(x)\,.
\end{aligned}$$

Differentiating g gives h, and differentiating h gives back g — the two functions swap into each other under differentiation. (Up to a factor

of 2, g and h are the *hyperbolic cosine* and *hyperbolic sine*, written $\cosh x$ and $\sinh x$; the name is not needed for anything below.) You will see this exact pattern again, with one added twist — alternating signs — when we compute the derivative of $\sin x$ later in the course.

1.2.10 Derivative of a Quotient

If $f(x) = \frac{h(x)}{g(x)}$ is the quotient of two differentiable functions, we can compute the quotient of the derivative, which is valid everywhere except where the denominator is zero. To prove the Quotient rule, we need the Chain Rule and the Power rule working together — so the proof is deferred to Chapter 2, right after the Chain Rule is introduced there.

Theorem 1.2.14 (*Quotient Rule*)

If $f(x) = \frac{h(x)}{g(x)}$, then

$$\mathcal{D}_x f(x) = \frac{g(x)\mathcal{D}_x h(x) - h(x)\mathcal{D}_x g(x)}{[g(x)]^2},$$

or more concisely

$$\mathcal{D}\,\frac{h}{g} = \frac{g\mathcal{D}\,h - h\mathcal{D}\,g}{g^2}.$$

1.3 Some Special Functions

There are several other functions the student should learn the derivative of. A summary of the derivatives which need to be committed to memory is given in Figure 2.1 in Chapter 2 (once the Chain Rule is available, the table can finally be shown in full).

Theorem 1.3.1 (*Derivative of* e^x)

One very important derivative to remember is the unique function which is the identity of the derivative operator—it is the only non-zero function which is the derivative of itself.

$$\mathcal{D}_x e^x = e^x \,.$$

This property makes e^x extraordinary—it is unchanged by differentiation, which means it describes processes whose rate of change is proportional to their current value. Exponential growth and decay, compound interest, radioactive decay, and the analysis of algorithms all involve this function.

We already proved this case above, in Section 1.2.9 (*Example: Differentiating a Power Series*), by differentiating the power series for e^x term by term. The same technique is applied to $\cos x$ in the homework problem *Derivatives via Taylor Series*, Section 4.1.2 of Chapter 4.

Theorem 1.3.2 (*Derivatives of* sin *and* cos)

$$\mathcal{D}\sin = \cos$$
$$\mathcal{D}\cos = -\sin$$

We can compute $\mathcal{D}\tan$ by applying the quotient rule to $\tan = \frac{\sin}{\cos}$. In particular

$$\mathcal{D}\tan = \sec^2$$

Corollary 1.3.3 (*Derivative of* tan)

$$\mathcal{D}\tan = \sec^2$$

Proof.

$$\begin{aligned}
\mathcal{D}\tan &= \mathcal{D}\frac{\sin}{\cos} \\
&= \frac{\cos \mathcal{D}\sin - \sin \mathcal{D}\cos}{\cos^2} \\
&= \frac{(\cos)(\cos) - (\sin)(-\sin)}{\cos^2} \\
&= \frac{\cos^2 + \sin^2}{\cos^2} \\
&= \frac{1}{\cos^2} = \sec^2
\end{aligned}$$

There is one more class of functions, the inverse trig functions, which are interesting, as many have elegant derivatives. These functions come in useful in Section 14.1 when we look at partial fraction decomposition. Many of these functions, such as arctan, and their derivatives are shown in Figure 2.1.

1.4 Anti-derivatives

If you have a function and you want to know its derivative, you can in many cases look up the derivative in a table such as Figure 2.1. If the function is not directly found in the table, you can express the function as a composition of several simpler functions which are found in the table. However, if you have f, and you know that f is the derivative of some other function, figuring that out from a derivative table is difficult, except in a few simple cases.

Such a function is called an anti-derivative. We say

an anti-derivative because any time $\mathcal{D}_x f(x) = g(x)$, then also $\forall\, C \in \mathbb{R}, \mathcal{D}_x(f(x) + C) = g(x)$. Thus if $f(x)$ is an anti-derivative of $g(x)$ then so is $f(x) + C$. In this sense, anti-derivatives are not unique.

The chain rule allows us to express the derivative of a composition as product of compositions. However, the chain rule DOES NOT allow us to express the anti-derivative of a composition as a composition of anti-derivatives. **Computing an anti-derivative is difficult in general.**

Nevertheless, we can compute the anti-derivatives of polynomials and simple trig functions using Table 2.1. If the derivative is $\mathcal{D}_x c x^n = n c x^{n-1}$, then a little algebra shows an anti-derivative of $c x^n = \frac{c}{n+1} x^{n+1}$, but this formula assumes $n + 1 \neq 0$. Thus we cannot compute an anti-derivative of $x^{-1} = \frac{1}{x}$ by the power rule. Plugging $n = -1$ into the formula requires dividing by $n + 1 = 0$, which is undefined. The anti-derivative of x^{-1} is certainly not $\frac{1}{0} x^0$ — the power rule simply breaks down here. The power rule explicitly supposes $n + 1 \neq 0$, and x^{-1} violates that assumption.

Does the fact that the power rule does not apply to x^{-1} mean that $\frac{1}{x}$ has no anti-derivative? To say that $\frac{1}{x}$ has not anti-derivative is to say that no function has $\frac{1}{x}$ as slope.

In fact, there is another function, namely $\ln|x|$, listed in the table, which is an anti-derivative of $\frac{1}{x}$. The two functions $\frac{1}{x}$ and $\ln|x|$ are illustrated in Figure 1.7.

Looking at the curve for $\frac{1}{x}$ it is clear that there exists some function, f (many in fact if we take into account constant offsets) such that $\frac{1}{x}$ is the slope of f at the point x. In fact the figure shows such a function which is labeled $\ln|x|$.

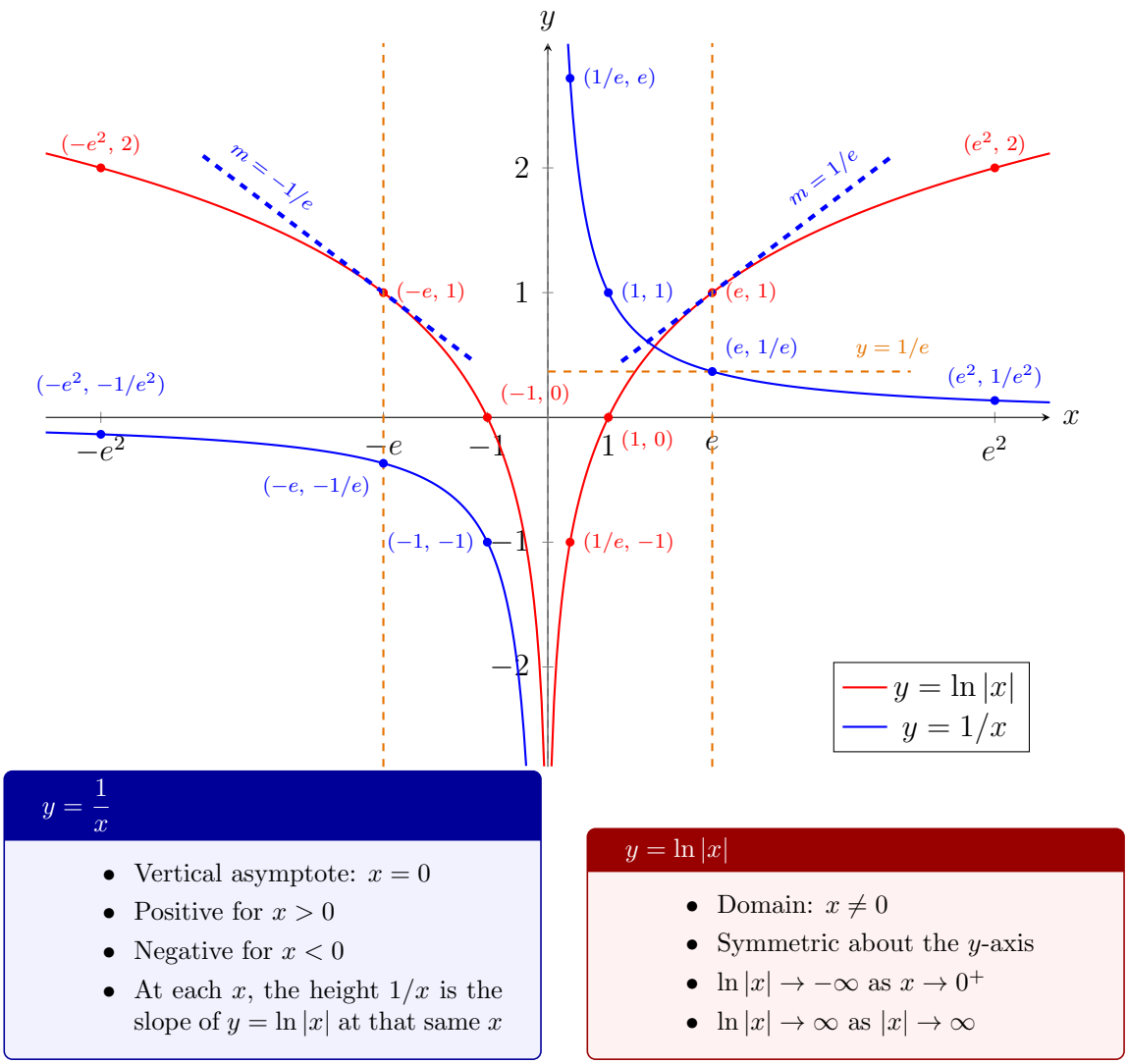


Figure 1.7: Graphs of $y = \ln|x|$ (red) and $y = \dfrac{1}{x}$ (blue). Both functions have a vertical asymptote at $x = 0$ (dashed).

1.5 Homework

`homework/secant.py` is an individually graded Intra assignment implementing the secant-line approximation of Section 1.1.1 procedurally, as a sequence of secant slopes converging to a derivative. The tests you are graded against are in `tests/secant_test.py`.

- `secant_slope(f, x1, x2)`: the slope of the secant line through $(x_1, f(x_1))$ and $(x_2, f(x_2))$, i.e. $m_{approx} = \dfrac{f(x_2) - f(x_1)}{x_2 - x_1}$ from Section 1.1.1.
- `converging_x_values(x1, x2)`: an infinite generator of x-values starting at x_2 and converging to x_1 — the first value yielded is x_2 itself, and each value after that is the average of the previous value and x_1.
- `approximate_derivative(f, x1, x2, epsilon)`: approximates $f'(x_1)$ by computing `secant_slope(f, x1, x)` for successive x from `converging_x_values(x1, x2)`, stopping once two consecutive slopes differ by less than `epsilon`.

? Open Questions

- We said computing an anti-derivative is "difficult in general," but difficult how? Is it just that we haven't found the right trick yet, or can a function provably have *no* anti-derivative expressible in elementary terms? (It's the latter — Chapter 16 answers this precisely, via the Risch algorithm.)
- The Power Rule was extended from integer exponents to rational exponents $x^{a/b}$ using the Chain Rule. What about irrational exponents, like x^{π}? Does $\mathcal{D}_x x^{\pi}$ even mean anything, and if so, how would you compute it? (This is answered in Chapter 7.)
- We claimed every differentiable function has a unique tangent line at every point of its domain. What does the graph of a function look like at a point where *no* tangent line exists, or where the tangent line is vertical? Try sketching $f(x) = |x|$ at $x = 0$ and $f(x) = x^{1/3}$ at $x = 0$ before reading further.

Chapter 2

Composition and the Chain Rule

Chapter Objectives

- Compute the derivative of a composition of functions using the Chain Rule, in point-free, Newtonian, and Leibniz notation.
- Prove the Quotient Rule, using the Chain Rule together with the Power Rule.
- Apply implicit differentiation to find dy/dx when y is defined by a relation $F(x, y) = 0$ rather than an explicit formula.
- Prove the derivative of $\arctan$ using implicit differentiation.

Chapter 1 covered every reduction rule except one: what happens when you differentiate a *composition* of functions, $h(g(x))$? Before we can answer that, it is worth understanding *composition itself* — how it behaves, and how it does not — since *applying* the rule correctly (recognizing which function is "outer" and which is "inner") is one of the more error-prone skills in this course, and that confusion usually comes from not having a solid feel for composition as an operation in its own right. Once we do, the rule for differentiating it (the Chain Rule) follows, and two things follow directly from *that*: a proof of the Quotient Rule (stated without proof in Chapter 1), and *implicit*

differentiation — a technique for finding dy/dx even when y is only defined implicitly by an equation, not by an explicit formula.

2.1 Composition of Functions

Let $g : I \to \mathbb{R}$ be defined on all of some interval I on the real line, or perhaps on all of the real line. Let J denote the image of g, $J = \{g(x) \mid x \in I\}$, and let $h : J \to \mathbb{R}$. Then $x \in I \implies h(g(x)) \in \mathbb{R}$, so the following defines a function.

Definition 2.1.1 (*Composition*)

The *composition* of h and g, written $h \circ g$, is the function that takes x as input and outputs $h(g(x))$:

$$(h \circ g)(x) = h(g(x)) \,.$$

2.1.1 The `compose` Function in Python

Everything above is easy to turn into runnable code, because Python functions are themselves values that can be passed around and returned. Composition is a two-line higher-order function:

```
def compose(f, g):
    return lambda x: f(g(x))
```

`compose(f, g)` builds and returns a *new* function — one that first applies `g`, then applies `f` — exactly matching $(f \circ g)(x) = f(g(x))$. Every algebraic property of composition proven in this section is directly checkable by running code built from this one function — not just symbolic claims. Sections 2.1.2, 2.1.3, and 2.1.6 each do exactly that, right after the corresponding proof.

This `compose` is intentionally the simplest possible implementation: it evaluates immediately, has no way to inspect its own structure, and cannot be differentiated, serialized to LaTeX, or pattern-matched. Chapter 5 builds something more elaborate for exactly that reason — a `Compose` AST *node*, not a Python closure — so that composition becomes a piece of *data* you can inspect and transform, not just a function you can call.

2.1.2 Composition Is Not Commutative

Swap which function is outer and which is inner, and you get a genuinely different function — not just a relabeling. Take $g(x) = x^2$ and $h(t) = \sin t$:

$$(h \circ g)(x) = \sin(x^2)\,, \qquad (g \circ h)(x) = (\sin x)^2\,.$$

$\sin(x^2) \neq (\sin x)^2$ as functions — compare their values at, say, $x = \frac{\pi}{2}$: $\sin\left(\frac{\pi^2}{4}\right) \approx 0.624$, while $\left(\sin\frac{\pi}{2}\right)^2 = 1$. In general, $h \circ g \neq g \circ h$: *composition is not commutative*. This is exactly why identifying which function is "outer" and which is "inner" is not a bookkeeping detail; get it backwards and you get a different function entirely. (Once we have the Chain Rule, Section 2.2.1 confirms their *derivatives* differ too.)

Using `compose` from Section 2.1.1, this is not just a symbolic claim; it is directly observable by running the code:

```python
import math

def square(x):
    return x ** 2

sin_after_square = compose(math.sin, square)    # sin(x**2)
square_after_sin = compose(square, math.sin)    # sin(x)**2

sin_after_square(math.pi / 2)    # 0.6242659526396992
square_after_sin(math.pi / 2)    # 1.0
```

The same two numbers computed above by hand.

2.1.3 Composition Is Associative

The other half of the monoid structure Chapter 5 builds on is associativity: it does not matter how you parenthesize a chain of compositions.

Corollary 2.1.2 (*Composition Is Associative*)

For all functions f, g, h,

$$(f \circ g) \circ h = f \circ (g \circ h)\,.$$

Proof. Directly from the definition, both sides send x to the same value:

$$\big((f \circ g) \circ h\big)(x) = (f \circ g)\big(h(x)\big) = f\big(g(h(x))\big)\,,$$

$$\big(f \circ (g \circ h)\big)(x) = f\big((g \circ h)(x)\big) = f\big(g(h(x))\big)\,.$$

Both reduce to $f(g(h(x)))$, so the two sides are the same function.

Unlike commutativity, associativity holds unconditionally — no exception like id is needed here.

Using the `compose` function from Section 2.1.1, this is checkable directly: three simple functions, composed in two different groupings.

```
def add_one(x):
    return x + 1

def double(x):
    return x * 2

def square(x):
    return x ** 2

left  = compose(compose(add_one, double), square)
right = compose(add_one, compose(double, square))

left(3)     # 19
right(3)    # 19
left is right   # False
```

Both groupings compute `add_one`(`double`(`square`(x))) — the same nested call, just parenthesized differently when it was built — so they agree at $x = 3$, and at every other x too. Yet `left` and `right` are two *different* function objects — different closures, built by different nested calls to `compose` — that happen to compute the same value for every input. That is exactly the structural-versus-semantic distinction Chapter 5 makes about AST trees (`Compose(Compose(f,g),h) == Compose(f,Compose(g,h))` is also `False` there, for the same reason), showing up here one level down, at the level of plain Python closures instead of an AST.

Composing across different types. Nothing above required f, g, h to share a domain or codomain — every proof went through for arbitrary functions. In general, let

$$h : A \to B\,, \qquad g : B \to C\,, \qquad f : C \to D\,.$$

Then $g \circ h : A \to C$ and $f \circ g : B \to D$, and both associativity group-

ings are functions $A \to D$:

$$(f \circ g) \circ h : A \to D\,, \qquad f \circ (g \circ h) : A \to D\,.$$

In Python's typing notation, this is

$$h : \texttt{Callable[[A], B]}\,,$$
$$g : \texttt{Callable[[B], C]}\,,$$
$$f : \texttt{Callable[[C], D]}\,,$$

and both `compose(compose(f, g), h)` and `compose(f, compose(g, h))` have type `Callable[[A], D]`.

Take four genuinely different types: $A =$ `float`, $B =$ `bool`, $C =$ `str`, $D =$ `tuple`.

```python
def h(x: float) -> bool:
    return x > 0

def g(b: bool) -> str:
    return "positive" if b else "non-positive"

def f(s: str) -> tuple:
    return (s, len(s))

left  = compose(compose(f, g), h)
right = compose(f, compose(g, h))

left(3.5)     # ('positive', 8)
right(3.5)    # ('positive', 8)

left(-2.0)    # ('non-positive', 12)
right(-2.0)   # ('non-positive', 12)
```

A `float` goes in, a `tuple` comes out, passing through `bool` and `str` along the way — and the two groupings still agree at every step, exactly as the arrow-notation argument above guarantees.

The types force more than just the same final answer: they force `f`, `g`, and `h` to be called in exactly the *same order* regardless of grouping.

Of the three functions, only `h` accepts a `float`, so `h` must run first, producing a `bool`. Of the remaining two, only `g` accepts a `bool`, so `g` must run second, producing a `str`. That leaves `f`, the only one that accepts a `str`, to run last. Neither $(f \circ g) \circ h$ nor $f \circ (g \circ h)$ has any freedom to call them in a different order or on different intermediate values — the types pin down h then g then f uniquely. So it is not merely that both groupings compute the same result: they compute it by running the *identical chain of calls*, $f(g(h(x)))$, every time. Associativity of composition with matching, generic types gives you the same value from what could in principle be different computations; associativity across genuinely distinct types like this forces the computation itself to be identical.

2.1.4 Typing `compose` Itself

`h`, `g`, and `f` above each had fixed, concrete types (`float`, `bool`, `str`, `tuple`). `compose` itself is different: it has to work for *any* choice of types — when we wrote it in Section 2.1.1, we had no idea yet whether it would end up composing functions over `float`s and `bool`s, or over some completely different pair of types. Python's type system expresses "works for any type, but the types still have to line up" with a *type variable*: a placeholder that stands for "some type, to be determined later by how the function is actually called," rather than for one fixed type like `float`.

```python
from typing import Callable, TypeVar

A = TypeVar("A")
B = TypeVar("B")
C = TypeVar("C")

def compose(f: Callable[[B], C],
            g: Callable[[A], B]
           ) -> Callable[[A], C] :
    return lambda x: f(g(x))
```

`A`, `B`, `C` are not real types the way `float` or `str` are — they are stand-ins, exactly like the A, B, C in the arrow notation $h : A \to B$, $g : B \to C$ from earlier in this section, which were never specific types either. When `compose` is actually called with concrete functions, a type checker such as PyCharm's built-in inspector (or the standalone tool `mypy`) works out what `A`, `B`, `C` must be *for that call* — $A =$ `float`, $B =$ `bool`, $C =$ `str` for the `h` / `g` / `f` example above — and checks that everything is consistent with that choice. The same generic `compose` is reused, unchanged, for a completely different triple of types the next time it is called; only `compose`'s own definition needs the type variables, not each call site.

2.1.5 Composing with a Constant Function

A *constant function* c ignores its input entirely: $c(t) = k$ for every t, for some fixed number k. Composing with one collapses the composition to a constant — in *either* order, though generally not to the *same* constant.

Corollary 2.1.3 (*Composing with a Constant*)

Let $c(t) = k$ for all t. For every function g,

$$(c \circ g)(x) = k \quad \text{for all } x\,, \qquad (g \circ c)(x) = g(k) \quad \text{for all } x\,.$$

Both $c \circ g$ and $g \circ c$ are themselves constant functions.

Proof. $(c \circ g)(x) = c(g(x)) = k$, since c returns k no matter what it is given — in particular, no matter what $g(x)$ is. $(g \circ c)(x) = g(c(x)) = g(k)$, since $c(x) = k$ for every x, so g is always evaluated at the same point, k.

Take $c(t) = 3$ and $g(x) = x^2$. It is tempting to read "3 composed with x^2" as the product $3x^2$ — it is not.

$$(c \circ g)(x) = c(x^2) = 3 \quad \text{for every } x\,,$$
$$(g \circ c)(x) = g(3) = 9 \quad \text{for every } x\,.$$

Two different constants (3 and 9), neither of them $3x^2$ — composing with a constant is not the same operation as multiplying by one, even though the two can look similar written down. (Once we have the Chain Rule, Section 2.2.1 checks that differentiating agrees.)

Using `compose` from Section 2.1.1:

```
def c(t):
    return 3

def g(x):
    return x ** 2

c_after_g = compose(c, g)   # the constant function 3
g_after_c = compose(g, c)   # the constant function 9

c_after_g(2)      # 3
c_after_g(100)    # 3, still --- the input never mattered
g_after_c(2)      # 9
g_after_c(100)    # 9, still --- also never mattered
```

A tempting mistake. Given `g_after_c = compose(g, c)` above, it is tempting to skip defining the constant function `c` at all and just hand `compose` the number directly:

```
mistake = compose(g, 3.4)
```

Without type declarations, `compose` (Section 2.1.1) accepts this without complaint — `f` and `g` were never restricted to be functions in the first place, so passing a plain `float` where `c` belongs raises no error yet. `compose` just builds and returns the closure `lambda x: g(3.4(x))`, without ever trying to call `3.4` as a function. The bug only surfaces later, if and when that closure is actually called:

```
mistake(2)
# TypeError: 'float' object is not callable
```

By then, the error message points at the *call* to `mistake`, not at the line `compose(g, 3.4)` that actually contains the mistake — which, in a longer program, could be far away, in a different function entirely.

Now contrast this with the typed `compose` from Section 2.1.4. There, `g`'s parameter is annotated `Callable[[A], B]`, and a type checker such as PyCharm's built-in inspector (or the standalone tool `mypy`) can tell, on sight, that `3.4` (type `float`) does not match `Callable[[A], B]` for any choice of `A`, `B`:

```
mistake = compose(g, 3.4)
# Expected type 'Callable[[A], B]', got 'float' instead
```

This warning appears *as the line is typed*, before the program is ever run — not after tracing a runtime crash back to its source. This is exactly the practical payoff of the type variables from Section 2.1.4: they let the type checker catch this whole category of mistake immediately, at the exact line where it was made, instead of at some unrelated call site later on.

2.1.6 Composing with the Identity

Section 2.1.2 showed that composition order matters — $h \circ g$ and $g \circ h$ are generally different functions. The identity function $\mathrm{id}(x) = x$ is the one exception: composing with it, in *either* order, changes nothing.

Corollary 2.1.4 (*Two-Sided Identity for Composition*)

For every function f,

$$(f \circ \mathrm{id})(x) = f(x) \qquad \text{and} \qquad (\mathrm{id} \circ f)(x) = f(x)\,,$$

i.e. $f \circ \mathrm{id} = \mathrm{id} \circ f = f$.

Proof. Directly from the definitions. $(f \circ \mathrm{id})(x) = f(\mathrm{id}(x)) = f(x)$, since $\mathrm{id}(x) = x$. And $(\mathrm{id} \circ f)(x) = \mathrm{id}(f(x)) = f(x)$, since id applied to *anything* returns that thing unchanged — in particular, applied to $f(x)$.

Using `compose` from Section 2.1.1, the identity law is just as checkable:

```
import math

def identity(x):
    return x

compose(math.sin, identity)(1.0) == math.sin(1.0)    # True
compose(identity, math.sin)(1.0) == math.sin(1.0)    # True
```

Together with Corollary 2.1.2, this identity law is exactly what makes composition a monoid — Section 2.1.7 makes that precise. (Once we have the Chain Rule, Section 2.2.1 checks that differentiating agrees with this too.)

2.1.7 Composition Forms a Monoid

Associativity (Corollary 2.1.2) and a two-sided identity (Corollary 2.1.4) are exactly the two properties that define a familiar algebraic structure, and nothing in either proof was specific to real-valued functions.

Definition 2.1.5 (*Monoid*)

A set M equipped with a binary operation $\triangle\colon M \times M \to M$ is a *monoid* if:

1. $\triangle$ is associative: $(a \triangle b) \triangle c = a \triangle (b \triangle c)$ for all $a, b, c \in M$.
2. There exists an identity element $e \in M$ such that $e \triangle a = a \triangle e = a$ for all $a \in M$.

Proposition 2.1.6 (*Self-Maps, Monoid Under Composition*)

For any set A, the set of functions $f : A \to A$, together with $\circ$ and id_A, forms a monoid.

Proof. Composing two functions $A \to A$ gives another function $A \to A$, so $\circ$ is a binary operation on this set. Associativity is Corollary 2.1.2, and id_A is a two-sided identity for $\circ$ by Corollary 2.1.4.

This shape of claim shows up constantly once you know to look for it: integers under multiplication (identity 1), strings under concatenation (identity the empty string), matrices under multiplication (identity the identity matrix) — all monoids, all for the same two reasons, on sets that have nothing else to do with each other. The pattern is useful precisely because it is so generic: any algorithm that only relies on associativity and an identity — fast exponentiation by squaring, for instance — works unchanged on *any* monoid. Real-valued functions

under composition are one instance of the pattern; Chapter 5, Section 5.5 meets another, `PointFree` expression trees under `Compose`, and puts the genericity to direct use.

2.1.8 Composing with a Sum

Composition is one operation, addition of functions is another — does one distribute over the other? There are two ways a sum could sit next to a composition, depending on which side of $\circ$ it is on:

$$(f_1 + f_2) \circ g \stackrel{?}{=} (f_1 \circ g) + (f_2 \circ g)\,, \qquad f \circ (g_1 + g_2) \stackrel{?}{=} (f \circ g_1) + (f \circ g_2)\,.$$

Exactly one of these holds for *every* choice of functions; the other fails in general.

Corollary 2.1.7 (*Composing a Sum on the Outside*)

For all functions f_1, f_2, g,

$$(f_1 + f_2) \circ g = (f_1 \circ g) + (f_2 \circ g)\,.$$

Proof. Addition of functions is defined pointwise: $(f_1 + f_2)(t) = f_1(t) + f_2(t)$ for every t. Apply this at $t = g(x)$:

$$\begin{aligned} \big((f_1 + f_2) \circ g\big)(x) &= (f_1 + f_2)\big(g(x)\big) \\ &= f_1\big(g(x)\big) + f_2\big(g(x)\big) \\ &= (f_1 \circ g)(x) + (f_2 \circ g)(x)\,. \end{aligned}$$

The other direction is *not* valid in general: $f \circ (g_1 + g_2) \neq (f \circ g_1) + (f \circ g_2)$. A counterexample is enough to show this, since the claim is that it holds for *every* f, g_1, g_2. Take

$f(t) = t^2$, $g_1(x) = 2x$, $g_2(x) = x + 1$, and $x = 1$:

$$\big(f \circ (g_1 + g_2)\big)(1) = f\big(g_1(1) + g_2(1)\big) = f(2 + 2) = f(4) = 16\,,$$

$$(f \circ g_1)(1) + (f \circ g_2)(1) = f(2) + f(2) = 4 + 4 = 8\,.$$

$16 \neq 8$. The difference is exactly $f(a + b)$ versus $f(a) + f(b)$: true for every f only when f is linear, which is not assumed here.

The numeric counterexample above shows the invalid direction can give the *wrong number*. With mismatched types, it can fail even more starkly: the "distributed" side is not just wrong, it is not even *well-typed*. Let $g_1, g_2 :$ `float` $\to$ `float` (so $g_1 + g_2$ is a well-defined `float` $\to$ `float` function), but let $f :$ `float` $\to$ `set`. Then $f \circ (g_1 + g_2)$ is perfectly well-typed — `float` $\to$ `set` — since no addition ever happens on the `set` side. But $(f \circ g_1) + (f \circ g_2)$ requires adding *two sets* with `+`, and Python sets do not support that operator at all:

```
def g1(x):
    return x

def g2(x):
    return x ** 2

def f(t):
    return {t}    # float -> set

# Valid direction: still well-typed, float -> set
lhs = compose(f, sum_of(g1, g2))
lhs(3)    # {12}

# Invalid direction: doesn't even run
rhs = sum_of(compose(f, g1), compose(f, g2))
rhs(3)    # TypeError: unsupported operand type(s) for +: 'set' and '
                                    set'
```

$f \circ (g_1 + g_2)$ never asked `set` to support `+`; the invalid rewrite does, unconditionally, whether or not the codomain of f happens to support addition. That is a second, independent way the two rewrites are not

interchangeable: one is well-typed for *every* choice of f, g_1, g_2, and the other is not even guaranteed to type-check.

Using `compose` from Section 2.1.1, plus a one-line `sum_of` for pointwise addition of functions:

```
def sum_of(f1, f2):
    return lambda x: f1(x) + f2(x)

def square(x):
    return x ** 2

def double(x):
    return x * 2

def add_one(x):
    return x + 1

# Valid: (f1 + f2) o g  ==  (f1 o g) + (f2 o g)
valid_left  = compose(sum_of(square, double), add_one)
valid_right = sum_of(compose(square, add_one), compose(double,
                                    add_one))

# Invalid in general: f o (g1 + g2)  !=  (f o g1) + (f o g2)
invalid_left  = compose(square, sum_of(double, add_one))
invalid_right = sum_of(compose(square, double), compose(square,
                                    add_one))

valid_left(1) == valid_right(1)       # True, and at every other x
                                    too
invalid_left(1) == invalid_right(1)   # False: 16 != 8
```

You will meet this exact question again in Chapter 6, Section 6.6.1's `distribute` homework, where `Compose(Sum(f1,f2), g)` and `Compose(f, Sum(g1,g2))` are two candidate AST rewrites and only one of them is a valid transformation to implement. The proof and counterexample above are exactly the calculus-side reasoning that settles which.

2.2 The Chain Rule

Even though it may come as a surprise, we can compute the derivative of $(h \circ g)$ in a simple, straightforward way, called the Chain Rule.

Theorem 2.2.1 (*Chain Rule*)

Let $g : \mathbb{R} \to \mathbb{R}$, and let $h : \mathit{domain}(g) \to \mathbb{R}$, then

$$\mathcal{D}\,(h \circ g) = (\mathcal{D}\,h) \circ g \cdot \mathcal{D}\,g\,.$$

Theorem 2.2.1 is stated in point-free form: both sides are functions, with no input variable named. The pointwise form, naming the variable x explicitly, is

$$\mathcal{D}_x(h \circ g)(x) = \mathcal{D}_x h(g(x)) \cdot \mathcal{D}_x g(x)\,.$$

To apply the Chain Rule can be tricky at first. We must determine exactly what the two functions are which are being composed. Let's look at an example.

Recall that in Example 1.2.4 we computed the derivative of $f(x) = x^6$ to be $6x^5$. In Example 2.2.2 we observe that $x^6 = (x^3)^2$, and that if we compute the derivative of $(x^3)^2$ using the Chain Rule, then we obtain the same result.

Example 2.2.2 (*Applying the Chain Rule*)

Let h be the function which squares its argument, and let g be the function which cubes its argument. So $h \circ g$ is the function which first cubes its argument, and then squares that result.

$$h(x) = x^2$$
$$g(x) = x^3$$
$$(h \circ g)(x) = (x^3)^2$$

We must remember that h and g are both unary functions, in

the most permissive case $h : \mathbb{R} \to \mathbb{R}$ and $g : \mathbb{R} \to \mathbb{R}$. So those derivatives can be computed independently of each other. Then to apply the chain rule after computing $\mathcal{D}\,h$, we simply apply that function, not to x but rather to $g(x)$. Applying to $g(x)$ is the part which is usually confusing to the student.

One source of confusion is that both h and g are written using the same input variable x. But the actual function does not depend on the naming of its input variable. For example $h(x) = x^2$ is just the function that squares its argument, so it is the same as $h(t) = t^2$. For this we must compute the derivatives of h by computing $\mathcal{D}_t h(t)$ and of g by computing $\mathcal{D}_x g(x)$.

$$\begin{aligned} \mathcal{D}_t h(t) &= \mathcal{D}_t t^2 = 2t^1 = 2t \\ \mathcal{D}_x g(x) &= \mathcal{D}_x x^3 = 3x^2 \end{aligned} \tag{2.1}$$

Now, we are ready to apply the Chain rule. We need to compute $\mathcal{D}_x h(g(x))$ and multiply that by $\mathcal{D}_x g(x)$.

What does it mean to compute $\mathcal{D}_x h(g(x))$? It means simply to replace t by the value of $g(x)$ which is x^3. Substituting x^3 for t in the formula for $\mathcal{D}_x h(t) = 2t$ and get $2x^3$.

As a final step, we must multiply $\mathcal{D}_x h(g(x)) = 2x^3$ by $\mathcal{D}_x g(x) = 3x^2$ (which was computed in Equation (2.1)).

$$\begin{aligned} \mathcal{D}_x(h \circ g)(x) &= \mathcal{D}_x h(g(x)) \cdot \mathcal{D}_x g(x) \\ &= 2x^3 \cdot 3x^2 = 6x^5 \end{aligned} \tag{2.2}$$

Thus, Example 2.2.2 verifies that the two methods of computing the derivative arrive at the same result: $6x^5$.

In the next example we will use two shift identities relating sin and cos — $\cos x = \sin\left(x + \frac{\pi}{2}\right)$ and its mirror image $-\sin x = \cos\left(x + \frac{\pi}{2}\right)$ (using that cos is an even function) — to recover $\mathcal{D}\cos$ from $\mathcal{D}\sin$

alone, without differentiating cos directly.

Example 2.2.3 (*Derivative of* cos *via the Chain Rule*)

We already know $\mathcal{D}\sin = \cos$ (Theorem 1.3.2).

Let $h(t) = \sin t$ and $g(x) = x + \frac{\pi}{2}$, so that $(h \circ g)(x) = \sin\left(x + \frac{\pi}{2}\right) = \cos x$.

$$\begin{aligned} \mathcal{D}_t h(t) &= \cos t \\ \mathcal{D}_x g(x) &= 1 \end{aligned}$$

By the Chain Rule,

$$\begin{aligned} \mathcal{D}_x \cos x &= \mathcal{D}_x (h \circ g)(x) \\ &= \mathcal{D}_x h(g(x)) \cdot \mathcal{D}_x g(x) \\ &= \cos\left(x + \frac{\pi}{2}\right) \cdot 1 \\ &= \cos\left(x + \frac{\pi}{2}\right) \\ &= -\sin x\,, \end{aligned}$$

so $\mathcal{D}_x \cos x = -\sin x$, matching Theorem 1.3.2.

Chapter 1 promised a second look at $\mathcal{D}_x e^{-x}$ once the Chain Rule was available (see the remark right after the Taylor-series computation of $\mathcal{D}_x e^{-x}$ in Section 1.2.9) — here it is, together with the general case e^{ax}.

Example 2.2.4 (*Derivative of* e^{-x} *via the Chain Rule*)

Recall $\mathcal{D}_x e^x = e^x$ (Theorem 1.3.1). Let $h(t) = e^t$ and $g(x) = -x$,

so that $(h \circ g)(x) = e^{-x}$.

$$\mathcal{D}_t h(t) = e^t$$
$$\mathcal{D}_x g(x) = -1$$

By the Chain Rule,

$$\mathcal{D}_x e^{-x} = \mathcal{D}_x h(g(x)) \cdot \mathcal{D}_x g(x) = e^{-x} \cdot (-1) = -e^{-x}\,,$$

matching the Taylor-series computation from Section 1.2.9. The extra minus sign, compared to $\mathcal{D}_x e^x = e^x$, comes entirely from $\mathcal{D}_x g(x) = \mathcal{D}_x(-x) = -1$ — the "inner" derivative contributed by the Chain Rule.

Example 2.2.5 (*Derivative of* e^{ax})

More generally, for any constant a, let $h(t) = e^t$ and $g(x) = ax$, so that $(h \circ g)(x) = e^{ax}$.

$$\mathcal{D}_t h(t) = e^t\,, \qquad \mathcal{D}_x g(x) = a\,.$$

By the Chain Rule,

$$\mathcal{D}_x e^{ax} = \mathcal{D}_x h(g(x)) \cdot \mathcal{D}_x g(x) = e^{ax} \cdot a = a\, e^{ax}\,.$$

Taking $a = 1$ recovers $\mathcal{D}_x e^x = e^x$; taking $a = -1$ recovers Example 2.2.4. The same computation with $g(x) = ax + b$ (so $\mathcal{D}_x g(x) = a$ still, by the Sum and Scalar Rules) gives $\mathcal{D}_x e^{ax+b} = a\, e^{ax+b}$ — one of the Computing Derivatives problems in Chapter 4.

2.2.1 Differentiating What We Composed

With the Chain Rule in hand, we can go back and confirm the three composition facts from Section 2.1 actually behave the way differentiation should expect.

Section 2.1.2 showed $\sin(x^2) \neq (\sin x)^2$ as functions; their derivatives, computed by the same Chain Rule applied in opposite order, are visibly different functions too:

$$\mathcal{D}_x \sin(x^2) = \cos(x^2) \cdot 2x = 2x\cos(x^2)\,, \qquad \mathcal{D}_x (\sin x)^2 = 2\sin(x) \cdot \cos(x)\,.$$

For a constant function $c(t) = k$ composed with any g (Section 2.1.5): since $\mathcal{D}\, c = 0$ (the Constant Rule, the very first differentiation rule in this course),

$$\mathcal{D}_x (c \circ g)(x) = \mathcal{D}\, c(g(x)) \cdot g'(x) = 0 \cdot g'(x) = 0\,,$$

matching $\mathcal{D}_x c \circ g = \mathcal{D}_x c = 0$ directly, since $c \circ g$ is itself just the constant function k.

Example 2.2.6 (*Composition of a Constant*)

In this example we differentiate a composition of a constant in two ways.

Let $c(t) = 5$ and $g(x) = \sin x$, so $g'(x) = \cos x$. Compute $\mathcal{D}_x(c \circ g)(x)$ and $\mathcal{D}_x(g \circ c)(x)$ (a) by the Chain Rule directly, and (b) by composing first and differentiating the result.

***c* ∘ *g*.** By the Chain Rule:

$$\mathcal{D}_x (c \circ g)(x) = \mathcal{D}\, c(g(x)) \cdot g'(x) = 0 \cdot \cos x = 0\,,$$

since $\mathcal{D}\, c = 0$ everywhere. Composing first: $(c \circ g)(x) = c(\sin x) = 5$ for every x — so $c \circ g$ *is* the constant function 5, and $\mathcal{D}_x 5 = 0$. Both routes give 0.

$g \circ c$**.** By the Chain Rule:

$$\mathcal{D}_x(g \circ c)(x) = \mathcal{D}\, g(c(x)) \cdot \mathcal{D}\, c(x) = \cos(5) \cdot 0 = 0\,.$$

Composing first: $(g \circ c)(x) = g(5) = \sin(5)$ for every x — a *different* constant than before, but still a constant, so $\mathcal{D}_x \sin(5) = 0$ (it is just a fixed number). Both routes give 0 again, even though $c \circ g$ and $g \circ c$ are different constant functions.

And for the identity (Section 2.1.6), using $\mathcal{D}_x \mathrm{id}(x) = 1$ (the Identity Rule):

$$\mathcal{D}_x(f \circ \mathrm{id})(x) = \mathcal{D}_x f(\mathrm{id}(x)) \cdot \mathcal{D}_x \mathrm{id}(x) = f'(x) \cdot 1 = f'(x)\,,$$

matching $\mathcal{D}_x f \circ \mathrm{id} = \mathcal{D}_x f$: composing with id is a no-op for the Chain Rule too, contributing a factor of 1 and nothing else.

Example 2.2.7 (*Composition of the Identity*)

In this example we differentiate a composition of the identity function two different ways.

Let $f(x) = \sin x$, so $f'(x) = \cos x$. Compute $\mathcal{D}_x(f \circ \mathrm{id})(x)$ and $\mathcal{D}_x(\mathrm{id} \circ f)(x)$ the same two ways.

$f \circ \mathrm{id}$**.** By the Chain Rule:

$$\mathcal{D}_x(f \circ \mathrm{id})(x) = \mathcal{D}\, f(\mathrm{id}(x)) \cdot \mathcal{D}_x \mathrm{id}(x) = \cos(x) \cdot 1 = \cos x\,.$$

Composing first: $(f \circ \mathrm{id})(x) = f(x) = \sin x$ — so $f \circ \mathrm{id} = f$, and $\mathcal{D}_x \sin x = \cos x$. Both routes give $\cos x$.

$\mathrm{id} \circ f$**.** By the Chain Rule:

$$\mathcal{D}_x(\mathrm{id} \circ f)(x) = \mathcal{D}\, \mathrm{id}(f(x)) \cdot f'(x) = 1 \cdot \cos x = \cos x\,.$$

Composing first: $(\mathrm{id} \circ f)(x) = \mathrm{id}(\sin x) = \sin x$ — so $\mathrm{id} \circ f = f$ too, and again $\mathcal{D}_x \sin x = \cos x$. Both routes give $\cos x$ in this order as well.

2.2.2 Composing with the Power Function

One composition pattern is common enough to deserve its own name: raising an entire expression to a power. $[f(x)]^n$ is itself a composition — the power function is the *outer* function, f is the inner one, exactly as in $(\sin x)^2$ above.

Corollary 2.2.8 (*Extended Power Rule*)

If f is differentiable and n is a real number, then

$$\mathcal{D}_x[f(x)]^n = n\,[f(x)]^{n-1}\,f'(x)\,.$$

Proof. Apply the Chain Rule with outer function $h(t) = t^n$ (so $\mathcal{D}\,h(t) = nt^{n-1}$) and inner function f:

$$\mathcal{D}_x[f(x)]^n = \mathcal{D}_x h(f(x)) = \mathcal{D}\,h(f(x)) \cdot f'(x) = n\,[f(x)]^{n-1}\,f'(x)\,.$$

This is the single most common Chain Rule pattern you will use in practice — far more often than a generic, unnamed composition. The $(\sin x)^2$ computation above is the case $n = 2$, $f = \sin$: $\mathcal{D}_x(\sin x)^2 = 2(\sin x)^1 \cdot \cos x = 2\sin x \cos x$, matching what we computed directly.

One rule versus many. x^n is a special case of $[f(x)]^n$ with $f = \mathrm{id}$ — but you already know how to differentiate it without the Chain Rule at all, by unfolding it as $n-1$ repeated multiplications and applying the Product Rule once per step. Both routes reach the same answer for $x^4 = x \cdot x \cdot x \cdot x$:

$$\begin{aligned}\mathcal{D}_x x \cdot x &= x \cdot 1 + x \cdot 1 = 2x \\ \mathcal{D}_x(x \cdot x) \cdot x &= x^2 \cdot 1 + x \cdot 2x = 3x^2 \\ \mathcal{D}_x(x^3) \cdot x &= x^3 \cdot 1 + x \cdot 3x^2 = 4x^3\end{aligned}$$

three separate Product Rule applications, each one needing the result of the last — versus one step of the Extended Power Rule above: $\mathcal{D}_x x^4 = 4x^3$. Same answer, but a real difference in how much work it took to get there. This pattern resurfaces in Chapter 5 for a reason that has nothing to do with calculus: representing $f(x)^n$ as a composition with the power function, rather than as n repeated multiplications, turns out to be dramatically more efficient once you build an AST for it — there, the difference is not just "three steps instead of one," but a difference in the *size of the expression itself.*

2.2.3 Chain Rule in Other Notations

The Chain Rule has several other representations in the literature other than the notation we've presented in Theorem 2.2.1. In the Newtonian notation, the derivative of a function f is denoted simply as f'. Thus, the chain rule is stated as

$$\begin{aligned}[h(g(x))]' &= h'(g(x)) \cdot g'(x) \\ [h(g)]' &= h'(g) \cdot g' \,.\end{aligned}$$

The Newtonian notation indicates that we compute, f', the derivative of f, apply it to $g(x)$, then multiply that simply by, $g'(x)$ the derivative of g. In Newton's notation, h' without an argument is the point-free form — it refers to the derivative as a function — while $h'(g(x))$ is the pointwise form: the derivative function evaluated at the point $g(x)$.

Example 2.2.9 (*Using Newtonian Chain Rule*)

We want to compute the derivative of $(x^3)^2$. Let

$$g(x) = x^3$$
$$h(x) = x^2$$

First, we compute $h'(x) = 2x$

Second, we compute $h'(g(x))$. Since $h'(x) = 2x$, then $h'(g(x)) = 2x^3$.

Third, we compute $g'(x) = 3x^2$.

Finally, we multiply $h'(g(x))$ by $g'(x)$ to get $\left(2x^3\right)\left(3x^2\right) = 6x^5$

The Leibniz notation for the derivative of f is $\frac{d}{dx}f$. The Leibniz formulation of the Chain Rule is

$$\frac{dy}{dx} = \frac{dy}{du} \cdot \frac{du}{dx} \tag{2.3}$$

Example 2.2.10 uses the Leibniz formulation to again compute the derivative of $(x^3)^2$.

Example 2.2.10 (*Using the Leibniz Chain Rule*)

We want to compute the derivative of $(x^3)^2$. Let

$$u = g(x) = x^3$$
$$y = h(t) = t^2$$

Now, compute the derivatives.

$$\frac{dt}{dx} = 3x^2$$
$$\frac{dy}{dt} = 2t$$

Now, multiply them

$$\begin{aligned}\frac{dy}{dx} &= \frac{dy}{dt} \cdot \frac{dt}{dx} \\ &= (2t)\left(3x^2\right)\end{aligned}$$

Finally, substitute back $t = x^3$

$$\begin{aligned}\frac{dy}{dx} &= (2t)\left(3x^2\right) \\ &= (2)\left(x^3\right)\left(3x^2\right) \\ &= 6x^5\end{aligned}$$

The Leibniz Chain rule is easy to remember because it appears to be the product of two fractions where the dt in the numerator cancels with the dt in the numerator. However, this handy mnomonic device is not at all algebraic cancellation; it is far deeper.

Leibniz notation is inherently pointwise: it always names the variable (x, u, y) and always refers to values, not functions. It has no point-free form. This is why the Leibniz chain rule requires introducing auxiliary variables u and y, and why the computation ends with a substitution step to eliminate u.

One feature of the Leibniz formulation is that it makes explicit the change of variables which we saw in Example 2.2.2.

Each of the three notations, $\mathcal{D}_x f$, f', and $\frac{dy}{dx}$ sheds a slightly different light on the Chain Rule—each has its advantages and disadvantages. The principle notation we have chosen, $\mathcal{D}_x f$, called the operator notation, may seem at first to be the most obscure. However, when you attempt to implement the homework problems in Section 5.6, you will see that the notation corresponds very closely to the Python code which you will write. In that chapter, you will see that $\mathcal{D}_x$ is actu-

ally an operator which transforms one AST (abstract syntax tree) to another.

2.2.4 Proof of the Quotient Rule

Chapter 1 stated the Quotient Rule (Theorem 1.2.14) but deferred its proof, since proving it needs the Chain Rule together with the Power Rule — both now in hand.

Proof of Theorem 1.2.14.

$$\begin{aligned}
\mathcal{D}\,\frac{h}{g} &= \mathcal{D}\left((h)\left(\frac{1}{g}\right)\right) \\
&= \underbrace{\mathcal{D}\left(hg^{-1}\right)}_{\text{use product rule}} \\
&= h \qquad \underbrace{\mathcal{D}\,g^{-1}}_{\text{use Chain and Power Rule}} \qquad + g^{-1}\mathcal{D}\,h \\
&= (h)(-1)\left(g^{-2}\right)\mathcal{D}\,g + g^{-1}\mathcal{D}\,h \\
&= \frac{\mathcal{D}\,h}{g}\underbrace{\left(\frac{g}{g}\right)}_{\times 1} - \left(\frac{h\,\mathcal{D}\,g}{g^2}\right) \\
&= \frac{g\mathcal{D}\,h - h\mathcal{D}\,g}{g^2}
\end{aligned}$$

With the Chain Rule and Quotient Rule both available, every differentiation rule from Chapter 1 and this chapter can finally be collected in one place.

Function		**Derivative**	
point-free f	*pointwise* $f(x)$	*point-free* $\mathcal{D}\, f$	*pointwise* $\mathcal{D}_x f(x)$
c	c	0	0
id	x	1	1
	x^n		nx^{n-1}
	$\frac{1}{n+1}x^{n+1}$		x^n
$c\, g$	$cg(x)$	$c\mathcal{D}\, g$	$c\mathcal{D}_x g(x)$
$h + g$	$h(x) + g(x)$	$\mathcal{D}\, h + \mathcal{D}\, g$	$\mathcal{D}_x h(x) + \mathcal{D}_x g(x)$
$h - g$	$h(x) - g(x)$	$\mathcal{D}\, h - \mathcal{D}\, g$	$\mathcal{D}_x h(x) - \mathcal{D}_x g(x)$
$h\, g$	$h(x)g(x)$	$h\mathcal{D}\, g + g\mathcal{D}\, h$	$h(x)\mathcal{D}_x g(x) + g(x)\mathcal{D}_x h(x)$
$h \circ g$	$h(g(x))$	$((\mathcal{D}\, h) \circ g)(\mathcal{D}\, g)$	$\mathcal{D}\, h(g(x))\ \mathcal{D}_x g(x)$
$\frac{h}{g}$	$\frac{h(x)}{g(x)}$	$\frac{g\mathcal{D}\, h - h\mathcal{D}\, g}{g^2}$	$\frac{g(x)\mathcal{D}_x h(x) - h(x)\mathcal{D}_x g(x)}{[g(x)]^2}$
$\sin$	$\sin x$	$\cos$	$\cos x$
$\cos$	$\cos x$	$-\sin$	$-\sin x$
$\tan$	$\tan x$	$\sec^2$	$\sec^2 x$
$\arcsin$	$\arcsin x$		$\frac{1}{\sqrt{1-x^2}}$
$\arccos$	$\arccos x$		$\frac{-1}{\sqrt{1-x^2}}$
$\arctan$	$\arctan x$		$\frac{1}{1+x^2}$
arccot	$\operatorname{arccot} x$		$\frac{-1}{1+x^2}$
$\exp$	e^x	$\exp$	e^x
	$\ln \lvert x \rvert$		$\frac{1}{x}$

Figure 2.1: Derivative Table

2.3 Implicit Differentiation

Sometimes a function $y = f(x)$ is not given by an explicit formula but is instead defined by an equation $F(x, y) = 0$ that the pair (x, y) must satisfy. We may not be able to solve for y in closed form, yet we can still find dy/dx by differentiating both sides of the equation with respect to x, treating y as a differentiable function of x, and then solving algebraically for dy/dx. This technique is *implicit differentiation*.

Example 2.3.1 (*Slope of the unit circle*)

The unit circle $x^2 + y^2 = 1$ defines y implicitly as a function of x near any point where $y \neq 0$. Differentiating both sides with respect to x:

$$2x + 2y\,\frac{dy}{dx} = 0 \implies \frac{dy}{dx} = -\frac{x}{y}\,.$$

No need to write $y = \pm\sqrt{1 - x^2}$ first.

The same strategy applies when the equation is far more complex, as the following example shows.

Example 2.3.2 (*Monotonicity via implicit differentiation*)

Let $\psi : \mathbb{R}^+ \to \mathbb{R}$ be the function defined implicitly by

$$e^{e^{\psi(x)+1}} - e^{e^{\psi(x)}} \;=\; e^{x-\psi(x)-1}\,. \tag{2.4}$$

We assume such a function ψ exists and is differentiable; the sign analysis below ($A + B > 0$) will show that ψ is unique if it exists, but proving existence requires the Implicit Function Theorem, which is beyond the scope of this course. We show $\psi'(x) > 0$ (i.e. ψ is strictly increasing) without solving for ψ explicitly.

First, we apply the Chain rule twice to compute $\mathcal{D}_x e^{e^{f(x)}}$, whose

value we will need later.

$$\begin{aligned}\mathcal{D}_x e^{e^{f(x)}} &= e^{e^{f(x)}} \cdot \mathcal{D}_x e^{f(x)} \\ &= e^{e^{f(x)}} \cdot e^{f(x)} \cdot \mathcal{D}_x f(x) \\ &= e^{f(x)+e^{f(x)}} \cdot \mathcal{D}_x f(x)\,.\end{aligned}$$

Implicit Differentiation:

$$\begin{aligned}\overbrace{\mathcal{D}_x\left(e^{e^{\psi(x)+1}} - e^{e^{\psi(x)}}\right)}^{\text{LHS}} &= \mathcal{D}_x e^{e^{\psi(x)+1}} - \mathcal{D}_x e^{e^{\psi(x)}} \\ &= e^{(\psi+1)+e^{\psi+1}} \cdot \psi' - e^{\psi+e^{\psi}} \cdot \psi' \\ &= \underbrace{\left(e^{\psi+1+e^{\psi+1}} - e^{\psi+e^{\psi}}\right)}_{=B} \psi'\end{aligned}$$

$$\overbrace{\mathcal{D}_x e^{x-\psi-1}}^{\text{RHS}} = \underbrace{e^{x-\psi-1}}_{=A} \cdot (1-\psi')\,.$$

LHS = RHS $\implies$ $B\,\psi' = A(1-\psi') \implies \psi' = \frac{A}{A+B}$
Sign analysis: Show A and B are both positive.
First, $A > 0$ as $e^{x-\psi-1}$ is always positive. What about B?

$$\begin{aligned} 1 + e^{\psi+1} &> e^{\psi+1} \\ &> e^{\psi} \\ \psi + 1 + e^{\psi+1} &> \psi + e^{\psi} \\ e^{\psi+1+e^{\psi+1}} &> e^{\psi+e^{\psi}} \\ \underbrace{e^{\psi+1+e^{\psi+1}} - e^{\psi+e^{\psi}}}_{=B} &> 0 \end{aligned}$$

Hence, $\psi' = \frac{A}{A+B}$ is a ratio of positive quantities, so $\psi'(x) > 0$ for all x and ψ is strictly increasing.

2.4 Proof of Derivative of arctan

In Figure 2.1 we listed the entry $\mathcal{D}_x \arctan(x) = \frac{1}{1+x^2}$ without any justification. The formula may seem puzzling: why does differentiating an inverse trigonometric function produce a rational expression with no trigonometry in sight? Now that we have implicit differentiation at our disposal, the answer is not difficult to find.

Theorem 2.4.1 (*Derivative of arctangent*)

$$\mathcal{D}_x \arctan(x) = \frac{1}{1+x^2}\,.$$

Proof. Set $y = \arctan(x)$, so that $\tan(y) = x$ with $y \in \left(-\frac{\pi}{2}, \frac{\pi}{2}\right)$. Differentiating both sides with respect to x (implicit differenti-

ation, treating y as a function of x):

$$\begin{aligned} y &= \arctan(x) \\ \tan(y) &= x \\ \mathcal{D}_x \tan(y) &= \mathcal{D}_x x \\ \sec^2(y) \cdot \mathcal{D}_x y &= 1 \\ \mathcal{D}_x y &= \frac{1}{\sec^2(y)} \\ \mathcal{D}_x y &= \cos^2(y)\,. \end{aligned} \tag{2.5}$$

It remains to express $\cos^2(y)$ in terms of x.

$$\begin{aligned} x &= \tan(y) \\ &= \frac{\sin(y)}{\cos(y)} \\ x\cos(y) &= \sin(y) \\ x^2\cos^2(y) &= \sin^2(y) \\ x^2\cos^2(y) + \cos^2(y) &= \underbrace{\sin^2(y) + \cos^2(y)}_{=1} \\ \cos^2(y)\big(x^2+1\big) &= 1 \\ \cos^2(y) &= \frac{1}{x^2+1} \\ \mathcal{D}_x y &= \frac{1}{x^2+1} \qquad \text{by Equation (2.5)} \end{aligned}$$

? Open Questions

- In Example 2.3.2 we *assumed* that the function ψ defined by Equation (2.4) exists and is differentiable, then reasoned about its sign. What actually guarantees that an equation $F(x, y) = 0$ defines y as a differentiable function of x near a given point in the first place? This question is answered by the *Implicit Function Theorem*, which is beyond the scope of this course but well worth looking up.
- The unit circle $x^2 + y^2 = 1$ does not define y as a single function of x globally — there are two branches, $y = \sqrt{1 - x^2}$ and $y = -\sqrt{1 - x^2}$. When we compute $\dfrac{dy}{dx} = -\dfrac{x}{y}$ by implicit differentiation, which branch does that formula describe, and how would the computation have warned us if we had tried to differentiate implicitly at a point like $(1, 0)$ where no branch is differentiable?

Chapter 3

The Shape of a Function

Chapter Objectives

- Use the sign of f' to determine regions where a function is increasing, decreasing, or constant, and locate local extrema from sign changes of f'.
- Use the sign of f'' to determine concavity and identify inflection points.
- Apply the First and Second Derivative Tests to classify critical points as local minima, local maxima, or neither.
- Find the global extrema of a continuous function on a closed interval by comparing critical-point values with endpoint values.

Throughout this chapter the guiding question is: *what is the shape of this function?* Given a function's first and second derivatives, we can determine where it increases and decreases, where it curves upward or downward, and where its extrema lie — turning two derivative computations into a full sketch of the graph, without plotting points one at a time. Two worked examples, $\sin x$ and e^{2x}, show these tools applied to functions whose shape is already familiar, so you can check the machinery against what you already know.

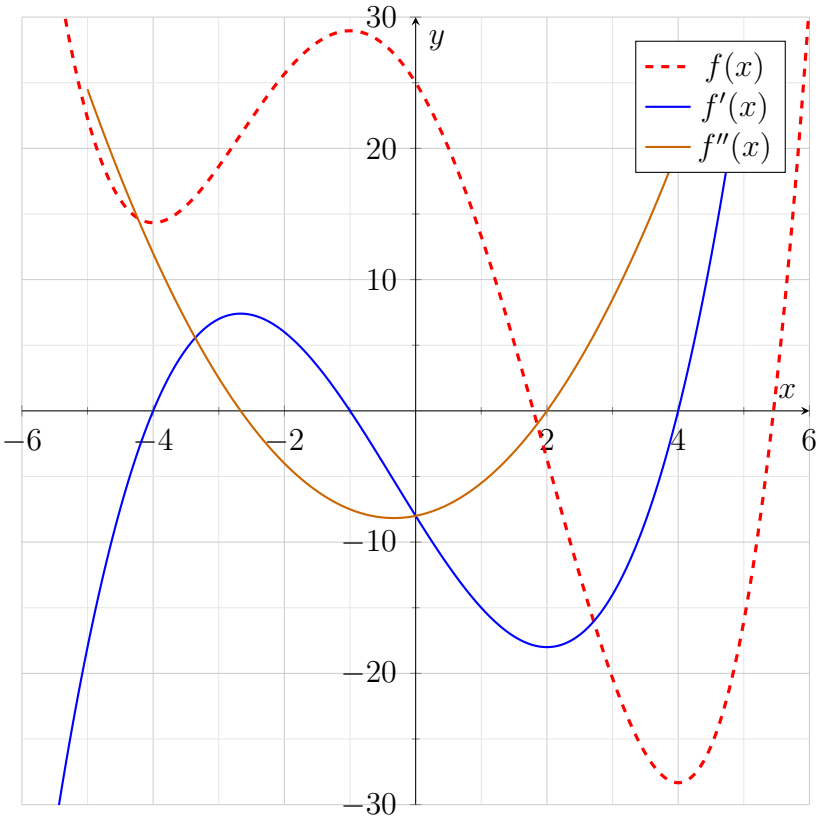


Figure 3.1: Graph of $f(x)$ in dashed red overlaid with $f'(x)$ in blue and $f''(x)$ in orange. $f(x)$ is drawn dashed to emphasize that this is the curve we are trying to derive.

3.1 Intuition of the Shape of a Graph

In this section, we look at a classic problem which all first semester calculus students are presented with: sketch the graph of a designated polynomial. The same techniques can be used to sketch the graph of a wide range of functions; nevertheless, we usually start with a polynomial as the first exercise. The strategy is to obtain information gleaned from the first and second derivative, and use that information as clues to construct the graph of the polynomial in question.

The function, f, Equation (3.1), we are going to consider is a quartic (4th degree) polynomial, its first derivative, f', Equation (3.2), is a cubic (3rd degree) polynomial, and its second derivative, f'', Equation (3.3) is quadratic.

$$f(x) = \frac{1}{8}x^4 + \frac{1}{6}x^3 - 4x^2 - 8x + 25 \tag{3.1}$$

$$\begin{aligned} f'(x) &= \mathcal{D}_x\left(\frac{1}{8}x^4 + \frac{1}{6}x^3 - 4x^2 - 8x + 25\right) \\ &= \frac{1}{2}x^3 + \frac{1}{2}x^2 - 8x - 8 \qquad (3.2) \\ &= \frac{1}{2}(x+4)(x+1)(x-4) \end{aligned}$$

$$\begin{aligned} f''(x) &= \mathcal{D}_x\left(\frac{1}{2}x^3 + \frac{1}{2}x^2 - 8x - 8\right) \\ &= \frac{3}{2}x^2 + x - 8 \qquad (3.3) \\ &= \frac{1}{2}(3x+8)(x-2) \end{aligned}$$

Figure 3.1 shows the graphs of $f(x)$, $f'(x)$, and $f''(x)$. We call the reader's attention to the three roots of the cubic, $f'(x)$, namely

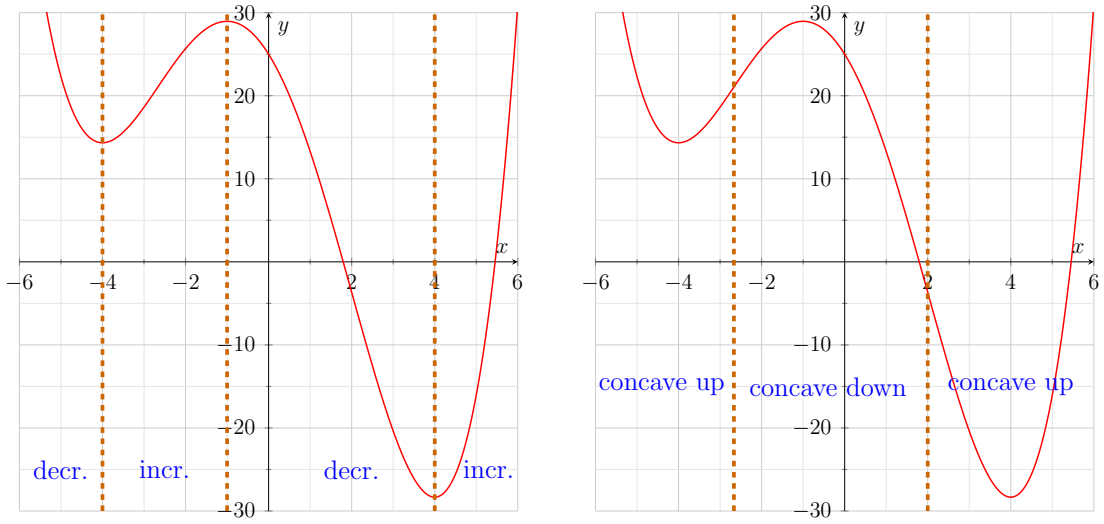


Figure 3.2: Graph of $f(x)$, constructed from $f'(x)$ and $f''(x)$. [Left] 4 regions of monotonicity. [Right] $f(x)$ 3 regions of concavity.

$x = -4$, $x = -1$, and $x = 4$, and the fact that those points exactly correspond to where the graph of $f(x)$ has local minima and maxima.

The vertical dashed lines are drawn on the plot in Figure 3.2 [left], each line corresponds to a root of $f'(x)$, *i.e.* $x = -4$, $x = -1$, and $x = 4$. The lines cut the plane into 4 vertical slices. From left to right, f is decreasing in the first slice up to the first dashed line, where f has a local minimum, which also happens to be its global minimum. Next, there is a slice where f is increasing up to a local maximum. Next, f, decreases through the 3rd slice up to a local minimum. Finally in the right-most slice f is increasing. The procedure for identifying these regions is to compute the derivative, and solve $f'(x) = 0$ for x, either explicitly or numerically.

There is even more information in the graph in Figure 3.1 which can help us to construct the graph of f. We find that the second derivative f'' has zeros exactly where f' has its extrema. At $x = -\frac{8}{3}$, $f''(x) = 0$ and $f'(x)$ is a local maximum. At $x = 2$, $f''(x) = 0$ and $f'(x)$ has a

local minimum. Thus, when we are trying to find the zeros of f', for example with a binary search, a good place to look is between the zeros of f''. Recall, that a binary search algorithm for finding a root of a polynomial typically needs an upper and lower bound to initialize the search.

From the orange curve, $f''(x)$, in Figure 3.1, we can also infer the *concavity* of f. Where $f''(x) > 0$, the graph of f opens upward (concave up, like a cup); where $f''(x) < 0$, it opens downward (concave down, like a cap). The two zeros of f'' at $x = -\frac{8}{3}$ and $x = 2$—are the *inflection points* of f, the places where the direction of curvature (*i.e.*, concavity) reverses. The two vertical dashed lines in Figure 3.2 [right] show the three regions: concave up, concave down, and concave up.

3.2 Regions of Monotone Behavior

A continuous, non-constant function on a closed interval $[a, b]$ need not be monotone everywhere; if it is not, it will have regions where it increases and regions where it decreases. More precisely, we can partition $[a, b]$ into subintervals on each of which f is strictly increasing, strictly decreasing, or constant. The derivative tells us which regime applies:

- $f'(x) > 0$ on an interval $\Rightarrow$ f is strictly increasing there;
- $f'(x) < 0$ on an interval $\Rightarrow$ f is strictly decreasing there;
- $f'(x) = 0$ on an interval $\Rightarrow$ f is constant there.

If f' is itself continuous, then wherever it transitions from positive to negative (or vice versa), the Intermediate Value Theorem forces f' to pass through zero at some point in between — informally, a continuous function cannot jump from a negative value to a positive

one without crossing zero somewhere in between; the theorem is stated precisely and used again in Chapter 8 (Theorem 8.2.2). That zero is a candidate for a local extremum.

The converse is not guaranteed: $f'(c) = 0$ does not by itself imply a sign change. The derivative may be zero at the boundary between two regions where it is positive on both sides. For example, $f(x) = x^3$ satisfies $f'(0) = 0$ yet f is strictly increasing on both sides of $x = 0$, so $x = 0$ is neither a local minimum nor a local maximum.

Once the sign of f' is known across the whole interval, the shape of the graph follows:

- wherever f switches from decreasing to increasing, it reaches a *local minimum*;
- wherever f switches from increasing to decreasing, it reaches a *local maximum*.

The extrema of f on $[a, b]$ are found either at these interior critical points or at the endpoints a and b. Figure 3.2 [left] illustrates all three cases for Equation (3.1): f' is negative, positive, negative, and positive across the four slices from left to right.

3.3 Critical Points

Definition 3.3.1 (*Critical Point*)

> A point c in the interior of the domain of f is a *critical point* of f if $f'(c) = 0$ or $f'(c)$ does not exist.

Every interior local extremum occurs at a critical point: if f has a local minimum or maximum at c and $f'(c)$ exists, then $f'(c) = 0$. This is because at an interior extremum the derivative cannot be strictly positive (the function would still be increasing) or strictly negative

(it would still be decreasing). For the function of Section 3.1, the critical points are $x \in \{-4, -1, 4\}$, the roots of Equation (3.2) visible in Figure 3.1.

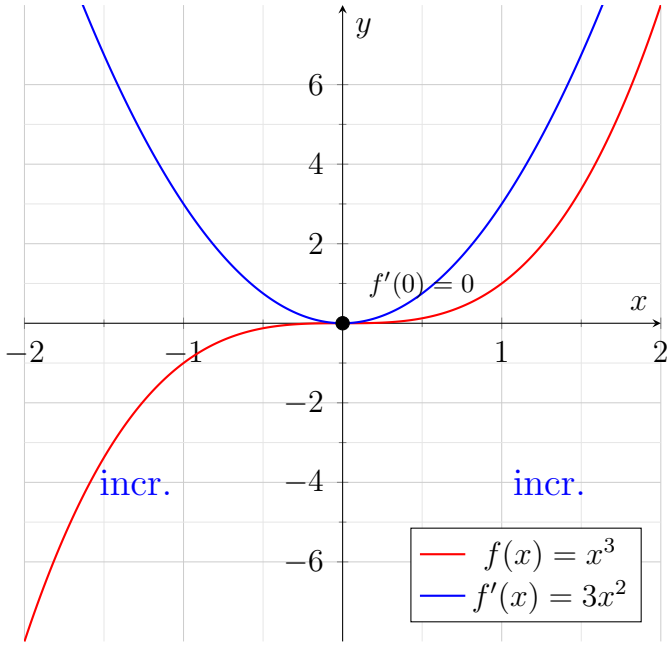


Figure 3.3: $f(x) = x^3$ (red) and $f'(x) = 3x^2$ (blue). Although $f'(0) = 0$, the derivative does not change sign at $x = 0$, so f is increasing on both sides and $x = 0$ is neither a local minimum nor a local maximum.

As the example $f(x) = x^3$ at $x = 0$ illustrates, the converse fails: a critical point is only a *candidate* for a local extremum and must be tested further. Figure 3.3 shows why: $f'(x) = 3x^2$ touches zero at $x = 0$ but never changes sign, so f is strictly increasing on both sides of the critical point.

If f is defined on a *closed interval* $[a, b]$ rather than on all of $\mathbb{R}$, the definition above still restricts critical points to the *interior* (a, b) — an endpoint is never a critical point, no matter how f behaves there. Yet a global extremum can still occur at an endpoint. For example, $f(x) = x + 1$ on $[0, 1]$ has $f'(x) = 1$ everywhere, so it has *no critical*

points at all; nevertheless f attains its global minimum $f(0) = 1$ at the left endpoint and its global maximum $f(1) = 2$ at the right endpoint. The lesson: when hunting for the global maximum or minimum of f on $[a, b]$, checking the critical points is not enough — you must also check the value of f at both endpoints a and b, and compare all of these values together.

3.4 Second Derivative and Concavity

Definition 3.4.1 (*Concavity*)

Let f be twice differentiable on an open interval I.

- f is *concave up* on I if $f''(x) > 0$ for all $x \in I$.
- f is *concave down* on I if $f''(x) < 0$ for all $x \in I$.

Equivalently, f is concave up when f' is increasing (the slope is growing), and concave down when f' is decreasing (the slope is shrinking).

Definition 3.4.2 (*Inflection Point*)

A point c is an *inflection point* of f if f'' changes sign at c, i.e. the concavity of f reverses at c.

A necessary condition for an inflection point is $f''(c) = 0$, but this is not sufficient. For example, $f(x) = x^4$ satisfies $f''(0) = 0$ yet f'' does not change sign at $x = 0$, so $x = 0$ is not an inflection point. For the function of Section 3.1, $f''(x) = 0$ at $x = -\frac{8}{3}$ and $x = 2$ (from Equation (3.3)), giving the three concavity regions visible in Figure 3.2 [right].

3.5 Classifying Extrema

The following two tests give precise criteria for classifying a critical point.

Theorem 3.5.1 (*First Derivative Test*)

Let c be a critical point of f and suppose f' is continuous on an open interval containing c.

- If f' changes from negative to positive at c: *local minimum.*
- If f' changes from positive to negative at c: *local maximum.*
- If f' does not change sign at c: neither.

Theorem 3.5.2 (*Second Derivative Test*)

Let $f'(c) = 0$ and suppose $f''(c)$ exists.

- If $f''(c) > 0$: *local minimum.*
- If $f''(c) < 0$: *local maximum.*
- If $f''(c) = 0$: inconclusive.

The first derivative test is more generally applicable: it works even when $f''(c) = 0$ or f'' does not exist. The second derivative test is quicker when $f''(c) \neq 0$, requiring only a single evaluation rather than a sign analysis over an interval.

The three-step procedure below finds the global extrema *assuming they exist.* That assumption is not automatic: a function can fail to attain a maximum or minimum at all (consider $f(x) = x$ on the open interval $(0, 1)$, which has neither). What guarantees existence here is the *Extreme Value Theorem*: a function continuous on a *closed, bounded* interval $[a, b]$ always attains both a global maximum and a

global minimum on that interval. This is precisely why the procedure requires a closed interval and a continuous function — drop either hypothesis and the search below can come up empty.

To find the *global* extrema of a continuous function on a closed interval $[a, b]$:

1. Find all critical points in the open interval (a, b).
2. Evaluate f at each critical point and at the endpoints a and b.
3. The largest value is the *global maximum*; the smallest is the *global minimum*.

Applying both tests to the function of Section 3.1, using $f'(x) = \frac{1}{2}(x+4)(x+1)(x-4)$ from Equation (3.2) and $f''(x) = \frac{1}{2}(3x+8)(x-2)$ from Equation (3.3):

c	sign change of f'	$f''(c)$	conclusion
-4	$- \to +$	$\frac{1}{2}(-4)(-6) = 12 > 0$	local minimum
-1	$+ \to -$	$\frac{1}{2}(5)(-3) = -\frac{15}{2} < 0$	local maximum
4	$- \to +$	$\frac{1}{2}(20)(2) = 20 > 0$	local minimum

Both tests agree, as expected. Since the leading coefficient of f is positive, $f(x) \to +\infty$ as $x \to \pm\infty$, so the global minimum is whichever local minimum has the smaller value: $f(-4) = \frac{43}{3}$ versus $f(4) = -\frac{85}{3}$, so the global minimum is at $x = 4$.

3.6 Worked Example: The Shape of $\sin x$

The polynomial of Section 3.1 required solving cubic and quadratic equations to locate its critical points and inflection points. The sine

function makes the same shape arguments *visual* instead.

$$f(x) = \sin x$$
$$f'(x) = \cos x$$
$$f''(x) = -\sin x$$

These are three curves every student already recognizes, so we can check the monotonicity and concavity criteria of this chapter by inspection, without solving anything.

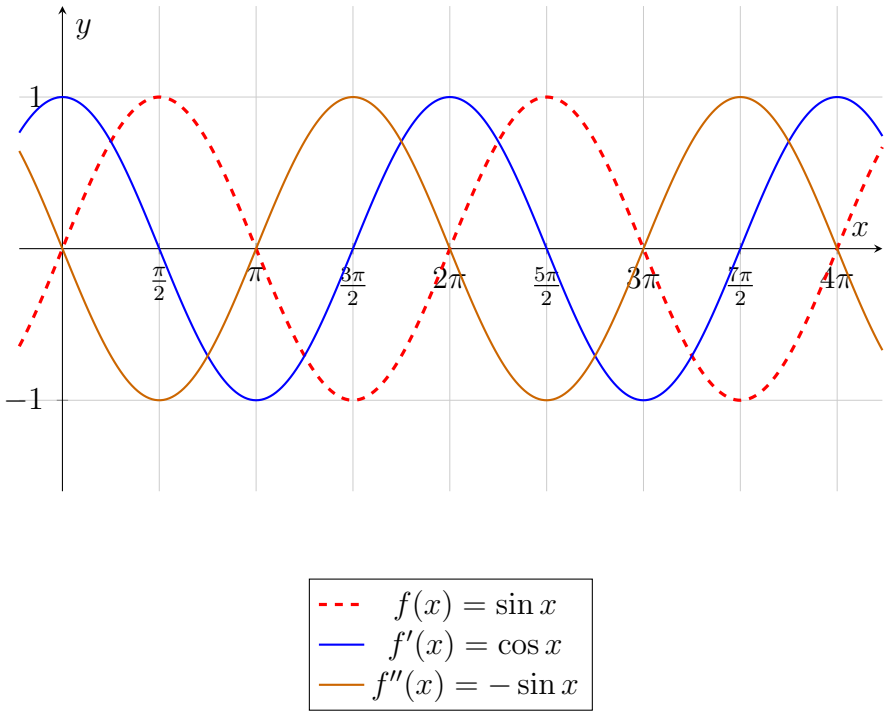


Figure 3.4: $f(x) = \sin x$ (red, dashed), $f'(x) = \cos x$ (blue), and $f''(x) = -\sin x$ (orange) over two periods.

Monotonicity. By the criterion of Section 3.1 ($f' > 0 \Rightarrow$ increasing, $f' < 0 \Rightarrow$ decreasing), the shape of $\sin x$ is read directly off the sign of $\cos x$ in Figure 3.4: everywhere the blue curve $\cos x$ lies above

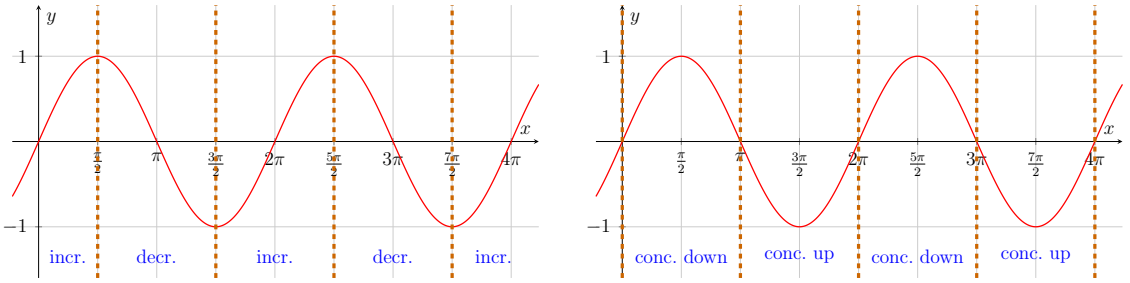


Figure 3.5: Shape of $f(x) = \sin x$. [Left] Regions of monotonicity, cut by the zeros of $f' = \cos x$ at $x = \frac{\pi}{2} + k\pi$. [Right] Regions of concavity, cut by the zeros of $f'' = -\sin x$ at $x = k\pi$.

the axis, the red curve $\sin x$ is climbing; everywhere $\cos x$ dips below the axis, $\sin x$ is falling. The two curves even cross zero together where one would expect: $\cos x = 0$ at $x = \frac{\pi}{2} + k\pi$, and by Definition 3.3.1 these are exactly the critical points of $\sin x$ — precisely where the red curve turns around and has a local max or local min.

Which of these critical points are maxima and which are minima follows from either test in Section 3.1. At $x = \frac{\pi}{2}$, $\cos x$ changes from positive to negative (First Derivative Test, Theorem 3.5.1), so $\sin x$ has a local maximum there — matching $\sin \frac{\pi}{2} = 1$, the peak of the curve. Equivalently, by the Second Derivative Test (Theorem 3.5.2), $f''\left(\frac{\pi}{2}\right) = -\sin \frac{\pi}{2} = -1 < 0$. At $x = \frac{3\pi}{2}$, $\cos x$ changes from negative to positive, giving a local minimum, matching $\sin \frac{3\pi}{2} = -1$; correspondingly $f''\left(\frac{3\pi}{2}\right) = -\sin \frac{3\pi}{2} = 1 > 0$. Figure 3.5 [left] shows all five regions of monotonicity across the two periods.

Concavity. By Definition 3.4.1, concavity is read from the sign of $f'' = -\sin x$, shown in orange in Figure 3.4. Since $f'' = -\sin x$, this sign is the *opposite* of the sign of $\sin x$ itself: where $\sin x > 0$ (the red curve is above the axis), $f'' < 0$ and $\sin x$ is concave down; where $\sin x < 0$ (below the axis), $f'' > 0$ and $\sin x$ is concave up. The inflection points (Definition 3.4.2) are the zeros of f'', i.e. the zeros of $\sin x$ itself, at $x = k\pi$ — Figure 3.5 [right] shows the resulting regions.

Because $f'' = -f$ for $f = \sin x$, the curve is always bending back toward the x-axis: wherever it rises above the axis it is concave down (curving back toward zero), and wherever it dips below the axis it is concave up (curving back toward zero). This is a feature special to $\sin x$ (and $\cos x$) among the functions in this chapter — most functions have no such fixed relationship between f and f''.

3.7 Worked Example: The Shape of e^{2x} — No Features to Find

Not every function has critical points or inflection points to locate, and $f(x) = e^{2x}$ is the cleanest illustration of why.

$$f(x) = e^{2x}$$
$$f'(x) = 2e^{2x}$$
$$f''(x) = 4e^{2x}$$

f' and f'' are both positive multiples of f itself, and $e^{2x} > 0$ for *every* real x — it is never zero. Hence $f'(x) > 0$ and $f''(x) > 0$ for every x as well: f is always increasing and always concave up. This one fact, verified directly from the formulas, rules out every feature this chapter has taught us to look for.

No critical points. By Definition 3.3.1, a critical point requires $f'(c) = 0$ or $f'(c)$ undefined. Here $f'(x) = 2e^{2x}$ exists for every x and is never 0, so f has no critical points at all. Equivalently, by the monotonicity criterion of Section 3.1, $f'(x) > 0$ everywhere means f is strictly increasing on all of $\mathbb{R}$ — there is no interval where it turns around, so there is nothing for a local max or min to be.

No inflection points. By Definition 3.4.2, an inflection point requires f'' to *change sign*. Here $f''(x) = 4e^{2x} > 0$ for every x, so by

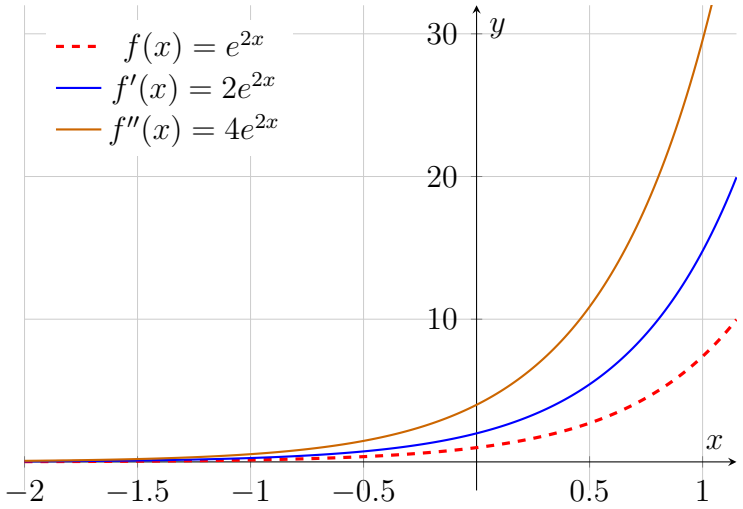


Figure 3.6: $f(x) = e^{2x}$ (red, dashed), $f'(x) = 2e^{2x}$ (blue), and $f''(x) = 4e^{2x}$ (orange). All three curves stay strictly positive; none of them ever touches the x-axis.

Definition 3.4.1, f is concave up on all of $\mathbb{R}$ and f'' never changes sign — there is no inflection point to find.

Contrast this with $f = \sin x$: there, $f'' = -f$ forces f'' to flip sign every time f does, producing infinitely many inflection points. Here, $f'' = 4f$ is a *positive* multiple of f, so f'' has the same sign as f always — and since $f = e^{2x}$ is itself always positive, f'' never changes sign. The lesson is not that e^{2x} is a boring function, but that "has no critical points" and "has no inflection points" are conclusions to be *verified from the derivatives*, exactly like "has a critical point at $x = 4$" was in Section 3.1.

? Open Questions

- The procedure for finding global extrema in this chapter requires a *closed, bounded* interval $[a, b]$. What happens on an open interval like $(0, 1)$, or an unbounded one like $[0, \infty)$? Can a continuous function still fail to have a global maximum or minimum there? Try to construct examples of each before reading on.

3.8 Enrichment: Two Tangent Lines and a Bézier Curve

This section is optional reading, not covered in lecture. It asks a natural follow-up to the tangent-line approximation from Chapter 1: if you know f and f' at *two* nearby points instead of one, how much better can you do — and what does it take to build a single combined estimate that agrees with f's direction at both points, not just its value?

3.8.1 Estimating the Value of a Function

Suppose all we know about a function f is its value and its derivative at a single point a: $f(a)$ and $f'(a)$. That's enough to approximate f near a — the natural thing to do is follow the tangent line, since it's the best *linear* description of f we have:

$$L_a(x) = f(a) + f'(a)(x - a). \tag{3.4}$$

Figure 3.7 shows why this works, and why it only works *locally*. Right at a, $L_a(a) = f(a)$ exactly. Close to a, $L_a(x)$ stays close to $f(x)$

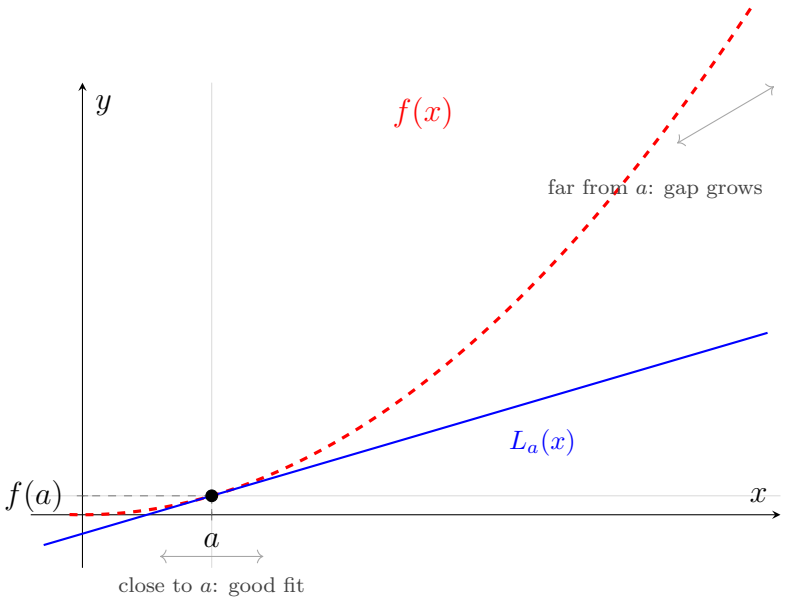


Figure 3.7: A single tangent-line estimate L_a (blue) of f (red, dashed), built from $f(a)$ and $f'(a)$ alone.

— the curve hasn't had room to bend away from its own tangent line yet. But as x moves farther from a, the approximation gets worse, and there's a reason L_a can't help that: it was built entirely from information *at* a, so it has no way to know that f is curving away underneath it. Whatever f does far from a is invisible to a straight line anchored only at a.

That limitation raises a natural question: what if we also knew f and f' at a second point nearby? Could a second estimate — built the same way, but anchored somewhere else — cover for the first one's blind spot?

3.8.2 Two Estimates for One Value

Suppose we also know $f(b)$ and $f'(b)$ at a second point b near a. That gives a *second* linear approximation, built the same way as Equa-

tion (3.4) but extrapolated backward from b:

$$L_b(x) = f(b) + f'(b)(x - b). \tag{3.5}$$

For x strictly between a and b, both $L_a(x)$ and $L_b(x)$ are estimates of $f(x)$ — computed from different data, and in general disagreeing (Figure 3.8). That leaves an obvious question: how should the two estimates be combined into one?

The first idea anyone reaches for is to just average them. It turns out that's the *worst* choice available, and specifically at the two points where we actually know the answer exactly.

3.8.3 Averaging Goes Wrong at the Endpoints

Try the simplest possible combination first: a plain 50/50 average of the two estimates, as a function of x,

$$\begin{aligned} L_{estimate}(x) &= \frac{L_a(x) + L_b(x)}{2} \\ &= \frac{1}{2}[f(a) + f'(a)(x - a) + f(b) + f'(b)(x - b)]. \end{aligned} \tag{3.6}$$

To judge whether this is any good, define the error at a point as

$$\text{error}(x) = \big|f(x) - L_{estimate}(x)\big|. \tag{3.7}$$

We can't minimize error(x) for every x — that would require knowing f, which is exactly what we don't have. But there *are* two values of x where we know f exactly: $x = a$ and $x = b$ themselves. So a reasonable minimum bar for any estimate is error$(a) = 0$ and error$(b) = 0$. Does $L_{estimate}$ clear that bar?

Evaluate Equation (3.6) at $x = a$ specifically — the $f'(a)(x - a)$ term vanishes since $x - a = 0$ there, leaving

$$L_{estimate}(a) = \tfrac{1}{2}f(a) + \tfrac{1}{2}\big[f(b) + f'(b)(a - b)\big]. \tag{3.8}$$

So

$$\text{error}(a) = \big|f(a) - L_{estimate}(a)\big| = \tfrac{1}{2}\big|f(a) - f(b) - f'(b)(a-b)\big|,$$

which is zero only if $f(a) = f(b) + f'(b)(a-b)$ — only if L_b happens to pass exactly through $(a, f(a))$, and nothing forces that. (The same computation at $x = b$, swapping the roles of a and b, gives the symmetric failure there.) A constant-weight average corrupts the one piece of information we know *exactly* at each end: the function's own value at the point we're sitting on — Figure 3.8 shows exactly this failure at both callouts.

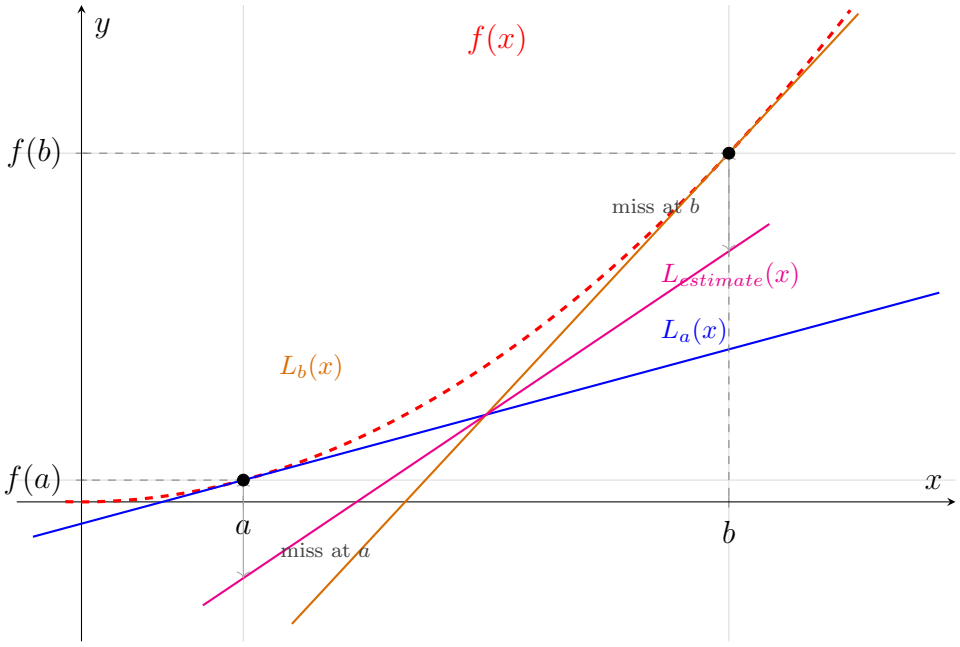


Figure 3.8: $L_{estimate}$ (magenta) against f (red, dashed), L_a (blue), and L_b (orange).

3.8.4 Matching the Slopes Too: A Bézier Curve

The naive average failed because a constant 50/50 blend can't land exactly on $f(a)$ and $f(b)$ (Figure 3.8) — fixing that means demanding an exact match at each endpoint, not just a compromise. But matching the *value* exactly at a and b still says nothing about the *slope* there: a curve can pass through $(a, f(a))$ and $(b, f(b))$ while heading in completely the wrong direction at either end, and still be a poor local approximation right where it's supposed to be most trustworthy.

Intuition says the fix is to make the slopes agree too — match not just $f(a)$ and $f(b)$, but $f'(a)$ and $f'(b)$ as well. That's exactly the idea behind the *Bézier curve*, the construction used throughout computer graphics — font outlines, vector-art paths, animation easing — anywhere a designer needs a curve that leaves one direction and settles smoothly into another.

A cubic Bézier curve is built from four *control points* B_0, B_1, B_2, B_3, blended by

$$B(t) = (1-t)^3 B_0 + 3(1-t)^2 t\, B_1 + 3(1-t)t^2\, B_2 + t^3 B_3, \qquad t \in [0,1]. \tag{3.9}$$

Anchor the two ends at the known points, $B_0 = (a, f(a))$ and $B_3 = (b, f(b))$, and place the other two *anywhere along the tangent lines* L_a and L_b:

$$B_1 = \big(a + s,\ L_a(a+s)\big), \qquad B_2 = \big(b - r,\ L_b(b-r)\big),$$

for any $s, r > 0$. Because B_1 sits on L_a, the curve leaves B_0 heading in exactly L_a's direction; because B_2 sits on L_b, it arrives at B_3 heading in exactly L_b's direction — true for *every* choice of s and r. That's the infinitely many curves: one for each pair of handle lengths, all agreeing with f in value and slope at both endpoints, yet visibly different curves in between. A common default is $s = r = h/3$ (with $h = b - a$), shown in Figure 3.9.

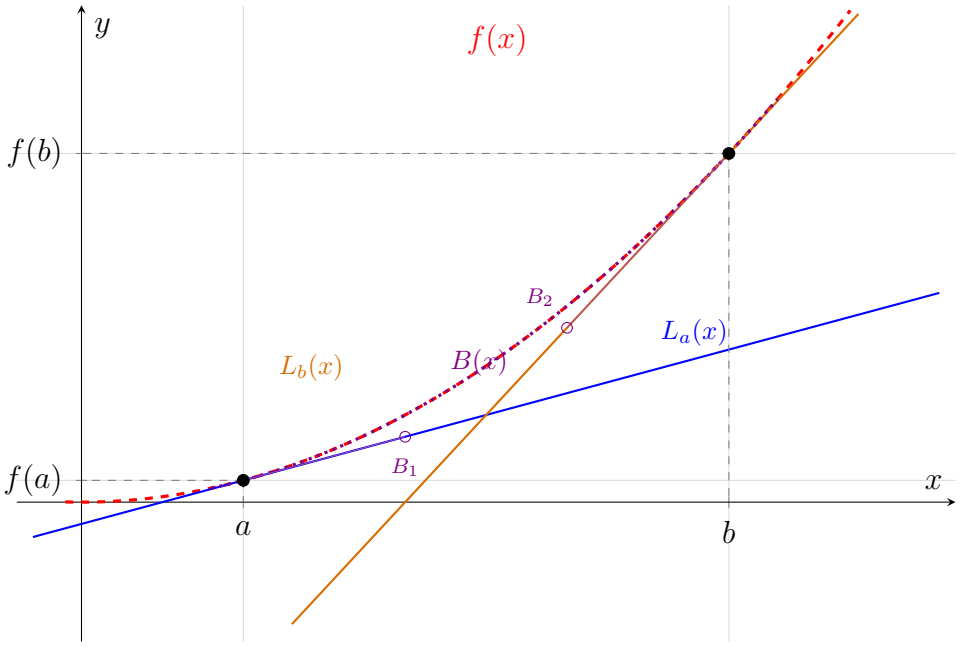


Figure 3.9: f (red, dashed), L_a (blue), L_b (orange), and the Bézier curve $B(x)$ (violet, dotted, Equation (3.9)) with handles B_1, B_2 (violet circles) sitting on L_a, L_b at distance $h/3$ from each anchor. $B(x)$ coincides with f exactly here, but the two remain distinguishable since they're drawn with different dash patterns; $B(x)$ runs alongside L_a leaving a and alongside L_b arriving at b, matching both value and slope at each endpoint.

In the running example, $f(x) = x^2/4$ is itself only quadratic, and for this particular handle length $h/3$ the resulting curve happens to trace f exactly, for its whole length, not only at a and b. That won't happen in general: a curve built only from $f(a)$, $f'(a)$, $f(b)$, $f'(b)$ has no way to see f's curvature in between, so for most choices of f and handle length it will still drift away from f somewhat — matching value and slope at the endpoints buys a much better local approximation than matching value alone, but not a perfect one.

? Open Questions

- Section 3.8.4's Bézier curve $B(x)$ used four numbers at two points, a and b. What if you're instead given data at three points a, m, b, and build one Bézier piece on $[a, m]$ and another on $[m, b]$? Each piece matches value and slope on its own, but what condition would you need at m to make the two pieces meet *smoothly* there too, not just continuously? (This is how a *spline* is built.)

Chapter 4

HW-1: Derivatives by Hand and Skip List Optimization

Chapter Objectives

- Compute derivatives of polynomial, trigonometric, exponential, and logarithmic functions by hand and verify them with your `derivative` implementation.
- Translate differentiation rules among four notations: point-free ($\mathcal{D}\, f$), pointwise ($\mathcal{D}_x f(x)$), Newton prime (f'), and Leibniz ($\frac{d}{dx}$); explain why Leibniz notation is inherently pointwise and has no point-free form.
- Apply derivatives to a CS problem: model the search time of a two-level linked list index as $T(k) = k + n/k$ and find the optimal index size $k = \sqrt{n}$ by differentiation.
- Generalize to a multi-level skip list, rewrite the cost function using natural logarithms, and show that the optimal promotion probability is $p = 1/e$.

In this problem set, we analyze a simplified two-level search structure inspired by skip lists, then generalize it to a multi-level skip list.

You will model search time as a function of one or more design parameters (k, then a promotion probability p) and use calculus to optimize performance.

Reminder. Throughout this sheet you will need the *change-of-base* identity for logarithms: 

$$\log_b(x) = \frac{\ln x}{\ln b},$$

where ln denotes the natural logarithm (base e).

Reminder. You will also need to rewrite exponentials in base e. 

Since $b = e^{\ln b}$, we have:

$$b^x = e^{x \ln b}.$$

From here, the chain rule applies directly.

4.1 Computing Derivatives

4.1.1 Computing Derivatives

Compute the derivative of each of the following, first using the Python code you have implemented for homework, and second by hand. Compare the results.

1. $f(x) = \frac{1}{3}x^3$
2. $f(x) = 3x^4 - 5x^3 + 3x^2 + x - 1$
3. $f(x) = x^3 \cos x$
4. $f(r) = \pi r^2$
5. $f(r) = \frac{4}{3}\pi r^3$
6. $f(x) = \sqrt[3]{x}$
7. $f(x) = e^x \cos x$
8. $f(x) = e^{ax+b}$
9. $f(x) = b^x$
10. $f(x) = \ln(ax + b)$
11. $f(x) = \log_b x$
12. $f(x) = \sin(x^2)$
13. $f(x) = \dfrac{\sin x}{x^2 + 1}$

14. $f(x) = \dfrac{1}{x}$: compute $\mathcal{D}_x f(x)$ two different ways — once using the Power Rule (writing $f(x) = x^{-1}$), and once using the Quotient Rule (writing $f(x)$ as the quotient $\dfrac{1}{x}$) — and show that the two results agree.

15. $f(\theta) = \sin^3\!\left(\theta + \frac{\cos 2\theta}{\theta^2 - 1}\right)$

4.1.2 Derivatives via Taylor Series

Reminder. Two familiar functions can be written as infinite power series:

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots, \qquad \cosh x = 1 + \frac{x^2}{2!} + \frac{x^4}{4!} + \frac{x^6}{6!} + \cdots.$$

Just as with finite polynomials, these series may be differentiated term by term — exactly the technique used in class (Chapter 1, Section 1.2.9) to differentiate e^x, e^{-x}, and $g = e^x + e^{-x}$, $h = e^x - e^{-x}$, and again in Chapter 7 to prove $\mathcal{D}_x \sin x = \cos x$.

1. **Derivative of** cos**.** Differentiate the series for $\cos x$ term by term. Show that the result equals the series for $-\sin x$ (recall $\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \cdots$). What does this tell you about the identity $\mathcal{D}_x \cos x = -\sin x$?

2. **Derivative of** cosh**.** Differentiate the series for $\cosh x$ term by term. Show that the result equals $x + \frac{x^3}{3!} + \frac{x^5}{5!} + \cdots$ — the series for $\sinh x$, i.e. for $h(x) = e^x - e^{-x}$ (up to the constant factor 2) from class. Unlike $\cos x$'s series, none of the signs alternate here: is that consistent with what you already showed in class about $\mathcal{D}_x g = h$ and $\mathcal{D}_x h = g$?

4.1.3 Prove the Power Rule for $n \in \mathbb{N}$

This exercise is an instance of *mathematical induction*: to prove that a proposition P_n holds for every natural number n, it suffices to prove a *base case* (P_0 and/or P_1 below) and an *inductive step* showing that P_n implies P_{n+1} for every n. Once both pieces are established, P_n is thereby proven for *all* n at once — the base case starts a chain of implications $P_1 \Rightarrow P_2 \Rightarrow P_3 \Rightarrow \cdots$ that reaches every natural number. This exercise walks you through the inductive step by hand; the same theorem, proven formally by induction, appears as Theorem 7.4.7 in Chapter 7.

Define the following propositions:

- $P_1 : \mathcal{D}_x x^1 = 1\, x^0 = 1$
- For each $n > 0$ define $P_n : \mathcal{D}_x x^n = n x^{n-1}$

1. **Base case.** Verify that P_1 holds, using the Identity Rule ($\mathcal{D}_x x = 1$) already established in class.

2. Assume P_{42} is true, and use that fact to prove P_{43}. To do this use the fact that $x^{43} = x\, x^{42}$, and apply the Product rule, to show $\mathcal{D}_x x^{43} = 43 x^{42}$.

3. Use the pattern of proving P_{43}. Assume that $n > 0$ and P_n is true. Prove P_{n+1} is true by expressing $x^{n+1} = x\, x^n$, and calculating its derivative using the Power Rule. This is the *inductive step* of a proof by induction that P_n holds for all $n \geq 1$.

4.1.4 Implicit Differentiation

1. **Warm-up.** Consider the curve $xy = 6$.

   (a) Verify that the points $(1, 6)$ and $(2, 3)$ lie on the curve.

(b) Differentiate both sides of $xy = 6$ with respect to x, treating y as a function of x, and solve for $\dfrac{dy}{dx}$.

(c) Find the slope of the tangent line at $(1, 6)$ and at $(2, 3)$.

(d) Check your answer by solving the equation explicitly for y and differentiating directly.

2. Show that if $x^2 + y^2 = 4$, then $y'' = \dfrac{-4}{y^3}$.

(a) Differentiate both sides of $x^2 + y^2 = 4$ implicitly with respect to x and solve for y'.

(b) Differentiate y' again with respect to x using the quotient rule. Your expression for y'' will still contain y'.

(c) Substitute the formula for y' from part (a) into your expression for y''.

(d) In the numerator, replace x^2+y^2 using the original equation and simplify to obtain $y'' = \dfrac{-4}{y^3}$.

3. **Slope of a slanted parabola.** The curve

$$(x + y)^2 = 4(x - y)$$

is a parabola whose axis makes a $45°$ angle with the x-axis.

(a) Verify that the points $(0, 0)$, $(4, 0)$, and $(0, -4)$ all lie on the parabola.

(b) Use implicit differentiation to find $\dfrac{dy}{dx}$.

(c) Find the slope of the tangent line at each of the three points above.

(d) The axis of this parabola has slope -1. The tangent at the vertex is perpendicular to the axis. Which of the three points is the vertex? How does the slope you computed confirm this?

Hint for (b): Treat $x+y$ as a single quantity and apply the chain rule to the left-hand side: $\dfrac{d}{dx}(x+y)^2 = 2(x+y)\cdot\dfrac{d}{dx}(x+y) = 2(x+y)(1+y')$.

4.1.5 Derivative of Known Identities (Optional)

The purpose of this exercise is to practice your understanding of the derivative rules, especially the chain rule.

1. Start with this known trigonometric identity: $\cos^2\theta + \sin^2\theta = 1$. Compute the derivative of both sides using rules such as the Chain Rule, Power Rule, Summation Rule, *etc.*, and show that the resulting equality is true.

2. Derive a formula for $\cos 2\theta$ by computing the derivative of both sides of the following trigonometric identity:

$$\sin 2\theta = 2\sin\theta\cos\theta.$$

   Specifically, show that

$$\cos 2\theta = \cos^2\theta - \sin^2\theta.$$

3. Derive a formula for $\sin^3\theta$ by computing the derivative of both sides of the following trigonometric identity:

$$\sin^4\theta = \frac{3 - 4\cos(2\theta) + \cos(4\theta)}{8}.$$

Specifically, show that

$$\sin^3\theta = \frac{2\sin 2\theta - \sin 4\theta}{8\cos\theta}.$$

Hint: do not use the Quotient Rule.

4.1.6 Finding Extrema

Apply the First and/or Second Derivative Test (Chapter 3) to find and classify the extrema of each function below.

Tip: once you've worked a problem out by hand, you can check your answer by plotting the function at `www.desmos.com` *(or any graphing tool). Do the calculus first, though — the point of these exercises is to find the shape of the graph* from *the derivatives, not to read the answer off a picture.*

1. $f(x) = x^2 + 1$. Find the critical point of f, and use the Second Derivative Test to determine whether it is a local maximum or a local minimum.

2. $g(x) = \dfrac{x^3}{3} + x$.

   (a) Compute $g''(x)$, and notice that it is exactly $f'(x)$ from Problem 1 — no new differentiation needed.

   (b) Notice also that $g'(x) = x^2 + 1$ is exactly $f(x)$ from Problem 1. Using that fact (again, no need to redo work already done), find the critical point(s) of g by solving $g'(x) = 0$.

   (c) What do you conclude? Does the Second Derivative Test have anything left to classify, and does g have any local extrema?

(d) What is the overall shape of g: is it increasing, decreasing, or neither, on all of $\mathbb{R}$?

3. $f(x) = x + \sin x$.

   (a) Find the critical point(s) of f.

   (b) Try applying the Second Derivative Test at each critical point. What happens?

   (c) Use the First Derivative Test instead (a sign analysis of f') to show that f has *no* local extrema at all, even though it has critical points.

4. $f(x) = e^{-x^2}$.

   (a) Compute $f'(x)$ and $f''(x)$.

   (b) Find every x where $f'(x) = 0$, and every x where $f''(x) = 0$. *Hint: recall that e^x is always positive (hence never zero), for any real x — keep that in mind when solving $f'(x) = 0$ and $f''(x) = 0$: in each case, factor out the exponential first, then solve what remains. (The two equations don't behave the same way — one of them has more solutions than the other.)*

   (c) Using only this information — we do not need an exact plot, only an approximate sketch based on what the critical points and the First and Second Derivative Tests tell us — what can you conclude about the shape of the graph of f: where is it increasing or decreasing, where is it concave up or down, and does it have any local extrema or inflection points?

5. $f(x) = \cos(x^2)$.

(a) Compute $f'(x)$ and $f''(x)$ (you will need the Chain Rule and the Product Rule together).

(b) Show that $f'(x) = 0$ exactly when $x = 0$ or $\sin(x^2) = 0$, and derive the general formula $x = \pm\sqrt{k\pi}$, $k = 0, 1, 2, \ldots$, that covers *every* critical point — including $x = 0$ itself, as the $k = 0$ case, so there is no separate branch to track. Explain why f has infinitely many critical points.

(c) Restrict attention to the interval $[0, 3]$. List the critical points of f that lie in this interval (there are three, including $x = 0$).

(d) Classify each of these three critical points using the Second Derivative Test. *Hint: at any critical point where* $\sin(x^2) = 0$, *the Pythagorean identity forces* $\cos(x^2) = \pm 1$ *there, which makes* f'' *easy to evaluate — except at one of the three points, where a leftover factor wipes out that value entirely. Which point is it, and what do you need to fall back on to classify it?*

6. Sketch the graph of $f(x) = \dfrac{4x - 3}{x^2 + 1}$.

(a) Compute $f'(x)$ using the Quotient Rule, and show that it simplifies to

$$f'(x) = \frac{-4x^2 + 6x + 4}{(x^2 + 1)^2}.$$

(b) Find the critical points of f by solving $-4x^2 + 6x + 4 = 0$ (equivalently $2x^2 - 3x - 2 = 0$). This one factors — no quadratic formula needed.

(c) The numerator of f' is a downward-opening parabola in x. Without computing any decimal approximations, use this fact to determine the sign of $f'(x)$ on each of the three

intervals cut out by the two critical points, and classify each critical point (First Derivative Test) as a local maximum or a local minimum.

(d) As a check, compute $f''(x)$ and show it equals

$$f''(x) = \frac{8x^3 - 18x^2 - 24x + 6}{(x^2+1)^3}.$$

This looks unpromising, but at a critical point c (a root of $2c^2 - 3c - 2 = 0$, so $c^2 = \frac{3c+2}{2}$) it collapses dramatically. *Hint: repeatedly substitute* $c^2 = \frac{3c+2}{2}$ *into* $8c^3 - 18c^2 - 24c + 6$ *(you will need it once to reduce* c^3 *to a linear expression in* c*, and again for the remaining* c^2*) to show* $f''(c) = \dfrac{-25c}{(c^2+1)^3}$. Use this exact expression to confirm your classification from part (c) via the Second Derivative Test.

(e) One more clue before you sketch: think about what happens to $f(x)$ when x is a huge positive number, or a huge negative number ($x \gg 0$ or $x \ll 0$). The numerator $4x - 3$ only grows as fast as x itself, but the denominator $x^2 + 1$ grows much faster, like x^2 — and a fraction whose bottom is growing much faster than its top gets closer and closer to 0. So far out to the left and to the right, the graph of f should hug the x-axis.

(f) Combine parts (a)–(c) with this observation about large $|x|$ to sketch the graph of f.

4.2 Derivative Notations

Chapter 1 states differentiation rules in *point-free form* using the $\mathcal{D}$ operator: functions are manipulated directly, with no input variable named. This is the natural style for the symbolic-differentiation code you will implement. The same rules can be expressed in three other notations that you will encounter in textbooks, lectures, and online resources.

Notation	**Point-free example**	**Pointwise example**
$\mathcal{D}$ operator	$\mathcal{D}\, f$	$\mathcal{D}_x f(x)$
Newton (prime)	f'	$f'(x)$
Leibniz	*none*	$\dfrac{d}{dx} f(x)$

Leibniz notation is inherently pointwise: it always names the variable, so there is no point-free Leibniz form.

For each rule in items 1–4, write the rule in:

(a) pointwise form using $\mathcal{D}_x$,

(b) Newton's prime notation (f', g', *etc.*),

(c) Leibniz notation ($\dfrac{d}{dx}$).

1. **Summation Rule.** Theorem 1.2.6 states in point-free form:

$$\mathcal{D}\,(h + g) = \mathcal{D}\,h + \mathcal{D}\,g.$$

2. **Scalar Rule.** Corollary 1.2.9 states in point-free form ($c \in \mathbb{R}$ a constant):

$$\mathcal{D}\,(c\,g) = c\,\mathcal{D}\,g.$$

3. **Product Rule.** Theorem 1.2.8 states in point-free form:

$$\mathcal{D}\,(h\,g) = h \cdot \mathcal{D}\,g + g \cdot \mathcal{D}\,h.$$

4. **Chain Rule.** Theorem 2.2.1 states in point-free form:

$$\mathcal{D}\,(h \circ g) = (\mathcal{D}\,h) \circ g \cdot \mathcal{D}\,g.$$

   For part (c), introduce $u = g(x)$ and $y = h(u)$, so that $y = (h \circ g)(x)$, and express the rule using $\dfrac{dy}{dx}$, $\dfrac{dy}{du}$, and $\dfrac{du}{dx}$.

5. **Reflection.** Leibniz notation required you to introduce auxiliary variables u and y for the Chain Rule, but not for the Summation or Product rules. In one sentence, explain why.

4.3 Optimize a 2-Level Linked List Index

Goal. We build an index of size k (described below) on top of a sorted list of n elements, and searching costs more or less depending on k. We want to find the value of k that minimizes the total search cost $T(k)$ — the time to find a target element that lives in L0.

We are given a sorted linked list of n elements (L0). We construct a second-level index (L1).

L1 is itself a linked list of *tails* of L0. Its first node references the head of L0 (the whole list); each subsequent node references a later tail of L0, marking the start of a new segment. Concretely, each L1 *cons cell* — a pair, using the traditional Lisp names `car` and `cdr` for its first and second fields — has:

- a **car** that is a pointer to some node in L0 (giving access to the tail of L0 from that point); and
- a **cdr** that points to the next L1 cell.

With k nodes in L1, L0 is partitioned into approximately k contiguous segments.

To search for an element:

- Traverse L1 sequentially (following cdrs) to find two consecutive L1 nodes t_i and t_{i+1} such that $\text{car}(t_i) \leq \text{key} < \text{car}(t_{i+1})$.
- Follow $\text{car}(t_i)$ into L0 and scan forward until the element is found or $\text{car}(t_{i+1})$ is reached.

We assume:

- Searching L1 requires time proportional to k.
- Searching within an L0 segment requires time proportional to $\frac{n}{k}$.

Thus the total search time is modelled by:

$$T(k) = k + \frac{n}{k}.$$

Figure 4.1 shows a small concrete example of this two-level structure.

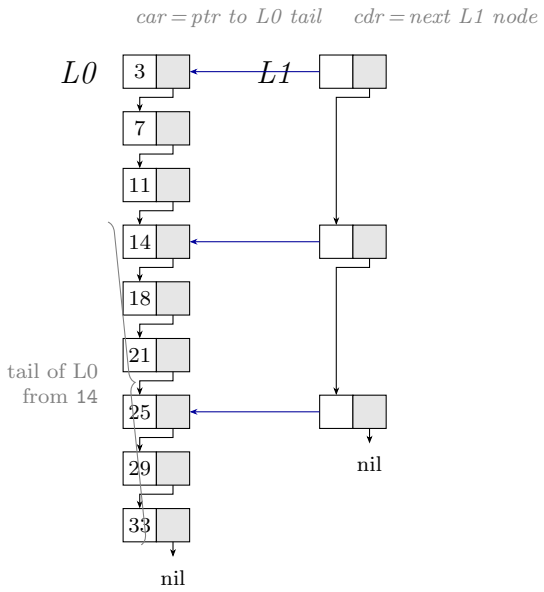


Figure 4.1: Two-level structure ($n = 9$, $k = 3$). Levels are columns; nodes within a level stack top-to-bottom. Each cell shows car (white, left) and cdr (grey, right): the cdr arrow descends to the next node in the same column; the blue car arrow points left into an L0 node, giving access to the tail of L0 from that point downward. A search scans L1 downward, then follows one car into L0 and scans from there.

4.3.1 Interpretation

Explain in words why:

- the term k corresponds to searching L1;

- the term $\frac{n}{k}$ corresponds to searching within L0.

4.3.2 Cost Function

Using the two assumptions above, explain how they combine into the formula $T(k) = k + \frac{n}{k}$ already given above, and explain why $k > 0$ is required.

4.3.3 Differentiation

Compute the derivative $T'(k)$ with respect to k.

4.3.4 Critical Points

Solve $T'(k) = 0$ to find the critical value(s) of k.

4.3.5 Optimal Structure Size

Show that the minimising value of k satisfies $k = \sqrt{n}$. Explain intuitively why this represents a balance between index traversal and segment search.

4.3.6 *(Optional)* Verification

Compare the behavior of $T(k)$ when $k \ll \sqrt{n}$ and when $k \gg \sqrt{n}$.

4.4 Modified Cost Model

Consider a modified search structure where the cost model is:

$$T(k) = 2k + \frac{n}{k}.$$

1. Find the value of k that minimizes $T(k)$.
2. Interpret how increasing the cost of Level 1 traversal affects the optimal index size.

4.5 Multi-Level Skip List Optimization

Goal. Instead of a single index parameter k, this structure is controlled by a *promotion probability* $p \in (0, 1)$ (described below), and every level of the structure is built using this same p. We want to find the value of p that minimizes the expected search cost $T(p)$ — the expected time to find a target element that lives in L0, once we account for both the number of levels the search passes through and the work done at each level.

We now generalize to a hierarchy of linked lists. L0 is the full sorted list. Ld is a linked list of *tails of* L$(d-1)$: each of its cons cells has a car pointing to some node in L$(d-1)$, giving access to the tail of L$(d-1)$ from that point onward.

Each element of L0 is independently promoted to L1 with probability p; each L1 node is independently promoted to L2 with probability p; and so on. The expected number of nodes per level is

$$n, \quad pn, \quad p^2n, \quad p^3n, \quad \ldots$$

Assume that:

- A search begins at the head of the highest level.
- At each level, follow cdrs until the next car would overshoot the target, then follow the current car down to the level below.
- The cost of examining a node is constant, independent of level.

Under these assumptions, a simplified expected search cost is

$$T(p) = \frac{1}{p} \log_{1/p}(n).$$

Figure 4.2 shows a small concrete example of this multi-level structure.

4.5.1 Number of Levels

Show that the expected number of levels is approximately $L = \log_{1/p}(n)$.

4.5.2 Horizontal Cost

Explain why the expected number of steps at each level is proportional to $\frac{1}{p}$.

4.5.3 Rewriting the Model

Using the change-of-base identity recalled in the Instructions, rewrite $T(p)$ using only natural logarithms.

4.5.4 Simplification

Show that minimising $T(p)$ is equivalent to maximising $-p \ln p$.

4.5.5 Optimization

Differentiate $-p \ln p$ and determine the critical point.

4.5.6 Conclusion

Show that the optimal promotion probability is $p = \frac{1}{e}$. Explain why this value balances the number of levels against the amount of searching at each level.

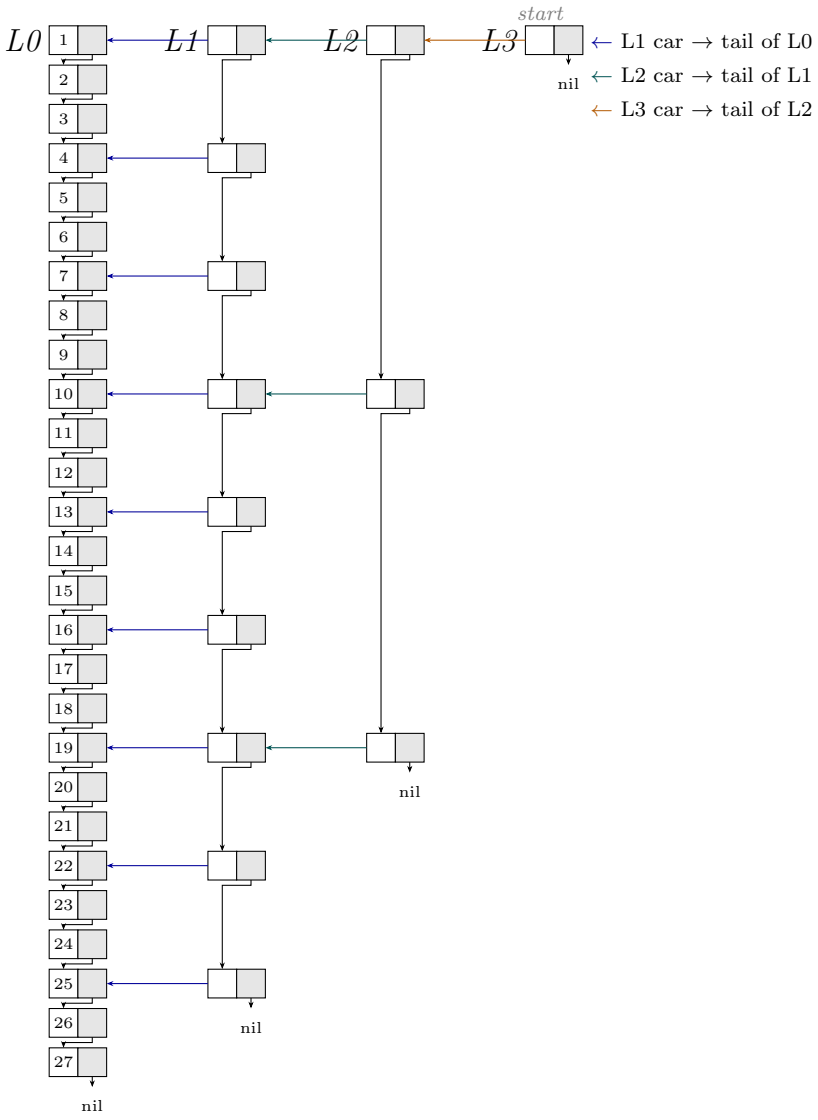


Figure 4.2: Multi-level structure ($n = 27$, $p = \frac{1}{3}$, four levels). Colored arrows are car pointers descending to the next level; black arrows are cdr pointers advancing within the current level.

Chapter 5

Symbolic Differentiation via Pattern Matching

Chapter Objectives

- Explain why the base class is called `PointFree` and describe the ADT it roots: leaf nodes (`Numeric`, `Symbol`, `Identity`, `Power`, `Sin`, `Cos`, `Exp`) and binary nodes (`Sum`, `Difference`, `Product`, `Quotient`, `Compose`).
- Construct and evaluate expression ASTs using Python class instances and `match` / `case` pattern matching.
- Explain why composition makes the set of `PointFree` objects a monoid, and why representing f^n as `Compose(Power(n), f)` keeps differentiating it a constant-size operation.
- Preview what `derivative(node)` must produce for a representative expression, and see the one-to-one correspondence between differentiation rules and `case` branches — implementing it for every node type is the group project, Chapter 6.
- Read `toPython`, provided complete, to see the same `evaluate` skeleton repurposed to serialize an AST to a Python expression string, including the `Compose` case's `var`-threading trick.

In this chapter we represent *differentiable functions* as *Abstract Syntax Trees* (ASTs) — tree-shaped data structures whose internal nodes are operators (sum, product, composition, ...) and whose leaves are constants, symbols, or elementary functions — which we can manipulate in programs. In Section 5.6 we actually look at the code for computing the derivative. However, in the current chapter we set up the data structure in a way that allows us to *construct* such functions, and serialize them with a human-readable representation. We will also construct an interface to evaluate a function at a given input value, mathematically representing a real number, but programmatically represented by an `int` or `float` in Python.

A recurring theme in this course is that mathematical structure and code structure are two sides of the same coin. When the shape of the code mirrors the shape of the mathematics, two things happen at once: the code becomes easier to read and easier to verify, and the mathematics becomes easier to understand — sometimes the clearest path to grasping a concept (a definition, technique, rule, *etc.*) is to implement it. Furthermore, when code and mathematics genuinely resemble each other, we gain a kind of *confidence* that is hard to achieve otherwise: if every differentiation rule corresponds to exactly one `case` branch, and every `case` branch corresponds to exactly one differentiation rule, then a correct rule and a correct branch are nearly the same thing.

Making code align with mathematics is not always easy; there are genuinely hard problems in computer science where the gap between mathematical specification and working implementation is wide and treacherous. But symbolic differentiation is a rare and fortunate exception: the problem is one where the alignment can be made almost perfect. The ADT (Algebraic Data Type) designed in this chapter is not chosen for convenience — its structure is *forced* by the mathematics, and that force is what makes the eventual code so elegant.

5.1 Pattern Matching: Intro

Before designing the `PointFree` class hierarchy from scratch in Section 5.2, it helps to see `match` / `case` actually run. Rather than an invented example, here is a first, partial look at `evaluate` — the function you will complete in Section 5.3 — restricted for now to four cases simple enough to need no recursion. You do not need to understand *why* these particular classes exist yet; Section 5.2 explains the design. For now, just watch how a function can dispatch on which class an object belongs to.

```
class Numeric:
  __match_args__ = ('c',)   # tells match/case what "c" below means
  def __init__(self, c):
    self.c = c

class Sin:
  pass

class Cos:
  pass

class Identity:
  pass
```

Before finishing `evaluate`, look at how each of these four classes is *instantiated* — building one object of each:

```
Numeric(3)      # a Numeric carries a value --- it must be given one
Identity()      # Identity, Sin, and Cos carry no data, so they take
Sin()           # no argument: none is needed, and none is allowed
Cos()
```

`Numeric` stores a number, so its constructor *requires* that number; the other three store nothing and are built with empty parentheses — passing an argument would be an error. Keep this in view when reading the `match` below: each `case` pattern is written to look *exactly* like the constructor call that would build the object, with one change

— where the constructor takes a stored *value*, the pattern takes a *variable* instead, and on a successful match that variable is bound to the value the object was constructed with. So `case Numeric(c):` mirrors `Numeric(3)`, with `c` capturing the `3`; and `case Sin():` mirrors `Sin()` exactly, capturing nothing.

Naming an instance and inspecting it:

```
n = Numeric(3)    # give the object a name
n.c               # --> 3       (the stored value, read directly)
print(n)          # --> <__main__.Numeric object at 0x104e2f9d0>
```

The stored value is always reachable as `n.c` — the `match` statement's binding is a convenience over exactly this attribute access, not a new capability. Notice what `print(n)` shows, though: with no `__repr__` defined, Python falls back to an opaque default. Readable output — `3`, or rendered LaTeX — comes from the serialization added in Section 5.4.

```
from math import sin, cos

def evaluate(expr, x):
  match expr:
    case Numeric(c):    return c
    case Identity():    return x
    case Sin():         return sin(x)
    case Cos():         return cos(x)
```

```
n = Numeric(3)
evaluate(n, 0)              # --> 3
evaluate(Identity(), 7)     # --> 7
evaluate(Sin(), 0)          # --> 0.0
evaluate(Cos(), 0)          # --> 1.0
```

Each `case` line is one *branch*: Python checks the branches top to bottom and runs the first one whose class matches the object's class. `case Numeric(c):` matches only `Numeric` instances, and simultaneously *binds* the number stored inside to the local name

`c` — no `expr.c` needed. `case Identity():`, `case Sin():`, and `case Cos():` bind nothing, because those classes carry no data of their own: the empty parentheses after the class name are part of the pattern, not decoration. This is exactly what "`case` branch" means every time the phrase appears later in this chapter.

A revealing example is `evaluate(Identity(), 7)`. From the code perspective, the `Identity` branch simply returns `x`, and `x` is `7`, so the result is `7`. From the mathematical perspective, `Identity` represents the identity function $\mathrm{id}(x) = x$; evaluating any function f at a point a means computing $f(a)$, so $\mathrm{id}(7) = 7$. The code and the mathematics say exactly the same thing by exactly the same reasoning — which is precisely the alignment this ADT is designed to achieve.

The rest of this chapter designs the `PointFree` class hierarchy — `Numeric`, `Sin`, `Cos`, `Identity`, and their relatives, including the binary node types `evaluate` above does not yet handle — so that `match`/`case` continues to work this cleanly no matter how deep the tree gets. Section 5.3 picks this same `evaluate` function back up and finishes it.

`evaluate` above could also have been written not as a single function, but rather as different methods, one on each of the classes `Numeric`, `Sin`, `Cos`, and `Identity`. Pattern matching, though, offers something plain object orientation has no clean way to express: matching on the *shape* of a sub-expression, not just the top-level class. The rest of this section demonstrates why that matters, using a class you will meet formally in Section 5.2: `Product`, which combines two sub-expressions.

```python
class Product:
  __match_args__ = ('f', 'g')    # tells match/case what f, g mean
  def __init__(self, f, g):
    self.f = f
    self.g = g
```

Activity 1

Do: As an exercise in futility, without using `match` / `case`, write a Python function `try_extract(expr)` that returns the tuple `(a1, a2, a3)` if `expr` has exactly the shape `Product(Product(a1, Product(a2, a3)), Sin())`, and returns `None` otherwise. You may only use `isinstance` and attribute access (`.f`, `.g`).

Discuss: How many `isinstance` calls did your solution need? How confident would you be reading it again in a week?

One possible solution:

```python
def try_extract(expr):
  if isinstance(expr, Product):
    if isinstance(expr.f, Product):
      a1 = expr.f.f
      if isinstance(expr.f.g, Product):
        a2 = expr.f.g.f
        a3 = expr.f.g.g
        if isinstance(expr.g, Sin):
          return (a1, a2, a3)
  return None
```

Four nested `isinstance` checks, four levels of indentation, and a chain of attribute lookups (`expr.f.g.f`) just to reach `a2` — and this is still one of the *shallower* patterns you will meet in this chapter. `case Product(Product(a1, Product(a2, a3)), Sin()):` says the same thing in one line, shaped exactly like the tree it describes. This is the payoff of *declarative* pattern matching: you state *what* shape you

are looking for, and the language handles *how* to check for it.

Here is the same `try_extract` rewritten with `match` / `case`:

```
def try_extract(expr):
  match expr:
    case Product(Product(a1, Product(a2, a3)), Sin()):
      return (a1, a2, a3)
    case _:
      return None
```

One pattern, one line, zero explicit `isinstance` calls and zero manual attribute lookups — the class names in the pattern (`Product`, `Sin`) already encode every type check the nested-`if` version had to spell out, and the variable names (`a1`, `a2`, `a3`) already encode every attribute lookup. The final `case _:` is a *wildcard* pattern: it matches anything, so it catches every `expr` that did not match the shape above — playing the same role as the `return None` at the end of the hand-written version. Notice, too, that the pattern `Product(Product(a1, Product(a2, a3)), Sin())` is written with the exact same syntax as the constructor call that would have *built* such a tree in the first place: pattern matching unifies construction syntax and extraction syntax, so a pattern already looks like the shape it matches, rather than reading as a sequence of imperative `isinstance` checks and attribute lookups.

Look back at your own nested-`isinstance` solution next to this one. Which is easier to *read*? And which one makes it easier to tell, at a glance, what shape of tree the function is even trying to recognize — the mechanics of checking, or the shape itself?

5.2 ADT

In Chapter 5, *Abstract Syntax Trees*, of your advanced data structures course, you already built exactly this kind of structure. An AST there

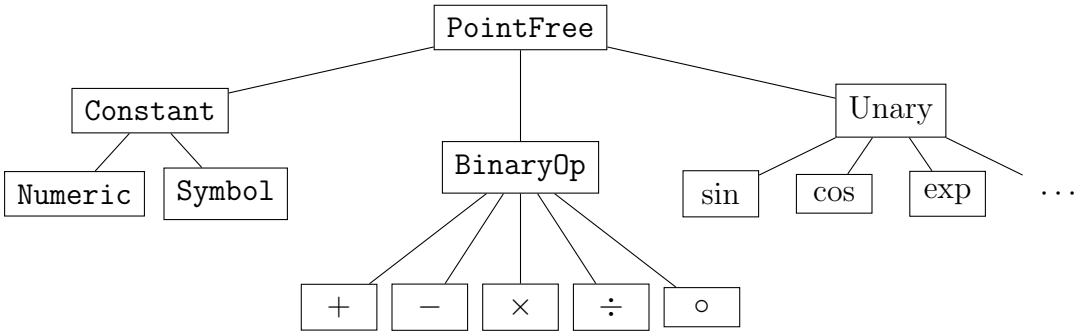


Figure 5.1: ADT for point-free functions. Leaves correspond to base cases in a `match` / `case` block; internal nodes are abstract categories grouping related cases. `Constant`, `BinaryOp`, and `Unary` are abstract base classes; `Unary`'s leaves are `Identity`, `Power(n)`, `Sin`, `Cos`, and `Exp`.

represented an arithmetic expression such as $(3 - \frac{20}{5}) \times (4 + 7)$, and an ADT — there too, an *Algebraic Data Type* — was a tree that exhaustively specifies the syntax an AST is allowed to have, implemented as a class hierarchy: abstract classes `Expression`, `ExpOperation`, `ExpConst`, `ExpUnary`, and `ExpBinary` at the top, and leaf classes — `ExpNumber`, `ExpVariable`, `ExpAdd`, `ExpSubtract`, `ExpMultiply`, `ExpDivide`, `ExpMinus` — doing the actual construction. Here we define a new ADT by exactly the same recipe, for *point-free functions* instead of arithmetic expressions: the identity function, numeric constants, the binary operations $+$, $-$, $\times$, $\div$, and composition $\circ$, the power function x^n, and the transcendental functions sin, cos, and exp. `Constant`, `BinaryOp`, and `Unary` (Figure 5.1) play the role that `ExpConst`, `ExpBinary`, and `ExpUnary` played there.

A word of caution: outside this course sequence, the acronym ADT very often means something else: *Abstract Data Type*, a type specified by the operations it supports (push, pop, enqueue, dequeue, ...), with its internal representation deliberately hidden from the client. That is *not* what ADT meant in Chapter 5 of the advanced data structures course, and it is not what it means here either: both there and here,

ADT means *Algebraic Data Type*, a type built as a closed, enumerable set of variants — the leaf and binary node classes of Figure 5.1 — over which we can pattern-match *exhaustively*. Algebraic data types are a first-class language feature in Haskell and other languages in the ML family (Standard ML, OCaml, ... — "ML" the programming language, not machine learning), declared with a `data` keyword; the language itself checks that a `case` analysis covers every variant. Python has no equivalent language construct: what we build in this chapter, like what you built in Chapter 5, is only a *pattern* — a class hierarchy that imitates an algebraic data type without the language enforcing exhaustiveness on our behalf. If you encounter the term *Abstract Data Type* elsewhere, remember the two notions share a name and both organize data around a fixed set of cases, but they answer different questions: an Abstract Data Type asks *what can I do with this value?*; an Algebraic Data Type asks *what shapes can this value take?*

5.2.1 Why `PointFree`?

In mathematics we draw a sharp distinction between two kinds of objects that look superficially similar.

- An **expression** such as $x^2 + 2x + 1$ *mentions the variable* x *explicitly*. It is a formula in an unknown; it becomes a number only once you substitute a value for x.
- A **function** such as $x \mapsto x^2 + 2x + 1$ does not name x as a free variable. The x is a bound placeholder: the function is a mapping that exists independently of any particular input. You *apply* it to a point to obtain a number.

This is the same distinction that underlies the two notation columns in the derivative table of Chapter 2 (Figure 2.1): the *pointwise* form

$\mathcal{D}_x \sin x = \cos x$ names the point x; the *point-free* form $\mathcal{D}\sin = \cos$ does not.

The nodes of our ADT belong to the second category. `Sin()` is *the sine function* — not the expression $\sin(x)$. No free variable x appears anywhere in the object; the input is supplied later, when you call `evaluate`. This style of defining functions without ever naming the input variable is called *point-free* (or *tacit*) programming. The "point" that is removed is the input point x itself. Point-free style is common in functional languages such as Haskell, and in category theory it is the default: functions are composed without reference to their arguments.

Translating ordinary (pointwise) notation into `PointFree` construction syntax takes practice, especially for expressions that mention x explicitly or that combine several operations. Table 5.1 works through a progression of examples. Two recurring traps: (1) a bare x does *not* disappear when you drop the point — it becomes `Identity()`, never nothing; and (2) "applying a function to a sub-expression" (composition) and "multiplying two sub-expressions together" (a product) can look deceptively similar in ordinary notation but use completely different constructors — $3x^2$ is a *product* of the constant 3 and x^2, while $\sin(x^2)$ is a *composition* of sin with x^2. (The table below also has two rows, `Compose(Sin(), Power(2))` and `Compose(Power(2), Sin())`, that look almost identical but compose in opposite order — *not* interchangeable; recall Section 2.1.2 for why swapping which function is outer and which is inner changes the function entirely.)

The base class is therefore called `PointFree`: every node in the hierarchy is a point-free function. Here is the actual class, from `src/common/PointFree.py` — you already have full access to this file:

Pointwise	Point-free	Python
$f(x) = x$	$f = \mathrm{id}$	`Identity()`
$f(x) = x^2$	$f = (\cdot)^2$	`Power(2)`
$f(x) = 3$	$f = 3$	`Numeric(3)`
$f(x) = 3x^2$	$f = 3 \cdot (\cdot)^2$	`Product(Numeric(3), Power(2))`
$f(x) = x^2 + 1$	$f = (\cdot)^2 + 1$	`Sum(Power(2), Numeric(1))`
$f(x) = \sin(x^2)$	$f = \sin \circ (\cdot)^2$	`Compose(Sin(), Power(2))`
$f(x) = \sin(x)^2$	$f = (\cdot)^2 \circ \sin$	`Compose(Power(2), Sin())`
$f(x) = \sin(x^2 + 1)$	$f = \sin \circ \big((\cdot)^2 + 1\big)$	`Compose(Sin(), Sum(Power(2), Numeric(1)))`
$f(x) = \frac{1}{x}$	$f = 1/\mathrm{id}$	`Quotient(Numeric(1), Identity())`
$f(x) = e^{-x}$	$f = \exp \circ (-1 \cdot \mathrm{id})$	`Compose(Exp(), Product(Numeric(-1), Identity()))`
$f(x) = cx$	$f = c \cdot \mathrm{id}$	`Product(Symbol('c'), Identity())`

Table 5.1: From pointwise notation, to point-free notation, to `PointFree` construction syntax. Every entry in the right column is a literal Python expression — type it at the interpreter, call `.evaluate(x)` on it, and check it against the pointwise formula on the left. (c in the last row is a `Symbol`, a named parameter — see Section 5.2.)

```
class PointFree:
    def __init__(self):
        pass

    __match_args__ = ()
```

That is the entire base case: no data, no children, just the object that every node in the ADT ultimately inherits from. `__match_args__ = ()` is the default every leaf and branch class either inherits unchanged or overrides with its own tuple of attribute names.

`PointFree` does more than this bare skeleton suggests. It defines `__repr__`, `__eq__`, and `__hash__` generically, purely in terms of what-

ever `__match_args__` a subclass declares — one implementation that automatically works correctly for every node type. It also overloads Python's arithmetic operators, so `Sin() + Cos()` literally constructs `Sum(Sin(), Cos())`: the algebra you write in Python *is* the algebra of the ADT. Read the full file when you are ready; nothing in it is hidden from you.

`Cos()` does not *call* the cosine function. It *constructs* a `PointFree` object that represents the cosine function. The cosine function does not compute its floating-point value until you supply an input:

```
f = Cos()      # construct a PointFree object; no number computed yet
f.evaluate(0) # now the cosine function is applied: returns 1.0
```

There is indeed some inefficiency here. Calling `Cos()` multiple times will construct several `PointFree` objects which are the same in all ways except that they are not identical according to the Python `is` operator.

```
f = Cos()      # construct a PointFree object
g = Cos()      # construct another PointFree object
f is g         # returns False
f.evaluate(1.2) == g.evaluate(1.2) # returns True
```

The connection to the `evaluate` function you will write is direct: `f.evaluate(x)` computes $[\![f]\!](x)$ — the semantic function denoted by the AST `f`, applied to the point `x`. The AST lives in the syntactic world $\mathcal{A}$; the result of `evaluate` lives in the semantic world $\mathcal{E}$. (This connection is developed fully in the enrichment section of Chapter 16.)

5.2.2 Constants: Numeric and Symbol

`Numeric` and `Symbol` are the *constant* leaves: functions whose output does not depend on x. A `Numeric(c)` is a literal number such as `42` or `3.14`; a `Symbol('n')` is a named symbol — list size n, a

probability p, or any other quantity that is fixed with respect to the differentiating variable x but has no specific numeric value. Both have derivative zero: $\mathcal{D}_x c = 0$ and $\mathcal{D}_x n = 0$. The pattern-match branches for both return `Numeric(0)` immediately with no recursive call.

It is worth pausing on the distinction: `Numeric(3)` knows its value is 3; `Symbol('c')` knows only its name. In the expression $f(x) = cx$, the c is a `Symbol` — it can stand for any real number, but for the purpose of differentiating with respect to x, it is constant.

Here are the real classes, unchanged from `src/common/PointFree.py`:

```
class Constant(PointFree):
    __match_args__ = ('c',)

    def __init__(self, c):
        assert type(self) is not Constant
        super().__init__()
        self.c = c

class Numeric(Constant):
    def __init__(self, c: float):
        assert isinstance(c, (int, float))
        super().__init__(c)

class Symbol(Constant):
    def __init__(self, c: str):
        assert isinstance(c, str)
        super().__init__(c)
        self.varname = c
```

`Constant` is itself a `PointFree` subclass, but it is not meant to be instantiated directly: `assert type(self) is not Constant` inside its own `__init__` enforces that — this is the actual mechanism behind Figure 5.1's claim that `Constant` is an *abstract* base class. `Numeric` and `Symbol` each add one more check (`isinstance(c, (int, float))` or `isinstance(c, str)`) before delegating to `Constant.__init__` via `super()`. Neither `Numeric` nor `Symbol` declares its own

`__match_args__`; both inherit `('c',)` from `Constant`, which is why `case Numeric(c):` and `case Symbol(name):` each bind a single positional value even though only `Constant` ever mentions `__match_args__` explicitly.

5.2.3 Binary Operations

A brief note on terminology. The *arity* of a function is the number of arguments it takes: a function of one argument is *unary* (arity one), a function of two arguments is *binary* (arity two), and a function of no arguments is *nullary*, or 0-ary. We reuse this vocabulary for the nodes of our ADT, where it counts something slightly different: not the number of arguments accepted by the mathematical function a node represents, but the number of `PointFree` *children* the node carries in the tree. By this reckoning `Sin()` has arity zero — it has no child subexpressions — even though sin itself is a unary function of x in the ordinary mathematical sense; Section 5.2.4 calls such nodes *unary* precisely because of that mathematical arity, not their tree arity of zero. Keep the two countings distinct: tree arity (how many `PointFree` children a node has) and mathematical arity (how many numeric arguments the function itself takes) coincide for the binary nodes discussed next, but diverge for the leaves discussed after them.

The nodes $+$, $-$, $\times$, and $\div$ are the *binary* nodes of the ADT: each has arity two, combining a left subexpression and a right subexpression. The derivative rules for all four — sum, difference, product, quotient — share the same procedural shape: apply a rule that references both subexpressions and recurs into each. The specific rule differs per operator, but every `case` branch for a binary node will destructure into two children and make two recursive calls.

Here is the real `BinaryOp` class, and how two of the five binary node types are built from it:

```
class BinaryOp(PointFree):
    __match_args__ = ('f', 'g')

    def __init__(self, f: PointFree, g: PointFree):
        assert isinstance(g, PointFree)
        super().__init__()
        self.f = f
        self.g = g

class Sum(BinaryOp):
    pass

class Product(BinaryOp):
    pass
```

`Difference`, `Quotient`, and `Compose` are declared the same way as `Sum` and `Product` above — each is just `class Difference(BinaryOp): pass`, contributing nothing beyond its own name. `BinaryOp` does all the real work: it declares `__match_args__ = ('f', 'g')` once, and every subclass inherits it unchanged, which is exactly why `case Sum(f, g):` and `case Product(f, g):` both destructure the same way even though neither `Sum` nor `Product` ever mentions `f` or `g` itself.

5.2.4 Unary Functions

The unary nodes — `Identity`, `Power`, sin, cos, and exp — are leaves of the AST (arity zero: no `PointFree` subexpression children) whose evaluation *does* depend on x. This is what distinguishes them from the constant leaves above: evaluating `Numeric(3)` or `Symbol('c')` ignores x entirely, whereas evaluating any unary node computes a value from x. `Identity` returns x itself; `Power(n)` returns x^n; sin returns $\sin(x)$; cos returns $\cos(x)$; and so on.

`Power(n)` carries a numeric exponent n, just as `Numeric(c)` carries a numeric value. The difference is that `Numeric(c)` ignores x

while `Power(n)` applies x to the exponent. Because n is a fixed number rather than a `PointFree` subexpression, `Power(n)` has no children to recur into; its `case` branch reads n from the node and uses x directly.

The `case` branch for each unary node therefore has the same shape: match the node (binding any numeric parameter it carries), use x directly, and return a result with no recursive call. The nodes differ only in *which* function of x they compute, and correspondingly in their derivative rule: `Identity` has derivative 1; `Power(n)` has derivative $n\,x^{n-1}$, i.e., `Product(Numeric(n), Power(n-1))`; sin has derivative cos; cos has derivative $-\sin$; exp has derivative exp; and so on.

Here is the real `Unary` class, and the leaf classes built from it:

```
class Unary(PointFree):
    pass

class Identity(Unary):
    pass

class Exp(Unary):
    pass

class Cos(Unary):
    pass

class Sin(Unary):
    pass

class Power(Unary):
    __match_args__ = ('exponent',)

    def __init__(self, exponent: int):
        assert isinstance(exponent, int)
        super().__init__()
        self.exponent = exponent
```

`Unary` itself is never constructed directly, carries no data, and declares

no `__match_args__` of its own — it exists purely to group these five leaf classes under one abstract base, the same role `Constant` plays for `Numeric` / `Symbol` and `BinaryOp` plays for `Sum` / `Product` /etc. `Identity`, `Exp`, `Cos`, and `Sin` carry no data at all — `pass` is the entire class body, and each inherits `Unary`'s (in turn `PointFree`'s) default `__match_args__ = ()`, which is why `case Sin():` and `case Cos():` take no arguments. `Power` is the one unary node that carries data: its own `__match_args__ = ('exponent',)` overrides that inherited empty tuple, which is what lets `case Power(n):` bind the exponent to a local name, `n`. Notice that the *local* pattern name (`n`) does not have to match the *attribute* name (`exponent`) — `__match_args__` only fixes the *position* of what gets bound, not what you choose to call it at the call site.

Just as `case BinaryOp(f, g):` matches any of the five binary node types in one arm, `case Unary():` matches any of `Identity`, `Exp`, `Cos`, `Sin`, or `Power(n)` in one arm — useful anywhere you only care that a node is a childless, x-dependent leaf, not which one.

The requirement that each function be defined and differentiable on all of $\mathbb{R}$ (or at least have a known, unconditional derivative rule) is what determines which functions the ADT supports.

```
>>> Power(0.5)
Traceback (most recent call last):
  ...
AssertionError
```

`Power` supports *integer* exponents only — `Power(0.5)` cannot even be constructed: the `assert isinstance(exponent, int)` above fires immediately, and `PointFree.__pow__` delegates straight to `Power.__init__`, so `Sin() ** 0.5` fails at *construction* time, not evaluation time. This is a deliberate limitation, not an oversight. The power rule itself extends effortlessly to any real exponent r —

$\mathcal{D}_x x^r = r\,x^{r-1}$ is the exact same formula, one line of code, whether r is an integer or not (Chapter 7, Section 7.5.3 proves this for general real r, once the Chain Rule is available). The problem is not differentiation; it is evaluation. Python's own `**` silently leaves the real numbers for a negative base and a non-integer exponent: `(-8) ** (1/3)` does not return -2, it returns a *complex* number, $\approx 1 + 1.73i$. Allowing `Power(0.5)` into the ADT would mean `evaluate` could return a complex number for some real `x` and some perfectly well-formed tree — silently, with no exception — breaking the `-> float` contract every other node in this ADT honors unconditionally. Restricting `Power` to integers, and enforcing that restriction at construction rather than merely documenting it, closes that door entirely: every `PointFree` tree that can be built evaluates to a real number, full stop.

5.3 AST

This section builds the real `PointFree` class hierarchy (Section 5.2) into actual expression trees, and finishes the `evaluate` function previewed in Section 5.1.

5.3.1 Constructing Expression Trees from Objects

In the advanced data structures course, ASTs were often represented as nested tuples or lists. Python imposes no constraints on what may appear inside a tuple, so it was possible to construct structurally invalid trees — a sum node whose left child is the string `'+'` instead of a subexpression, for instance — without any error. Here we use class instances instead. The `__init__` constructor of each class asserts that its children are valid `PointFree` objects, so an invalid tree cannot be constructed: the error is caught at the moment of construction rather than discovered later during a traversal.

A word on Python syntax: constructing a class instance requires parentheses, even when the class carries no data. `Sin()` constructs the sine function; `Cos()` constructs the cosine function; `Power(3)` constructs the function $f(x) = x^3$. This mirrors the mathematical convention that sin, cos, and $x \mapsto x^3$ name *functions*, not values — the parentheses in `Sin()` are Python's construction syntax, not function application.

`Sin` (no parentheses) is the *class* — a Python type object, not a function object. `Sin()` *constructs* an instance of that class, which is what the ADT actually uses. Writing `Sin` where you mean `Sin()` will not produce an immediate error, because Python happily stores a class as a value, but the pattern `case Sin():` matches *instances* of `Sin`, not the class itself. Always write `Sin()`, `Cos()`, `Identity()`, etc.

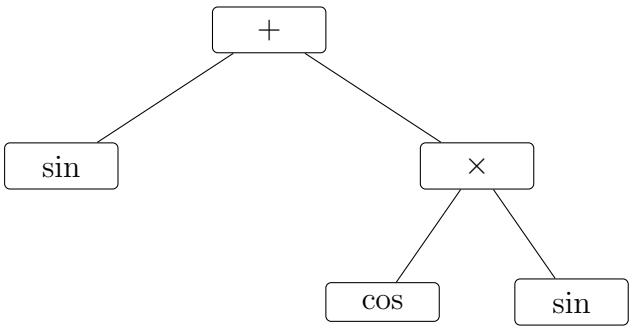


Figure 5.2: AST for $f(x) = \sin(x) + \cos(x) \cdot \sin(x)$, constructed as `Sum(Sin(), Product(Cos(), Sin()))`. Binary nodes ($+$, $\times$) carry two children; function-object leaves (sin, cos) carry none.

Figure 5.2 shows the expression $f(x) = \sin(x) + \cos(x) \cdot \sin(x)$ as an AST. In Python this tree is constructed as:

```
1 expr = Sum(Sin(), Product(Cos(), Sin()))
```

Each constructor call in the Python expression corresponds to exactly one node in the figure, and the nesting of the calls corresponds to the

parent–child relationships in the tree.

5.3.2 From OOP Dispatch to Pattern Matching

In the advanced data structures course, tree traversal was implemented by giving each class its own method and relying on Python's runtime dispatch to select the right one. Section 5.1 already showed the alternative this course uses instead, on four leaf cases. Here is `evaluate` in full, extended with the binary nodes: `case Sum(f, g):` and its four siblings bind their two children the same way `case Product(f, g):` did in Section 5.1, and every one of them recurs into both children before combining the results.

```
def evaluate(expr: PointFree, x: float, env: dict = None) -> float:
  if env is None:
    env = {}
  match expr:
    case Identity():    return x
    case Numeric(c):    return c
    case Symbol(name):
      if name not in env:
        raise ValueError(f"Symbol '{name}' not in environment")
      else:
        return env[name]
    case Power(n):      return x ** n
    case Sin():         return sin(x)
    case Cos():         return cos(x)
    case Exp():         return exp(x)
    case Sum(f, g):
      return evaluate(f, x, env) + evaluate(g, x, env)
    case Difference(f, g):
      return evaluate(f, x, env) - evaluate(g, x, env)
    case Product(f, g):
      return evaluate(f, x, env) * evaluate(g, x, env)
    case Quotient(f, g):
      return evaluate(f, x, env) / evaluate(g, x, env)
    case Compose(f, g):
      return evaluate(f, evaluate(g, x, env), env)
```

The leaf branches (arity zero or one numeric argument) return immediately; the binary branches recur into both children. When you implement the derivative function in Section 5.6, the structure will be identical: one branch per differentiation rule, returning a result directly or recurring depending on the arity.

The optional `env` argument is a dictionary mapping `Symbol` names to numeric values. It is only consulted in the `Symbol` branch; every other branch ignores it. Expressions that contain no `Symbol` nodes can be evaluated without supplying `env` at all:

```
evaluate(Sin(), 0)                          # 0.0
evaluate(Power(3), 2)                       # 8
evaluate(Sum(Sin(), Cos()), 0)              # 1.0
```

```
>>> evaluate(Product(Symbol('c'), Sin()), 1.0)
Traceback (most recent call last):
  ...
ValueError: Symbol 'c' not bound in environment
```

When a `Symbol` is present, the corresponding value must appear in the given `env`; otherwise an exception is thrown.

```
evaluate(Product(Symbol('c'), Sin()), 1.0, {'c': 3.0})   # 3*sin(1)
```

5.4 Serializing Expressions

An AST can be serialized — walked and converted to a string — by following exactly the same `match` / `case` skeleton as `evaluate`. This section covers `toPython`, provided complete in `common/toPython.py` — reading it is enrichment, not a required exercise. A companion function `toLaTeX`, also provided ready-made, is described in Section 6.3.

5.4.1 `toPython`

`toPython` serializes an expression AST to a Python expression string. For example, the AST for $x \cdot \sin(x)$ becomes the string `'(x * math.sin(x))'`. It follows the same skeleton as `evaluate`: a `match` / `case` block with one branch per node type, where each branch returns a string instead of a number. This is a recurring pattern in compiler construction: the same tree traversal, repurposed for different output targets.

Table 5.2 gives the expected output for every node type. For the binary combinators (`Sum`, `Difference`, `Product`, `Quotient`) the two operand strings are obtained by recurring into the left and right children.

Node	String returned by `toPython`
`Identity()`	`'x'`
`Numeric(3)`	`'3'`
`Symbol('c')`	`'c'`
`Power(2)`	`'(x ** 2)'`
`Sin()`	`'math.sin(x)'`
`Cos()`	`'math.cos(x)'`
`Exp()`	`'math.exp(x)'`
`Sum(f, g)`	`'(' + toPython(f) + ' + ' + toPython(g) + ')'`
`Difference(f, g)`	`'(' + toPython(f) + ' - ' + toPython(g) + ')'`
`Product(f, g)`	`'(' + toPython(f) + ' * ' + toPython(g) + ')'`
`Quotient(f, g)`	`'(' + toPython(f) + ' / ' + toPython(g) + ')'`
`Compose(f, g)`	see below

Table 5.2: Expected output of `toPython(expr)` for each node type, with variable name `'x'`.

Because `Sin`, `Cos`, and `Exp` produce strings containing

`math.sin`, `math.cos`, and `math.exp`, the `math` module must be passed to `eval` as a namespace:

```
import math
f = eval('lambda x: ' + toPython(expr), {'math': math})
f(1.0)  # same as evaluate(expr, 1.0)
```

`Compose` and the `var` parameter. For every node except `Compose`, the variable name appears as a simple placeholder: `Power(2)` produces `'(x ** 2)'`, dropping the name `'x'` into a template. `Compose` cannot follow this pattern, because Python has no compose operator: it must build the nested call by string manipulation.

The key idea is a second parameter `var` (default `'x'`) that threads through every recursive call. Leaf nodes substitute `var` directly into their template. `Compose(f, g)` uses two steps:

1. Serialize the *inner* function g with the current variable:
   `inner = toPython(g, var)`
   The call produces a full Python expression string, not just a name.

2. Serialize the *outer* function f, using `inner` as the new variable:
   `return toPython(f, inner)`

In step 2, `var` is not a simple name like `'x'`—it is a complete expression string. `toPython` accepts any string as `var` and substitutes it wherever the variable appears. A concrete trace for `Compose(Sin(), Power(2))`:

```
inner  = toPython(Power(2), 'x') # step 1: '(x ** 2)'
result = toPython(Sin(), inner)  # step 2: 'math.sin((x ** 2))'
```

Figure 5.3 contrasts two compositions to show why argument order matters. `Compose(Sin(), Power(2))` places sin as the outer

`Compose(Sin(), Power(2))`
$f(g(x)) = \sin(x^2)$
`'math.sin((x ** 2))'`

`Compose(Power(2), Sin())`
$f(g(x)) = (\sin x)^2$
`'(math.sin(x) ** 2)'`

Figure 5.3: Two compositions with the same nodes in opposite roles. Each tree shows f (outer, left child) and g (inner, right child). Below each tree: the Python constructor, the mathematical result, and the string returned by `toPython`.

function f and x^2 as the inner g, giving $\sin(x^2)$. Swapping to `Compose(Power(2), Sin())` makes x^2 outer and sin inner, giving $(\sin x)^2$.

How would you convince yourself `toPython` is correct, if it were not provided? One useful technique: the function `gen(n)` in `gen.py` generates a random `PointFree` AST of size n. Wrap the string returned by `toPython` in `eval('lambda x: ...', {'math': math})` and check that the resulting callable agrees numerically with `evaluate` on the same expression, for many randomly generated expressions — exactly what `tests/toPython_test.py` does.

5.4.2 A Note on Meta-programming

Writing programs that generate other programs is called *meta-programming*, and it is one of the recurring big ideas of computer science. You will encounter it many times in your career, in forms ranging from the principled to the alarming.

At the principled end sit Lisp-family languages (Common Lisp,

Scheme, Clojure). In these languages, code and data share the same structure — a program is itself a list, an AST you can manipulate with ordinary functions. The mechanism, called *macros*, lets you extend the language by writing functions that transform code before it is evaluated. There is no gap between the program and its representation. Compilers occupy the same principled end: they transform a source-program AST through a sequence of *intermediate representations* (IRs), each one closer to machine code, using exactly the tree-traversal pattern you see in `toPython`.

At the other end of the spectrum you will find ad-hoc string-based code generation: SQL queries assembled by string concatenation, HTML templates built with `+` and `format()`, configuration files produced by shell scripts. These approaches are everywhere, they often work, and they are the root cause of entire classes of security vulnerabilities — SQL injection, cross-site scripting — that have cost the industry billions of dollars. When you encounter them in the wild, you may gasp. Now you will at least know what you are looking at.

`toPython` sits honestly between these two worlds: it generates strings, but it does so by walking a well-defined AST rather than by ad-hoc concatenation. That discipline — always work from a structured representation, never concatenate raw code strings — is the lesson to carry forward.

5.5 Composition

`Compose` has ridden along with `Sum`, `Difference`, `Product`, and `Quotient` since Section 5.2 as just another two-child node. Structurally it is one. But composition is the operation the Chain Rule is written on, so before the derivative code reaches it (Section 5.6) it is worth pinning down how `Compose` behaves — starting with its algebra.

Composition is *associative*, $(f \circ g) \circ h = f \circ (g \circ h)$ (Corollary 2.1.2), and has a *two-sided identity*, `Identity()`, since $\mathrm{id} \circ f = f \circ \mathrm{id} = f$ for every f (Corollary 2.1.4, proven in Section 2.1.6 of Chapter 2). Those are exactly the two conditions for a *monoid* (Definition 2.1.5), and Proposition 2.1.6 already established that the functions $\mathbb{R} \to \mathbb{R}$ form one under $\circ$. A `PointFree` object *is* a function $\mathbb{R} \to \mathbb{R}$, so that result transfers with nothing new to prove: `Compose` is the operation, `Identity()` is the unit. (`Sum` and `Product` form monoids too, with units `Numeric(0)` and `Numeric(1)`; the `fast_power` discussion below returns to this. `Difference` and `Quotient` form none — they are not associative and have no two-sided identity.)

What sets composition apart from `Sum` and `Product` is that it is *not commutative* (Section 2.1.2): `Compose(Sin(), Power(2))` is $\sin(x^2)$, while `Compose(Power(2), Sin())` is $\sin(x)^2$. The nesting order of `Compose` nodes is part of what the tree means, and the subsections below work through the cases the derivative code will meet — composing with `Power` (Section 5.5.1), with a constant (Section 5.5.2), and with `Identity()` (Section 5.5.3).

It is tempting to think `Compose` only makes sense between "simple" nodes — a `Unary` leaf composed with another `Unary` leaf, say. It is not restricted that way, and it should not be: every `PointFree` node, regardless of its own tree shape, represents a complete function $\mathbb{R} \to \mathbb{R}$. `Compose(Sum(Sin(), Cos()), Power(2))` is perfectly valid — it is $(\sin + \cos) \circ (x \mapsto x^2)$, i.e. $\sin(x^2) + \cos(x^2)$ — and evaluates, differentiates, and renders to LaTeX exactly like any other `PointFree` tree. The "arity" vocabulary from Section 5.2 (zero `PointFree` children for `Constant` and `Unary`, two for `BinaryOp`) describes tree *structure*, not composability: `Compose(f, g)` requires only that `f` and `g` both be `PointFree` objects — exactly what `BinaryOp.__init__`

asserts.

You may already have met this shape of claim: fast exponentiation by squaring — the algorithm behind `fast_power` — is usually introduced as a technique that works for *any* monoid: integers under multiplication, matrices under multiplication, strings under concatenation. `fast_power` itself exploits a *different* monoid on `PointFree` objects than the one this section studies: the *multiplicative* monoid, (PointFree, $\times$, `Numeric(1)`), where `Product` is the operation and `Numeric(1)` is the identity. `PointFree` objects are *also* an *additive* monoid, (PointFree, $+$, `Numeric(0)`), under `Sum` with `Numeric(0)` as the identity — the same two-sided-identity law, with the same caveat that `Sum(f, Numeric(0)) == f` is `False` even though the two sides compute the same function. `PointFree` under `Compose` — the subject of this section — is a *third*, independent monoid on the very same set, with its own identity element, `Identity()`. The fast-exponentiation trick does not care which monoid you feed it; nothing about the trouble `fast_power` ran into was specific to multiplication — which is exactly what makes it worth asking whether the same trouble would show up for composition too.

5.5.1 Composing with Power

The worked example above never needed `Compose`, but the chain rule does, and `PointFree`'s operator overloading (Section 5.2) lets you write compositions in a form that looks deceptively like ordinary arithmetic. `g ** n` does not construct a product of `g` with itself — it constructs `Compose(Power(n), g)`:

```
def __pow__(self, exponent: int):
    return Compose(Power(exponent), self)
```

Recall the Extended Power Rule (Corollary 2.2.8, Chapter 2): $[f(x)]^n$

is a composition with `Power(n)` as the outer function, exactly like $(\sin x)^2$ was. `__pow__` above is the direct AST translation of that same fact — `Power(n)` is not a special case, it is just another node you can compose with.

This choice is not just notational. Chapter 2 already showed, by hand, that differentiating x^4 via the Extended Power Rule takes one step, while unfolding it as $x \cdot x \cdot x \cdot x$ and differentiating via the Product Rule takes three — same answer, more work. Representing f^n with `Compose(Power(n), f)` instead of n repeated `Product` nodes is that same observation, one level up: not just fewer steps to compute the derivative, but a dramatically smaller *tree* to compute it from, and the gap only widens as n grows.

```
def naive_pow(f, n):
    if n == 0:
        return Numeric(1)
    return Product(f, naive_pow(f, n - 1))
```

`homework/fast_power.py` is available in the repository if you want to investigate a cleverer-looking alternative — exponentiation by squaring, computing g^n with $O(\log n)$ multiplications instead of `naive_pow`'s $O(n)$. You are encouraged to implement it and investigate for yourself whether it actually evaluates faster than `naive_pow`; the answer is not what the algorithm's usual reputation would suggest.

For $n = 100$, `Sin() ** 100` (the actual implementation) constructs a tree of depth 1 and 3 nodes total, regardless of n. `naive_pow(Sin(), 100)` constructs a chain of 100 `Product` nodes terminating in `Numeric(1)` — depth 100, 201 nodes. The gap gets far worse the moment you differentiate: `(Sin() ** 100).derivative()` is a 7-node tree (a single chain-rule step reusing the closed-form rule $\mathcal{D}_x x^n = nx^{n-1}$), while differentiating the naive 100-deep product chain — one product-rule application per level, each one duplicating everything below it — produces a tree of *10,497* nodes for the exact same

mathematical function. Composing with `Power(n)` does not just happen to be correct; it is what keeps "differentiate x^{100}" a constant-size operation instead of a quadratic-size one.

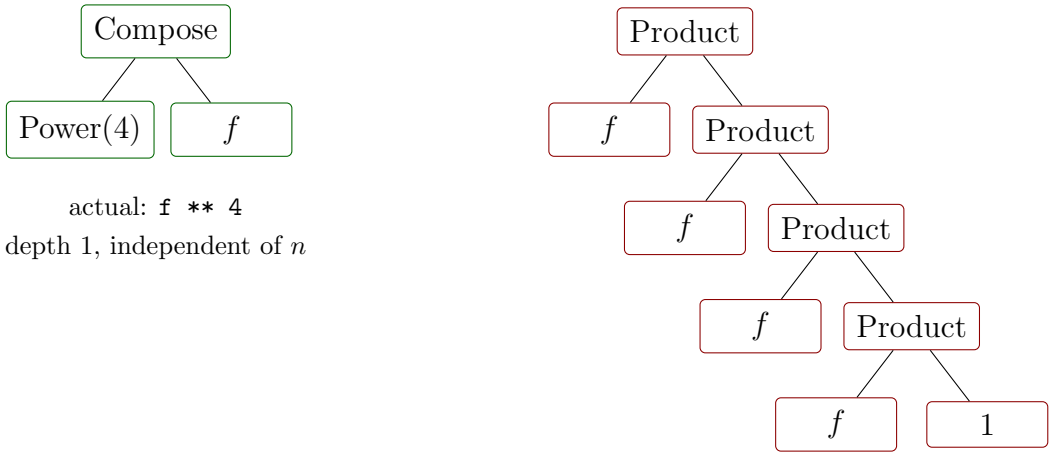


Figure 5.4: The two tree shapes for f^n, shown at $n = 4$ since a $n = 100$-deep chain cannot fit on a page. **Left:** the actual implementation, `Compose(Power(n), f)` — one `Compose` node no matter how large n is. **Right:** `naive_pow(f, n)` — a chain that adds one more `Product` node, and one more level of depth, for every unit increase in n, bottoming out in `Numeric(1)`, the multiplicative identity, when n finally reaches 0.

Figure 5.4 shows both shapes at a size small enough to draw. Imagine the right-hand comb extended 96 more levels to see what $n = 100$ actually looks like.

`naive_pow` has a correctness problem too, not just a size problem, but writing it recursively already fixes half of it. `naive_pow(f, 0)` now correctly returns `Numeric(1)` — writing the function recursively forces an explicit base case, and 0 is the only exponent that terminates the recursion, so you cannot avoid deciding what happens there. `naive_pow(f, -2)`, however, is a differ-

ent story: n counts down by 1 each call $(-2, -3, -4, \ldots)$, moving *away* from 0 forever, so the function never terminates — Python raises `RecursionError: maximum recursion depth exceeded` instead of returning a wrong answer. Recursion turned a silent bug into a loud one, but did not remove it: a base case only helps if every input actually reaches it, and nothing about writing `n - 1` guarantees that for negative n. `Sin() ** 0` and `Sin() ** -2` both come out correct with no special-casing at all, because `Power`'s `evaluate` rule just defers to Python's own `x ** n`, which already handles zero and negative exponents correctly. Composing with `Power(n)` does not just avoid the size blowup — it inherits correct behavior at $n \leq 0$ for free, precisely because that case was never special-cased in the first place.

This section only ever raises `Power` to *integer* exponents. That is not a choice specific to `fast_power` or `naive_pow` — `Power` itself refuses to be constructed with anything else, deliberately (Section 5.2.4).

You do not have to take any of the depth or node-count figures above on faith — `pattern_match.py`, another Intra assignment (Section 6.6), gives you the tools to check them yourself.

Exponentiation by squaring — computing g^n with $O(\log n)$ multiplications instead of $O(n)$ — is the standard algorithm for computing large powers efficiently, and it is a natural question to ask whether it would help here too. This turns out to be a genuinely instructive rabbit hole, but a time-consuming one, so it is presented here as optional reading rather than a required assignment: `homework/fast_power.py` and `tests/fast_power_test.py` exist in the repository if you want to implement it yourself, but neither is graded, and it is not covered in lecture.

A correct `fast_power` produces a tree of depth 8 for $n = 100$ — compare that with depth 100 for `naive_pow` and depth 1 for `Compose(Power(100), f)` directly. But exponentiation by squaring does not reduce the *total* size of the tree by nearly as much as its

depth would suggest: it still comes out to 199 nodes, barely fewer than `naive_pow`'s 201, and it evaluates no faster either. The culprit is sharing: squaring is naturally implemented as `Product(f, f)`, and without memoization, `evaluate` recurs into that shared `f` object twice, paying for it as if it were two separate subexpressions. Curious what the fix looks like? Chapter 16 (Section 16.9) works through one.

5.5.2 Composing with a Constant

What does it mean to compose *with a constant*? Consider `Compose(Numeric(3), Power(2))`. Reading the tree, you might guess it represents $3x^2$ — "3 combined with x^2" sounds like multiplication. It does not: `Numeric(3)` *is* the constant function $x \mapsto 3$ (Section 5.2), and Corollary 2.1.3 (Section 2.1.5, Chapter 2) already established that composing with a constant function collapses to a constant, in either order — never a product. Here, `Numeric(3)` $\circ$ `Power(2)` $= 3$, not $3x^2$.

Check this directly against `evaluate`: the `Compose` branch computes `evaluate(f, evaluate(g, x, env), env)`, and the `Numeric` branch returns c without ever inspecting the value it was handed. At $x = 2$, `Compose(Numeric(3), Power(2))` evaluates to 3, not $3 \cdot 2^2 = 12$. Composing with a constant discards whatever you composed it with.

This is not a curiosity — it is what makes the general chain-rule implementation `df(g) * dg` correct even when `df` itself turns out to be a constant. Take $(\mathrm{id} \circ \sin)(x) = \sin(x)$, i.e. `Compose(Identity(), Sin())`. Here f = `Identity()`, so `df` $= f.$`derivative()` $=$ `Numeric(1)`: the derivative of the outer function is the *constant* function 1, because `Identity`'s rate of change does not depend on where you evaluate it. The chain rule computes `df(g) * dg`, i.e. `Compose(Numeric(1), Sin())` $\cdot$ `Cos()`. By the

same reasoning as above, `Compose(Numeric(1), Sin())` evaluates to 1 at *every* x — not $1 \cdot \sin(x)$ — so the product correctly reduces to $\cos(x)$ everywhere, matching $\mathcal{D}_x \sin x = \cos x$ exactly. Had composing with a constant instead *multiplied* by it, the bug would hide itself in the most common case: an outer derivative of `Numeric(0)` (by far the most frequent constant a derivative reduces to) gives the same answer either way, since "0 times anything" and "discard and return 0" both vanish. The bug only surfaces for a *nonzero* constant derivative, exactly the case above: at $x = 0$, the correct value is $\cos(0) = 1$, while the "multiply" bug would compute $\sin(0) \cdot \cos(0) = 0$ instead.

5.5.3 Composing with Identity

`Identity()` is the function $x \mapsto x$: composing it with anything should do nothing — *semantically*, i.e. as a claim about the functions themselves, this is exactly Corollary 2.1.4 (Section 2.1.6, Chapter 2): $\mathrm{id} \circ g = g \circ \mathrm{id} = g$ for every g. What `evaluate` adds here is confirmation that the *code* agrees: the `Identity` branch of `evaluate` returns its input completely unchanged, on either side of `Compose`, for every concrete x.

"Semantically" is doing real work in that sentence.  `Compose(Identity(), g) == g` is `False` — and so is `Compose(g, Identity()) == g` — because `==` compares tree structure (Section 5.2), and `Compose(Identity(), g)` is a two-node tree while `g` may be a single leaf. The identity law holds for the *function* each side denotes, not for the *AST* that represents it: exactly the structural-versus-semantic distinction from Section 6.2, showing up here as an algebraic law rather than a testing strategy.

The last piece is associativity — $(f \circ g) \circ h = f \circ (g \circ h)$, proven directly from what $\circ$ means in Corollary 2.1.2 (Section 2.1.3, Chapter 2). That proof needs no `PointFree`-specific argument: it is a basic fact

about function composition in general, true for any three functions regardless of how they are represented. It is still worth checking that `Compose` and `evaluate` actually respect it, since it is exactly as easy to get an implementation of composition subtly wrong as it was to get `Product(f,f)`-based squaring subtly slow.

```
f, g, h = Cos(), Sin(), Power(2)
left  = Compose(Compose(f, g), h)
right = Compose(f, Compose(g, h))
left == right           # False -- different trees
left.evaluate(0.5)       # 0.9695514213944606
right.evaluate(0.5)      # 0.9695514213944606 -- same function
```

`Compose(Compose(f,g),h)` and `Compose(f,Compose(g,h))` are different trees — `==` says so — but they evaluate to the same number at every x, because both represent $\cos(\sin(x^2))$, just built with the parentheses in a different place.

Proposition 5.5.1 (*PointFree: Monoid Under Composition*)

> The set of `PointFree` objects, together with `Compose` and the identity element `Identity()`, forms a monoid (Definition 2.1.5) — another instance of Proposition 2.1.6 (Section 2.1.7, Chapter 2): `Compose` is associative, and `Compose(Identity(), g)` and `Compose(g, Identity())` both denote the same function as `g`, for every `g`.

5.6 Derivative by Pattern Matching

Your task is to implement a function `derivative(node: PointFree) -> PointFree` that takes an expression AST and returns the AST of its derivative. The structure is identical to `evaluate`: one `case` branch per node type, with leaf branches returning immediately and binary branches recurring

into their children. The difference is that instead of computing a number, each branch applies a differentiation rule and returns a new `PointFree` tree.

To see concretely what the function must produce, consider the expression from Section 5.3:

```
expr = Sum(Sin(), Product(Cos(), Sin()))
```

This represents $f(x) = \sin(x) + \cos(x) \cdot \sin(x)$. Applying the differentiation rules from Chapter 1:

$$
\begin{aligned}
f'(x) &= \mathcal{D}_x \sin(x) + \mathcal{D}_x(\cos(x) \cdot \sin(x)) && \text{(sum rule)} \\
&= \cos(x) + \mathcal{D}_x(\cos(x) \cdot \sin(x)) \\
&= \cos(x) + \big(\mathcal{D}_x \cos(x)\big) \cdot \sin(x) + \cos(x) \cdot \big(\mathcal{D}_x \sin(x)\big) && \text{(product rule)} \\
&= \cos(x) + (-\sin(x)) \cdot \sin(x) + \cos(x) \cdot \cos(x)
\end{aligned}
$$

The AST that `derivative(expr)` must return is therefore:

```
Sum(
  Cos(),
  Sum(
    Product(Product(Numeric(-1), Sin()), Sin()),
    Product(Cos(), Cos())
  )
)
```

Figure 5.5 draws this tree out in full.

Every node in this result was produced by one rule: the outermost `Sum` by the sum rule, the inner `Sum` and its two `Product` children by the product rule, and the leaf nodes `Cos()`, `Sin()`, and `Numeric(-1)` by the rules for sin, cos, and constants. There is a one-to-one correspondence between the four lines of the mathematical derivation above and the four recursive calls your implementation will make.

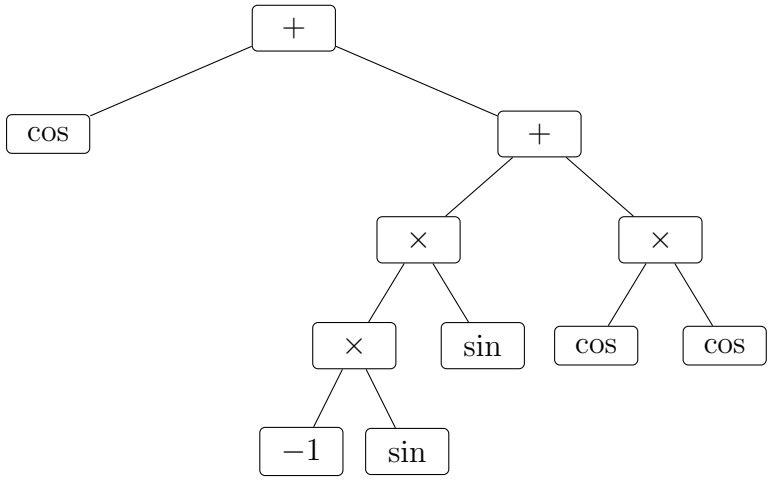


Figure 5.5: AST of $f'(x) = \cos(x) + (-\sin(x)) \cdot \sin(x) + \cos(x) \cdot \cos(x)$, the derivative of $f(x) = \sin(x) + \cos(x) \cdot \sin(x)$ from Figure 5.2. The tree is deeper and wider than the original, which is typical: derivatives are structurally more complex than the expressions they come from.

Implementing the rest of `derivative`, testing it, and defending it live is the subject of the group project — Chapter 6 picks up exactly here.

? Open Questions

- The ADT stops at `Sin`, `Cos`, and `Exp`. The Chapter 2 derivative table (Figure 2.1) also lists $\ln$, $\arctan$, $\arcsin$, and $\arccos$. Adding them as `Unary` leaves is mechanical for `evaluate`, `toLaTeX`, and `derivative` — their rules are already in that table. *Testing* them is the hard part. Every `PointFree` function you can build so far is total: `Sin`, `Cos`, `Exp`, `Power`, `Identity`, and every sum, product, and composition of them is defined for *every* real x. So the numerical correctness test used for the group project (Section 6.5.1) — evaluate `f.derivative()` and a secant approximation at sample points chosen anywhere on the line — always has a valid point. `Ln` breaks that: $\ln$ is defined only on $(0, \infty)$, so `Compose(Ln(), g)` is defined only where $g(x) > 0$, a set that depends on g and can be awkward to describe. `Arcsin` and `Arccos` are narrower still, defined only on $[-1, 1]$. (`Arctan` happens to be total, so it is the easy one.) A random sample point now often lands outside the domain, where `evaluate` raises a domain error — and because the secant approximation needs $f(x \pm h)$, a point can be inside the domain while a neighbour is not. How should the test harness handle this? Compute each expression's domain and sample only inside it? Catch domain errors and skip those points — and if so, how many surviving points make the test meaningful rather than vacuous? Or restrict the random generator so it only produces total expressions? This is the `Quotient` zero-denominator problem (Section 6.4.3) again, enlarged from isolated bad points to whole excluded intervals.

Chapter 6

Group Project: PointFree Derivative

Chapter Objectives

- As your group project (Section 6.1), implement `derivative` for every node type in the ADT, design and defend a test suite for it, and defend the whole in a live oral exam.
- Explain why structural AST equality is too strict a correctness test for `derivative`, and design a numerical testing strategy instead.
- Use `print_ast`, `display` / `toLaTeX`, and `simplify` to diagnose why a computed derivative disagrees with an expected one.
- Explain what `gen` generates, and the difference between syntactic and semantic uniformity.
- Implement `pattern_match.py` and `checks.py` individually: reusable tools for counting, measuring, and rewriting trees, and for numerically comparing functions.

Chapter 5 built the `PointFree` ADT and previewed what `derivative` must produce, node type by node type (Section 5.6). The group project (Section 6.1) is to finish `derivative` and defend it. This chapter gives you what that takes: why you cannot test a symbolic-differentiation implementation by comparing two ASTs for

equality, a numerical strategy that works instead, the debugging tools for when a test disagrees with what you expected, and `gen` for generating test inputs. Two smaller, individually-graded Intra assignments (Section 6.6) share the same machinery.

6.1 What is the Project?

This is the one graded assignment in the course that you do as a group and defend in person. Working in a group of two or three, you complete the symbolic differentiation function `derivative`, write a test suite for it, and defend both in a live oral exam. Your goal is two-fold: implement it correctly, and convince yourselves — and your instructor — that it is correct.

Deliverables.

- `homework/derivative.py` — every `# CHALLENGE` case completed (see *What to implement* below).
- `tests/derivative_test.py` — your own test suite (Section 6.5.1); no test file is provided.
- A live oral defense (Section 6.5.3): each group member is questioned individually.

6.1.1 Forming a group

Students may divide themselves into groups of two or three persons. It is your responsibility to find a group or to form one. If you have trouble, talk to the class representative. Once your group is settled, tell the instructor who is in it.

6.1.2 What to implement

`homework/derivative.py` is provided as a skeleton. The leaf cases and a few of the binary cases are written for you; every case you must supply is marked with a `# CHALLENGE` comment. Complete all of them, so that `derivative(node)` returns a `PointFree` tree representing the derivative of `node`, for *every* node type in the ADT.

The skeleton already handles `Constant` (derivative `0`), `Identity` (derivative `1`), `Sum` (Summation Rule), and `Cos`. The `# CHALLENGE` cases you fill in, and the rule for each — none of the mathematics is new:

- `Difference`: the Difference Rule (Corollary 1.2.10).
- `Product` whose left or right factor is a constant (a `Numeric` or a `Symbol`): the Scalar Rule (Corollary 1.2.9).
- any other `Product`: the Product Rule (Theorem 1.2.8).
- `Quotient`: the Quotient Rule (Theorem 1.2.14).
- `Power(n)`: the Power Rule (Theorem 1.2.3).
- `Compose`: the Chain Rule (Theorem 2.2.1, Chapter 2).
- `Sin` and `Exp`: read straight off the derivative table (Figure 2.1).

`case` branches are tried in order, so the two constant-factor `Product` branches must be written *before* the general `Product(f, g)` branch, and must apply the Scalar Rule. Applying the full Product Rule to $f(x) = 3\sin(x)$ is not *wrong*, but it returns `Sum(Product(Numeric(0), Sin()), Product(Numeric(3), Cos()))` — the derivative of the constant 3 is `Numeric(0)`, and that $0 \cdot \sin(x)$ term is dead weight the Scalar Rule avoids, yielding just

`Product(Numeric(3), Cos())`. A messier tree makes every later structural comparison harder.

Section 5.6 works one non-trivial expression from tree to derivative tree, one `case` per line of the hand derivation; use it as the model for the rest.

The remaining sections of this chapter are the toolkit for the other half of the job — convincing yourself the implementation is right: why you cannot test it by comparing ASTs (Section 6.2), the debugging aids (Section 6.3), random test generation (Section 6.4), and how to design the suite you submit and what to expect at the defense (Section 6.5).

6.2 Testing the Derivative

How can you know whether the code for computing the derivative is correct?

An important part of the development of your code must entail implementing a way to test your code. A significant hurdle is that the computed AST of the derivative inevitably is far more complex than a human would derive if computing the derivative on paper.

One dubious solution might be to attempt to write a function which will compare two ASTs for semantic equality. For example: compute the derivative of x^3x^7 by the product rule and compare this to the derivative of $(x^5)^2$ by the chain rule, and then compare the results. It is possible to make this equality checker work for simple cases, but it is an exceedingly difficult problem to make it work for anything except very simple cases.

Before abandoning the pursuit entirely, it is potentially worthwhile to look at the programmatic results of $\mathcal{D}_x x^3x^7$ vs $\mathcal{D}_x(x^5)^2$, and ponder how you would determine these expressions to be equal programmatically.

In the advanced data structures course you implemented

`equiv_exprs` for Boolean expressions: two expressions are equivalent if they agree on every possible assignment of their variables. That approach worked because Boolean variables take only two values, making exhaustive search feasible. For real-valued functions the domain is infinite, so exhaustive search is impossible. How can you adapt the idea?

6.3 Debugging Tools

When your derivative implementation produces an unexpected result, comparing two `PointFree` trees by eye using `__repr__` can be difficult: the nested constructor notation grows long quickly and the tree structure is hard to parse visually. The module `common/astprint.py` provides a `print_ast` function that renders any `PointFree` tree as an indented text diagram:

```
from common.astprint import print_ast

expr = Sum(Sin(), Product(Cos(), Sin()))
print_ast(expr.derivative())
```

For the expression above the terminal output is:

```
Sum
|-- Cos
\-- Sum
    |-- Product
    |   |-- Product
    |   |   |-- Numeric(-1)
    |   |   \-- Sin
    |   \-- Sin
    \-- Product
        |-- Cos
        \-- Cos
```

Each node is printed on its own line; the branch connectors show the parent–child relationships at every level. Compare the output against the expected AST in Chapter 5, Section 5.6 to locate exactly which branch of your `derivative` function is producing the wrong result.

A second visual tool is `toLaTeX` together with `display`. `toLaTeX` follows the same pattern as `toPython` but produces a LaTeX string instead of a Python expression. LaTeX is a markup language — like HTML or Markdown — used by mathematicians to typeset formulas. You do not need to understand LaTeX syntax; the function is provided ready-made and you will use it only as a black box.

Suppose we want to typeset and display an expression generated within your Python code. A call to `display` will typeset the expression and display in your web browser somewhat similar to Equation (6.1).

```
expr = Sum(Exp(),Quotient(Sum(Power(2),Sin()),
                          Difference(Power(3), Cos())))
display(expr)
```

$$e^x + \frac{x^2 + \sin x}{x^3 - \cos x} \tag{6.1}$$

The companion function `display` calls `toLaTeX`, wraps the result in the minimal boilerplate needed to render it, and opens the rendered formula in a web browser. The `display` function thus gives an immediate visual check that the AST represents the expression you intended.

A third black-box tool: `expr.simplify()` rewrites a `PointFree` tree into an equivalent but more compact, canonically-ordered form — folding constant arithmetic, cancelling terms like `f - f`, sorting commutative operands into a fixed order, and so on. You are not expected to know how it works, only that it exists and is safe to call on any `PointFree` value.

It is particularly useful for debugging `derivative`: two expressions that compute the same function can still produce structurally different ASTs (Section 6.2), so a raw `==` comparison against an expected tree can fail even when your derivative is correct. Simplifying both sides first often resolves that:

```
expr.derivative().simplify() == expected.simplify()
```

If the two still disagree after simplifying, that is a much stronger signal that `derivative` itself, not just superficial tree shape, is at fault.

6.4 Randomly Generating Expressions

The function `gen(n)` in `gen.py` generates a random `PointFree` AST of size n. It is provided ready-made for you to use in your tests; you do not need to modify it to complete the homework. This section explains how it works and what its limitations are.

6.4.1 How `gen` Works

`gen` is defined recursively on the size parameter:

- If $n = 1$, return a randomly chosen *leaf* from a fixed list: `Numeric(0)`, `Numeric(1)`, `Numeric(-1)`, `Identity()`, `Sin()`, `Cos()`, `Exp()`, `Power(2)`, `Power(3)`, `Power(4)`.
- If $n > 1$, choose a random pivot k with $1 \leq k < n$, recursively generate a left subtree of size k and a right subtree of size $n - k$, then combine them with a randomly chosen binary constructor: `Sum`, `Product`, `Difference`, `Quotient`, or `Compose`.

```
def gen(size: int) -> PointFree:
    if size == 1:
        return random.choice(leaves)
    pivot = random.randint(1, size - 1)
    left  = gen(pivot)
    right = gen(size - pivot)
    return random.choice([lambda: Sum(left, right),
                          lambda: Product(left, right),
                          lambda: Difference(left, right),
                          lambda: Quotient(left, right),
                          lambda: Compose(left, right)])()
```

The pivot splits the budget n into two positive parts; the left and right subtrees then recur independently. Every internal node is one of the five binary constructors chosen with equal probability.

`gen` deliberately does *not* use `match` / `case`, even though this entire chapter has used pattern matching for everything else. The reason is not stylistic: `match` / `case` dispatches on the *shape of an existing value* — `evaluate`, `derivative`, and `toPython` all take a `PointFree` tree and ask "which class is this node?" `gen` never receives a `PointFree` at all; its only input is an `int`. There is no

existing shape to take apart — the job is to *construct* something that does not exist yet, and the two decision points reflect that: `if size == 1` is a plain condition on an integer, not a class dispatch, and `random.choice([...])` is *selecting* which constructor to call, not matching a value against several possible shapes. Pattern matching earns its keep when you are tearing an ADT value down; a function whose entire job is building new instances has nothing of that shape to exploit, no matter how thoroughly the rest of the codebase leans on `match` / `case`.

6.4.2 Syntax Uniformity vs. Semantic Uniformity

What does it mean for a random generator to be *uniform*? There are two quite different answers.

Uniform over the syntax space means that every structurally distinct AST of size n has an equal chance of being produced. `gen` is approximately uniform in this sense: the pivot is uniform in $\{1, \ldots, n-1\}$, the constructor is uniform over five choices, and the leaf is uniform over ten choices. The number of distinct binary trees of a given size is given by the *Catalan numbers*

$$C_n = \frac{1}{n+1}\binom{2n}{n},$$

a sequence $(1, 2, 5, 14, 42, \ldots)$ that grows exponentially; even moderate sizes produce an astronomical variety of structures.

Uniform over the semantic space would mean that every mathematically distinct function has an equal chance of being produced. This is a far harder goal, and `gen` does not attempt it. Many syntactically different ASTs compute the same function: for example, `Sum(Identity(), Identity())` and

`Product(Numeric(2), Identity())` both represent $2x$. Avoiding such duplicates would require first deciding whether two arbitrary expressions are semantically equal — precisely the hard problem raised in Section 6.2. For testing purposes, syntax uniformity is sufficient: the goal is to explore a wide variety of tree shapes, not to enumerate distinct functions.

6.4.3 Division by Zero

Whenever the random constructor is `Quotient`, the right subtree becomes a denominator. At generation time, however, x is not yet known; `gen` has no way to evaluate the denominator to check whether it is zero. A denominator like `Difference(Identity(), Identity())` represents $x - x$, which is identically zero. A denominator like `Sin()` is zero at $x = 0, \pi, 2\pi, \ldots$

Whether this is a problem depends on what you are testing:

- To test that your code *handles* division-by-zero gracefully (raises the right exception, does not crash silently), you *want* to generate such cases.
- To run large-scale numerical tests without spurious failures, you can simply catch `ZeroDivisionError` and skip the offending $(expression, x)$ pair.

There is no single correct policy. The test file `tests/toPython_test.py` takes the second approach: it catches `ZeroDivisionError`, `OverflowError`, and `ValueError` and skips any pair that falls outside the domain, so that random tests run cleanly at scale.

6.5 Designing Your Test Suite and the Defense

Section 6.1 stated the deliverables; this section is about the second one — the test suite — and about the oral defense.

6.5.1 Where your tests go, and what the grader does

No test file is provided for `derivative`. Unlike this chapter's Intra assignments (Section 6.6), where the grading tests are handed to you, designing and writing the test suite is itself part of this assignment. Put your tests in `tests/derivative_test.py`, following the same `unittest` structure as the test files already in `tests/`, and run them with `pytest` from `src/`. A hidden test suite also runs on submission and counts toward the grade; it reports only pass or fail, never which cases failed, so your own suite is what tells you where you stand.

Design a test suite that *you* are confident in. A good suite will include:

- Structural tests: for simple inputs (`Numeric`, `Identity`, `Sin`, `Power(n)`, ...) check that `derivative` returns the exact expected AST.
- Numerical tests: for more complex expressions, evaluate `f.derivative()` and a numerical approximation of f' at several sample points, and assert they agree within a small tolerance. This sidesteps the AST-equality problem raised in Section 6.2. You already have exactly this numerical approximator: Intra-1's `secant.py` implements `approximate_derivative`, built on the same secant-line idea Section 1.1.1 introduced. Reuse it directly rather than writing a second approximator from scratch:

```
from homework.secant import approximate_derivative

x = 1.7
numeric = approximate_derivative(f.evaluate, x, x + 1.0, 1e-6)
symbolic = f.derivative().evaluate(x)
self.assertAlmostEqual(numeric, symbolic, delta=1e-3)

```

Note what `approximate_derivative` takes: a plain `Callable[[float], float]` (here, the bound method `f.evaluate`), not a `PointFree` expression. That makes it a different kind of tool from `checks.py`'s `functions_agree` (Section 6.6), which compares two `PointFree` trees to each other and cannot take a plain numerical procedure as either argument. Use `approximate_derivative` here, where one side of the comparison (the numerical approximation) is not itself a `PointFree` expression; use `functions_agree` below, where both sides of an identity are.

- Edge cases: the product rule, the chain rule via `Compose`, and expressions involving `Symbol` nodes.
- Algebraic identity tests: verify that your derivative satisfies mathematical identities across many functions f, g, and h. Good identities to try:
  - Commutativity of addition and multiplication:

    $$\text{deriv}(f + g) \stackrel{?}{=} \text{deriv}(g + f)\,,$$

    and

    $$\text{deriv}(f \cdot g) \stackrel{?}{=} \text{deriv}(g \cdot f)\,.$$

  - Scaling:

    $$\text{deriv}(f + f) \stackrel{?}{=} \text{deriv}(2 \cdot f)\,.$$

– Associativity:

$$\text{deriv}(f+(g+h)) \stackrel{?}{=} \text{deriv}((f+g)+h)\,,$$

and similarly for $\cdot$ and $\circ$.

6.5.2 Structural versus Numerical Equality

These two notions look similar but behave very differently in practice. *Structural equality* compares ASTs node by node after simplification — using the `.simplify()` tool from Section 6.3 — while *numerical equality* evaluates both sides at several sample points and checks they agree within a tolerance. The commutativity identities above can usually be verified *structurally*: a good simplifier can sort operands into a canonical order so that deriv$(f+g)$ and deriv$(g+f)$ produce identical trees. The associativity identities almost never can: applying the sum rule to $f+(g+h)$ versus $(f+g)+h$ yields ASTs whose parenthesisation differs, and the rewriting rules that would normalize it interact with the commutative sorter to create infinite loops. The practical lesson: **if a structural test fails, check numerically before assuming the derivative is wrong.** Agreement in value with disagreement in structure means the *simplifier* has a limitation; a mismatch in value means the derivative itself is buggy.

6.5.3 Submission and defense

Submit `derivative.py` and `derivative_test.py` to Forge/Intra following the same procedure as previous courses.

Each group then presents and defends its solution in a live session with the instructor. *Every member is questioned individually* and must be able to explain and defend any part of the submission — not only the code they personally wrote. Be ready to:

- walk through your test suite and say why you are confident it is adequate;
- work, on the spot, a derivative your code has not seen;
- justify any single `case` branch — including why the branches are in the order they are. In particular, be able to explain why the constant-factor `Product` branches come before the general one and use the Scalar Rule: what tree does the full Product Rule produce for $3\sin(x)$, and why is the Scalar Rule's result preferable?

Instructions for scheduling the defense session are provided separately (a sign-up sheet is circulated near the submission deadline).

6.6 Homework

Two Intra assignments accompany this chapter: `pattern_match.py` and `checks.py`. Both are submitted to Forge/Intra and graded automatically, the same way as in your previous courses: the tests you are graded against are given to you, so there are no surprises. Both are graded individually, but you are encouraged to work through them with your group alongside `derivative.py` — they are easier problems over the same ADT, and `checks.py` in particular produces tools you will want for your own `derivative` test suite (Section 6.5.1).

6.6.1 `pattern_match.py`: Counting, Measuring, and Rewriting Trees

Implement three functions in `homework/pattern_match.py`, each a recursive walk over a `PointFree` tree using the same `match` / `case`

skeleton as `evaluate`. `count_nodes` and `max_depth` are the tools Chapter 5, Section 5.5.1 used to justify its depth and node-count figures; `distribute` rewrites an expression using the distributive law.

- `count_nodes(node)`: the total number of nodes in the `PointFree` tree rooted at `node` (a leaf counts as 1; a binary node counts itself plus both children, recursively).
- `max_depth(node)`: the depth of that tree (a leaf has depth 0; a binary node has depth 1 plus the *deeper* of its two children — not the sum of both).
- `distribute(node)`: rewrite `node` into an equivalent tree in which no `Product` has a `Sum` as either child, by applying the distributive law

  $$f \cdot (g + h) = f \cdot g + f \cdot h\,, \qquad (f + g) \cdot h = f \cdot h + g \cdot h\,,$$

  wherever it applies — including to a sum that is only nested *inside* another sum, or that only appears after one of its own children has itself been rewritten. `Difference` is left alone; only `Sum` is in scope for this exercise.

The tests you are graded against are in `tests/pattern_match_test.py`. Unlike `gen` (Section 6.4), this *is* a natural `match` / `case` problem: you are tearing down an existing tree by shape, exactly the pattern `evaluate` and `derivative` already use.

You do not need a separate `case` arm for each of `Sum`, `Difference`, `Product`, `Quotient`, and `Compose`. All five inherit `__match_args__ = ('f', 'g')` from the same abstract class, `BinaryOp` (Chapter 5, Section 5.2) — and `match` / `case` patterns can name an *abstract* class just as well as a concrete one.

`case BinaryOp(f, g):` matches an instance of any of the five, binds its two children, and needs writing only once.

`Compose` and `distribute`: a puzzle. `Compose` might plausibly distribute over `Sum` on either side:

$$(f_1 + f_2) \circ g \stackrel{?}{=} (f_1 \circ g) + (f_2 \circ g)\,, \qquad f \circ (g_1 + g_2) \stackrel{?}{=} (f \circ g_1) + (f \circ g_2)\,.$$

Exactly one of these is a valid rewrite for *every* choice of f, f_1, f_2, g, g_1, g_2 — the other is not. Work out which is which, by hand or by testing a candidate rewrite numerically against `evaluate`, before you write any code. Implement only the valid direction; leave the invalid one unrewritten. (Once you have worked it out — or if you get stuck — Chapter 2, Section 2.1.8 works through this exact question from the calculus side, with a proof and a counterexample.)

`distribute` must reach a genuine *fixed point*: a sum nested inside another sum, or one that only becomes visible after one of its own children has already been rewritten, must be fully expanded, not left half done. The repetition this requires is handled for you by `find_fixed_point`, a small generic utility in `common/fixed.py`: it calls a function repeatedly on its own output until the result stops changing, then returns it — the same strategy `simplify` (Section 6.3) uses internally, both built on the identical helper. `distribute(node)` itself is just `find_fixed_point(recur, node)`, where `recur` is the nested function defined just above it in the same file.

If you go looking for the loop and cannot find one, that is by design, not a sign you are missing something. Your job is `recur`: describe what a *single* pass does at one node — that is where every `# CHALLENGE` arm lives. The looping and the "are we done yet" check are entirely `find_fixed_point`'s job, not `recur`'s and not yours. You are welcome to read `common/fixed.py` if you are curious how the

repetition and termination check work, but you do not need to understand, call, or modify it yourself to finish this exercise — `distribute` already calls it for you, exactly once, with your `recur` as the function to repeat.

6.6.2 `checks`: Building Blocks for Testing `derivative`

Section 6.5.1 asks you to design your own test suite for `derivative`, but does not hand you the tools to build it. `homework/checks.py` closes that gap: implement three small, reusable functions that you will then use yourself when testing `derivative`. The tests you are graded against are in `tests/checks_test.py`.

- `inf_norm(f, x_min, x_max, x_step)`: the *infinity norm* (sup-norm) of f on $[x\_min, x\_max]$ — the largest value of $|f(x)|$ among the points $x\_min, x\_min + x\_step, x\_min + 2\,x\_step, \ldots$ up to x_max. Any sample point where f is undefined (`ZeroDivisionError`, `OverflowError`, `ValueError`) is skipped rather than treated as an error.
- `distance(f, g, x_min, x_max, x_step)`: the sup-norm distance between `f` and `g` on $[x\_min, x\_max]$ — the infinity norm of their difference. Since `PointFree` overloads Python's arithmetic operators (Chapter 5, Section 5.2), `f - g` is itself a valid `PointFree` expression, so `distance` is just `inf_norm(f - g, ...)`: it inherits the domain-error handling of `inf_norm` for free, with no exception handling of its own to write.
- `functions_agree(f, g, x_min, x_max, x_step, tol=1e-4)`: returns `True` exactly when `distance(f, g, ...) < tol`.

This is the "numerical test" technique Section 6.5.1 recommends, packaged as a single function call instead of a comparison loop you would otherwise rewrite for every test case.

`inf_norm` is the only one of the three that needs its own exception handling; `distance` and `functions_agree` are each one-line wrappers that reuse it.

`distance` measures the sup-norm: the single worst disagreement between `f` and `g` anywhere on the interval. Chapter 9 introduces a second, genuinely different notion of distance — the L^2 (root-mean-square) distance, `inner_product.py`'s `distance` — which *aggregates* disagreement across the whole interval via an integral, rather than reporting only the worst point. That version has to wait: it is built on `simpsons_sum` from `riemann_sum.py` (Chapter 8), which you have not seen yet.

? Open Questions

- Chapter 5, Section 5.3 claimed that OOP dispatch (one method per class) and `match` / `case` (one function with a branch per class) are "equivalent in power." Equivalent in what they can compute, yes — but not in what they make easy. Suppose later you want to add a new node type, `Tan()`. How many places in the code must change under each style? Now suppose instead you want to add a new operation, say `simplify`, alongside `evaluate` and `toPython`. How many places change under each style now? (This tension has a name in programming-language theory: the *Expression Problem*.)

- Section 6.2 raised, but did not solve, the problem of deciding whether two `PointFree` expressions are *semantically* equal — computing the same function even though their ASTs differ. Boolean expressions admit exhaustive testing because there are only finitely many input assignments; real-valued expressions do not. What partial guarantees could you build instead? Is agreement at many random sample points convincing? How many, and how would you choose them?

- In an Algebraic Data Type as understood in this chapter, exhaustive pattern matching means every variant is covered by some `case` branch. In Haskell, the compiler checks this for you and refuses to compile a non-exhaustive `case`. Python's `match` / `case` performs no such check — an unhandled node type simply falls through silently unless you add a catch-all branch that raises an error. How would you convince yourself, or a grader, that your `derivative` function really does handle every node type in the ADT?

Chapter 7

Definition of Derivative

Chapter Objectives

- Recall the ε-δ definition of limit and the limit laws from Algebra-1.
- Apply the Limit Cancellation Recipe to resolve $\frac{0}{0}$ indeterminate forms by factoring (algebraic case) or by trigonometric identity (transcendental case).
- Define the derivative rigorously as a limit of a difference quotient and connect it to the reduction rules already studied.
- Identify non-differentiable functions: corners, cusps, and discontinuities.
- Re-derive the product rule, chain rule, and quotient rule from the limit definition.
- Derive derivatives of the transcendental functions ($\sin$, $\cos$, e^x, $\ln$) from first principles.

This entire chapter is optional reading, not covered in lecture. Every differentiation rule you have used since Chapter 1 — the reduction rules, the Chain Rule, the Quotient Rule, and the derivatives of sin, cos, e^x, ln, and the inverse trigonometric functions — has already been stated, and in most cases justified, without needing anything in this

chapter. What this chapter adds is *rigor*: a precise ε-δ definition of limit and of the derivative, and first-principles proofs, from that definition, of rules you have already been using. Read it if you want to see *why* those rules are true all the way down to the definition of limit, rather than just *that* they are true.

In this chapter we make precise what we have been doing since Chapter 1. Students have already encountered limits, continuity, and a numerical implementation of the derivative in the Algebra-1 course; the present chapter recalls that material briefly and uses it to give a rigorous definition of differentiability.

7.1 Recall: Limits and Continuity

The following definitions and results were developed in Algebra-1. We recall them here without proof, since they underpin every argument in this chapter.

Definition 7.1.1 (*Limit*)

Let f be defined on an open interval containing a, except possibly at a itself. The *limit* of $f(x)$ as x approaches a is L, written $\lim_{x\to a} f(x) = L$, if for every $\varepsilon > 0$ there exists $\delta > 0$ such that

$$0 < |x - a| < \delta \implies |f(x) - L| < \varepsilon.$$

Figure 7.1 illustrates this definition geometrically.

Proposition 7.1.2 (*Limit Laws*)

If $\lim_{x\to a} f(x) = L$ and $\lim_{x\to a} g(x) = M$ and $c \in \mathbb{R}$, then:

1. $\lim_{x\to a}[f(x) + g(x)] = L + M$

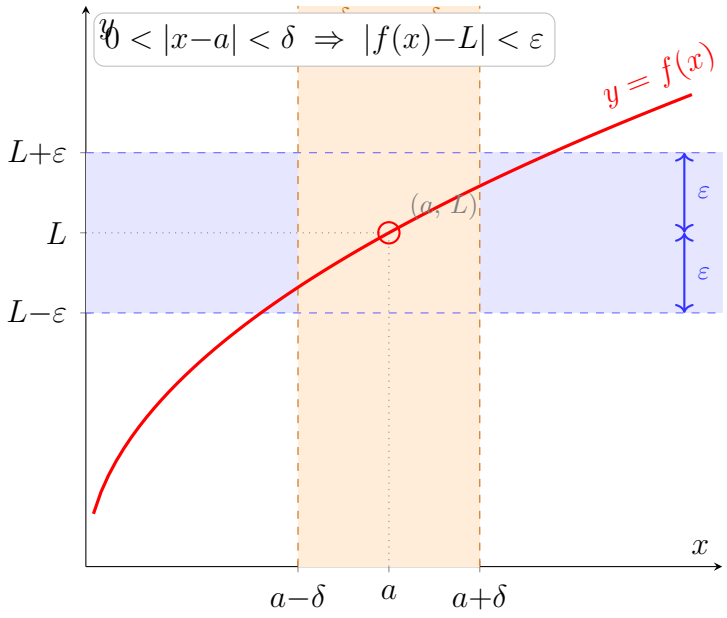


Figure 7.1: The ε-δ definition of limit. The blue band marks $|f(x) - L| < \varepsilon$ (all y-values within ε of L). The orange band marks $|x - a| < \delta$ (all x-values within δ of a). For this choice of ε, the value δ is chosen small enough that the graph of f stays entirely inside the blue band whenever x is in the orange band (excluding $x = a$ itself, marked by the open circle).

2. $\lim_{x \to a} c\, f(x) = c\, L$

3. $\lim_{x \to a} f(x) g(x) = L \cdot M$

4. $\lim_{x \to a} \dfrac{f(x)}{g(x)} = \dfrac{L}{M}$ (provided $M \neq 0$)

Definition 7.1.3 (*Continuity*)

f is *continuous at* a if $\lim_{x \to a} f(x) = f(a)$; in particular the limit exists and equals the function value. f is *continuous on an interval* if it is continuous at every point of that interval.

Proposition 7.1.4 (*Substitution Rule*)

If $\lim_{x\to a} g(x) = L$ and f is continuous at L, then

$$\lim_{x\to a} f(g(x)) = f\left(\lim_{x\to a} g(x)\right) = f(L).$$

The Substitution Rule is the result we use when we write $\lim_{h\to 0} f(x+h) = f(x)$ in limit proofs: continuity of f at x is exactly what justifies moving the limit inside f.

The limit laws handle sums, products, and quotients of functions whose limits are already known — but some limits cannot be decomposed that way. The Squeeze Theorem provides an alternative: trap the unknown function between two functions that both converge to the same value, and conclude that it converges there too. It is used later in this chapter to establish $\lim_{h\to 0} \frac{\sin h}{h} = 1$, the key step in proving that $\mathcal{D}_x \sin x = \cos x$ from first principles.

Proposition 7.1.5 (*Equal Near a Implies Equal Limits*)

Let f and g be defined near a (except possibly at a itself). If $f(x) = g(x)$ for all $x \neq a$, then

$$\lim_{x\to a} f(x) = \lim_{x\to a} g(x),$$

in the sense that if either limit exists, both exist and are equal.

Proof. The ε-δ definition requires $0 < |x - a| < \delta$, which excludes $x = a$ entirely. Any ε-δ argument for f therefore reads word-for-word for g, since f and g agree at every point the definition inspects.

This proposition is what licenses every algebraic cancellation in a dif-

ference quotient: canceling a common factor creates a new function that agrees with the original *except at the limit point*, and Proposition 7.1.5 guarantees the limits are the same.

Theorem 7.1.6 (*Squeeze Theorem*)

Let g, f, and h be functions defined near a (except possibly at a itself). If $g(x) \le f(x) \le h(x)$ for all x near a, and

$$\lim_{x\to a} g(x) = \lim_{x\to a} h(x) = L,$$

then $\lim_{x\to a} f(x) = L$.

Example 7.1.7 (*Squeeze Theorem*)

This example shows how ot use the Squeeze Theorem to verify a limit of a particular function at an undefined point.

Show that $\lim_{x\to 0} \left| x \sin\left(\frac{1}{x}\right) \right| = 0$, even though $f(x) = |x \sin(1/x)|$ is not defined at $x = 0$: the formula requires evaluating $\sin(1/x)$, and $1/x$ is undefined at $x = 0$. Note that f is never negative.

Verification. Since $|\sin(1/x)| \le 1$ for every $x \neq 0$,

$$0 \;\le\; f(x) \;=\; |x| \cdot |\sin(1/x)| \;\le\; |x| \qquad \text{for all } x \neq 0.$$

Let $g(x) = 0$ and $h(x) = |x|$ (Figure 7.2); both are defined everywhere, including at $x = 0$, and

$$\lim_{x\to 0} g(x) = \lim_{x\to 0} h(x) = 0.$$

By the Squeeze Theorem (Theorem 7.1.6), $\lim_{x\to 0} f(x) = 0$ — even though $f(0)$ itself does not exist.

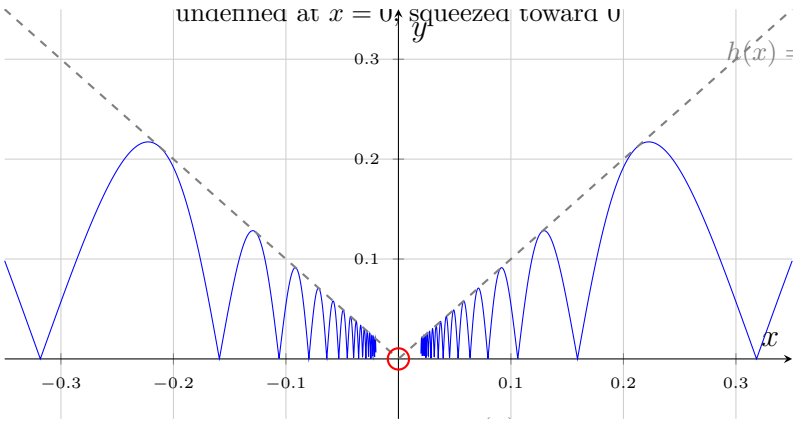


Figure 7.2: $f(x) = |x \sin(1/x)|$, trapped between $g(x) = 0$ and $h(x) = |x|$ (dashed). Since f is never negative, it is squeezed from below by the x-axis itself. The open circle marks that f is undefined at $x = 0$; both bounds converge there anyway, forcing $\lim_{x \to 0} f(x) = 0$.

This is exactly what the Squeeze Theorem is for: pinning down a limit at a point where direct substitution is impossible, by trapping f between two simpler functions whose limits are already known. The same bound $|x \sin(1/x)| \leq |x|$ reappears later in this chapter (Example 7.3.3), where it is used to show that $x \sin(1/x)$ — with $f(0) := 0$ — is continuous at 0, although still not differentiable there.

Example 7.1.8 (*A Limit at a Point of Undefinedness*)

Compute $\displaystyle\lim_{x \to 1} \frac{x^2 - 1}{x - 1}$.

The function $f(x) = \dfrac{x^2 - 1}{x - 1}$ is not defined at $x = 1$: the denominator is zero there, so any attempt to evaluate it directly raises a division-by-zero error. We can verify this in Python:

```python
def f(x):
    return (x**2 - 1) / (x - 1)

for x in [0.9, 0.99, 0.999, 1.001, 1.01, 1.1]:
    print(f"f({x}) = {f(x)}")

f(1)   # ZeroDivisionError: float division by zero
```

Running the code above, the values approach 2 from both sides:

$$\begin{aligned} f(0.9) &= 1.9 \\ f(0.99) &= 1.99 \\ f(0.999) &= 1.999 \\ f(1.001) &= 2.001 \\ &\vdots \end{aligned}$$

The numerical evidence suggests the limit is 2. To prove it, factor the numerator:

$$\frac{x^2 - 1}{x - 1} = \frac{(x-1)(x+1)}{x-1} = x + 1 \qquad (x \neq 1).$$

The cancellation holds for all $x \neq 1$, so $\dfrac{x^2-1}{x-1}$ and $x+1$ agree everywhere except at $x = 1$. By Proposition 7.1.5 their limits as $x \to 1$ are equal, and the Substitution Rule gives:

$$\lim_{x \to 1} \frac{x^2 - 1}{x - 1} = \lim_{x \to 1} (x + 1) = 1 + 1 = 2.$$

This same $\frac{0}{0}$ situation arises in the definition of the derivative: the difference quotient $\dfrac{f(a+h) - f(a)}{h}$ is undefined at $h = 0$, and finding the derivative means computing the limit as $h \to 0$.

Pattern 7.1.9 (*The Limit Cancellation Recipe*)

When evaluating $\lim_{x\to a} \dfrac{N(x)}{D(x)}$ (or $\lim_{h\to 0} \frac{N(h)}{D(h)}$) and direct substitution gives $\frac{0}{0}$, the following steps resolve the indeterminate form.

1. **Identify** — confirm that direct substitution gives $\frac{0}{0}$: both numerator and denominator vanish at the limit point.
2. **Transform** — use algebra or a known identity to make the vanishing denominator factor appear explicitly in the numerator:
   - *Algebraic*: factor the numerator. The Factor Theorem guarantees a factor $(x-a)$ exists whenever $N(a)=0$ and N is a polynomial.
   - *Transcendental*: apply a trigonometric or other identity to expose a recognisable limit, such as $\frac{\sin h}{h}$.
3. **Cancel** — remove the common factor from numerator and denominator, producing a new function that agrees with the original for all $x \neq a$ (resp. $h \neq 0$). By Proposition 7.1.5, the two functions have the same limit.
4. **Evaluate** — apply the Substitution Rule to the simplified expression if it is now continuous at the limit point, or quote a previously established limit.

7.2 Differentiability

Definition 7.2.1 (*Derivative at a Point*)

Let f be defined on an open interval containing a. The *derivative of f at a* is

$$f'(a) = \lim_{h \to 0} \frac{f(a+h) - f(a)}{h},$$

provided this limit exists. If it does, f is *differentiable at a*. f is *differentiable on an interval* if it is differentiable at every point of that interval.

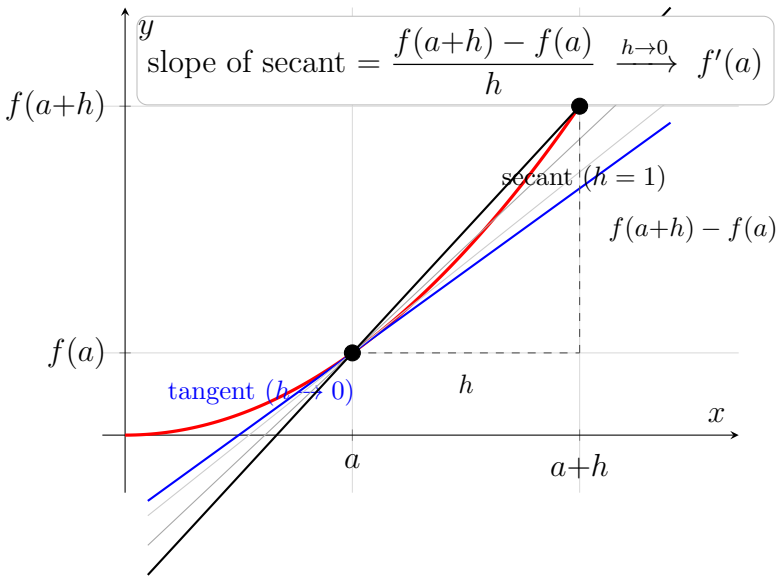


Figure 7.3: The derivative $f'(a)$ as the limit of secant slopes. For $f(x) = x^2$ at $a = 1$, the secant through $(a, f(a))$ and $(a+h, f(a+h))$ has slope $\frac{f(a+h)-f(a)}{h}$. As h decreases (dark to light), the secant rotates onto the tangent line (blue), whose slope is $f'(1) = 2$.

The definition coincides with the numerical approximation $\dfrac{f(a+h) - f(a)}{h}$ for small h that students computed in Algebra-1; here

we require the limit to exist exactly. Figure 7.3 shows geometrically how secant lines converge to the tangent as $h \to 0$.

Theorem 7.2.2 (*Differentiability Implies Continuity*)

If f is differentiable at a, then f is continuous at a.

Proof. We must show $\lim_{h\to 0} f(a+h) = f(a)$, i.e. $\lim_{h\to 0}[f(a+h) - f(a)] = 0$. Write

$$f(a+h) - f(a) = \frac{f(a+h) - f(a)}{h} \cdot h.$$

Taking the limit as $h \to 0$ and applying the product rule for limits:

$$\lim_{h\to 0}[f(a+h) - f(a)] = \lim_{h\to 0} \frac{f(a+h) - f(a)}{h} \cdot \lim_{h\to 0} h = f'(a) \cdot 0 = 0.$$

The converse fails: continuity does not imply differentiability. Section 7.3 gives canonical counterexamples.

Note (connection to Chapter 4). The cost functions $T(k) = k + \dfrac{n}{k}$ (differentiable for $k > 0$) and $-p \ln p$ (differentiable for $p \in (0, 1)$) encountered in Chapter 4 are both differentiable on their respective domains. This is what justified applying the first-derivative test to locate their optimal values. The tools needed to prove these functions are differentiable are assembled in this section and in Section 7.5 below.

7.3 Non-Differentiable Functions

Theorem 7.2.2 says differentiability implies continuity, but not the other way around. The three examples below show three ways a function can fail to be differentiable: a *corner* (the function is continuous

but the one-sided slopes disagree), a *jump discontinuity* (the function is not even continuous), and an *oscillating difference quotient* (the function is continuous, yet the difference quotient itself has no limit).

Example 7.3.1 (*Absolute Value Has a Corner at the Origin*)

Let $f(x) = |x|$. f is continuous everywhere, but it is not differentiable at $a = 0$.

Verification. Compute the one-sided limits of the difference quotient at 0:

$$\lim_{h \to 0^-} \frac{|0+h| - |0|}{h} = \lim_{h \to 0^-} \frac{-h}{h} = -1$$

$$\lim_{h \to 0^+} \frac{|0+h| - |0|}{h} = \lim_{h \to 0^+} \frac{h}{h} = 1.$$

The left and right limits exist but are unequal, so $\lim_{h \to 0} \frac{|h|}{h}$ does not exist. Therefore $f'(0)$ is undefined.

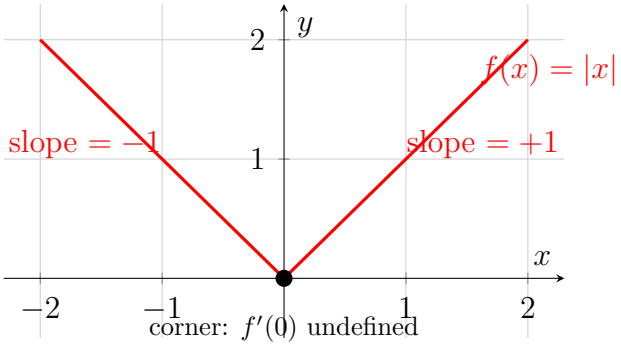


Figure 7.4: $f(x) = |x|$: continuous everywhere but not differentiable at $x = 0$. The left-hand limit of the difference quotient is -1; the right-hand limit is $+1$. Since they disagree, $f'(0)$ does not exist.

Geometrically, the graph of $|x|$ has a sharp corner at the origin: the tangent line from the left has slope -1 and the tangent line from the right has slope $+1$, and there is no single tangent line at that point (Figure 7.4).

Example 7.3.2 (*A Jump Discontinuity*)

Let $f(x) = \begin{cases} 0 & x < 0 \\ 1 & x \geq 0 \end{cases}$. f is not differentiable at $a = 0$.

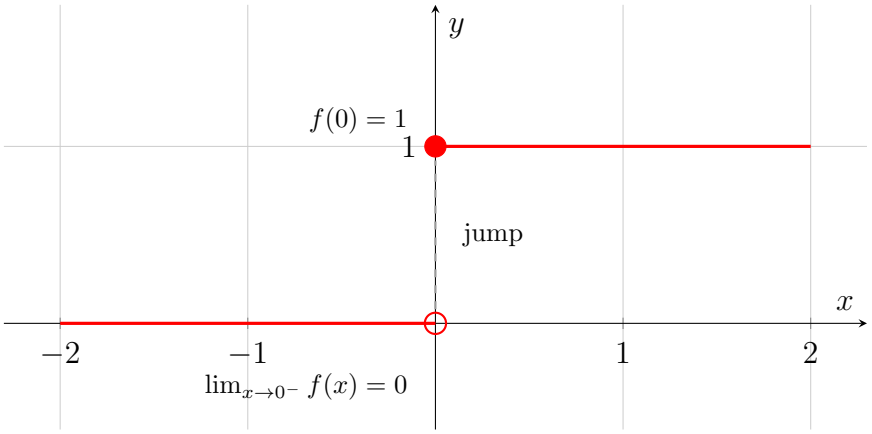


Figure 7.5: The step function $f(x) = \begin{cases} 0 & x < 0 \\ 1 & x \geq 0 \end{cases}$: not continuous at $x = 0$. The open circle marks $\lim_{x \to 0^-} f(x) = 0$; the filled circle marks $f(0) = 1$. Since f is not continuous at 0, it cannot be differentiable there.

Verification. f is not continuous at 0: $\lim_{x \to 0^-} f(x) = 0 \neq 1 = f(0)$ (see Figure 7.5). By the contrapositive of Theorem 7.2.2, f is not differentiable at 0.

This example shows that the contrapositive form of Theorem 7.2.2 — *not continuous implies not differentiable* — is often the quickest

way to rule out differentiability without computing any limits.

Example 7.3.3 (*Continuous but Not Differentiable*)

In this example, the function is a so-called oscillating difference quotient. The closer we look toward 0, the faster the function oscilates, even if its value appoaches 0. Let

$$f(x) = \begin{cases} x\sin(1/x) & x \neq 0 \\ 0 & x = 0 \end{cases}.$$

f is continuous everywhere, but it is not differentiable at $x = 0$.

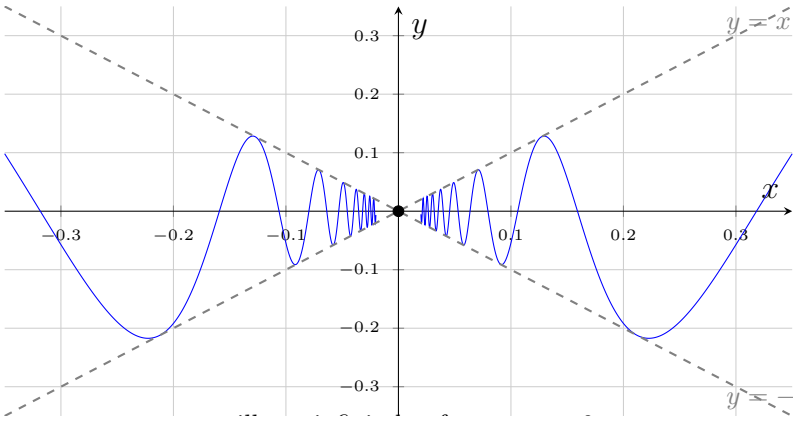


Figure 7.6: $f(x) = x\sin(1/x)$ (with $f(0) = 0$): squeezed between $y = x$ and $y = -x$ (dashed), so f is continuous at $x = 0$; but it oscillates infinitely often as $x \to 0$, so $f'(0)$ does not exist.

Verification. Continuity at 0: since $|\sin(1/x)| \leq 1$ for all $x \neq 0$,

$$|f(x)| = |x|\,|\sin(1/x)| \leq |x|\,.$$

By the Squeeze Theorem, $\lim_{x\to 0} f(x) = 0 = f(0)$, so f is continuous at 0 (see Figure 7.6).

Differentiability at 0: the difference quotient simplifies to

$$\frac{f(0+h)-f(0)}{h} = \frac{h\sin(1/h)-0}{h} = \sin(1/h), \qquad h \neq 0.$$

Take $h_n = \dfrac{2}{(4n+1)\pi} \to 0$, for which $\sin(1/h_n) = \sin\!\left(\frac{\pi}{2} + 2n\pi\right) = 1$, and $h'_n = \dfrac{2}{(4n+3)\pi} \to 0$, for which $\sin(1/h'_n) = \sin\!\left(\frac{3\pi}{2} + 2n\pi\right) = -1$. Both sequences tend to 0, but the difference quotient tends to 1 along one and -1 along the other, so $\lim_{h\to 0} \sin(1/h)$ does not exist. Therefore $f'(0)$ is undefined.

Unlike the jump discontinuity above, the contrapositive shortcut gives no help here: f *is* continuous at 0, so Theorem 7.2.2 says nothing about whether f is differentiable there. Differentiability has to be checked directly from the limit definition — and here it is the difference quotient itself, not f, that oscillates without settling.

7.4 Deriving the Differentiation Rules

The rules introduced in Chapter 1 and applied throughout Chapters 1–4 all follow from the limit definition. In this section we derive the most important ones from first principles, thereby putting the computational machinery of the earlier chapters on a rigorous footing.

7.4.1 Constant Rule

Theorem 7.4.1 (*Constant Rule*)

If $f(x) = c$ for all x, then $f'(x) = 0$.

Proof. By the limit definition,

$$f'(x) = \lim_{h\to 0} \frac{f(x+h) - f(x)}{h} = \lim_{h\to 0} \frac{c - c}{h} = \lim_{h\to 0} \frac{0}{h} = \lim_{h\to 0} 0 = 0.$$

7.4.2 Sum and Constant-Multiple Rules

Theorem 7.4.2 (*Linearity of the Derivative*)

Let f and g be differentiable at x, and let $\alpha, \beta \in \mathbb{R}$. Then $\alpha f + \beta g$ is differentiable at x and

$$(\alpha f + \beta g)'(x) = \alpha\, f'(x) + \beta\, g'(x).$$

Proof. Applying the limit definition of the derivative to $\alpha f + \beta g$:

$$\begin{aligned}
(\alpha f + \beta g)'(x) &= \lim_{h \to 0} \frac{(\alpha f + \beta g)(x+h) - (\alpha f + \beta g)(x)}{h} \\
&= \lim_{h \to 0} \frac{\alpha f(x+h) + \beta g(x+h) - \alpha f(x) - \beta g(x)}{h} \\
&= \lim_{h \to 0} \frac{\alpha\big(f(x+h) - f(x)\big) + \beta\big(g(x+h) - g(x)\big)}{h} \\
&= \lim_{h \to 0} \left[\alpha \frac{f(x+h) - f(x)}{h} + \beta \frac{g(x+h) - g(x)}{h}\right] \\
&= \alpha \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} + \beta \lim_{h \to 0} \frac{g(x+h) - g(x)}{h} \\
&= \alpha\, f'(x) + \beta\, g'(x).
\end{aligned}$$

The penultimate step uses the sum and scalar-multiple rules for limits, which hold because both limits exist (since f and g are differentiable at x by hypothesis).

Corollary 7.4.3 (*Constant-Multiple Rule*)

If f is differentiable at x and $c \in \mathbb{R}$, then $(cf)'(x) = c\, f'(x)$.

Proof. Set $\alpha = c$, $\beta = 0$, and $g = 0$ in Theorem 7.4.2.

Corollary 7.4.4 (*Sum Rule*)

If f and g are differentiable at x, then $(f+g)'(x) = f'(x) + g'(x)$.

Proof. Set $\alpha = \beta = 1$ in Theorem 7.4.2.

Corollary 7.4.5 (*Difference Rule*)

If f and g are differentiable at x, then $(f-g)'(x) = f'(x) - g'(x)$.

Proof. Set $\alpha = 1$ and $\beta = -1$ in Theorem 7.4.2.

7.4.3 Product Rule

Theorem 7.4.6 (*Product Rule*)

If f and g are differentiable at x, then fg is differentiable at x and

$$(fg)'(x) = f'(x)\,g(x) + f(x)\,g'(x).$$

Proof. Apply the limit definition. Adding and subtracting $f(x+h)g(x)$ in the numerator:

$$\begin{aligned}
(fg)'(x) &= \lim_{h\to 0} \frac{f(x+h)g(x+h) - f(x)g(x)}{h} \\
&= \lim_{h\to 0} \frac{f(x+h)(g(x+h)-g(x)) + g(x)(f(x+h)-f(x))}{h} \\
&= \lim_{h\to 0} \left[f(x+h)\,\frac{g(x+h)-g(x)}{h} + g(x)\,\frac{f(x+h)-f(x)}{h} \right] \\
&= \lim_{h\to 0} f(x+h) \cdot \lim_{h\to 0} \frac{g(x+h)-g(x)}{h} \\
&\qquad + g(x) \cdot \lim_{h\to 0} \frac{f(x+h)-f(x)}{h} \\
&= f(x)\,g'(x) + g(x)\,f'(x).
\end{aligned}$$

The penultimate step splits the limit using the sum rule for limits and the fact that $g(x)$ is constant with respect to h. The final step

uses $\lim_{h\to 0} f(x+h) = f(x)$, which holds because differentiability implies continuity.

Alternative Proof via Logarithmic Differentiation — restricted to $f, g \neq 0$. This gives a second, illuminating derivation of the Product Rule, but — unlike the proof above — it only establishes the formula at points where *neither* f nor g vanishes. That restriction is not just a technicality: the theorem itself makes no such assumption (e.g. $(x \cdot x)' = 2x$ must hold at $x = 0$ too), and this argument cannot reach a zero of f or g, since $\ln|0|$ is undefined there.

Suppose $f(x) \neq 0$ and $g(x) \neq 0$. Since $|fg| = |f|\,|g|$,

$$\ln|f(x)g(x)| = \ln|f(x)| + \ln|g(x)|\,.$$

Differentiate both sides: the left side by the Chain Rule (Theorem 7.4.9), the right side by Linearity together with the Chain Rule and $\mathcal{D}_x \ln|u| = 1/u$ (Theorem 7.5.6, proven later in this chapter):

$$\frac{(fg)'(x)}{f(x)g(x)} = \frac{f'(x)}{f(x)} + \frac{g'(x)}{g(x)}.$$

Multiplying both sides by $f(x)g(x)$:

$$(fg)'(x) = f(x)g(x)\left(\frac{f'(x)}{f(x)} + \frac{g'(x)}{g(x)}\right) = g(x)f'(x) + f(x)g'(x).$$

7.4.4 Power Rule

Theorem 7.4.7 (*Power Rule for* $\mathbb{N} \setminus \{0\}$)

For all $n \in \mathbb{N}$ with $n \geq 1$,

$$\mathcal{D}_x x^n = n x^{n-1}.$$

Proof. By induction on n. Let P_n denote the proposition $\mathcal{D}_x x^n = n x^{n-1}$.

Base case ($n = 1$). Cancellation Recipe (Pattern 7.1.9): the difference quotient gives $\frac{0}{0}$ at $h = 0$; h already appears explicitly in the numerator (Step 2), cancel it (Step 3), then apply the Substitution Rule (Step 4).

$$\mathcal{D}_x x = \lim_{h \to 0} \frac{(x+h) - x}{h} = \lim_{h \to 0} \frac{h}{h} = \lim_{h \to 0} 1 = 1 = 1 \cdot x^0,$$

so P_1 holds.

Inductive step. Assume P_n holds for some $n \geq 1$. The proof that $P_n \Rightarrow P_{n+1}$, using Theorem 7.4.6, is Exercise 4.1.3 in Chapter 4.

Theorem 7.4.8 (*Power Rule (General)*)

For all $t \in \mathbb{R}$ and $x > 0$,

$$\mathcal{D}_x x^t = t x^{t-1}.$$

For $t \in \mathbb{N} \setminus \{0\}$ the result holds for all x; for $t \in \mathbb{Z}$ with $t \leq 0$ it holds for all $x \neq 0$.

Proof. We treat each class of exponent in turn.

Case $t \in \mathbb{N} \setminus \{0\}$. Theorem 7.4.7.

Case $t = 0$. $\mathcal{D}_x x^0 = \mathcal{D}_x 1 = 0 = 0 \cdot x^{-1}$ by the Constant Rule.

Case $t \in \mathbb{Z}$, $t < 0$. Write $t = -m$ with $m \in \mathbb{N} \setminus \{0\}$. Differentiating the identity $x^m \cdot x^{-m} = 1$ with the Product Rule and Theorem 7.4.7:

$$\begin{aligned} \mathcal{D}_x\left(x^m \cdot x^{-m}\right) &= \mathcal{D}_x 1 \\ m x^{m-1} \cdot x^{-m} + x^m \cdot \mathcal{D}_x x^{-m} &= 0 \\ \mathcal{D}_x x^{-m} &= \frac{-m\, x^{m-1} \cdot x^{-m}}{x^m} \\ &= -m\, x^{(m-1)-m-m} \\ &= -m\, x^{-m-1} \\ &= t\, x^{t-1}. \end{aligned}$$

Case $t \in \mathbb{Q} \setminus \mathbb{Z}$. Requires the Chain Rule, which has not yet been proven at this point in the chapter. This case (and the next) is completed in Section 7.5.3 once the Chain Rule and the derivative of ln are available.

Case $t \in \mathbb{R}$, $x > 0$. Requires the Chain Rule and the derivative of ln, proven later in this chapter. Completed alongside the previous case in Section 7.5.3.

7.4.5 Chain Rule

Theorem 7.4.9 (*Chain Rule*)

If g is differentiable at x and f is differentiable at $g(x)$, then $f \circ g$ is differentiable at x and

$$\mathcal{D}_x(h(g(x)) = (\mathcal{D}\, h)(g(x)) \cdot \mathcal{D}_x g(x).$$

In Newtonian notation:

$$(f \circ g)'(x) = f'(g(x)) \cdot g'(x).$$

In Leibniz notation, setting $u = g(x)$ and $y = f(u)$:

$$\frac{dy}{dx} = \frac{dy}{du} \cdot \frac{du}{dx}.$$

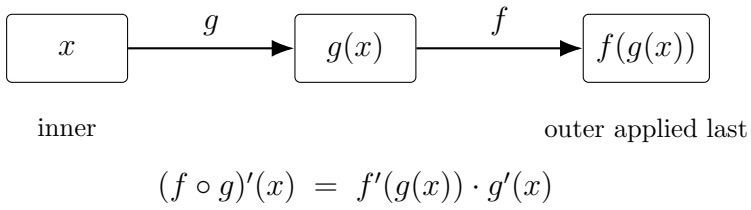


Figure 7.7: Composition: g acts first (inner), then f acts on the result (outer). The chain rule multiplies the two local rates of change, $g'(x)$ and $f'(g(x))$, in that same order.

Figure 7.7 lays out the composition this theorem differentiates.

Motivation. The Leibniz form suggests a proof by canceling du, exactly as one would cancel a common factor in a product of fractions. This is the right intuition, and it is a powerful mnemonic. The difficulty is that $\frac{dy}{du}$ and $\frac{du}{dx}$ are limits, not fractions, and du is not a number that can be canceled. We make the intuition rigorous below.

Proof (naive attempt). The natural approach is to multiply and divide by $g(x+h)-g(x)$:

$$\begin{aligned}(f\circ g)'(x) &= \lim_{h\to 0}\frac{f(g(x+h))-f(g(x))}{h}\\ &= \lim_{h\to 0}\frac{f(g(x+h))-f(g(x))}{g(x+h)-g(x)}\cdot\frac{g(x+h)-g(x)}{h}.\end{aligned}$$

The Cancellation Recipe (Pattern 7.1.9) appears to apply here: multiply and divide by $g(x+h)-g(x)$ to expose $f'(g(x))$ and $g'(x)$. Step 3 of the recipe fails, however, because $g(x+h)-g(x)$ can equal zero for arbitrarily small $h\neq 0$ even when g is differentiable — so the cancellation is not always valid. The rigorous proof below patches this gap with an auxiliary function Q that removes the division-by-zero case.

Proof (rigorous). Define the auxiliary function

$$Q(k)=\begin{cases}\dfrac{f(g(x)+k)-f(g(x))}{k} & k\neq 0,\\ f'(g(x)) & k=0.\end{cases}$$

Since f is differentiable at $g(x)$, the limit of the difference quotient exists and equals $f'(g(x))$, so Q is continuous at $k=0$.

The key observation is that Q satisfies the multiplicative identity

$$f\big(g(x)+k\big)-f\big(g(x)\big) \;=\; Q(k)\cdot k$$

for *all* k: when $k\neq 0$ this is just the definition of Q; when $k=0$ both sides equal zero.

Now substitute $k = g(x+h) - g(x)$:

$$f(g(x+h)) - f(g(x)) \;=\; Q\big(g(x+h)-g(x)\big)\cdot\big(g(x+h)-g(x)\big).$$

Divide both sides by h:

$$\frac{f(g(x+h)) - f(g(x))}{h} \;=\; Q\big(g(x+h)-g(x)\big)\cdot\frac{g(x+h)-g(x)}{h}.$$

Take the limit as $h \to 0$:

$$\begin{aligned}(f\circ g)'(x) &= \lim_{h\to 0} Q\big(g(x+h)-g(x)\big)\cdot\frac{g(x+h)-g(x)}{h}\\ &= Q(0)\cdot g'(x)\\ &= f'(g(x))\cdot g'(x).\end{aligned}$$

The step $Q\big(g(x+h)-g(x)\big)\to Q(0)$ holds because $g(x+h)\to g(x)$ (differentiability implies continuity) and Q is continuous at 0.

Historical note. For Leibniz and Newton the cancellation was not a mnemonic — it was a proof, and a correct one within their framework. Leibniz's dy, du, dx were genuine *infinitesimals*: nonzero quantities smaller than any positive real number. Division by du was legal (it was not zero), so the cancellation was valid. Newton's *fluxions* carried the same assumption through the notion of *ultimate ratios* of vanishing increments. The flaw only surfaces when calculus is rebuilt on the ε-δ limit definition of Cauchy (1820s) and Weierstrass (1860s), where du is a limiting process, not a number. Remarkably, Abraham Robinson's *non-standard analysis* (1960) later constructed a rigorous extension of the real numbers containing genuine infinitesimals, in which the Leibniz cancellation proof is entirely correct. Leibniz's intuition was right; it simply preceded the machinery needed to justify it by three centuries.

7.4.6 Quotient Rule

Theorem 7.4.10 (*Quotient Rule*)

If f and g are differentiable at x and $g(x) \neq 0$, then f/g is differentiable at x and

$$\left(\frac{f}{g}\right)'(x) = \frac{f'(x)\,g(x) - f(x)\,g'(x)}{g(x)^2}.$$

Proof. Write $f/g = f \cdot g^{-1}$ and apply the Product Rule (Theorem 7.4.6):

$$\left(\frac{f}{g}\right)'(x) = f'(x)\,g(x)^{-1} + f(x)\,\left(g^{-1}\right)'(x).$$

By the Chain Rule (Theorem 7.4.9) with outer function $t \mapsto t^{-1}$ and inner function g, using the General Power Rule at $t = -1$ (Theorem 7.4.8):

$$\left(g^{-1}\right)'(x) = -g(x)^{-2} \cdot g'(x).$$

Substituting and collecting over the common denominator $g(x)^2$:

$$\left(\frac{f}{g}\right)'(x) = \frac{f'(x)}{g(x)} - \frac{f(x)\,g'(x)}{g(x)^2} = \frac{f'(x)\,g(x) - f(x)\,g'(x)}{g(x)^2}.$$

The Quotient Rule can also be proven directly from the limit definition without invoking the Chain Rule, by the same add-and-subtract technique used for the Product Rule. The derivation via $f \cdot g^{-1}$ is shorter and illustrates how the rules compose.

7.5 Derivatives of Transcendental Functions

The table of derivatives introduced in Figure 2.1 (page 71) of Chapter 2 includes several transcendental functions. The algebraic rules proven in this section are now sufficient to justify most of them.

- $\mathcal{D}_x e^x = e^x$: follows immediately from the definition of exp given in Section 7.5 below.
- $\mathcal{D}_x \sin x = \cos x$ and $\mathcal{D}_x \cos x = -\sin x$: derived below via term-by-term differentiation of the Taylor series for $\sin x$ (the version covered in lecture); Theorem 7.5.2 and Corollary 7.5.3 also give the classical proof from first principles (enrichment). The analogous Taylor-series computation for $\cos x$ (and for e^x) is Section 4.1.2 in Chapter 4.
- $\mathcal{D}_x \tan x$: derived in Chapter 1 using the Quotient Rule applied to $\sin x / \cos x$.
- $\mathcal{D}_x \ln x = \dfrac{1}{x}$: proven in Theorem 7.5.6 below via implicit differentiation.
- Inverse trigonometric functions: proven in the optional Section 7.5.2 below via the same implicit differentiation schema used for arctan in Chapter 2, Section 2.4.

Proposition 7.5.1 (*Two Key Trigonometric Limits*)

$$\lim_{h\to 0} \frac{\sin h}{h} = 1 \qquad \text{and} \qquad \lim_{h\to 0} \frac{\cos h - 1}{h} = 0.$$

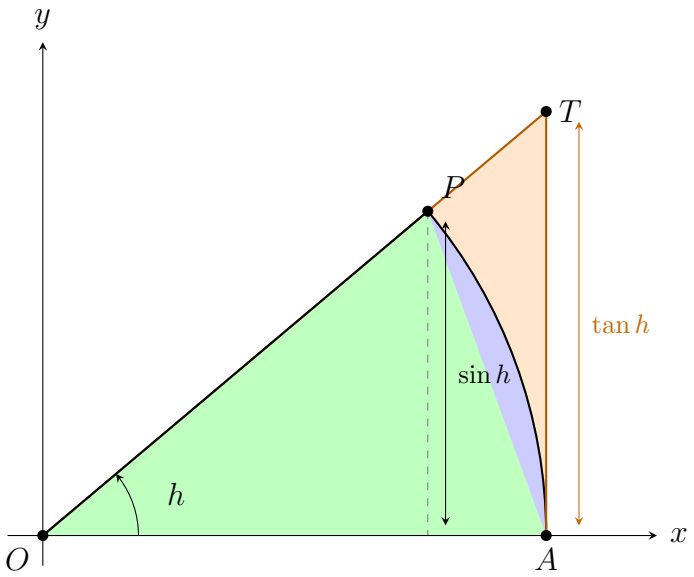


Figure 7.8: The three nested regions used to prove Proposition 7.5.1. Triangle OAP (green, area $\frac{1}{2}\sin h$) is contained in the circular sector OAP of angle h (blue, area $\frac{h}{2}$), which is contained in triangle OAT (orange, area $\frac{1}{2}\tan h$). Point $P = (\cos h, \sin h)$ lies on the unit circle; $T = (1, \tan h)$ is where the line through O and P meets the vertical $x = 1$.

Proof of Proposition 7.5.1. **First limit.** It suffices to prove the limit as $h \to 0^+$; the case $h \to 0^-$ follows because $\dfrac{\sin(-h)}{-h} = \dfrac{\sin h}{h}$ (both sin and h are odd functions of h).

Fix $0 < h < \pi/2$ and place three regions inside the unit circle (Figure 7.8), all sharing the vertex at the origin O:

- Triangle OAP, where $A = (1, 0)$ and $P = (\cos h, \sin h)$. Area $= \frac{1}{2}\sin h$.
- Circular sector OAP of angle h. Area $= \frac{h}{2\pi} \cdot \pi = \frac{h}{2}$.

- Triangle OAT, where $T = (1, \tan h)$ is where the tangent to the circle at A meets the ray OP. Area $= \frac{1}{2}\tan h$.

Containment gives the area inequality:

$$\frac{\sin h}{2} \leq \frac{h}{2} \leq \frac{\tan h}{2}.$$

Divide through by $\dfrac{\sin h}{2} > 0$ and take reciprocals (reversing inequalities):

$$\cos h \leq \frac{\sin h}{h} \leq 1.$$

Since $\lim_{h\to 0^+} \cos h = 1$, the Squeeze Theorem (Theorem 7.1.6) gives

$$\lim_{h\to 0^+} \frac{\sin h}{h} = 1\,.$$

Second limit. Cancellation Recipe (Pattern 7.1.9): apply the double-angle identity (Step 2, transcendental transform) to expose the first limit with h replaced by $h/2$ (Step 4).

Apply the double-angle identity $\cos h - 1 = -2\sin^2\!\left(\frac{h}{2}\right)$:

$$\frac{\cos h - 1}{h} = \frac{-2\sin^2\!\left(\frac{h}{2}\right)}{h} = -\frac{\sin\frac{h}{2}}{\frac{h}{2}} \cdot \sin\frac{h}{2}.$$

As $h \to 0$, the first factor tends to 1 (the first limit with h replaced by $h/2$) and $\sin\frac{h}{2} \to 0$, so the product tends to $-1 \cdot 0 = 0$.

Theorem 7.5.2 (*Derivative of* sin)

$\mathcal{D}_x \sin x = \cos x$.

Proof via Taylor Series — covered in lecture. Recall the power series

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n+1}}{(2n+1)!}.$$

Differentiate term by term:

$$\begin{aligned} \mathcal{D}_x \sin x &= 1 - \frac{3x^2}{3!} + \frac{5x^4}{5!} - \frac{7x^6}{7!} + \cdots = \sum_{n=0}^{\infty} \frac{(-1)^n (2n+1)\, x^{2n}}{(2n+1)!} \\ &= 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n}}{(2n)!} \\ &= \cos x. \end{aligned}$$

The middle step cancels a factor from each numerator into the factorial below it (e.g. $\frac{3}{3!} = \frac{1}{2!}$, or in general $\frac{2n+1}{(2n+1)!} = \frac{1}{(2n)!}$), which is exactly the series for $\cos x$.

Term-by-term differentiation of a power series is a general technique, not a trick specific to $\sin x$: whenever a function is only known to you as a series, differentiating (or integrating) it term by term is often the fastest way to compute its derivative — you will use the same move for $\cos x$ and e^x in Section 4.1.2 (Chapter 4). This does require that the series be differentiated within its radius of convergence, a fact made precise in Chapter 11; until then, term-by-term differentiation is taken as given, as it was above.

Proof from First Principles. Cancellation Recipe (Pattern 7.1.9): the difference quotient gives $\frac{0}{0}$ at $h = 0$; the angle-addition formula

(Step 2, transcendental transform) rewrites the numerator, exposing $\frac{\sin h}{h}$ and $\frac{\cos h - 1}{h}$, whose limits are known from Proposition 7.5.1 (Step 4).

Recall $\sin(x+h) = \sin x \cos h + \cos x \sin h$, the angle-addition formula:

$$\begin{aligned}
\mathcal{D}_x \sin x &= \lim_{h \to 0} \frac{\sin(x+h) - \sin x}{h} \\
&= \lim_{h \to 0} \frac{\sin x \cos h + \cos x \sin h - \sin x}{h} \\
&= \lim_{h \to 0} \left[\sin x \cdot \frac{\cos h - 1}{h} + \cos x \cdot \frac{\sin h}{h} \right] \\
&= \sin x \cdot 0 + \cos x \cdot 1 \\
&= \cos x.
\end{aligned}$$

The final step applies Proposition 7.5.1, whose own proof uses the Squeeze Theorem (Theorem 7.1.6).

Corollary 7.5.3 (*Derivative of* cos)

$\mathcal{D}_x \cos x = -\sin x.$

The analogous Taylor-series computation for $\cos x$ — differentiate its series term by term and recognize the result as $-\sin x$ — is Section 4.1.2 in Chapter 4.

Proof from First Principles. Cancellation Recipe (Pattern 7.1.9), identical structure to the proof of Theorem 7.5.2 above.

Using the angle-addition formula

$\cos(x+h) = \cos x \cos h - \sin x \sin h$:

$$\begin{aligned}\mathcal{D}_x \cos x &= \lim_{h\to 0} \frac{\cos x \cos h - \sin x \sin h - \cos x}{h} \\ &= \lim_{h\to 0} \left[\cos x \cdot \frac{\cos h - 1}{h} - \sin x \cdot \frac{\sin h}{h}\right] \\ &= \cos x \cdot 0 - \sin x \cdot 1 = -\sin x.\end{aligned}$$

Definition 7.5.4 (*The Exponential Function*)

The *exponential function* $\exp\colon \mathbb{R} \to \mathbb{R}$ is the unique function satisfying

$$\exp'(x) = \exp(x) \quad \text{for all } x \in \mathbb{R}, \qquad \exp(0) = 1.$$

We write e^x for $\exp(x)$ and call $e := \exp(1)$ *Euler's number*. In particular, $\mathcal{D}_x e^x = e^x$ for all x.

The equation $\exp'(x) = \exp(x)$ is an example of an *ordinary differential equation* (ODE): an equation relating an unknown function to its own derivative(s). ("Ordinary" distinguishes it from a *partial* differential equation, which involves derivatives with respect to more than one variable.) ODEs are solved numerically in Chapter 12.

Existence of such a function is confirmed in Chapter 11 by the power series $e^x = \sum_{n=0}^{\infty} x^n/n!$, which can be shown to satisfy both conditions. Uniqueness requires a tool (the zero-derivative theorem) that becomes available once the Fundamental Theorem of Calculus is proven in Chapter 10. Until then, Definition 7.5.4 is taken as an axiom.

Theorem 7.5.5 (*Exponential Addition Formula*)

For all $x, y \in \mathbb{R}$, $\quad e^{x+y} = e^x \cdot e^y$.

Proof. Fix $y \in \mathbb{R}$ and define $g(x) = e^{x+y}$. By the Chain Rule, $g'(x) = e^{x+y} = g(x)$, so g satisfies the same ODE as exp. The rescaled function $h(x) = g(x)/e^y$ then satisfies $h' = h$ and $h(0) = e^y/e^y = 1$. By uniqueness, $h = \exp$, so $e^{x+y} = e^y \cdot e^x$.

7.5.1 Alternative Definitions of the Exponential

The ODE definition is not the only route to the exponential function; the following three approaches are equivalent and appear in many textbooks.

- **Via $\ln$ as antiderivative (no integral required).**

  Assert that there exists a unique function $\ln\colon (0,\infty) \to \mathbb{R}$ satisfying $(\ln x)' = 1/x$ and $\ln 1 = 0$. The algebraic property $\ln(xy) = \ln(x) + \ln(y)$ then follows from uniqueness by the same trick as Theorem 7.5.5: fix $a > 0$ and observe that $x \mapsto \ln(ax) - \ln(a)$ has derivative $1/x$ and value 0 at $x = 1$, so by uniqueness it equals $\ln x$.

  Define exp as the inverse function of ln, so that

  $$e^x := \ln^{-1}(x)\,;$$

  the derivative $(e^x)' = e^x$ follows by implicit differentiation of $\ln(e^x) = x$. Existence and uniqueness of ln carry the same gap as Definition 7.5.4: both are resolved once the tools of Chapter 10 are available.

- **Via the natural logarithm as an integral.**

  Define

  $$\ln x = \int_1^x \frac{1}{t}\,dt$$

for $x > 0$. By the Fundamental Theorem of Calculus (Chapter 10), $(\ln x)' = 1/x$, so ln is strictly increasing and maps $(0, \infty)$ onto $\mathbb{R}$. Define exp as the inverse function of ln, so that

$$e^x := \ln^{-1}(x)\,.$$

This approach resolves existence automatically — the integral of a continuous function is always well-defined — but requires the full theory of integration as a prerequisite.

- **Via the power series.** 🎓

  Set $e^x = \displaystyle\sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$. Differentiating term by term returns the same series, confirming $\exp' = \exp$ and $\exp(0) = 1$, and simultaneously establishing existence. This is carried out in Section 1.2.9 of Chapter 1. The analogous derivation for cos, by the same power-series technique, is Exercise 4.1.2 of Chapter 4; the sin case is Exercise 11.3 of Chapter 11.

"Differentiating term by term" above deserves a closer look, now 🎓 that Definition 7.2.1 is available. Writing $f_n(x) = \dfrac{x^n}{n!}$ for the n-th term and $f(x) = \sum_{n=0}^{\infty} f_n(x)$, the claim $\mathcal{D}_x f(x) = \sum_{n=0}^{\infty} \mathcal{D}_x f_n(x)$ says, spelled out in full using Definition 7.2.1 for both f and each f_n:

$$\lim_{h\to 0} \frac{\lim\limits_{N\to\infty} \sum\limits_{n=0}^{N} f_n(x+h) - \lim\limits_{N\to\infty} \sum\limits_{n=0}^{N} f_n(x)}{h} = \lim_{N\to\infty} \sum_{n=0}^{N} \lim_{h\to 0} \frac{f_n(x+h) - f_n(x)}{h}$$

The left side takes the limit defining the series first ($N \to \infty$) and the limit defining the derivative second ($h \to 0$); the right side takes the very same two limits in the *opposite* order. Nothing proven so

far justifies that these agree — swapping the order of two limiting processes is not automatically valid in general. Theorem 1.2.13 of Chapter 1 (already used there, without this justification) is exactly the statement that for a power series, within its radius of convergence, this particular swap *is* valid. A full proof needs *uniform convergence*, a topic from real analysis this course does not cover.

All four definitions — ODE, antiderivative of $1/x$, integral, power series — produce the same function. The ODE approach is used here because it identifies the property central to calculus without requiring the integral or infinite series as prerequisites.

Remark: the depth behind the definition. How to define e^x rigorously is one of the genuinely fascinating questions in mathematics. Elementary school gives n^m as repeated multiplication — a concrete, finite operation. Extending this first to rational exponents ($n^{1/2} = \sqrt{n}$), then to irrational exponents (2^π), then to complex arguments ($e^{i\pi} = -1$), and finally to matrix arguments (e^M for a square matrix M, which solves systems of differential equations) requires definitions that are progressively harder to justify, and were far from obvious even to the mathematicians who first encountered exponential growth — through compound interest, geometric series, and logarithm tables — in the seventeenth century. The three approaches sketched above give a small glimpse of that depth. For this course, none of it is needed: the two properties $\mathcal{D}_x e^x = e^x$ and $e^{x+y} = e^x e^y$, delivered immediately by Definition 7.5.4, are sufficient for every calculation that follows.

Theorem 7.5.6 (*Derivative of* ln)

For $x > 0$, $\mathcal{D}_x \ln x = \dfrac{1}{x}$.

Proof. The identity $e^{\ln x} = x$ holds for all $x > 0$. Differentiating both sides with respect to x using the Chain Rule (Theorem 7.4.9)

and $\mathcal{D}_x e^x = e^x$:

$$e^{\ln x} \cdot \mathcal{D}_x \ln x = 1 \implies x \cdot \mathcal{D}_x \ln x = 1 \implies \mathcal{D}_x \ln x = \frac{1}{x}.$$

Note (connection to Chapter 4). Students who completed the skip-list exercises in Section 4.5 of Chapter 4 differentiated $-p \ln p$ and $p(\ln p)^2$ to optimize the promotion probability. Those calculations rely on $\mathcal{D}_x \ln p = \frac{1}{p}$, which is proven in Theorem 7.5.6 above.

7.5.2 Derivatives: Inverse Trigonometric Functions

In this section we prove the following equalities.

$$\begin{aligned}
\mathcal{D}_x \arcsin x &= \frac{1}{\sqrt{1-x^2}}, && |x| < 1, \\
\mathcal{D}_x \arccos x &= \frac{-1}{\sqrt{1-x^2}}, && |x| < 1, \\
\mathcal{D}_x \arctan x &= \frac{1}{1+x^2}, && x \in \mathbb{R}, \\
\mathcal{D}_x \operatorname{arccot} x &= \frac{-1}{1+x^2}, && x \in \mathbb{R}, \\
\mathcal{D}_x \operatorname{arcsec} x &= \frac{1}{|x|\sqrt{x^2-1}}, && |x| > 1, \\
\mathcal{D}_x \operatorname{arccsc} x &= \frac{-1}{|x|\sqrt{x^2-1}}, && |x| > 1.
\end{aligned}$$

Each inverse trigonometric derivative follows the same schema used for arctan in Chapter 2, Section 2.4. Given $y = \operatorname{arcf}(x)$:

1. Write the equivalent direct statement $f(y) = x$.
2. Differentiate implicitly: $f'(y) \cdot y' = 1$, so $y' = 1/f'(y)$.
3. Apply a Pythagorean identity to express $f'(y)$ purely in terms of $f(y) = x$, eliminating y.

The Pythagorean identity needed is one of:

- $\sin^2 + \cos^2 = 1$,
- $1 + \tan^2 = \sec^2$, or
- $1 + \cot^2 = \csc^2$,

depending on the function.

Figure 7.9 is a reminder of the geometric meaning of all six trigonometric functions. The angle θ determines a point $P = (\cos\theta, \sin\theta)$ on the circle; the ray from O through P meets the vertical tangent at $x = 1$ in Q and the horizontal tangent at $y = 1$ in R.

Function	**Geometric description**
$\sin\theta$	y-coordinate of P (Figure 7.9)
$\cos\theta$	x-coordinate of P
$\tan\theta$	$\lvert(1,0)Q\rvert$
$\sec\theta$	$\lvert OQ\rvert$
$\cot\theta$	$\lvert(0,1)R\rvert$
$\csc\theta$	$\lvert OR\rvert$

All cases follow the three-step schema given at the start of Section 7.5.2 above. **arctan** is proven in Theorem 2.4.1; the remaining cases are shown here.

Theorem 7.5.7 (*Derivative of* arcsin)

$\mathcal{D}_x \arcsin x = \frac{1}{\sqrt{1-x^2}}$ for $|x| < 1$.

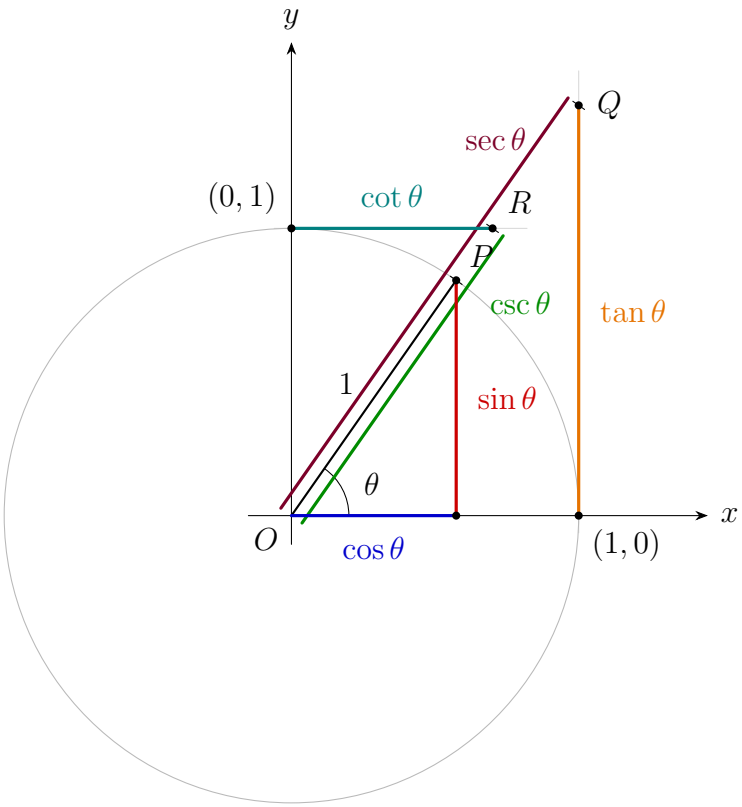


Figure 7.9: Geometric definitions of six trigonometric functions on the unit circle.

Proof. Set $y = \arcsin x$, so $\sin y = x$ with $y \in \left(-\frac{\pi}{2}, \frac{\pi}{2}\right)$. Differentiating both sides implicitly:

$$\cos y \cdot y' = 1 \qquad \Longrightarrow \qquad y' = \frac{1}{\cos y}.$$

Apply $\sin^2 y + \cos^2 y = 1$ and use $\cos y > 0$ on the chosen branch:

$$\cos y = \sqrt{1 - \sin^2 y} = \sqrt{1 - x^2}.$$

Therefore $y' = \dfrac{1}{\sqrt{1-x^2}}$.

A note on a second proof. Theorem 7.5.7 is proven here by implicit differentiation, but that is not the only route. Section 13.2.3 (Chapter 13) re-derives this same derivative from the substitution $x = \sin\theta$ applied to $\int \frac{dx}{\sqrt{1-x^2}}$, using FTC Part 1 instead of implicit differentiation.

Theorem 7.5.8 (*Derivative of* arccos)

$\mathcal{D}_x \arccos x = \frac{-1}{\sqrt{1-x^2}}$, for $|x| < 1$.

Proof. Set $y = \arccos x$, so $\cos y = x$ with $y \in (0, \pi)$. Differentiating implicitly:

$$-\sin y \cdot y' = 1 \qquad \Longrightarrow \qquad y' = \frac{-1}{\sin y}.$$

Since $\sin y > 0$ on $(0, \pi)$:

$$\sin y = \sqrt{1 - \cos^2 y} = \sqrt{1 - x^2}.$$

Therefore $y' = \dfrac{-1}{\sqrt{1-x^2}}$.

A note on a second proof. As with arcsin above, this result can also be reached without implicit differentiation, by the substitution $t = \cos\theta$ applied to $\int_1^x \frac{-dt}{\sqrt{1-t^2}}$ — see Exercise 15.2 in Chapter 15.

Theorem 7.5.9 (*Derivative of arccot*)

$\mathcal{D}_x \operatorname{arccot} x = \frac{-1}{1+x^2}$, for $x \in \mathbb{R}$.

Proof. Set $y = \operatorname{arccot} x$, so $\cot y = x$ with $y \in (0, \pi)$.

Differentiating implicitly:

$$-\csc^2 y \cdot y' = 1 \qquad \Longrightarrow \qquad y' = \frac{-1}{\csc^2 y}.$$

Apply $1 + \cot^2 y = \csc^2 y$:

$$\csc^2 y = 1 + \cot^2 y = 1 + x^2.$$

Therefore $y' = \dfrac{-1}{1 + x^2}$.

Theorem 7.5.10 (*Derivative of arcsec*)

$\mathcal{D}_x \operatorname{arcsec} x = \frac{1}{|x|\sqrt{x^2-1}}$, for $|x| > 1$.

Proof. Set $y = \operatorname{arcsec} x$, so $\sec y = x$ with $y \in \left[0, \frac{\pi}{2}\right) \cup \left(\frac{\pi}{2}, \pi\right]$. Differentiating implicitly:

$$\sec y \tan y \cdot y' = 1 \qquad \Longrightarrow \qquad y' = \frac{1}{\sec y \tan y} = \frac{1}{x \tan y}.$$

Apply $\tan^2 y = \sec^2 y - 1 = x^2 - 1$, so $|\tan y| = \sqrt{x^2 - 1}$. The sign of $\tan y$ matches the sign of x on this branch, giving $x \tan y = |x|\sqrt{x^2 - 1}$, and therefore:

$$y' = \frac{1}{|x|\sqrt{x^2 - 1}}.$$

Theorem 7.5.11 (*Derivative of arccsc*)

$\mathcal{D}_x \operatorname{arccsc} x = \frac{-1}{|x|\sqrt{x^2-1}}$, for $|x| > 1$.

Proof. The derivation is analogous to arcsec, with an extra sign change from differentiating $\csc y = x$ (giving $-\csc y \cot y \cdot y' = 1$). The result is $y' = \dfrac{-1}{|x|\sqrt{x^2-1}}$.

7.5.3 Completing the Power Rule

The proof of Theorem 7.4.8 deferred two cases — $t \in \mathbb{Q} \setminus \mathbb{Z}$ and $t \in \mathbb{R}$ with $x > 0$ — because they require the Chain Rule and the derivative of ln, neither of which was available yet at that point in the chapter. Both tools are now in hand (Section 7.4.5, Theorem 7.5.6), so we finish the proof here.

Proof of Case $t \in \mathbb{Q} \setminus \mathbb{Z}$, $x > 0$.
Write $t = p/q$ in lowest terms, with $p \in \mathbb{Z}$, $q \in \mathbb{N}$, $q > 1$. Set $y = x^{p/q}$, so that $y^q = x^p$. Differentiate both sides with respect to x — the left side by the Chain Rule (treating y as a function of x) together with Theorem 7.4.7, the right side directly by Theorem 7.4.7:

$$q\, y^{q-1} \cdot \mathcal{D}_x y = p\, x^{p-1} \qquad \Longrightarrow \qquad \mathcal{D}_x y = \frac{p\, x^{p-1}}{q\, y^{q-1}}\,.$$

Substitute $y^{q-1} = \left(x^{p/q}\right)^{q-1} = x^{p-p/q}$ and simplify the exponent:

$$\mathcal{D}_x x^{p/q} = \frac{p\, x^{p-1}}{q\, x^{p-p/q}} = \frac{p}{q}\, x^{(p-1)-(p-p/q)} = \frac{p}{q}\, x^{p/q-1} = t\, x^{t-1}\,.$$

Proof of Case $t \in \mathbb{R}$, $x > 0$.

This case subsumes all the others, but is proven separately because writing x^t this way requires ln to already be available. Since $x > 0$, write $x^t = e^{t \ln x}$. Let $u(x) = t \ln x$, so $x^t = \exp(u(x))$. By the Chain Rule, using $\mathcal{D}_x e^x = e^x$ (Definition 7.5.4) and

$$\begin{aligned} \mathcal{D}_x u(x) &= \mathcal{D}_x (t \ln x) \\ &= t \, \mathcal{D}_x \ln x \\ &= t \frac{1}{x} \qquad \text{Theorem 7.5.6} \end{aligned}$$

So,

$$\begin{aligned} \mathcal{D}_x x^t &= \mathcal{D}_x \exp(u(x)) \\ &= \exp(u(x)) \cdot \mathcal{D}_x u(x) \\ &= e^{t \ln x} \cdot \frac{t}{x} \\ &= x^t \cdot \frac{t}{x} \\ &= t \, x^{t-1} . \end{aligned}$$

Together with the cases already proven, this completes Theorem 7.4.8 for every $t \in \mathbb{R}$.

Why does the rate of change of x^t exactly correspond to t (the exponent) multiplied by x^{t-1}, with the exponent decreased exactly by 1? This proof contains something insightful and intuitive. Answer: because of the Chain Rule: to compute the derivative of x^t, we express it as an exponential, $e^{u(t)}$, and then differentiating—we get back $e^{u(t)}$

but scaled by $\mathcal{D}_x u(t) = \frac{t}{x}$ which has the effect of exactly scaling by t (the exponent) and dividing by x which decrements the exponent exactly by 1.

Now that this proof covers every real exponent, it is worth noting that Chapter 5's `PointFree` ADT deliberately does not: `Power(n)`, the class representing x^n in that chapter's symbolic differentiation engine, only accepts integer `n` (Section 5.2.4) — enforced by an assertion in `Power`'s constructor, not merely documented. That restriction is not because the power rule fails for non-integer exponents; this proof shows it does not. It is because the *values* can leave the real numbers ($x < 0$ raised to a fractional power is complex), and the symbolic engine promises every expression it can build evaluates to a real number. The rule is true; the ADT simply chooses not to represent the case where evaluating it could break that promise.

? Open Questions

- Definition 7.5.4 calls $\exp$ the *unique* function with $\exp' = \exp$ and $\exp(0) = 1$, but says uniqueness "requires a tool (the zero-derivative theorem)" without stating what that theorem says or proving it. What would such a theorem claim? (It says: if $g'(x) = 0$ for every x in an interval, then g is constant on that interval. Try to see why this would settle uniqueness — apply it to the difference of two solutions of the same ODE.)

- The chapter objectives promise "corners, cusps, and discontinuities" as the ways a function can fail to be differentiable, but Section 7.3 only exhibits a corner ($|x|$) and a jump discontinuity. A *cusp* is a third case: sketch $f(x) = x^{1/3}$ near $x = 0$. Is f continuous at 0? What happens to the difference quotient $\frac{f(0+h)-f(0)}{h}$ as $h \to 0$ — does it fail to exist the same way it did for $|x|$, or does something different go wrong?

- The proof of the Power Rule (General) handles integer exponents in full, but for $t \in \mathbb{Q} \setminus \mathbb{Z}$ and $t \in \mathbb{R}$ it only says "requires the Chain Rule" and points to a later section, without carrying out the computation. Finish it yourself: write $x^t = e^{t \ln x}$ and differentiate using the Chain Rule together with $\mathcal{D}_x e^x = e^x$ and $\mathcal{D}_x \ln x = 1/x$ (both proven later in this chapter).

- The historical note at the end of Section 7.4.5 observes that Robinson's non-standard analysis makes Leibniz's "cancel the du" argument for the Chain Rule literally rigorous, three centuries after the fact. If infinitesimals can be made fully rigorous, why do essentially all calculus courses — this one included — teach ε-δ limits instead? What would be gained, or lost, by teaching infinitesimals from the start?

Chapter 8

The Integral

Chapter Objectives

- Approximate the area under a curve using left, right, midpoint, trapezoidal, and Simpson's rule Riemann sums.
- Define the definite integral as the limit of a Riemann sum, and state the conditions under which the limit exists.
- Apply the properties of the definite integral: zero-width, reversal, splitting, and linearity.
- Compare the convergence rates of the numerical methods and explain why higher-order rules converge faster.
- Implement the five summation rules in Python and verify them against `scipy.integrate.quad`.

Calculus was built to answer two questions that, on the surface, look unrelated. The first is geometric: for a rectangle or a triangle, "area" already has a precise meaning you learned long before calculus, but what does "area" even mean when the boundary is curved — say, under $f(x) = x^2$ or $f(x) = \sin x$? There is no elementary formula for that.

The second example is about physical rates of change. A capacitor with voltage $v(t)$ across it draws current $i(t) = C\,v'(t)$, where C (the

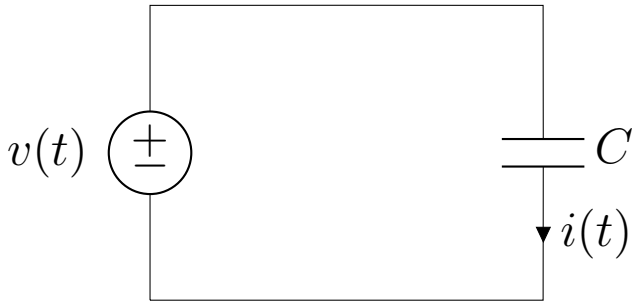


Figure 8.1: A voltage source $v(t)$ drives current $i(t)$ through a capacitor. Physics gives $i(t) = C\,\frac{d}{dt}v(t)$; an engineer often knows the current she wants and needs to recover the voltage that produces it — the reverse direction.

capacitance) depends only on the capacitor's geometry; if you know v, finding i is differentiation. An engineer designing the circuit often needs the reverse: given the desired current $i(t)$ — possibly a real, measured signal with no tidy formula at all — find the voltage $v(t)$ that produces it. Figure 8.1 shows the circuit.

$$
\begin{aligned}
i(t) &= C\,\frac{d}{dt}v(t) \\
\int i(t)dt &= \int C\,\frac{d}{dt}v(t)dt \\
&= Cv(t) + v_0 && \text{by Fundamental Theorem of Calculus} \\
v(t) &= \frac{1}{C}\left(\int i(t)dt - v_0\right)
\end{aligned}
$$

The same question shows up whenever a rate is known and the accumulated quantity is not: a cannonball's height, given a constant downward acceleration; a population, given its growth rate.

Nothing said so far suggests these two questions — "what is the area under this curve" and "given a rate, recover the quantity" — have anything to do with each other. One is static and geometric; the

other is about undoing a derivative. This chapter builds the tool that answers the first question directly, and only much later reveals, in Section 8.6, why it secretly answers the second one too. That connection, properly proven, is the Fundamental Theorem of Calculus (Chapter 10) — one of the most surprising results in all of mathematics, and worth not spoiling here. For now, keep both questions in mind and let the geometric one lead.

Even restricted to the geometric question, there is no single canonical answer. History has produced several different precise definitions of "the integral," each an attempt to make "area" (or, as it will turn out, "accumulation") rigorous for a wider class of functions than the one before it. Cauchy's definition, built first below, requires f to be continuous and uses evenly spaced rectangles. Riemann's, covered later in this chapter, allows unevenly spaced partitions and some discontinuous functions Cauchy's definition cannot touch. Still more general theories — Lebesgue's integral, and gauge (Henstock–Kurzweil) integration, among them — push further still, each at some real cost, and are left to later coursework in real analysis. Keep this in mind as new notation appears over the rest of this chapter: $\int_a^b f(x)\,dx$, a Riemann sum, a partition, a mesh. None of it *is* the integral; each is one particular formalization of an informal idea this chapter is about to make precise.

That precision starts with the simplest possible idea: approximate the region under the curve with rectangles.

8.1 Riemann Sums with Rectangles

To approximate the area under the curve $y = f(x)$ between $x = a$ and $x = b$, we divide the interval $[a, b]$ into n equal subintervals, each of width

$$\Delta x = \frac{b - a}{n}\,.$$

The partition points are

$$x_i = a + i\,\Delta x, \quad i = 0, 1, \ldots, n\,,$$

so $x_0 = a$ and $x_n = b$. On each subinterval $[x_{i-1}, x_i]$ we erect a rectangle whose width is Δx and whose height is the value of f at some chosen point in the subinterval. Summing the areas of all n rectangles gives a *Riemann sum* approximating the integral. The choice of where in each subinterval to evaluate f gives rise to several rules.

8.1.1 Left and Right Sums

The simplest choices are the left endpoint x_{i-1} and the right endpoint x_i of each subinterval.

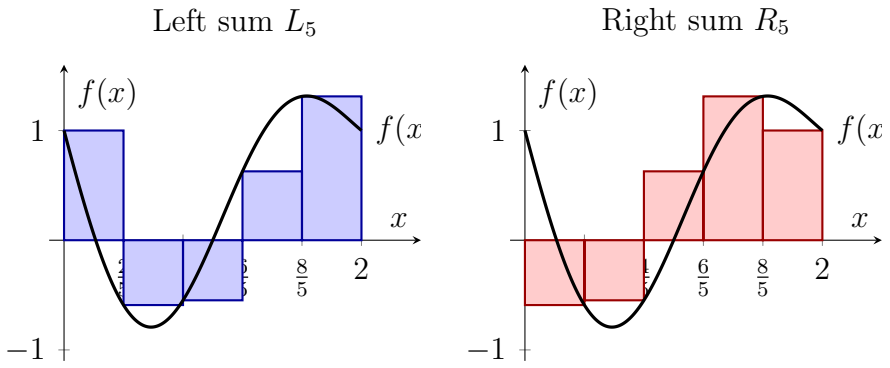


Figure 8.2: Left and right Riemann sums for $f(x) = (x-1)^2 - \sin(\pi x)$ on $[0, 2]$ with $n = 5$ subdivisions; true integral $= \frac{2}{3} \approx 0.667$. Both $L_5 = R_5 = 0.72$ because $f(0) = f(2) = 1$; the individual rectangle placements differ, but their totals coincidentally match. Bars below the x-axis contribute negative signed area.

Definition 8.1.1 (*Left and Right Riemann Sums*)

The *left Riemann sum* with n subintervals is

$$L_n = \sum_{i=1}^{n} f(x_{i-1})\,\Delta x\,.$$

The *right Riemann sum* with n subintervals is

$$R_n = \sum_{i=1}^{n} f(x_i)\,\Delta x\,.$$

Whether the sum over- or underestimates the true area depends on the shape of f, not simply on the left-or-right choice. Figure 8.2 illustrates this with $f(x) = (x-1)^2 - \sin(\pi x)$ on $[0, 2]$: the function dips below zero on part of the interval, so some rectangles contribute negative signed area. In all cases the error shrinks as n grows.

8.1.2 The Midpoint Rule

A better choice is the midpoint of each subinterval, $\bar{x}_i = \frac{x_{i-1}+x_i}{2} = a + (i - \frac{1}{2})\Delta x$. Using the midpoint tends to cancel the over- and underestimation of the left and right sums, since the rectangle sticks above the curve on one side of the midpoint by roughly the same amount it falls short on the other.

Definition 8.1.2 (*Midpoint Rule*)

The *midpoint Riemann sum* with n subintervals is

$$M_n = \sum_{i=1}^{n} f\Big(a + \Big(i - \tfrac{1}{2}\Big)\Delta x\Big)\,\Delta x\,.$$

Like the left and right sums, the midpoint rule evaluates f at an actual point in the subinterval, so the top edge of each rectangle lies on the curve by construction.

8.2 The Trapezoidal Rule

Instead of a rectangle on each subinterval, we fit a trapezoid whose two parallel sides have heights $f(x_{i-1})$ and $f(x_i)$. The area of a trapezoid with parallel sides h_1 and h_2 and width w is $\frac{h_1+h_2}{2}\cdot w$, so each trapezoid contributes $\frac{f(x_{i-1})+f(x_i)}{2}\,\Delta x$. Summing and simplifying:

Definition 8.2.1 (*Trapezoidal Rule*)

$$T_n = \frac{\Delta x}{2}\Big(f(x_0) + 2f(x_1) + 2f(x_2) + \cdots + 2f(x_{n-1}) + f(x_n)\Big).$$

Note that $T_n = \frac{L_n+R_n}{2}$: the trapezoidal rule is exactly the average of the left and right sums. Because a trapezoid fits any linear function exactly, T_n is exact whenever f is linear, and its error on smooth functions comes only from the curvature.

The trapezoidal rule does not evaluate f at a single chosen point per subinterval: it uses the average of the two endpoint values. This raises a natural question: does the trapezoid area correspond to any rectangle whose top edge lies on the curve? The Intermediate Value Theorem answers yes.

Theorem 8.2.2 (*Intermediate Value Theorem*)

If f is continuous on $[a, b]$ and v is any value satisfying $f(a) \le v \le f(b)$ (or $f(b) \le v \le f(a)$), then there exists $c \in [a, b]$ such that $f(c) = v$.

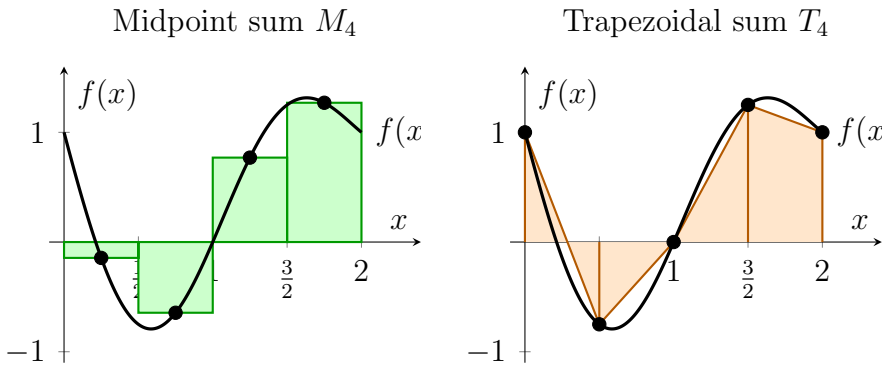


Figure 8.3: Midpoint rule $M_4 = 0.625$ (left, green) and trapezoidal rule $T_4 = 0.75$ (right, orange) for $f(x) = (x-1)^2 - \sin(\pi x)$ on $[0, 2]$ with $n = 4$; true integral $= \frac{2}{3} \approx 0.667$. Dots mark the midpoint evaluation points. The midpoint rule underestimates; the trapezoidal rule overestimates; the trapezoidal error is roughly twice the midpoint error — the relationship Simpson's rule exploits. Compare with Figure 8.2.

Geometrically: a continuous curve cannot jump from height $f(a)$ to height $f(b)$ without passing through every intermediate value. The average $\frac{f(x_{i-1})+f(x_i)}{2}$ lies between $f(x_{i-1})$ and $f(x_i)$, so Theorem 8.2.2 guarantees a point $c_i \in [x_{i-1}, x_i]$ with

$$f(c_i) = \frac{f(x_{i-1}) + f(x_i)}{2}\,.$$

Therefore the trapezoid area and the rectangle area are equal:

$$\frac{f(x_{i-1}) + f(x_i)}{2}\,\Delta x \;=\; f(c_i)\,\Delta x\,.$$

The top edge of this rectangle lies on the curve at the interior point c_i. In other words, the trapezoidal rule is a legitimate Riemann sum in disguise, with sample points c_i that we cannot locate explicitly but whose existence is guaranteed by continuity.

8.3 Simpson's Rule

Simpson's rule fits a parabola through each consecutive triple of partition points, two subintervals at a time. Accordingly n must be even.

Definition 8.3.1 (*Simpson's Rule*)

With n subintervals (n even),

$$S_n = \frac{\Delta x}{3} \sum_{j=0,\,2,\,\ldots}^{n-2} \Big(f(x_j) + 4f(x_{j+1}) + f(x_{j+2}) \Big) .$$

Because a parabola fits any polynomial of degree at most 3 exactly, S_n is exact for cubics as well as quadratics. The derivation of the formula and the analysis of why it converges so much faster than the other methods are deferred to the homework and to Chapter 12.

Figure 8.4 shows S_4 applied to $f(x) = (x-1)^2 - \sin(\pi x)$ on $[0, 2]$. Two violet parabolic arcs are fitted through the five partition points, two subintervals at a time; dots mark where the parabolas are pinned to f. Compare the curved tops with the straight-line tops of the trapezoidal rule (Figure 8.3): the parabolas follow f much more closely, which is why Simpson's rule converges at $O(1/n^4)$ rather than $O(1/n^2)$. For this function S_4 is exact: $(x-1)^2$ is a quadratic (which Simpson integrates exactly) and the Simpson approximation of $\int_0^2 \sin(\pi x)\,dx$ also evaluates to 0.

8.4 The Cauchy Integral

As $n \to \infty$, all of the approximations above — left sum, right sum, midpoint, trapezoid, Simpson's — converge to the same limit, provided

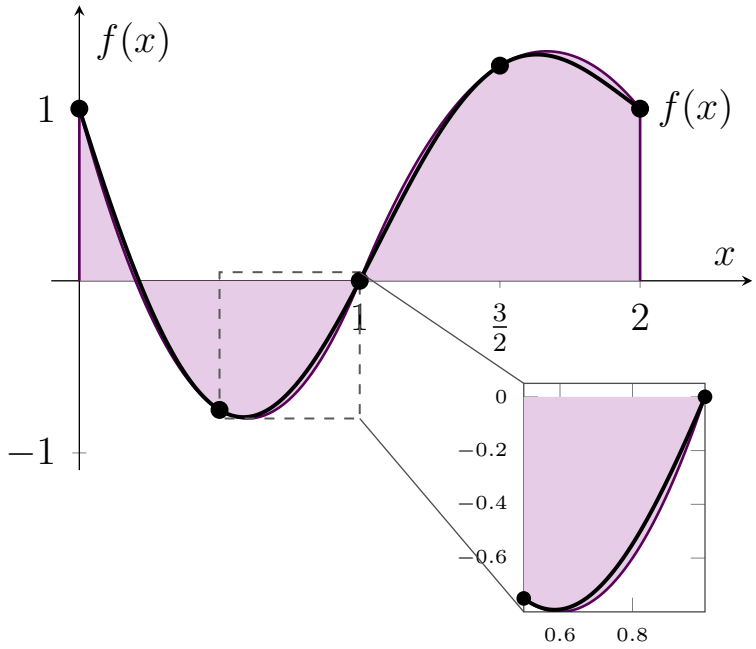


Figure 8.4: Simpson's rule S_4 for $f(x) = (x-1)^2 - \sin(\pi x)$ on $[0, 2]$ with $n = 4$ subintervals. Violet parabolic arcs are pinned to f at the partition points (dots).

f is continuous on $[a, b]$. This common limit is the *definite integral* of f, defined as follows.

Definition 8.4.1 (*The Cauchy Integral*)

Let f be continuous on $[a, b]$. Divide $[a, b]$ into n equal subintervals of width $\Delta x = (b-a)/n$, with partition points $x_i = a + i\,\Delta x$ for $i = 0, 1, \ldots, n$. Choose any sample point $x_i^* \in [x_{i-1}, x_i]$ in each subinterval. The *Cauchy integral* of f over $[a, b]$ is

$$\int_a^b f(x)\,dx \;=\; \lim_{n\to\infty} \sum_{i=1}^{n} f(x_i^*)\,\Delta x\,.$$

The limit exists and is the same regardless of how the sample points are chosen.

The left sum ($x_i^* = x_{i-1}$), right sum ($x_i^* = x_i$), and midpoint rule ($x_i^* = \bar{x}_i$) fit this definition directly. The trapezoidal rule and Simpson's rule also converge to the same limit, even though they use multiple function values per subinterval rather than a single sample point. The justification is the Intermediate Value Theorem (Theorem 8.2.2): the weighted average of function values used by each rule lies between the minimum and maximum of f on the subinterval, so the IVT guarantees a point c_i in the subinterval where $f(c_i)$ equals that weighted average — making the rule equivalent to a standard Riemann sum.

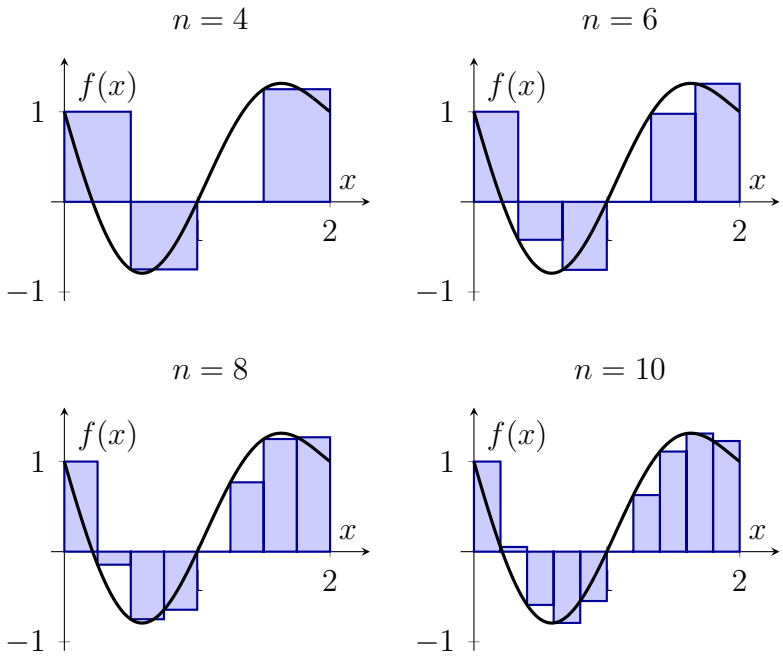


Figure 8.5: Left Riemann sum L_n for $f(x) = (x-1)^2 - \sin(\pi x)$ on $[0, 2]$ with $n = 4, 6, 8, 10$ intervals. As n grows, the rectangles approximate the signed area more closely and L_n converges to the Cauchy integral $\int_0^2 f(x)\,dx = \frac{2}{3}$.

This is what you will implement in Python in the homework: for large n, the sum $\sum_{i=1}^{n} f(x_i^*)\,\Delta x$ approximates the integral to any desired precision (Figure 8.5).

8.5 The Riemann Integral *(Optional)*

The Cauchy integral requires a uniform partition and a continuous function. The *Riemann integral*, developed by Bernhard Riemann in 1854, generalizes both restrictions: it allows arbitrary (non-uniform) partitions and handles a broader class of functions, including some discontinuous ones.

Definition 8.5.1 (*Partition and Mesh*)

A *partition* P of $[a, b]$ is a finite sequence

$$a = x_0 < x_1 < \cdots < x_n = b.$$

Write $\Delta x_i = x_i - x_{i-1}$ for the width of the i-th subinterval. The *mesh* of P is the width of the widest piece:

$$\|P\| = \max_{1 \le i \le n} \Delta x_i.$$

Example 8.5.2 (*Mesh does not always go to zero*)

For each $n \ge 2$, consider the non-uniform partition of $[0, 1]$:

$$P_n = \left\{0,\ \tfrac{1}{2},\ 1 - \tfrac{1}{2^2},\ 1 - \tfrac{1}{2^3},\ \ldots,\ 1 - \tfrac{1}{2^{n-1}},\ 1\right\}.$$

The subinterval widths are $\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \ldots, \frac{1}{2^{n-1}}$: clearly non-uniform, and each new point added is closer to 1 than the last. Yet the

mesh is

$$\|P_n\| = \tfrac{1}{2} \quad \text{for every } n,$$

because the first subinterval $[0, \frac{1}{2}]$ is always present and always the widest. No matter how many points are piled up near $x = 1$, the mesh never shrinks.

This is a warning: a sequence of partitions that keeps adding points is *not* necessarily getting finer in the sense required by the Riemann integral. What matters is that the *widest* piece shrinks to zero. The uniform partition $P_n = \{0, \frac{1}{n}, \frac{2}{n}, \dots, 1\}$ does satisfy $\|P_n\| = \frac{1}{n} \to 0$, because every subinterval shrinks together.

Definition 8.5.3 (*Definite Riemann Integral*)

f is *Riemann integrable* on $[a, b]$ if there exists a number L such that, for every sequence of partitions with $\|P\| \to 0$ and every choice of sample points $x_i^* \in [x_{i-1}, x_i]$, the Riemann sums converge to L:

$$\sum_{i=1}^{n} f(x_i^*)\,\Delta x_i \;\longrightarrow\; L \quad \text{as } \|P\| \to 0\,.$$

This number L is the *definite integral* of f on $[a, b]$, written $\int_a^b f(x)\,dx$.

Figure 8.6 shows an example: the non-uniform partition $\{0,\ 0.3,\ 0.7,\ 1.0,\ 1.2,\ 1.6,\ 2.0\}$ divides $[0, 2]$ into 6 subintervals of unequal width, with mesh $\|P\| = 0.4$. A sample point x_i^* is chosen freely inside each subinterval, and the corresponding rectangles form one Riemann sum.

Every continuous function is Riemann integrable, and for continuous functions the Riemann integral agrees with the Cauchy integral — the generalization costs nothing on the functions engineers encounter.

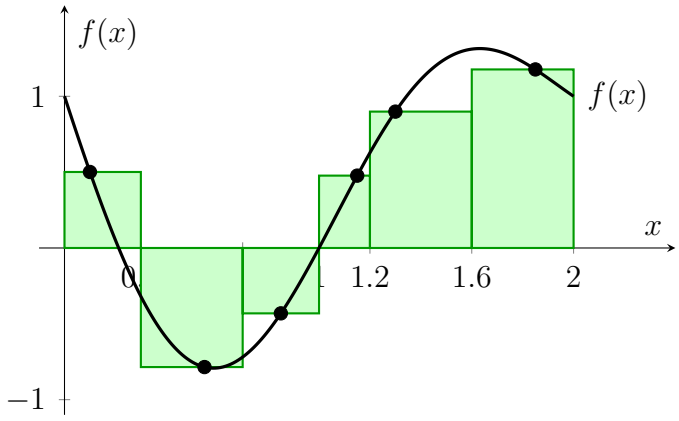


Figure 8.6: A Riemann sum for $f(x) = (x-1)^2 - \sin(\pi x)$ on $[0, 2]$ using the non-uniform partition $\{0,\ 0.3,\ 0.7,\ 1.0,\ 1.2,\ 1.6,\ 2.0\}$ with 6 subintervals of unequal width. The dot in each subinterval marks the chosen sample point x_i^*, which need not be the midpoint or an endpoint. As the mesh $\|P\|$ tends to zero, these sums converge to the definite integral regardless of how the sample points are chosen.

Lemma 8.5.4 (*Cauchy and Riemann Integrals Agree*)

If f is continuous on $[a, b]$, then f is Riemann integrable and its Riemann integral equals its Cauchy integral. In particular, the uniform-partition limit

$$\int_a^b f(x)\,dx \;=\; \lim_{n\to\infty} \sum_{i=1}^{n} f(x_i^*)\,\Delta x$$

gives the correct value for any choice of sample points.

The extra generality is not vacuous: there are functions the Riemann integral handles that the Cauchy integral, as defined above, cannot touch at all.

Example 8.5.5 (*A Function Cauchy Cannot Integrate, But Riemann Can*)

Let

$$f(x) = \begin{cases} 0, & 0 \le x < 1 \\ 1, & 1 \le x \le 2 \end{cases}$$

on $[0, 2]$. Geometrically, the region under f is a unit square sitting on $[1, 2]$, so its area is obviously 1.

The Cauchy integral does not apply. Definition 8.4.1 requires f to be *continuous* on $[a, b]$ before it says anything at all. f jumps from 0 to 1 at $x = 1$, so that hypothesis fails outright — not just the conclusion, the definition itself is silent on this f.

The Riemann integral gives 1, regardless of the partition or the sample points. Let $P = \{0 = x_0 < x_1 < \cdots < x_n = 2\}$ be any partition with mesh $\|P\|$, and let x_i^* be any sample point chosen in each subinterval $[x_{i-1}, x_i]$. Suppose 1 is not itself a partition point (the argument is even simpler if it is), and let x_k be the unique subinterval $[x_{k-1}, x_k]$ containing 1 in its interior. Every subinterval entirely inside $[0, 1)$ has $f \equiv 0$ on it and contributes 0 to the Riemann sum; every subinterval entirely inside $[1, 2]$ has $f \equiv 1$ on it and contributes exactly its width. Only the one straddling subinterval $[x_{k-1}, x_k]$ is ambiguous: it contributes either 0 or Δx_k to the sum, depending on whether $x_k^* < 1$ or $x_k^* \ge 1$. Comparing the Riemann sum to the true area 1 — which is exactly the total width of the subintervals lying in $[1, 2]$, plus whatever $[x_{k-1}, x_k]$ *should* contribute — the only possible discrepancy is on that one straddling subinterval, so

$$\left| \sum_{i=1}^{n} f(x_i^*)\,\Delta x_i \;-\; 1 \right| \;\le\; \Delta x_k \;\le\; \|P\|\,.$$

As $\|P\| \to 0$ this bound forces the sum to 1, for *every* sequence of partitions with shrinking mesh and *every* choice of sample points

— exactly what Definition 8.5.3 requires. So f is Riemann integrable on $[0, 2]$ with $\int_0^2 f(x)\,dx = 1$, even though it never qualified for the Cauchy integral in the first place.

Even Riemann's more general definition has its own limits, though: there exist functions for which the Riemann integral does not exist: the sums may converge to different values depending on the partition or sample-point choice. For some such functions a clear notion of area still exists — the Riemann integral simply cannot capture it. This limitation led to more general theories such as the Lebesgue integral, studied in courses on real analysis and measure theory.

8.6 Area under the Curve of a Function

When $f(x) \geq 0$ on $[a, b]$, the definite integral has a direct geometric meaning: it equals the area of the region bounded above by $y = f(x)$, below by the x-axis, and on the sides by $x = a$ and $x = b$.

When $f(x) < 0$ on part of $[a, b]$, the rectangles in the Riemann sum have negative height, contributing negative area. The integral then measures *signed area*: regions above the x-axis count positively, regions below count negatively. The integral of $\sin x$ over a full period $[0, 2\pi]$ is zero, for example, because the positive and negative regions cancel exactly.

Figure 8.7 shows this cancellation for $f(x) = \sin x$.

A surprising connection, not yet visible. Nothing said so far connects area to the derivative. We built the integral from Riemann sums — adding up thin rectangles — with no reference to slopes, tangent lines, or rates of change. Computed purely as a geometric quantity, using only what this chapter has given you, $\int_a^b f(x)\,dx$ gives no hint that it has anything to do with $f'(x)$.

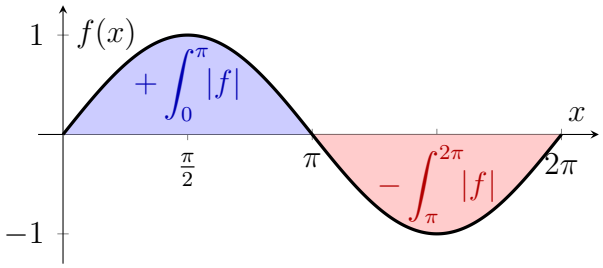


Figure 8.7: Signed area for $f(x) = \sin x$ on $[0, 2\pi]$. The blue lobe on $[0, \pi]$ contributes positive area; the red lobe on $[\pi, 2\pi]$ contributes negative area. The two lobes have equal magnitude, so $\int_0^{2\pi} \sin x \, dx = 0$, even though the curve is never identically zero on the interior of either half.

Yet it does, intimately. The Fundamental Theorem of Calculus (Chapter 10) shows that accumulating area and taking derivatives are inverse operations of one another. This is genuinely surprising: area is a static, geometric quantity — it does not care how you got there, only what shape you end up with — while the derivative is a local, dynamic quantity, the instantaneous rate of change at a single point. There is no obvious reason these should be related at all, let alone be exact inverses. Mathematicians of antiquity computed areas (Archimedes' method of exhaustion for a circle or a parabolic segment) for nearly two thousand years without anyone connecting that work to tangent lines. It took Newton and Leibniz, independently and famously controversially, to recognize in the 17th century that these were two views of the same underlying operation — a discovery that unified what had been separate branches of geometry into the single theory now called calculus. Keep this question in mind through the rest of this chapter: why should adding up thin rectangles have anything at all to do with slopes?

A recurring intuition. Reading $\int_a^b f(x)\,dx$ as "the area under this curve" is more than a geometric curiosity: it is often the fastest route to an antiderivative that resists direct algebraic attack. When a table lookup or a substitution does not obviously apply, it is worth

asking instead whether the region under the curve is a recognizable shape — a triangle, a trapezoid, a slice of a circle — whose area can be computed geometrically instead of symbolically. Section 13.2 uses exactly this trick to evaluate $\int \sqrt{1-x^2}\,dx$, an integral with no direct table entry, by recognizing the region under the curve as part of a circle.

8.7 Properties of the Definite Integral

Once the integral is defined as the limit of a Riemann sum, several properties become self-evident from pictures, or directly from the sum itself. We state each one first in words, then formally.

Renaming the integration variable. The symbol x in $\int_a^b f(x)\,dx$ names the sample points and the subinterval widths in the Riemann sum $\sum_i f(x_i^*)\,\Delta x$ used to define the integral. Relabeling every occurrence of x in that sum — to t, or u, or any symbol not already in use — produces exactly the same partition, the same sample points, and therefore the same limit. The letter x carries no meaning outside the integral; only f, a, and b determine its value.

Proposition 8.7.1 (*Renaming the Integration Variable*)

For any function f continuous on $[a,b]$ and any symbol t not otherwise in use,

$$\int_a^b f(x)\,dx = \int_a^b f(t)\,dt\,.$$

This freedom matters whenever one letter is needed for two different jobs. Defining a function by accumulation, for example, requires integrating up to a variable endpoint: $g(x) = \int_0^x f(x)\,dx$ would use x both as the endpoint and as the variable summed over inside the in-

tegral — two different roles crammed into one symbol. Renaming the integration variable, $g(x) = \int_0^x f(t)\,dt$, keeps the roles separate: x is the point where g is evaluated, and t is the variable that sweeps from 0 to x inside the sum. We rely on this renaming freely from Chapter 10 onward.

Zero-width integral. If the upper and lower limits are equal, the interval $[a, a]$ has zero width. There are no rectangles to sum; an empty sum is zero.

Proposition 8.7.2 (*Zero-Width Integral*)

For any function f and any $a \in \mathbb{R}$,

$$\int_a^a f(x)\,dx = 0\,.$$

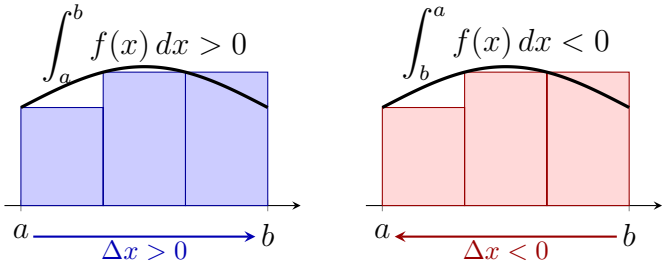


Figure 8.8: Reversing the limits negates the integral. The same three rectangles appear in both panels; only the direction of traversal differs. Left-to-right steps give $\Delta x > 0$; right-to-left steps give $\Delta x < 0$, flipping every rectangle's signed area.

Reversing the limits. When we write $\int_a^b f(x)\,dx$ with $a < b$, each rectangle has a positive width $\Delta x = (b-a)/n > 0$, and the sum is positive (for a positive function). If we swap the limits and write $\int_b^a$, we are traversing the same rectangles right to left: each step is now $\Delta x = (a-b)/n < 0$. The heights $f(x_i)$ are identical, but every width

has changed sign, so the entire sum — and its limit — changes sign. Figure 8.8 shows both traversals side by side.

Proposition 8.7.3 (*Reversing the Limits of Integration*)

If f is integrable on $[a, b]$, then

$$\int_a^b f(x)\,dx = -\int_b^a f(x)\,dx\,.$$

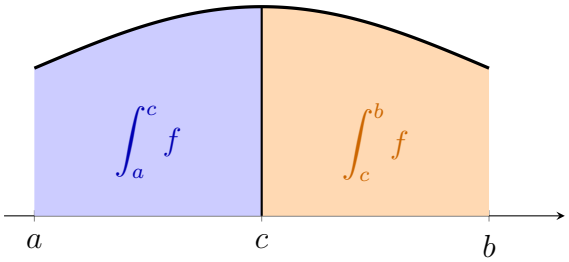


Figure 8.9: A vertical line at c splits the area under f into a left piece (blue, $\int_a^c f$) and a right piece (orange, $\int_c^b f$). Their sum equals the total area $\int_a^b f$.

Splitting the interval. Consider the area under f from a to b. If c lies between a and b, drawing a vertical line at c splits the shaded region into two pieces: the area from a to c, and the area from c to b. The total is the sum of the parts, as shown in Figure 8.9.

Proposition 8.7.4 (*Splitting the Interval*)

If f is integrable on an interval containing a, b, and c, then

$$\int_a^b f(x)\,dx = \int_a^c f(x)\,dx + \int_c^b f(x)\,dx\,.$$

Note that c need not lie between a and b; the identity holds in general by the sign-flipping rule of Proposition 8.7.3.

8.8 Linearity of the Integral

Linearity of the integral is not a deep theorem — it is arithmetic on rectangles. Consider approximating $\int_a^b (\alpha f(x) + \beta g(x))\,dx$ with a Riemann sum. Each rectangle has width Δx and height $\alpha f(x_i) + \beta g(x_i)$, so its area is

$$(\alpha f(x_i) + \beta g(x_i))\,\Delta x = \alpha\, f(x_i)\,\Delta x + \beta\, g(x_i)\,\Delta x\,.$$

The second step is just the distributive law of arithmetic. Summing over all rectangles,

$$\sum_i (\alpha f(x_i) + \beta g(x_i))\,\Delta x = \alpha \sum_i f(x_i)\,\Delta x \;+\; \beta \sum_i g(x_i)\,\Delta x\,.$$

Taking the limit as $\Delta x \to 0$, the limit distributes across the sum and the constants α, β pass through unchanged:

$$\int_a^b (\alpha f(x) + \beta g(x))\,dx = \alpha \int_a^b f(x)\,dx + \beta \int_a^b g(x)\,dx\,.$$

In other words: you can distribute the integral sign across a sum and pull constants outside, handling each term independently. This is the same pattern as linearity of the derivative (Section 1.2.7), and for the same reason: both operators interact with addition and scalar multiplication in the simplest possible way. The payoff is that any time you face an integral of a linear combination — a polynomial, a Taylor series, a Fourier series — you can integrate one term at a time.

8.9 Convergence

The three methods — rectangles, trapezoids, and Simpson's rule — all converge to the same limit as $n \to \infty$, but at very different rates. Left and right rectangle sums improve proportionally to $1/n$: doubling the number of subdivisions halves the error. The trapezoidal rule and the midpoint rule improve proportionally to $1/n^2$: doubling n divides the error by four. However, they are not equally good — the midpoint rule incurs half the error of the trapezoidal rule for the same n. This surprises most people: the trapezoid's slanted top follows the curve's slope and looks like a better fit, but looking like the curve and enclosing the correct area are different things. The midpoint rectangle overshoots on one side of the midpoint and undershoots symmetrically on the other, so the two errors cancel; the trapezoid has no such cancellation. Simpson's rule improves proportionally to $1/n^4$: doubling n divides the error by sixteen.

For smooth functions — and especially for polynomials — the differences in practice are far larger than these rates suggest. The trapezoidal rule integrates linear functions *exactly*, so its error on a polynomial comes only from higher-order terms. Simpson's rule integrates polynomials of degree at most three exactly. The homework exercises make these differences concrete: students will find that reaching a given precision requires vastly different values of n depending on the method.

8.10 Numerical Integration in Python

The homework exercises ask you to implement all three methods as Python loops, measure how many subdivisions each method needs to reach a target precision, and compare the results. Use `scipy.integrate.quad` as the oracle: the error of your implementation at a given n is the absolute difference between your result and `quad`'s result.

8.10.1 Implementing the Functions

The homework file `riemann_sum.py` contains five functions for you to complete: `left_sum`, `right_sum`, `mid_sum`, `trapezoid_sum`, and `simpsons_sum`. Each has the same signature:

```
def left_sum(f, left, right, n):
```

where `f` is a unary function, `left` and `right` are the endpoints of the interval, and `n` is the number of subintervals. Each function returns the numerical approximation of $\int_{\texttt{left}}^{\texttt{right}} f(x)\,dx$.

The step size is `dx = (right - left) / n`. Each method iterates over exactly n subintervals; the question is which endpoint of each subinterval to evaluate.

- **Left sum.** Evaluate `f` at the *left* endpoint of each subinterval: $k = 0, 1, \ldots, n-1$ (*excluding* n). In Python: `range(n)`.
- **Right sum.** Evaluate `f` at the *right* endpoint: $k = 1, 2, \ldots, n$ (*including* n). In Python: `range(1, n+1)`.
- **Midpoint sum.** Evaluate `f` at the center of each subinterval, `left + (k + 0.5)*dx`, for $k = 0, 1, \ldots, n-1$ (*excluding* n). In Python: `range(n)`.
- **Trapezoidal sum.** Each subinterval contributes $\frac{f(x_k)+f(x_{k+1})}{2} \cdot \Delta x$, so you need *both* endpoints: iterate $k = 0, 1, \ldots, n-1$ (*excluding* n) and use `f(left + k*dx)` and `f(left + (k+1)*dx)` inside the loop.
- **Simpson's rule.** n must be even. Iterate over *pairs* of subintervals: $k = 0, 2, 4, \ldots, n-2$ (*excluding* n).

In Python: `range(0, n-1, 2)`. Each pair contributes $\frac{\Delta x}{3}(f(x_k) + 4f(x_{k+1}) + f(x_{k+2}))$.

8.10.2 Measuring Convergence

The function `expected_convergence_order(simp_n, mid_n, trap_n)` takes the minimum number of subdivisions each of the three methods needs to reach a target precision ε, and returns `True` if those counts are in the order the theory predicts.

From the Convergence section: the trapezoidal and midpoint rules both converge as $O(1/n^2)$, but the midpoint error is half the trapezoidal error; Simpson's rule converges as $O(1/n^4)$. Therefore Simpson's needs the fewest subdivisions to reach any fixed ε, and the midpoint rule needs fewer than the trapezoidal rule:

$$\texttt{simp\_n} \leq \texttt{mid\_n} \leq \texttt{trap\_n}.$$

To find the minimum n for a given method and tolerance, start with a small n (say 2) and double it until the error drops below ε. Doubling keeps n even, which Simpson's rule requires.

8.10.3 Using `scipy.integrate.quad` as Oracle

`scipy` is **not available** on the cloud service used to grade your submission. Do not import it in `riemann_sum.py` or any file you submit — the import statement itself will cause the grader to fail, even if you never call any `scipy` function. Use `scipy` only in your own local scratch scripts.

`scipy.integrate.quad` computes definite integrals to near machine precision. Use it as the reference value against which to measure the error of your implementations:

```python
# local scratch only -- do not submit
from scipy.integrate import quad

def error(method, f, a, b, n):
    reference, _ = quad(f, a, b)
    return abs(method(f, a, b, n) - reference)
```

The second return value of `quad` is an internal error estimate that you can ignore here (hence the `_`).

A good test function is $f(x) = \sin(\pi x)$ on $[0, 1]$, whose exact integral is $2/\pi \approx 0.6366$. Pick a target precision such as $\varepsilon = 10^{-6}$ and find the minimum n each method needs to achieve `error(...) < eps`. You should find that Simpson's rule reaches $\varepsilon = 10^{-6}$ with far fewer subdivisions than the rectangle or trapezoidal methods.

? Open Questions

- The Cauchy Integral (Definition 8.4.1) asserts that for continuous f the limit "exists and is the same regardless of how the sample points are chosen," but never says why. What property of a function continuous on a *closed* interval guarantees this — and why might it fail on an open interval, or for a continuous function on an unbounded interval?
- Section 8.5 mentions in passing that "there exist functions for which the Riemann integral does not exist," and that this gap motivated the Lebesgue integral. Can you construct such a function on $[0, 1]$? (Hint: think about a function that behaves one way on the rationals and another way on the irrationals, and reconsider Example 8.5.2 — does making the partition finer always help?)
- Example 8.5.2 shows that piling up more partition points does not by itself guarantee convergence — only a shrinking *mesh* does. Can you think of an algorithm or data structure where doing more work, or adding more steps, feels like it should improve the worst case but provably does not, unless the work is distributed a particular way?

Chapter 9

NumPy: Integrals Involving Inverse Trig Functions

Chapter Objectives

- Identify the domains and ranges of the six inverse trigonometric functions and explain why restricted domains are necessary.
- Graph $\arctan$ and interpret its horizontal asymptotes as $x \to \pm\infty$.
- Verify the derivative of $\arctan$ numerically using finite differences in NumPy.
- Recognize the definite integral as an inner product on a function space, and use the induced norm and metric to make convergence of functions precise.
- Use NumPy to evaluate definite integrals whose antiderivatives involve inverse trigonometric functions.
- Discover antiderivatives empirically by comparing numerical integrals against symbolic guesses.

9.1 What is NumPy?

NumPy is not part of the Python standard library. If it is not already available in your environment, install it once with:

```
pip install numpy
```

After that, import it at the top of every script that uses it. The conventional alias is `np`:

```
import numpy as np
```

Activity 1

Do: Open a Python session and run `import numpy as np`. If you see a `ModuleNotFoundError`, run `pip install numpy` in a terminal and restart the session. Confirm success with `print(np.__version__)` — you should see a version string such as `2.0.0`.

Discuss: NumPy is not part of the Python standard library, so it must be installed separately and is not available everywhere. Why does this matter for your Oral Defense and Intra submissions? What would happen if you accidentally left an `import numpy` at the top of a submitted file?

NumPy is a Python library for numerical computation. Its central data structure is the `ndarray`: a fixed-length, homogeneous array of floating-point (or integer) values stored contiguously in memory. Unlike a Python list, an `ndarray` supports *vectorized operations* — arithmetic and mathematical functions applied to every element simultaneously, without an explicit loop.

```python
import numpy as np

# Python list: loop required
xs = [0.0, 0.5, 1.0, 1.5, 2.0]
ys = [x**2 for x in xs]           # explicit loop

# NumPy array: vectorized
xs = np.linspace(0, 2, 5)         # [0.0, 0.5, 1.0, 1.5, 2.0]
ys = xs**2                         # [0.0, 0.25, 1.0, 2.25, 4.0]
```

`np.linspace(a, b, n)` produces an array of n evenly spaced values from a to b inclusive. This is the standard way to build a grid of sample points.

NumPy also provides vectorized versions of all the standard mathematical functions — `np.sin`, `np.cos`, `np.exp`, `np.log`, `np.arctan`, and so on — each of which accepts an `ndarray` and returns an `ndarray` of the same shape.

```python
xs = np.linspace(-3, 3, 300)
ys = np.arctan(xs)                # arctan evaluated at 300 points
```

This vectorized style maps directly onto the mathematical idea of evaluating a function $f : \mathbb{R} \to \mathbb{R}$ at every point of a grid, and it is the computational pattern we will use throughout this chapter.

NumPy and SciPy are forbidden in Oral Defense and Intra submissions. The graded coding submissions—`derivative.py` (Oral Defense) and `riemann_sum.py` (Intra)—require you to build everything from first principles, without relying on numeric libraries. Neither `numpy` nor `scipy` may appear in any file you submit.

This is not merely a stylistic constraint. The cloud grading service that evaluates your submission does not have these libraries installed. If an `import numpy` or `import scipy` statement appears anywhere in your submitted files—even at the top of a helper module, even if that import is never used—the grader will fail to load your code and award zero points.

NumPy and SciPy are available for the TP (lab) sessions in this chapter, where numerical computation is the whole point. Keep your TP scratch scripts strictly separate from your Oral Defense and Intra submission files.

9.2 Math Notation vs. Python Names

Moving between a textbook and a Python interpreter requires translating between two sets of naming conventions that are almost — but not quite — consistent.

9.2.1 Trigonometric Functions

The six trig functions are spelled the same in both worlds; only the call syntax differs.

Math	**`math` module**	**NumPy**
$\sin x$	`math.sin(x)`	`np.sin(x)`
$\cos x$	`math.cos(x)`	`np.cos(x)`
$\tan x$	`math.tan(x)`	`np.tan(x)`
$\cot x$	—	—
$\sec x$	—	—
$\csc x$	—	—

Neither library provides `cot`, `sec`, or `csc` directly. Compute them as reciprocals: `1/np.tan(x)`, `1/np.cos(x)`, `1/np.sin(x)`.

9.2.2 Inverse Trigonometric Functions

Here the two libraries disagree with each other and with common textbook notation.

Math	**`math` module**	**NumPy**
$\arcsin x$	`math.asin(x)`	`np.arcsin(x)`
$\arccos x$	`math.acos(x)`	`np.arccos(x)`
$\arctan x$	`math.atan(x)`	`np.arctan(x)`
	`math.atan2(y,x)`	`np.arctan2(y,x)`

The `math` module drops the *arc* prefix and contracts to `asin`, `acos`, `atan`; NumPy keeps the full spelling `arcsin`, `arccos`, `arctan`. Both compute exactly the same mathematical function.

9.2.3 The $\cos^{-1}$ Notation and Why It Is Confusing

In many textbooks — and on most calculators — the inverse cosine is written $\cos^{-1} x$ rather than $\arccos x$. This notation is standard and you will encounter it constantly. It is also, from a purely logical standpoint, inconsistent.

Where is the inconsistency? For any number x, the exponent -1 denotes the reciprocal:

$$x^{-1} = \frac{1}{x}.$$

For trig functions, the positive exponents follow this pattern too:

$$\cos^2 x = (\cos x)^2, \qquad \cos^3 x = (\cos x)^3, \qquad \ldots$$

By strict analogy, $\cos^{-1} x$ should mean $(\cos x)^{-1} = \sec x$. It does not. Instead, $\cos^{-1}$ is a notational exception carved out historically for the

inverse *function*:

$$\cos^{-1} x \;=\; \arccos x, \quad \text{not} \quad \frac{1}{\cos x}.$$

The reciprocal $\frac{1}{\cos x}$ has its own name: $\sec x$.

This creates a genuinely ambiguous corner case: what does $\cos^{-2} x$ mean? The literature is inconsistent. Some authors write it to mean $(\arccos x)^2$; others mean $(\cos x)^{-2} = \sec^2 x$. The safest choice is to avoid the exponent notation altogether for negative powers and write $\arccos x$ or $\sec x$ explicitly.

This handout always uses arcsin, arccos, arctan (and similarly for arccot, arcsec, arccsc) instead of the $\sin^{-1}$, $\cos^{-1}$, $\tan^{-1}$ notation. When you read other sources, mentally translate $\cos^{-1} x \mapsto \arccos x$, and do not confuse it with $\sec x$.

9.2.4 Logarithms: ln vs. log

In calculus courses, $\ln x$ denotes the *natural* logarithm (base e), and $\log x$ often means the same thing. In computer science, $\log x$ usually means $\log_2 x$ (base 2) because algorithm complexity is measured in bits. In a calculator or spreadsheet, $\log x$ often means $\log_{10} x$ (base 10).

Python follows the calculus convention:

Math	`math` module	NumPy
$\ln x$	`math.log(x)`	`np.log(x)`
$\log_{10} x$	`math.log10(x)`	`np.log10(x)`
$\log_2 x$	`math.log2(x)`	`np.log2(x)`
$\log_b x$	`math.log(x, b)`	`np.log(x)/np.log(b)`

`math.log(x)` with no second argument returns $\ln x$. Passing a second argument `b` returns $\log_b x$. NumPy has no two-argument `np.log`; use the change-of-base formula `np.log(x)/np.log(b)` instead.

9.3 Domain of Inverse Trig Functions

A function has a *true* inverse only if it is one-to-one. The six trigonometric functions are periodic and therefore *not* one-to-one on their natural domains. To define arcsin, arccos, and the rest, we first *restrict* each function to a carefully chosen interval on which it is one-to-one, then invert only that piece.

Function	Restricted domain	Range of inverse
arcsin	$[-\frac{\pi}{2},\ \frac{\pi}{2}]$	$[-1,\ 1]$
arccos	$[0,\ \pi]$	$[-1,\ 1]$
arctan	$(-\frac{\pi}{2},\ \frac{\pi}{2})$	$\mathbb{R}$
arccot	$(0,\ \pi)$	$\mathbb{R}$
arcsec	$[0, \frac{\pi}{2}) \cup (\frac{\pi}{2}, \pi]$	$(-\infty, -1] \cup [1, \infty)$
arccsc	$[-\frac{\pi}{2}, 0) \cup (0, \frac{\pi}{2}]$	$(-\infty, -1] \cup [1, \infty)$

The restricted domains are *conventions*, not theorems. Different textbooks (and different software libraries) occasionally make different choices. NumPy follows the standard mathematical convention for `np.arcsin`, `np.arccos`, and `np.arctan`; arccot, arcsec, and arccsc have no dedicated NumPy function and must be expressed in terms of the three that exist:

```python
import numpy as np

# arccot(x) = arctan(1/x)
arccot = lambda x: np.arctan(1 / x)   # valid for x != 0

# arcsec(x) = arccos(1/x)
arcsec = lambda x: np.arccos(1 / x)

# arccsc(x) = arcsin(1/x)
arccsc = lambda x: np.arcsin(1 / x)
```

9.3.1 `math.atan`, `np.arctan`, and `math.atan2`

Python gives you three arctangent-related functions, and the differences matter.

`math.atan(x)` is the standard-library scalar arctangent. It accepts a single `float` and returns a single `float` in $(-\frac{\pi}{2}, \frac{\pi}{2})$. It is the same mathematical function as arctan, but it cannot accept an `ndarray`.

`np.arctan(x)` is the NumPy vectorized version. It applies the same function element-wise to an `ndarray` (or a scalar), returning an `ndarray` of the same shape. Use this whenever your input is a grid of sample points.

```python
import math, numpy as np

math.atan(1.0)                # 0.785... (pi/4)

xs = np.linspace(-2, 2, 5)
np.arctan(xs)                 # array of 5 values
# math.atan(xs)               # TypeError: not a scalar
```

`math.atan2(y, x)` is a different function entirely. It computes the angle $\theta \in (-\pi, \pi]$ of the vector (x, y) measured from the positive x-axis. Unlike $\arctan(y/x)$, it uses *both* components to determine the correct quadrant, and it handles $x = 0$ without raising an error.

```python
import math

math.atan2(1,  1)   # pi/4  (first quadrant,  northeast)
math.atan2(1, -1)   # 3*pi/4  (second quadrant, northwest)
math.atan2(-1, -1)  # -3*pi/4  (third quadrant,  southwest)
math.atan2(-1,  1)  # -pi/4 (fourth quadrant, southeast)
math.atan2(1,  0)   # pi/2  (straight up; atan(1/0) would crash)
```

The rule of thumb: use `np.arctan` when you have a single ratio y/x and do not need quadrant information; use `math.atan2(y, x)` when you have separate y and x coordinates and need the full $(-\pi, \pi]$ range. NumPy provides `np.arctan2(y, x)` as the vectorized counterpart of `math.atan2`.

A common mistake is to assume that $\arcsin(\sin x) = x$. This is true *only* when $x \in [-\frac{\pi}{2}, \frac{\pi}{2}]$. Outside that interval, `np.arcsin(np.sin(x))` returns a value in $[-\frac{\pi}{2}, \frac{\pi}{2}]$ that is *not* equal to `x`. This is not a NumPy bug — it is the definition of arcsin. The restricted domain is the price of having an inverse at all.

A second, subtler domain violation arises from floating-point round-off. You encountered this in the Linear Algebra course when implementing the `Vector.angle` method. The angle between two vectors is defined by

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

so $\theta = \arccos\left(\frac{u \cdot v}{\|u\|\,\|v\|}\right)$. Mathematically the ratio is always in $[-1, 1]$, but floating-point arithmetic can produce a result such as `1.0000000002`, which causes `math.acos` to raise a `ValueError`. The fix in the Linear Algebra handout used a three-way `if` / `elif` / `else` guard.

With NumPy the standard idiom is `np.clip`, which clamps every element of an array into a specified interval before passing it to `np.arccos`:

```python
import numpy as np

def angle_between(u, v):
    ratio = np.dot(u, v) / (np.linalg.norm(u) * np.linalg.norm(v))
    return np.arccos(np.clip(ratio, -1.0, 1.0))
```

`np.clip(x, a, b)` replaces every value below `a` with `a` and every value above `b` with `b`, leaving values already in $[a, b]$ unchanged. Without the clip, a round-off error of 10^{-15} past the boundary returns `nan` silently — no exception, just a wrong answer that propagates through the rest of the computation.

Activity 2

Do: Open a Python session and run `import numpy as np`, then `np.arcsin(np.sin(3.0))`. Record the result.

Discuss: The result is not `3.0`. Use the identity $\sin(\pi - x) = \sin x$ to explain the value that appears. In what kind of real code would this silent substitution produce a hard-to-find bug?

9.4 Graph of arctan

Among the six inverse trig functions, arctan is the most useful in practice: its domain is all of $\mathbb{R}$ (no restriction needed), and it appears directly in integrals of the form $\int \frac{1}{1+x^2}\,dx$. Plotting it reveals the key features.

```python
import numpy as np
import matplotlib.pyplot as plt

xs = np.linspace(-15, 15, 500)
plt.plot(xs, np.arctan(xs))
plt.axhline( np.pi/2, color='gray', linestyle='--')
plt.axhline(-np.pi/2, color='gray', linestyle='--')
plt.xlabel('x')
plt.ylabel(r'$\arctan(x)$')
plt.grid(True)
plt.show()
```

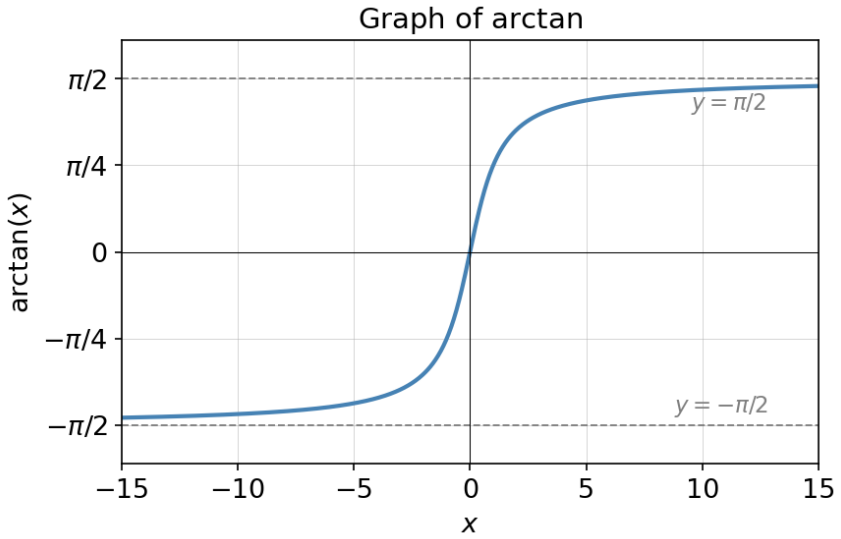


Figure 9.1: Graph of $\arctan x$, showing the horizontal asymptotes at $y = \pm\frac{\pi}{2}$.

Figure 9.1 shows the S-shaped curve, which rises steeply near the origin and flattens toward two horizontal asymptotes.

Key features of the graph.

- **Domain**: $(-\infty, \infty)$ — arctan accepts every real number.
- **Range**: $\left(-\frac{\pi}{2}, \frac{\pi}{2}\right)$ — the output is strictly between $-90°$ and $90°$, never reaching the endpoints.

- **Horizontal asymptotes**:

$$\lim_{x\to+\infty} \arctan x = \frac{\pi}{2}, \qquad \lim_{x\to-\infty} \arctan x = -\frac{\pi}{2}.$$

  These follow directly from the restricted domain of tan: as θ approaches $\pm\frac{\pi}{2}$ from inside the interval, $\tan\theta \to \pm\infty$, so the inverse goes the other way.

- **Odd function**: $\arctan(-x) = -\arctan(x)$, so the graph is symmetric about the origin.

- **Passes through the origin**: $\arctan(0) = 0$.

Activity 3

Do: From Figure 9.1, read off: (a) $\arctan(0)$; (b) the two asymptotic values as $x \to \pm\infty$; (c) the approximate slope of the curve at $x = 0$.

Discuss: The derivative formula gives $\arctan'(0) = \frac{1}{1+0^2} = 1$. Does the graph confirm a slope of 1 at the origin? What does a slope of exactly 1 look like geometrically (think of the angle the tangent line makes with the x-axis)?

9.5 Experimentally Verifying the Derivative of arctan

We proven in Chapter 2 that

$$\mathcal{D}_x \arctan x = \frac{1}{1+x^2}.$$

Figure 9.2 plots both functions together: the claim is that the bottom panel is the slope of the top panel at every point. Here we

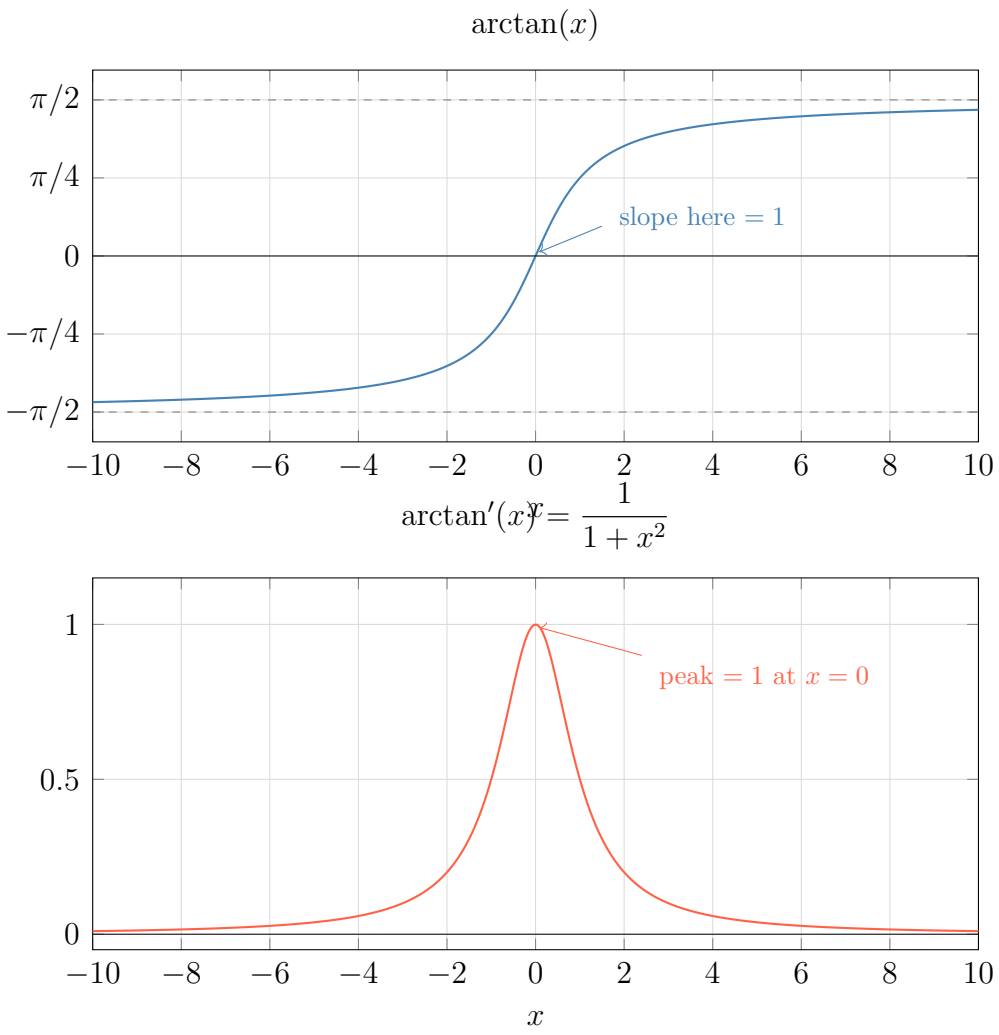


Figure 9.2: Top: $\arctan x$ on $[-10, 10]$. Bottom: its derivative $\frac{1}{1+x^2}$. Where $\arctan$ rises steeply (near $x = 0$), the derivative peaks at 1; where $\arctan$ flattens toward its asymptotes, the derivative approaches 0.

verify this numerically using the *finite-difference approximation*: for small h,

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}.$$

This is the *central difference* formula. It approximates the slope of the tangent line by the slope of a secant spanning a small interval $[x-h, x+h]$ centred on x. The error is $O(h^2)$: halving h reduces the error by a factor of 4.

```
import numpy as np

h = 1e-5
xs = np.linspace(-5, 5, 500)

numerical  = (np.arctan(xs + h) - np.arctan(xs - h)) / (2 * h)
analytical = 1 / (1 + xs**2)

max_error = np.max(np.abs(numerical - analytical))
print(f'max error: {max_error:.2e}')   # expect ~1e-10
```

Running the code above prints a maximum pointwise error on the order of 10^{-10}, which is consistent with the $O(h^2)$ accuracy of the central difference with $h = 10^{-5}$.

What this confirms.

- The formula $\mathcal{D}_x \arctan x = \frac{1}{1+x^2}$ is correct: the numerical approximation and the closed form agree to ten decimal places across the entire grid.
- Numerical differentiation is a useful *sanity check*, not a replacement for a proof. It cannot handle discontinuities, and the accuracy degrades as $h \to 0$ due to floating-point cancellation (subtracting two nearly equal numbers).
- The central difference is more accurate than the one-sided formula $\frac{f(x+h)-f(x)}{h}$ for the same h, because errors on either side of x partially cancel.

Activity 4

Do: In the central-difference code, change `h` to `1e-1`, then `1e-10`, then `1e-15`. Print the max error for each value.

Discuss: There is an optimal h somewhere between those extremes. Why does the error *increase* again as $h \to 0$? (Hint: when h is very small, what happens to $\arctan(x+h) - \arctan(x-h)$ in floating-point arithmetic?)

9.5.1 Inner Product Spaces

We won't repeat the axioms defining a vector space at this time. Those axioms can easily be found in any Linear Algebra textbook. However, we do repeat the definition of the inner product.

Definition 9.5.1 (*Inner Product*)

Let V be a vector space over the field $\mathbb{R}$. A binary function, $\langle \cdot, \cdot \rangle : V \times V \to \mathbb{R}$, is called an *inner product* provided:

1. $\langle v, v \rangle \geq 0 \, \forall v \in V$
2. $\langle v, v \rangle = 0$ if and only if $v = 0$.
3. $\langle u, v \rangle = \langle v, u \rangle \; \forall u, v \in V$
4. $\langle \alpha u + \beta w, v \rangle = \alpha \langle u, v \rangle + \beta \langle w, v \rangle \; \forall u, v, w \in V, \forall \alpha, \beta \in \mathbb{R}$

The set of functions $f : [a, b] \to \mathbb{R}$ such that $\int_a^b f(x)\, dx < \infty$ is a linear space. We sometimes define an inner product on this space as

follows.

$$\langle f, g \rangle = \int_a^b f(x)g(x)dx$$

With this definition of the inner product, we can define a norm

$$\|f\| = \sqrt{\langle f, f \rangle}$$

And once this norm has been defined, we can define the distance between two vectors in the vector space as follows

$$d(f, g) = \|f - g\|$$

so the normed vector space is also a metric space: a metric $d(f, g) \geq 0$ that equals zero only when $f = g$, is symmetric, and satisfies the triangle inequality — exactly what we need to make sense of a sequence of functions converging to a target.

We can compute $\langle f, g \rangle$ and $\|f\|$ numerically with `np.trapezoid`, the same way Section 9.5.2 will:

```
import numpy as np

xs = np.linspace(0, 2 * np.pi, 10000)

inner = np.trapezoid(np.sin(xs) * np.sin(xs), xs)
norm_sin = np.sqrt(inner)
print(f'<sin, sin> = {inner:.4f}')
print(f'||sin||    = {norm_sin:.4f}')
```

This prints `<sin, sin> = 3.1416` and `||sin|| = 1.7725`.

Activity 5

Do: Extend the code above to compute `distance(f, g)` $= \|f - g\|$ for two functions sampled on the same grid. Verify three cases: `distance(sin, sin)` is 0; `distance(sin, cos)` is clearly non-zero; and `distance(sin(x), sin(1.0001*x))` is small — close to 0, but not exactly 0.

Discuss: `sin(1.0001*x)` is barely a different function from `sin(x)` at all. Why does that intuition match what `distance` reports? What would you expect `distance(sin(x), sin(1.01*x))` to look like compared to `distance(sin(x), sin(1.0001*x))`, and why? Section 9.5.2 makes this idea precise: a sequence of functions converges exactly when their pairwise distance shrinks to zero.

9.5.2 Convergence and Inter-Function Distance

Saying that $\phi_h \to \frac{1}{1+x^2}$ as $h \to 0$ is a claim about *functions*, not just numbers. The previous section already supplies exactly what we need to make this precise: the L^2 inner product induces the norm $\|f\| = \sqrt{\langle f, f \rangle}$, which induces the metric $d(f, g) = \|f - g\|$. We can compute $d(f, g)^2 = \|f - g\|^2$ numerically with `np.trapezoid`.

The previous section's `inner_product.py` used `simpsons_sum` instead, because it is the more accurate of the two ($O(h^4)$ versus `trapezoid_sum`'s $O(h^2)$) — and you had already built both yourself in Chapter 8. `np.trapezoid` here is simply what NumPy itself provides: NumPy has no built-in Simpson's rule at all. That arrives with SciPy (Chapter 12), which is also where `simpsons_sum` gets compared directly against `scipy.integrate.quad` for accuracy.

```python
import numpy as np

xs   = np.linspace(-10, 10, 1000)
ana  = 1.0 / (1.0 + xs**2)

for h in [1.0, 0.5, 0.25, 0.125]:
    phi  = (np.arctan(xs + h) - np.arctan(xs)) / h
    err  = ana - phi
    dist = np.trapezoid(err**2, xs)
    print(f'h = {h:.3f}    ||error||^2 = {dist:.6f}')
```

The loop above prints:

```
h = 1.000    ||error||^2 = 0.168740
h = 0.500    ||error||^2 = 0.047132
h = 0.250    ||error||^2 = 0.012145
h = 0.125    ||error||^2 = 0.003060
```

Each time h is halved, $\|\text{error}\|^2$ drops by roughly a factor of 4 — consistent with the $O(h)$ accuracy of the one-sided difference quotient.

The L^2 *distance* between ϕ_h and $\frac{1}{1+x^2}$ is

$$d_h = \left\| \tfrac{1}{1+x^2} - \phi_h \right\| = \sqrt{\int_{-7}^{7} \left[\tfrac{1}{1+x^2} - \phi_h(x) \right]^2 dx}.$$

What `np.trapezoid` computes above is d_h^2, i.e. $\langle \text{err}, \text{err} \rangle$ — showing the squared distance avoids an unnecessary square root. The claim "$\phi_h \to \frac{1}{1+x^2}$ as $h \to 0$" is precisely the statement that $d_h \to 0$.

Figure 9.3 shows this convergence visually for $h \in \{1, 0.5, 0.25\}$. Each row corresponds to one value of h. The left panel overlays ϕ_h (blue dashed) on $\frac{1}{1+x^2}$ (red); the right panel shows the pointwise error with d_h^2 in the title. As h decreases, the two curves on the left become indistinguishable and the error collapses toward zero. A script with four values of h (including $h = 0.125$) is provided in `src/tp/arctan-convergence.py`.

Activity 6

Do: Run `src/tp/arctan-convergence.py` and record the four $\|\text{err}\|^2$ values for $h = 1,\ 0.5,\ 0.25,\ 0.125$. Compute the ratio between consecutive values.

Discuss: Each time h is halved, $\|\text{err}\|^2$ drops by roughly a factor of 4. The one-sided difference quotient has $O(h)$ accuracy, so the error is $\approx Ch$ for some constant C. Why does halving h then reduce $\|\text{err}\|^2 \approx C^2h^2$ by a factor of 4 rather than 2?

To explore this interactively, run the script `arctan-widget.py` (provided alongside this handout), as shown in Figure 9.4. It displays a slider for h; dragging it updates the plot of ϕ_h, the error curve, and the $\|\text{error}\|^2$ value in real time.

Activity 7

Do: Run `src/tp/arctan-widget.py`. Drag h slowly from 2.0 down to 0.05, watching both panels.

Discuss: Describe qualitatively what happens to the blue dashed curve and to the green error curve as h decreases. For what approximate value of h does the error curve become essentially flat at zero across the whole domain? Is the convergence faster near $x = 0$ or near $x = \pm 7$, and why?

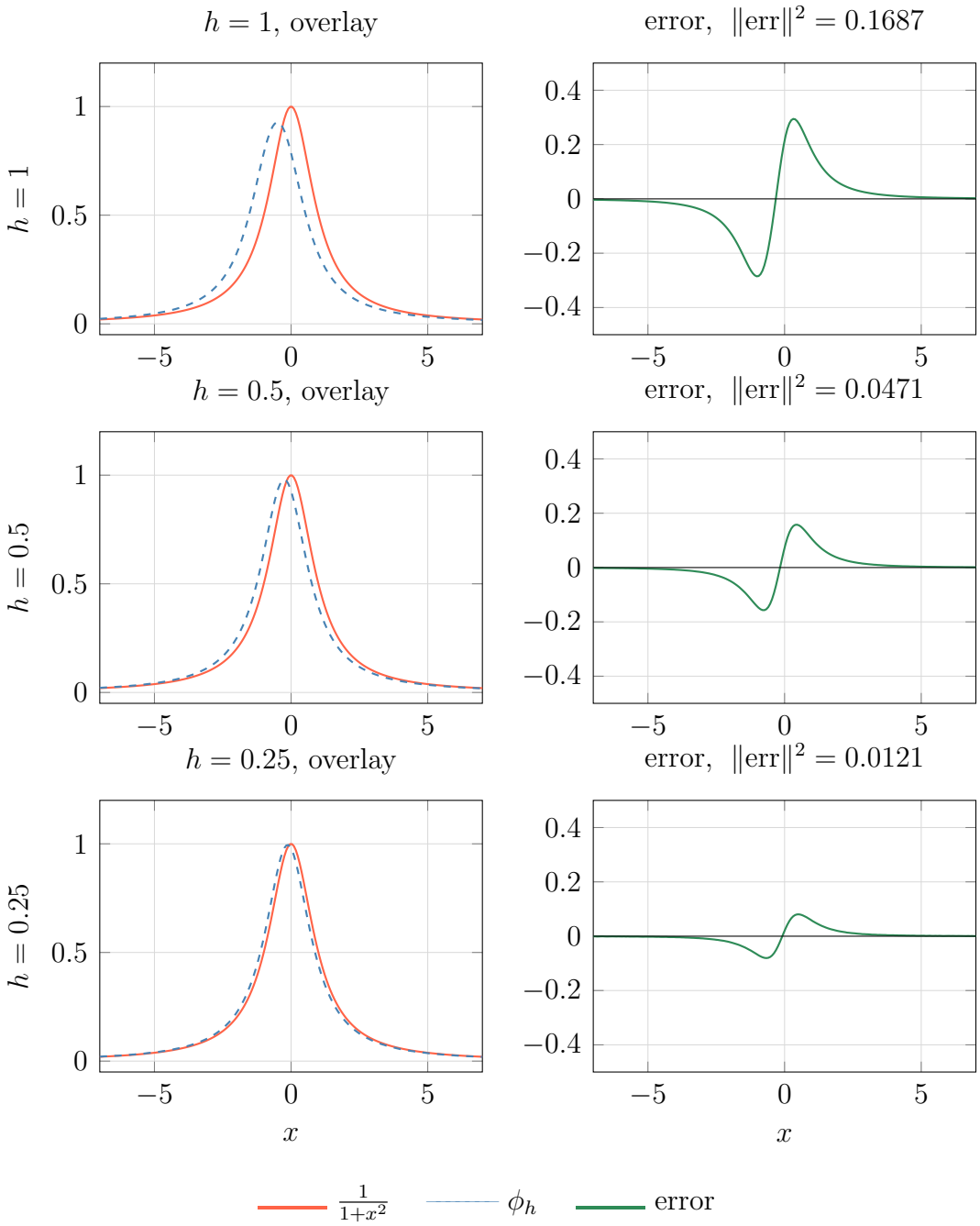


Figure 9.3: $\phi_h \to \frac{1}{1+x^2}$ as $h \to 0$, for $h = 1, 0.5, 0.25$.

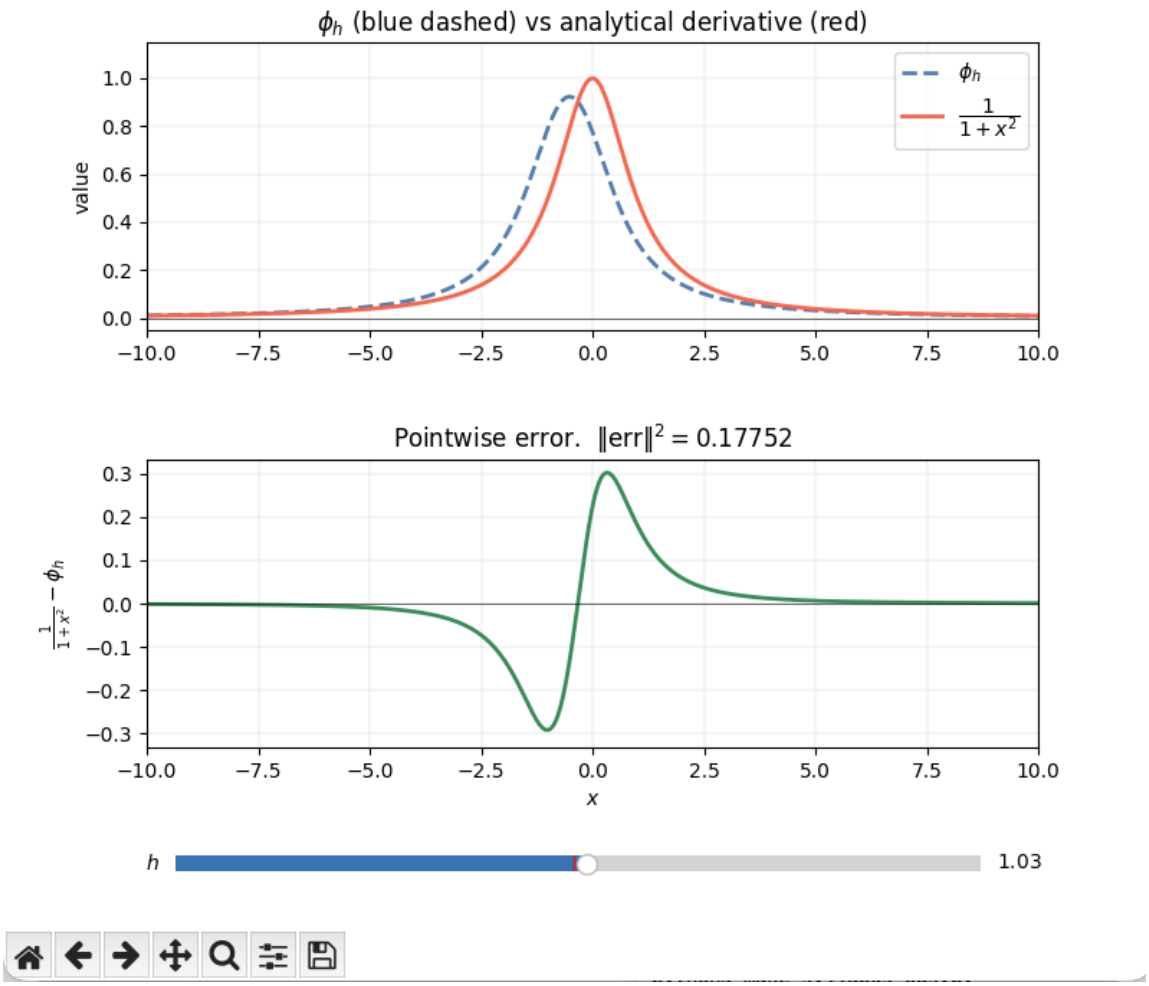


Figure 9.4: Widget created by python script `arctan-widget.py`

9.6 Integrals Involving Inverse Trigonometric Functions

The derivative formula we verified in the previous section,

$$\mathcal{D}_x \arctan x = \frac{1}{1+x^2},$$

runs equally well in the other direction as an antiderivative formula:

$$\int \frac{1}{1+x^2}\,dx = \arctan x + C.$$

Differentiation tables and antiderivative tables are two views of the same data: read left-to-right for derivatives, right-to-left for antiderivatives.

9.6.1 Numerically Computing Definite Integrals with NumPy

A definite integral has a specific numerical value. NumPy can approximate it directly with `np.trapezoid`, the same function we used in Section 9.5.2 to measure the L^2 distance.

Example: computing π from an integral. By the Fundamental Theorem of Calculus,

$$\int_0^1 \frac{1}{1+x^2}\,dx = \Big[\arctan x\Big]_0^1 = \arctan 1 - \arctan 0 = \frac{\pi}{4}.$$

NumPy can verify this directly:

```python
import numpy as np

xs     = np.linspace(0, 1, 10_000)
approx = np.trapezoid(1.0 / (1.0 + xs**2), xs)
exact  = np.pi / 4

print(f'NumPy:  {approx:.10f}')
print(f'Exact:  {exact:.10f}')
print(f'Error:  {abs(approx - exact):.2e}')
```

With 10 000 sample points, the output is:

```
NumPy:  0.7853981628
Exact:  0.7853981634
Error:  5.55e-10
```

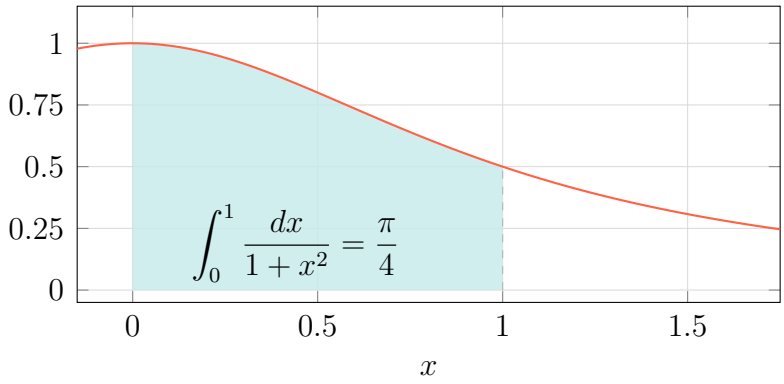


Figure 9.5: The shaded area equals $\pi/4 \approx 0.785$, computed as $\int_0^1 \frac{1}{1+x^2}\,dx = \arctan 1 - \arctan 0$.

The trapezoidal rule recovers $\pi/4$ to nine decimal places. This is a striking result: the number π emerges from integrating a purely algebraic function — no circles, no geometry, just $\frac{1}{1+x^2}$ (Figure 9.5).

The complete script, including the plot, is in `src/tp/arctan-integral.py`.

Activity 8

Do: Run `src/tp/arctan-integral.py`. Then modify the upper limit from `1` to `np.tan(np.pi/3)` (≈ 1.732) and run again.

Discuss: The FTC gives $\int_0^{\tan(\pi/3)} \frac{dx}{1+x^2} = \arctan(\tan(\frac{\pi}{3})) = \frac{\pi}{3}$. Does the numerical result match? What familiar constant is $4 \times \frac{\pi}{4}$, and how could you compute it using only this integral?

9.6.2 Discovering Antiderivatives Empirically

The Fundamental Theorem of Calculus tells us that if f is continuous, then

$$F(x) = \int_a^x f(t)\,dt$$

is an antiderivative of f, meaning $F'(x) = f(x)$. This opens an empirical discovery strategy. Given a function f whose antiderivative you do not know, you can:

1. Fix a base point a (often $a = 0$).
2. Compute $F(x) = \int_a^x f(t)\,dt$ numerically for a dense grid of x values using `np.cumsum`.
3. Plot the resulting curve and try to recognise it.
4. Test your guess G by checking that $G' \approx f$ numerically.

With a uniform grid of step Δx, the left Riemann sum gives

$$F(x_k) \;\approx\; \Delta x \sum_{i=0}^{k-1} f(x_i),$$

which `np.cumsum(f) * dx` computes in one vectorized call.

```
import numpy as np
import matplotlib.pyplot as plt

xs = np.linspace(0, 5, 1000)
dx = xs[1] - xs[0]

f = 1.0 / (1.0 + xs**2)
F = np.cumsum(f) * dx          # numerical antiderivative, F(0) = 0

plt.plot(xs, F,              label=r'numerical $F(x)$',
         color='steelblue', linewidth=2)
plt.plot(xs, np.arctan(xs), label=r'$\arctan x$',
         color='tomato',    linewidth=2, linestyle='--')
plt.legend()
plt.xlabel('$x$')
plt.title(r'Discovering the antiderivative of $\dfrac{1}{1+x^2}$')
plt.grid(True, linewidth=0.3, alpha=0.6)
plt.show()
```

The two curves are indistinguishable to the eye. Checking numerically:

```
print(
  f'Max pointwise error: {np.max(np.abs(F - np.arctan(xs))):.4f}')
```

gives a maximum error of about 5×10^{-3}—small, but not zero, because the left Riemann sum has $O(\Delta x)$ accuracy. Switching to the trapezoidal rule (averaging adjacent values) reduces the error by more than three orders of magnitude, to around 10^{-6}.

Figure 9.6 shows f alongside the discovered F.

The complete script, including the overlay plot, is in `src/tp/arctan-antideriv.py`.

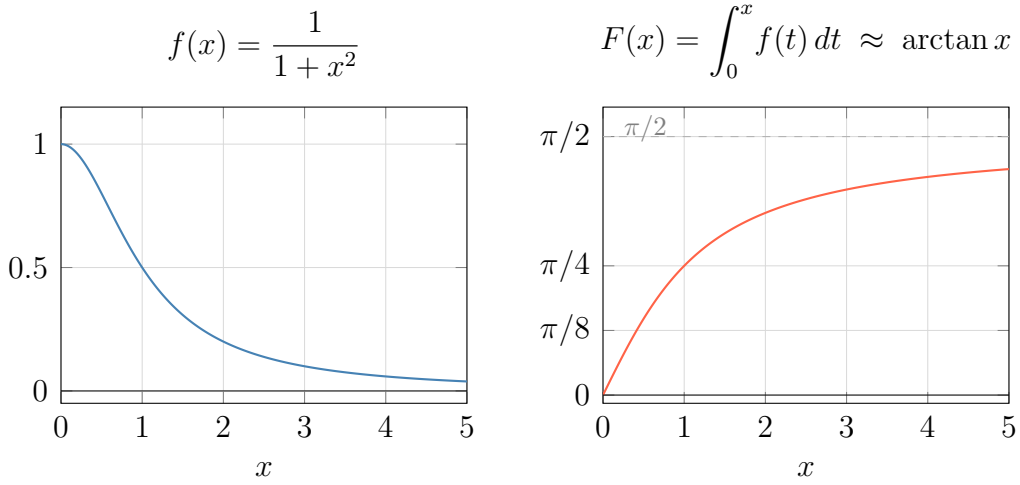


Figure 9.6: Left: the integrand $f(x) = \frac{1}{1+x^2}$. Right: the antiderivative $F(x) = \int_0^x f(t)\,dt$ computed numerically via `np.cumsum`, which is indistinguishable from $\arctan x$.

Activity 9

Do: In `src/tp/arctan-antideriv.py`, replace `f` with `1 / np.sqrt(np.maximum(1 - xs**2, 1e-9))` and change the domain to `np.linspace(0, 0.99, 1000)`. Run the script and inspect the discovered F.

Discuss: The numerical F should resemble a familiar function. Which one? Use the derivative rule $(\arcsin x)' = \frac{1}{\sqrt{1-x^2}}$ to confirm your guess algebraically. What antiderivative formula does this experiment suggest?

Why this works. The FTC guarantees $F' = f$. If a candidate G also satisfies $G' = f$, then $(G - F)' = 0$ everywhere, so $G - F$ is a constant. The base point pins the constant: $F(0) = 0$ and $\arctan 0 = 0$, so the constant is zero and $G = F$ exactly. The empirical check therefore

confirms the symbolic formula $\int \frac{1}{1+x^2}\,dx = \arctan x + C$ without any algebraic derivation.

9.7 Take Away

- NumPy enables *vectorized* computation: evaluate, differentiate, and integrate a function across an entire grid with no explicit loop.
- Naming conventions differ: `math.atan` vs. `np.arctan` compute the same function; `np.log` is the natural logarithm, matching the calculus convention ln.
- Library functions make silent choices about domains and principal branches. `np.arcsin(np.sin(x))` does not return `x` for $|x| > \pi/2$ — this is not a bug, but it will surprise you if you have not thought about restricted domains. Always check what range a function returns before relying on it.
- The derivative formula $\mathcal{D}_x \arctan x = \frac{1}{1+x^2}$ can be verified numerically: the finite-difference quotient $\phi_h \to \frac{1}{1+x^2}$ in L^2 distance as $h \to 0$.
- The same formula, read as an antiderivative, gives $\int \frac{1}{1+x^2}\,dx = \arctan x + C$. A special case: $\int_0^1 \frac{1}{1+x^2}\,dx = \frac{\pi}{4}$, which NumPy approximates to nine decimal places with 10 000 points.
- The Fundamental Theorem of Calculus makes antiderivative discovery possible empirically: compute $F(x) = \int_0^x f(t)\,dt$ with `np.cumsum`, plot it, and match it against known functions.

9.8 Homework

`homework/inner_product.py` is an individually graded Intra assignment implementing the inner product, norm, and distance defined in Section 9.5.1, each built directly on the one before it. The tests you are graded against are in `tests/inner_product_test.py`.

- `inner_product(f, g, x_min, x_max, n=1000)`: approximates $\langle f, g \rangle = \int_{x_{\min}}^{x_{\max}} f(x)g(x)\,dx$ using `simpsons_sum` from `riemann_sum.py`.
- `norm(f, x_min, x_max, n=1000)`: $\|f\| = \sqrt{\langle f, f \rangle}$, i.e. the square root of `inner_product(f, f, x_min, x_max, n)`.
- `distance(f, g, x_min, x_max, n=1000)`: $d(f, g) = \|f - g\|$, i.e. `norm` applied to the difference of `f` and `g`.

Everything in this chapter used `numpy` freely — this assignment is the one place in the chapter where you must not. `inner_product.py` is graded on the cloud (Section 9.1), which does not have `numpy` installed. This is not just "don't call `np` functions": even a bare, unused `import numpy` at the top of the file is enough to fail the whole submission, because the import itself raises `ModuleNotFoundError` before any of your code runs, and the grader awards zero points if your file fails to load. Compute `inner_product`, `norm`, and `distance` using only `simpsons_sum` from `riemann_sum.py` and plain Python arithmetic.

`f` and `g` here are plain `Callable[[float], float]`, not necessarily `PointFree` objects: write `inner_product` and `norm` generically, calling `f(x)` and `g(x)` directly, rather than relying on `PointFree` operator overloading (`f - g`, `f * g`), which does not

exist for arbitrary Python callables. A `PointFree` instance still works as an argument with no conversion needed: `PointFree` overloads `__call__`, so calling `f(x)` on a `PointFree` instance already invokes `f.evaluate(x)`.

This `distance` measures something different from `checks.py`'s `distance` (Section 6.6.2, Chapter 6): that one is the sup-norm — the single worst pointwise disagreement between `f` and `g` anywhere on the interval. This one is the L^2, or root-mean-square, distance — an aggregate over the *whole* interval instead of a single worst point. Two functions that differ briefly but sharply can have a small L^2 distance yet a large sup-norm distance, and vice versa for two functions that differ mildly everywhere. Between the two chapters you now have two complementary ways to numerically compare two functions when testing `derivative`.

? Open Questions

- The "Why this works" argument closing Section 9.6.2 concludes $G = F$ exactly from $(G - F)' = 0$ everywhere plus $(G - F)(0) = 0$ — the same zero-derivative-implies-constant fact that Chapter 7 invoked (also without proof) to argue $\exp$ is unique. It is actually a consequence of the Mean Value Theorem, which this course does not cover. Assuming it, can you see why a function with zero derivative everywhere *on an interval* must be constant? Would the argument still work if the domain had a gap in it — say, f defined on $(-\infty, 0) \cup (0, \infty)$ with $f' = 0$ everywhere it is defined?
- The central-difference activity in Section 9.5 asks you to observe that the error gets worse again once h is very small, due to floating-point cancellation. Is there a principled way to predict the best h in advance — balancing the $O(h^2)$ truncation error against round-off error, which grows like (machine epsilon)$/h$ — rather than just trying values and eyeballing the results?
- Section 9.3 notes that the restricted domains for arcsec and arccsc are *conventions*, and that different sources disagree (some define arcsec's range using $[0, \pi/2) \cup (\pi/2, \pi]$ as here, others use a range that keeps the formula for the derivative simpler at the cost of an extra absolute value). If you combined two libraries that silently made different convention choices, what would go wrong, and how would you notice?

Chapter 10

The Fundamental Theorem of Calculus

Chapter Objectives

- Explain the Fundamental Theorem of Calculus using the tank-and-flow-meter analogy: the integral of a rate gives the net change.
- State and apply FTC Part 1 (differentiation of an accumulation function) and Part 2 (evaluation via antiderivatives).
- Compute definite integrals as linear combinations using the antiderivative table.
- Expand a function as a Taylor series and differentiate or integrate term by term.

10.1 Intuition: A Tank and a Flow Meter

Imagine a tank of water with a single valve. The valve can let water in or out, and it contains a meter — internally a small wheel whose spin rate is proportional to the flow. The meter displays the instantaneous flow rate $r(t)$: positive when water enters the tank, negative when it leaves. For example, $r(t) = -0.8\,\mathrm{L/sec}$ (liters per second) means water is leaving the tank at 0.8 litres per second; $r(t) = 2.1\,\mathrm{L/sec}$ means water

is entering at 2.1 litres per second.

After some time has passed, from $t = a$ to $t = b$, you want to know the net change in the volume of water in the tank. There are two ways to find out.

Method 1: Look at the tank. Measure the volume at the start, $V(a)$, and at the end, $V(b)$. The net change is $V(b) - V(a)$.

Method 2: Watch the meter. Read the flow rate at many short intervals of duration Δt. During each interval the approximate volume change is $r(t_i)\,\Delta t$ — positive if water is entering, negative if leaving. Summing over all intervals and taking the limit as $\Delta t \to 0$ gives a Riemann sum that converges to the integral:

$$\text{net change} \approx \sum_i r(t_i)\,\Delta t \xrightarrow{\Delta t \to 0} \int_a^b r(t)\,dt\,.$$

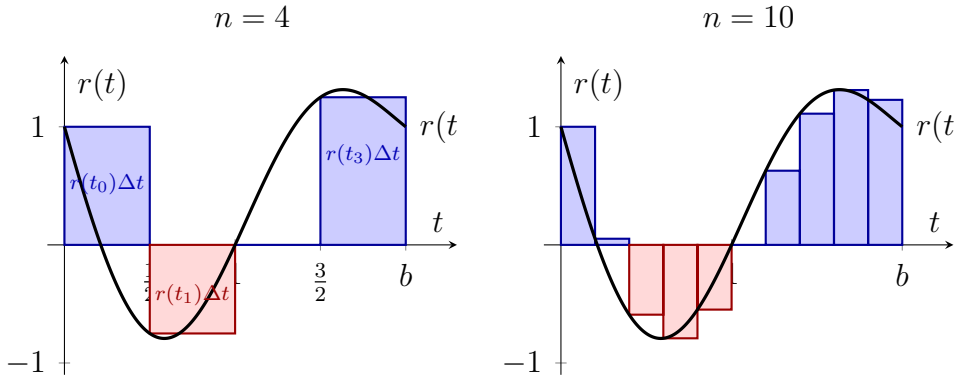


Figure 10.1: Left ($n = 4$): the left Riemann sum $\sum_{i=0}^{3} r(t_i)\,\Delta t$ for $r(t) = (t-1)^2 - \sin(\pi t)$ on $[a, b] = [0, 2]$ — the same rectangle construction as Figure 8.2, applied to a flow rate. **Right ($n = 10$):** a finer partition reduces the error; compare with Figure 8.5.

Figure 10.1 shows this sum for $r(t) = (t-1)^2 - \sin(\pi t)$: the same left-sum construction as Figure 8.2, now applied to a flow rate. Blue

rectangles accumulate inflow; red rectangles subtract outflow. The right panel ($n = 10$) uses a finer partition and visibly hugs the curve more closely, exactly as in Figure 8.5.

Both methods measure the same quantity, so they must agree:

$$\int_a^b r(t)\,dt = V(b) - V(a)\,.$$

This is the Fundamental Theorem of Calculus, Part 2. The key observation is that $r(t) = V'(t)$: the flow rate is the derivative of the volume, so V is an antiderivative of r. The theorem says that *integrating a rate of change over an interval gives the net change*, and you can evaluate that integral simply by finding any antiderivative and subtracting its values at the two endpoints.

Part 1 is the other direction of the same coin. Suppose you define $F(x)$ to be the volume accumulated from time a to time x:

$$F(x) = \int_a^x r(t)\,dt\,.$$

Notice the integration variable here is t, not x. This is not a typo: x is already doing a different job, naming the time at which F is evaluated, so it cannot also name the variable swept over inside the integral. Proposition 8.7.1 says the integration variable can be renamed freely, and here we must use that freedom — writing $\int_a^x r(x)\,dx$ would make x mean two different things in the same expression. We will keep this distinction — outer variable versus integration variable — every time we build a function by accumulation.

At what rate is $F(x)$ growing? At time x, water is entering at rate $r(x)$, so the accumulated volume is growing at exactly rate $r(x)$. That is, $F'(x) = r(x)$ — differentiating the accumulated integral gives back the integrand. This is Part 1.

Figure 10.2 shows both methods side by side for a concrete flow rate $r(t)$.

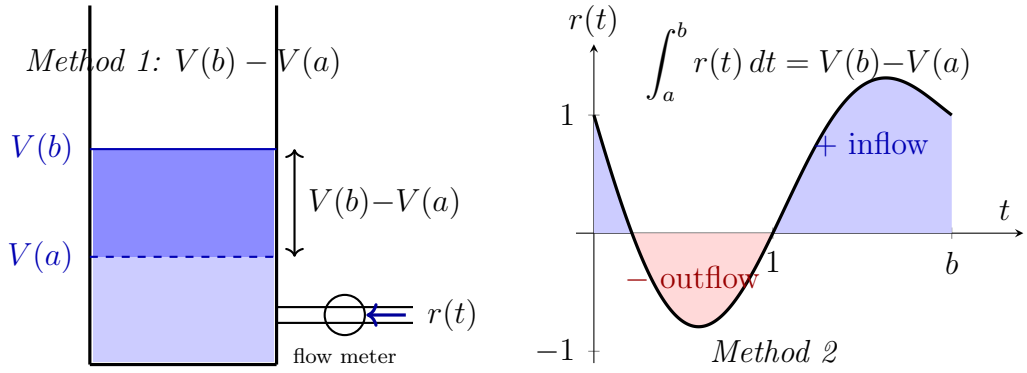


Figure 10.2: Two ways to measure the net change in water volume. **Left (Method 1):** read $V(b)-V(a)$ directly from the tank. **Right (Method 2):** integrate the flow rate $r(t) = (t-1)^2 - \sin(\pi t)$ over $[a, b]$; blue area is net inflow, red area is net outflow. Both give the same answer — that equality is FTC Part 2.

10.2 The Fundamental Theorem of Calculus [FTC]

The Fundamental Theorem of Calculus explains how integration and differentiation relate and in which sense they are inverse operations.

Theorem 10.2.1 (*Fundamental Theorem of Calculus, Part 1*)

Let f be continuous on $[a, b]$, and define $F : [a, b] \to \mathbb{R}$ by

$$F(x) = \int_a^x f(t)\,dt\,.$$

Then F is differentiable on (a, b) and $\mathcal{D}\,F = f$, *i.e.*

$$\mathcal{D}_x F(x) = f(x)\,.$$

Figure 10.3 illustrates both directions of this relationship; the com-

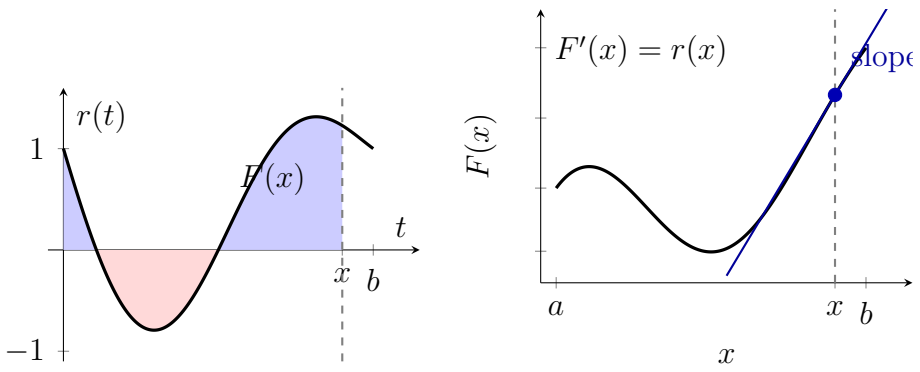


Figure 10.3: FTC Part 1 for $r(t) = (t-1)^2 - \sin(\pi t)$ on $[0, 2]$. **Left:** $r(t)$ with the shaded area equal to $F(x) = \int_a^x r(t)\,dt$. **Right:** the accumulated volume $F(x)$; the tangent slope at each x equals $r(x)$, confirming $F'(x) = r(x)$.

putation is worked out in Section 10.4.

Theorem 10.2.2 (*Fundamental Theorem of Calculus, Part 2*)

Let f be continuous on $[a, b]$, and let F be any antiderivative of f on $[a, b]$, *i.e.* $\mathcal{D}\,F = f$. Then

$$\int_a^b f(x)\,dx = F(b) - F(a)\,.$$

10.3 Linearity of the Integral

In Section 8.8 we showed directly from the Riemann sum that

$$\int_a^b \big(\alpha f(x) + \beta g(x)\big)\,dx = \alpha \int_a^b f(x)\,dx + \beta \int_a^b g(x)\,dx\,.$$

The argument was arithmetic: distributing Δx across the height of each rectangle, then pulling the constants α and β outside the sum before taking the limit. We record it here as a theorem for reference.

Theorem 10.3.1 (*Linearity of the Integral*)

For any integrable functions f and g on $[a, b]$ and constants $\alpha, \beta \in \mathbb{R}$,

$$\int_a^b \big(\alpha f(x) + \beta g(x)\big)\, dx = \alpha \int_a^b f(x)\, dx + \beta \int_a^b g(x)\, dx\,.$$

Theorem 10.3.1 lets us integrate any polynomial term by term, applying the power rule $\int x^n\, dx = \frac{x^{n+1}}{n+1} + C$ to each monomial.

Example 10.3.2 (*Integrating a polynomial*)

Compute $\displaystyle\int_0^3 (4x^3 - x^2 + 1)\, dx$.

Apply linearity to split into three integrals, pull out the constants, then use the power rule on each term:

$$\begin{aligned}
\int_0^3 (4x^3 - x^2 + 1)\, dx &= 4\int_0^3 x^3\, dx \;-\; \int_0^3 x^2\, dx \;+\; \int_0^3 1\, dx \\
&= 4\left[\frac{x^4}{4}\right]_0^3 - \left[\frac{x^3}{3}\right]_0^3 + \big[x\big]_0^3 \\
&= \big[x^4\big]_0^3 - \left[\frac{x^3}{3}\right]_0^3 + \big[x\big]_0^3 \\
&= (81 - 0) - (9 - 0) + (3 - 0) \\
&= 75\,.
\end{aligned}$$

In the third line the factor of 4 canceled against the denominator 4 in $x^4/4$, leaving $[x^4]_0^3$ directly.

In practice, once linearity is routine, the antiderivative is written

in one step by integrating each term in place:

$$\int_0^3 (4x^3 - x^2 + 1)\,dx = \left[x^4 - \frac{x^3}{3} + x\right]_0^3 = 81 - 9 + 3 = 75\,.$$

10.4 Relation to Anti-Derivatives

Figure 10.3 shows FTC Part 1 concretely. **Left:** the flow rate $r(t)$ is plotted over time; the shaded region from a to x represents the volume accumulated up to time x, that is $F(x) = \int_a^x r(t)\,dt$. **Right:** that same $F(x)$ is plotted as a function of time; the slope of the tangent at any x equals $r(x)$, confirming $F'(x) = r(x)$.

By Part 2, we can evaluate the net change exactly by finding an antiderivative and subtracting its values at the endpoints.

Example 10.4.1 (*Net volume from a flow rate*)

Let $r(t) = (t-1)^2 - \sin(\pi t)$ on $[0, 2]$ (flow rate in L/s, liters per second). Find the net change in volume $\displaystyle\int_0^2 r(t)\,dt$.

Step 1: find an antiderivative. By linearity (Theorem 10.3.1), we integrate each term separately. We need F with $F'(t) = r(t)$:

$$F(t) = \frac{(t-1)^3}{3} + \frac{\cos(\pi t)}{\pi}\,.$$

(Use $\int (t-1)^2\,dt = \frac{(t-1)^3}{3}$ and $\int \sin(\pi t)\,dt = -\frac{\cos(\pi t)}{\pi}$, so $-\int \sin(\pi t)\,dt = \frac{\cos(\pi t)}{\pi}$.)

Step 2: apply FTC Part 2.

$$\begin{aligned}\int_0^2 r(t)\,dt &= F(2) - F(0)\\ &= \left(\frac{(2-1)^3}{3} + \frac{\cos 2\pi}{\pi}\right) - \left(\frac{(0-1)^3}{3} + \frac{\cos 0}{\pi}\right)\\ &= \left(\frac{1}{3} + \frac{1}{\pi}\right) - \left(-\frac{1}{3} + \frac{1}{\pi}\right) = \frac{2}{3}.\end{aligned}$$

The $\frac{1}{\pi}$ terms cancel exactly, leaving $\frac{2}{3}$ L net inflow.

10.5 Taylor Series and Integration

A Taylor series expresses a smooth function as an infinite polynomial. Linearity (Theorem 10.3.1) is a finite statement, proven by induction on the number of terms, and does *not* automatically extend to infinite sums. Swapping an integral and an infinite series requires a separate justification: if a series of functions converges *uniformly* on $[a, b]$, then it may be integrated term by term — that is, the integral of the limit equals the limit of the integrals. The Taylor series of e^x, $\sin x$, and $\cos x$ converge uniformly on every bounded interval, so the interchange is valid for them.

This is especially useful for functions that have no closed-form an-

tiderivative. A short list of series that appear repeatedly:

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$$

$$\sin x = \sum_{n=0}^{\infty} \frac{(-1)^n\, x^{2n+1}}{(2n+1)!} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$$

$$\cos x = \sum_{n=0}^{\infty} \frac{(-1)^n\, x^{2n}}{(2n)!} = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots$$

Each series converges for all $x \in \mathbb{R}$ and may be integrated term by term on any bounded interval. You will differentiate and integrate these series term by term in Chapter 11, Section 11.3.

Example 10.5.1 (*Integrating the sinc function*)

The function $\operatorname{sinc}(x) = \sin(x)/x$ appears throughout signal processing and Fourier analysis. At $x = 0$ we set $\operatorname{sinc}(0) = 1$ (the limiting value), making it continuous everywhere. It has no closed-form antiderivative, but dividing the series for $\sin x$ by x gives an ordinary power series:

$$\frac{\sin x}{x} = \frac{1}{x}\left(x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots\right) = 1 - \frac{x^2}{6} + \frac{x^4}{120} - \frac{x^6}{5040} + \cdots$$

Integrating term by term from 0 to 1:

$$\begin{aligned}\int_0^1 \frac{\sin x}{x}\,dx &= \left[x - \frac{x^3}{3\cdot 6} + \frac{x^5}{5\cdot 120} - \frac{x^7}{7\cdot 5040} + \cdots\right]_0^1 \\ &= 1 - \frac{1}{18} + \frac{1}{600} - \frac{1}{35280} + \cdots \\ &\approx 1 - 0.05556 + 0.00167 - 0.00003 + \cdots \approx 0.94608\,.\end{aligned}$$

The series alternates in sign with decreasing terms, so by the alternating series test the error after k terms is smaller than the $(k{+}1)$-th term. Four terms already give six correct decimal places.

The same technique applies to e^{-x^2}, which arises in probability as the Gaussian density. It too has no elementary antiderivative; its Taylor series $e^{-x^2} = 1 - x^2 + x^4/2! - x^6/3! + \cdots$ integrates term by term to give the *error function* $\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\,dt$, a standard special function implemented in every scientific library (`math.erf` in Python).

? Open Questions

- The Taylor-series section says term-by-term integration is valid when a series "converges *uniformly*" on $[a, b]$, but never defines uniform convergence or says what goes wrong without it. Ordinary (pointwise) convergence only requires that $f_n(x) \to f(x)$ at each individual x, with no control over how fast. Can you find, or look up, a sequence of continuous functions on $[0, 1]$ that converges pointwise to a function f, yet $\int_0^1 f_n \not\to \int_0^1 f$? (Hint: try a sequence of increasingly tall, increasingly thin spikes.)
- sinc and erf are examples of functions with no elementary antiderivative that were important enough to be given their own names and built-in implementations (`math.erf`) in every scientific library. What other integrals have you encountered, or can you find, that were "promoted" to a named special function for the same reason?

Chapter 11

HW-2: Computing Integrals

Chapter Objectives

- Evaluate the inner product of two functions on an interval and verify orthogonality.
- Compute antiderivatives by table lookup, including compositions requiring a change of variable.
- Differentiate and integrate Taylor series term by term within their radius of convergence.
- Evaluate definite integrals using the Fundamental Theorem of Calculus and verify numerically using `simpsons_sum` from `riemann_sum.py`.

11.1 Inner Product on Function Spaces

Recall from the linear algebra course that an inner product on a vector space V is a binary function $\langle \cdot, \cdot \rangle : V \times V \to \mathbb{R}$ satisfying four axioms (see Definition 9.5.1).

Let V be the set of continuous functions $f : [0, 1] \to \mathbb{R}$. Define

$$\langle f, g \rangle = \int_0^1 f(x)\, g(x)\, dx\,.$$

1. **Axiom 1 (non-negativity).** Show that $\langle f, f \rangle \geq 0$ for all $f \in V$.

2. **Axiom 3 (symmetry).** Show that $\langle f, g \rangle = \langle g, f \rangle$ for all $f, g \in V$.

3. **Axiom 4 (linearity in the first argument).** Show that $\langle \alpha f + \beta g,\, h \rangle = \alpha \langle f, h \rangle + \beta \langle g, h \rangle$ for all $f, g, h \in V$ and $\alpha, \beta \in \mathbb{R}$.

4. **Axiom 2 (definiteness) — a subtle point.** Axiom 2 requires that $\langle f, f \rangle = 0$ if and only if $f = 0$. The direction "$f = 0 \Rightarrow \langle f, f \rangle = 0$" is immediate. The other direction is more delicate.

   Consider the function

   $$f(x) = \begin{cases} 1 & x = \frac{1}{2} \\ 0 & \text{otherwise.} \end{cases}$$

   Compute $\langle f, f \rangle = \int_0^1 [f(x)]^2\, dx$ and state its value. Is f the zero function?

   Explain in your own words what this example shows about whether the integral definition of inner product strictly satisfies Axiom 2 for Riemann-integrable functions.

   Note for the curious: this difficulty is one of the reasons mathematicians replaced the Riemann integral with the Lebesgue integral. In the resulting space $L^2[0, 1]$, functions that agree except on a "negligible" set of points are identified, and Axiom 2 holds exactly.

11.2 Antiderivatives by Table Lookup

Compute the following antiderivatives using linearity and the derivative table from Chapter 2. No integration technique is required.

1. $\displaystyle\int \left(3x^2 + 2x - 7\right) dx$
2. $\displaystyle\int \left(5e^x - 4\cos x\right) dx$
3. $\displaystyle\int \frac{3}{1+x^2}\, dx$
4. $\displaystyle\int \left(\frac{1}{x} + e^x\right) dx$
5. $\displaystyle\int \left(2\sin x + \frac{5}{\sqrt{1-x^2}}\right) dx$
6. $\displaystyle\int \left(x^4 - \frac{2}{x} + 3e^x\right) dx$
7. $\displaystyle\int \frac{x^5 - 2 + 3xe^x}{x} dx$ *(Hint: use the previous result.)*

11.3 Taylor Series: Differentiation and Integration

Recall that several familiar functions can be expressed as infinite power series:

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n+1}}{(2n+1)!}$$

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n}}{(2n)!}$$

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \cdots = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

Just as with finite polynomials, these series may be differentiated and integrated term by term. You already used this technique in HW-1 (Section 4.1.2) to differentiate the series for $\cos x$ and e^x, and Chapter 7 proved $\mathcal{D}_x \sin x = \cos x$ by an entirely different method (the Squeeze Theorem, from first principles). Below, part 1 runs the same power-series technique on $\sin x$ instead — a genuinely independent confirmation of $\mathcal{D}_x \sin x = \cos x$, by a method with no ε-δ argument anywhere in it — and parts 2–3 run the idea in the other direction: integrating a series term by term.

1. **Derivative of** sin**, via its power series.** Differentiate the series for $\sin x$ term by term. Show that the result equals the series for $\cos x$.

2. **Integral of** cos**.** Integrate the series for $\cos x$ term by term to obtain an antiderivative. Determine the constant of integration

by using the fact that $\sin 0 = 0$. Show that the result equals the series for $\sin x$.

3. e^x **integrates to itself.** Integrate the series for e^x term by term to obtain an antiderivative series. Determine the constant of integration by evaluating at $x = 0$ and using the fact that $e^0 = 1$. Show that the result is again the series for e^x (plus a constant). What does this confirm about the relationship between differentiation and integration of e^x?

11.4 Evaluating Definite Integrals

Evaluate the following definite integrals using the Fundamental Theorem of Calculus, Part 2 (Theorem 10.2.2). For each one: (a) find an antiderivative F, (b) compute $F(b) - F(a)$, and (c) state the exact value.

1. $\displaystyle\int_0^2 (3x^2 + 1)\,dx$

2. $\displaystyle\int_0^{\pi} \sin x\,dx$

3. $\displaystyle\int_1^e \frac{1}{x}\,dx$

4. $\displaystyle\int_0^1 e^x\,dx$

5. $\displaystyle\int_0^1 \frac{1}{1+x^2}\,dx$ *(Hint: the answer involves π.)*

6. $\displaystyle\int_0^{1/2} \frac{1}{\sqrt{1-x^2}}\,dx$ *(Hint: what is $\arcsin\frac{1}{2}$?)*

Numerical verification. Pick any two integrals above and verify your exact answers using your own `simpsons_sum` from `riemann_sum.py`. Place `riemann_sum.py` in the same directory as your script and run:

```python
from riemann_sum import simpsons_sum
import math

# verify integral 5: integral of 1/(1+x^2) from 0 to 1
val = simpsons_sum(lambda x: 1 / (1 + x**2), 0, 1, 1000)
print(f'numerical: {val:.10f}')        # should be pi/4
print(f'exact:     {math.pi/4:.10f}')
print(f'error:     {abs(val - math.pi/4):.2e}')
```

With $n = 1000$ subintervals, Simpson's rule achieves an error well below 10^{-8} on smooth integrands — far more than enough to confirm your hand-computed answers. Use n even (Simpson's rule requires it); doubling n reduces the error by a factor of roughly 16 (the rule is $O(1/n^4)$).

Chapter 12

Calculus with SciPy

Chapter Objectives

- Use `scipy.integrate.quad` to evaluate definite integrals numerically, including those with no closed-form antiderivative.
- Use Python's `timeit` module to measure and compare the speed of `simpsons_sum` and `quad`, and explain why the two differ in both accuracy and speed.
- Use `scipy.optimize.minimize_scalar` to find minima and maxima of functions numerically, and compare with analytical solutions obtained by differentiation.
- Preview `scipy.integrate.solve_ivp` for solving initial-value problems and connect the ODE formulation to the Fundamental Theorem of Calculus.

In Chapter 9 you used NumPy and your own `riemann_sum.py` to approximate definite integrals from first principles. SciPy is the next tier: it provides *black-box solvers* for the three core calculus tasks — integration, optimization, and differential equations. You feed it a Python callable and some parameters; it returns a highly accurate numerical answer together with an estimate of its own error. This session develops the habit of reaching for the right tool for each job.

12.1 What is SciPy?

SciPy is organised into submodules. Unlike NumPy, which is usually imported wholesale as `np`, the convention is to import only the specific functions you need from SciPy:

```
from scipy.integrate import quad, solve_ivp
from scipy.optimize  import minimize_scalar
```

Activity 1

Do: Open a Python session and run `from scipy.integrate import quad`. If you see a `ModuleNotFoundError`, SciPy is not yet installed; run `pip install scipy` in a terminal and restart. Confirm success with `print(quad(lambda x: 1, 0, 1))` — you should see a pair such as `(1.0, 1.11e-14)`.

Discuss: SciPy is not part of the Python standard library, so it must be installed separately and is not available everywhere. Why does this matter for your Oral Defense and Intra submissions? What would happen if you accidentally left a `from scipy.integrate import quad` line in a submitted file?

NumPy and SciPy are forbidden in Oral Defense and Intra submissions. This reminder appears in Chapter 9 as well, but it bears repeating. Neither `numpy` nor `scipy` may appear in any file you submit for the graded programming assignments. The cloud grading service does not have these libraries installed; a stray import line anywhere in your submitted files will cause the grader to fail with zero points, even if that import is never executed.

Keep your TP scratch scripts strictly separate from your submission files.

12.2 Numerical Integration with `scipy.integrate.quad`

`quad(f, a, b)` computes $\int_a^b f(x)\,dx$ numerically and returns a pair `(value, error_estimate)`. Unlike your `riemann_sum.py` (which evaluates f at n uniformly spaced points), `quad` uses *adaptive Gauss-Kronrod quadrature*: it subdivides $[a, b]$ non-uniformly, concentrating sample points wherever the integrand varies most rapidly. For smooth integrands, this typically achieves errors near 10^{-12} with far fewer function evaluations than a uniform grid.

```
from scipy.integrate import quad
import numpy as np

val, err = quad(lambda x: 1 / (1 + x**2), 0, 1)
print(f'value: {val:.15f}')   # pi/4
print(f'error: {err:.2e}')    # quad's own error estimate
print(f'pi/4:  {np.pi/4:.15f}')
```

Running the code above prints:

```
value: 0.785398163397448
error: 8.72e-15
pi/4:  0.785398163397448
```

The agreement is exact to fifteen decimal places — far more accurate than the trapezoidal rule with 10 000 points used in Chapter 9.

The complete script is in `src/tp/scipy-quad-demo.py`.

Activity 2

Do: Run `src/tp/scipy-quad-demo.py`. It compares `quad` with your own `simpsons_sum` at several values of n:

```
from riemann_sum import simpsons_sum

f      = lambda x: 1 / (1 + x**2)
exact = np.pi / 4
for n in [10, 100, 1000, 10_000]:
    val = simpsons_sum(f, 0, 1, n)
    print(f'n={n:6d}  error={abs(val - exact):.2e}')
```

Discuss: At what value of n does `simpsons_sum` match the accuracy that `quad` achieves with no n at all? Recall that Simpson's rule is $O(1/n^4)$: doubling n reduces the error by a factor of 16. How many doublings (and thus how many extra function evaluations) does it take to close the gap?

12.2.1 Integrals with No Closed Form

The real power of `quad` appears when the antiderivative cannot be expressed in terms of elementary functions. A famous example is the *Gaussian integral*:

$$\int_0^\infty e^{-x^2}\,dx = \frac{\sqrt{\pi}}{2}.$$

This result is not obvious and cannot be derived by the techniques in this course. Nevertheless, `quad` evaluates it instantly, and it can handle the infinite upper limit via `np.inf`:

```python
from scipy.integrate import quad
import numpy as np

val, err = quad(lambda x: np.exp(-x**2), 0, np.inf)
print(f'quad:          {val:.15f}')
print(f'sqrt(pi)/2:    {np.sqrt(np.pi)/2:.15f}')
print(f'error:         {err:.2e}')
```

Output:

```
quad:          0.886226925452758
sqrt(pi)/2:    0.886226925452758
error:         8.57e-10
```

Notice that your `simpsons_sum` cannot handle this integral at all: the upper limit is ∞ and there is no antiderivative to plug in. `quad` handles both limitations automatically.

The Gaussian integral is not a curiosity. Every probability calculation involving a normal (bell-curve) distribution reduces to evaluating integrals of e^{-x^2}. Machine learning loss landscapes, error bounds in statistics, and noise models in signal processing all depend on this number.

Activity 3

Do: Run `src/tp/scipy-quad-demo.py`. Then use `quad` to evaluate $\int_0^1 \frac{\sin x}{x}\,dx$ (the *sinc integral*). Note that the integrand is not defined at $x = 0$ — Python will raise a `ZeroDivisionError` if you naively write `lambda x: np.sin(x)/x`. Fix this with `lambda x: np.sinc(x/np.pi)`, which NumPy evaluates at $x = 0$ correctly.

Discuss: The exact value is approximately 0.9460831. Like e^{-x^2}, the function $\frac{\sin x}{x}$ has no elementary antiderivative. What does this tell you about the boundary between integrals that are tractable by hand and those that require numerical methods?

12.3 Speed of Computation

Accuracy is not the only reason to prefer `quad` over your own `simpsons_sum`. A simulation may evaluate the same integral millions of times, so the time per call matters as much as the error.

The two implementations differ in two fundamental ways. First, `simpsons_sum` evaluates f at exactly $n+1$ uniformly spaced points; `quad` uses an *adaptive* strategy that concentrates evaluations where f changes most rapidly, typically requiring only 21–300 evaluations regardless of the integrand. Second, `simpsons_sum` executes a Python loop; `quad` calls QUADPACK, a Fortran library compiled to native machine code that has been optimised for decades.

The script below measures both effects at once:

```
import timeit, numpy as np
from scipy.integrate import quad
from riemann_sum import simpsons_sum

f      = lambda x: 1.0 / (1.0 + x**2)
exact = np.pi / 4.0

print(f"{'method':<18}  {'n':>8}  {'error':>10}  {'ms/call':>8}")
print("-" * 50)

for n in [1_000, 10_000, 100_000]:
    val = simpsons_sum(f, 0.0, 1.0, n)
    err = abs(val - exact)
    ms  = timeit.timeit(lambda n=n: simpsons_sum(f, 0.0, 1.0, n),
                        number=10) / 10 * 1000
    print(f"{'simpsons_sum':<18}  {n:>8d}  {err:>10.2e}  {ms:>8.2f}"
                                    )

val_q, _ = quad(f, 0.0, 1.0)
err_q    = abs(val_q - exact)
ms_q     = timeit.timeit(lambda: quad(f, 0.0, 1.0),
                         number=1000) / 1000 * 1000
print(f"{'quad':<18}  {'(adaptive)':>8}  {err_q:>10.2e}  {ms_q:>8.4f
                                    }")
```

Typical output on a modern laptop (your numbers will vary):

```
method                    n        error    ms/call
-------------------------------------------------
simpsons_sum           1000    1.33e-13       0.87
simpsons_sum          10000    8.88e-16       8.72
simpsons_sum         100000    8.88e-16      87.10
quad              (adaptive)    8.88e-16     0.0375
```

`quad` matches the best accuracy `simpsons_sum` can achieve (floating-point arithmetic bottoms out near 10^{-16}) while running more than **2000 times faster** than the $n = 100\,000$ call. You can also ask `quad` how many function evaluations it used:

```python
_, _, info = quad(f, 0.0, 1.0, full_output=True)
print(info['neval'])   # typically 21
```

For this smooth integrand, `quad` needed 21 evaluations to reach machine precision; `simpsons_sum` needed 100 001 to reach the same accuracy.

The full timing script is `src/tp/scipy-speed-demo.py`. 

Activity 4

Do: Run `src/tp/scipy-speed-demo.py`. Add $n = 1\,000\,000$ to the `simpsons_sum` loop and observe how the timing scales. Then print `info['neval']` for a rougher integrand such as `lambda x: abs(np.sin(10*x))` and compare with the smooth case.

Discuss: The speedup has two sources: fewer function evaluations and compiled Fortran code. To isolate the second factor, rewrite `simpsons_sum` using NumPy vectorisation (compute all f values at once with `np.linspace` instead of a Python loop) and re-time it for $n = 100\,000$. Does the gap shrink? What fraction of the original speedup came from the algorithm versus the implementation language?

12.4 Numerical Optimization with `scipy.optimize`

Differentiation gives us an analytical recipe for finding extrema: compute $f'(x) = 0$, solve for x, classify. When $f'(x) = 0$ cannot be solved in closed form, we need a numerical alternative.

`scipy.optimize.minimize_scalar` finds a local minimum of a scalar function $f : \mathbb{R} \to \mathbb{R}$. Its `bounded` method restricts the search to an interval $[a, b]$, which is usually necessary in practice to avoid unintended local minima:

```
from scipy.optimize import minimize_scalar

f      = lambda x: (x - 2)**2 + 1
result = minimize_scalar(f,
                         bounds=(0, 5),
                         method='bounded')
print(f'minimum at x = {result.x:.6f}')   # should be 2.0
print(f'f(x_min)     = {result.fun:.6f}') # should be 1.0
```

The `result` object is a namespace with (at least) two fields: `result.x` is the minimiser and `result.fun` is the minimum value.

Maximisation. `minimize_scalar` only minimises. To *maximise* f, minimise $-f$:

```
result = minimize_scalar(lambda x: -f(x),
                         bounds=(0, 5),
                         method='bounded')
x_max = result.x
f_max = -result.fun
```

`minimize_scalar` finds a *local* minimum inside the given bounds — not necessarily the global minimum. If f has multiple local minima, the result depends on the search interval. Always plot the function first to choose bounds that contain exactly one minimum.

12.4.1 Recovering the Skip-List Optimum

In HW-1 (Section 4.5) you found analytically that the optimal promotion probability for a skip list of size n is $p = \frac{1}{e}$, minimising the expected search cost

$$T(p) = \frac{\ln n}{-p \ln p}, \qquad p \in (0, 1).$$

Here we verify that result numerically.

```
from scipy.optimize import minimize_scalar
import math

n = 10_000    # list size; try changing this
T = lambda p: math.log(n) / (-p * math.log(p))

result = minimize_scalar(T,
                         bounds=(0.01, 0.99),
                         method='bounded')
print(f'optimal p   = {result.x:.8f}')
print(f'1/e         = {1/math.e:.8f}')
print(f'T(p_opt)    = {result.fun:.6f}')
```

Output (for $n = 10\,000$):

```
optimal p   = 0.36787944
1/e         = 0.36787944
T(p_opt)    = 25.036301
```

The numerical minimiser matches $1/e$ to eight decimal places, confirming the analytical answer you computed by setting $T'(p) = 0$.

The complete script is in `src/tp/scipy-optimize-demo.py`.

Activity 5

Do: Run `src/tp/scipy-optimize-demo.py` with $n = 10\,000$. Then change n to 100, 10^6, and 10^9. Record the optimal p and the minimum cost $T(p_{\text{opt}})$ for each case.

Discuss: The optimal $p = 1/e$ does not depend on n, but $T(p_{\text{opt}})$ grows. How does the cost scale with n? Use the formula $T(p) = \frac{\ln n}{-p \ln p}$ at $p = 1/e$ to predict the ratio $T(10^6)/T(10^3)$, then check against the numerical values.

12.5 Solving Differential Equations

12.5.1 What is a Differential Equation?

A *differential equation* is an equation that involves an unknown function *and its derivative*. When there is a single independent variable, as in every example below, it is called an *ordinary differential equation*, or **ODE** for short — introduced already in Chapter 7 via the defining equation $\exp' = \exp$. The simplest kind is a *first-order initial-value problem* (IVP):

$$\frac{dy}{dt} = g(t,\, y), \qquad y(t_0) = y_0.$$

We are given the *rate of change* at every point — not as a formula for y, but as a rule that computes y' from t and the current value y. The initial condition $y(t_0) = y_0$ pins the solution uniquely.

This structure will feel familiar: the Fundamental Theorem of Calculus is a special case. When g does not depend on y at all — say $g(t, y) = f(t)$ — the IVP reduces to:

$$y(t) = y_0 + \int_{t_0}^{t} f(s)\, ds.$$

`solve_ivp` is a general version of `quad` that handles the case where g depends on y as well.

12.5.2 Exponential Growth

The archetypal first-order ODE is:

$$\frac{dy}{dt} = y, \qquad y(0) = 1.$$

Its analytical solution is $y(t) = e^t$ — the exponential function is its own derivative. We verified this symbolically in Chapter 10 using the Taylor series. Here we recover it numerically.

`scipy.integrate.solve_ivp` expects the right-hand side as a Python function `f(t, y)`, where `y` is a *vector* of state variables (even if there is only one):

```
from scipy.integrate import solve_ivp
import numpy as np

def dydt(t, y):
    return y            # dy/dt = y; y is a length-1 array

sol = solve_ivp(dydt,
                t_span=(0, 3),
                y0=[1.0],
                dense_output=True,
                rtol=1e-10,
                atol=1e-12)

ts       = np.linspace(0, 3, 200)
ys_num   = sol.sol(ts)[0]          # numerical solution
ys_exact = np.exp(ts)              # analytical y(t) = e^t

max_err = np.max(np.abs(ys_num - ys_exact))
print(f'max error: {max_err:.2e}')   # expect < 1e-8
```

The `solve_ivp` solver (`RK45`) achieves an error near 10^{-9} here by automatically choosing step sizes — the ODE analogue of `quad`'s adaptive quadrature. (Its default tolerances, `rtol=1e-3` and `atol=1e-6`, are much looser — tightening them, as above, is what buys the extra accuracy.)

The complete script, including the overlay plot, is in `src/tp/scipy-ivp-demo.py`.

Activity 6

Do: Run `src/tp/scipy-ivp-demo.py`. You will see the numerical solution plotted against e^t. Then change the right-hand side to `return -y` and the initial condition to `y0=[1.0]`. Run again.

Discuss: What is the analytical solution to $y' = -y$, $y(0) = 1$? Does the numerical plot match? If the ODE $y' = y$ models uninhibited growth, what physical process might $y' = -y$ model?

12.5.3 Beyond Simple Integration: Logistic Growth

When g depends on y, the IVP cannot be reduced to a plain integral and generally has no elementary closed form. The *logistic growth model*

$$\frac{dy}{dt} = r\,y\left(1 - \frac{y}{K}\right), \qquad y(0) = y_0$$

describes a population $y(t)$ that grows rapidly when small but levels off at a *carrying capacity* K. Here $r > 0$ is the intrinsic growth rate.

```python
from scipy.integrate import solve_ivp
import numpy as np

r, K, y0 = 1.0, 100.0, 5.0   # growth rate, capacity, initial pop.

def dydt(t, y):
    return r * y[0] * (1 - y[0] / K)

sol = solve_ivp(dydt,
                t_span=(0, 10),
                y0=[y0],
                dense_output=True)

ts = np.linspace(0, 10, 300)
ys = sol.sol(ts)[0]
print(f'y(10) ~= {ys[-1]:.2f}   (carrying capacity = {K})')
```

The S-shaped solution curve rises steeply near $t = 0$ and flattens toward $K = 100$ — the same qualitative behavior as arctan, which should not surprise you: both are bounded, monotone, and approach their asymptote exponentially fast.

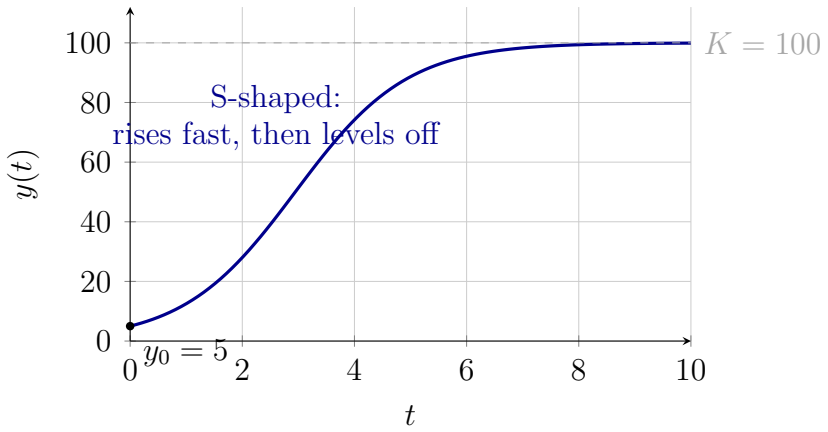


Figure 12.1: The logistic growth curve for $r = 1$, $K = 100$, $y_0 = 5$. Growth is nearly exponential while $y \ll K$, then slows as y approaches the carrying capacity K.

Figure 12.1 plots this curve.

Activity 7

Do: Add `import matplotlib.pyplot as plt` and plot `ys` against `ts`. Then try changing `y0` to `50.0` and `150.0` and rerun.

Discuss: What happens when $y_0 > K$? Does the model remain biologically sensible? If you were implementing a population simulation and this ODE produced negative values, would that be a bug in `solve_ivp` or a limitation of the model?

12.6 Take Away

- `scipy.integrate.quad` evaluates $\int_a^b f(x)\,dx$ to near machine precision using adaptive quadrature. It handles infinite limits (`np.inf`) and integrands with no closed-form antiderivative — far beyond what your `riemann_sum.py` can reach.

- `scipy.optimize.minimize_scalar` finds a local minimum of f inside a given interval. It is the numerical counterpart of setting $f'(x) = 0$ and solving. Use it when the critical-point equation has no closed form, or as a sanity-check on an analytical result (as we did for the skip-list optimal $p = 1/e$).

- `scipy.integrate.solve_ivp` solves an initial-value problem $y' = g(t, y)$, $y(t_0) = y_0$. When g does not depend on y, this reduces to plain integration. When it does, `solve_ivp` extends the reach of calculus to processes — population dynamics, electrical circuits, heat flow — where the rate of change depends on the current state.

- The common thread: all three tools accept a Python callable and return a high-accuracy numerical answer with an error estimate. This is the SciPy contract: you provide the problem; the library chooses the algorithm.

? Open Questions

- The warning in Section 12.4 says `minimize_scalar` finds only a *local* minimum, and recommends plotting the function first to pick good bounds. What would you do if the function had many local minima and you couldn't plot it — say, a loss function of a thousand parameters, as in machine learning? Is there any way to be *certain* you have found the global minimum, short of checking every point?
- Section 12.5.1 states that the initial condition $y(t_0) = y_0$ "pins the solution uniquely," without qualification. Is that always true, for *every* right-hand side $g(t, y)$? Try to find (or look up) an initial-value problem with more than one solution satisfying the same initial condition. (Hint: try $\frac{dy}{dt} = \sqrt{|y|}$, $y(0) = 0$ — $y \equiv 0$ is one solution; can you find a second one that stays at 0 for a while and then peels away?)
- `quad`'s adaptive quadrature "concentrates sample points wherever the integrand varies most rapidly." But to know where a function varies rapidly, you would seem to need to have already evaluated it there. How might an adaptive algorithm decide where to refine its sampling without already knowing the answer it's trying to compute?

Chapter 13

Integration by Substitution

Chapter Objectives

- Explain why antidifferentiation has no systematic algorithm analogous to the derivative rules.
- Apply u-substitution to evaluate integrals by reversing the chain rule.
- Evaluate $\int \sqrt{1-x^2}\,dx$ both geometrically (area of a circular sector and a triangle) and via the substitution $x = \sin\theta$, and explain why the two methods agree.

Why do we need techniques to integrate? Why can't we just do the opposite of what we did when computing the derivative?

This is a very reasonable question. Before answering it, consider this concrete example. Let

$$f(x) = \sin(x^2) \cdot \cos(x^2)\,.$$

Both factors involve the inner function x^2, so both require the chain

rule. Applying the product rule:

$$\begin{aligned} f'(x) = & \underbrace{2x\cos(x^2)}_{\text{chain rule on } \sin(x^2)} \cdot \cos(x^2) + \sin(x^2) \cdot \underbrace{(-2x\sin(x^2))}_{\text{chain rule on } \cos(x^2)} \\ &= 2x\cos^2(x^2) - 2x\sin^2(x^2) \\ &= 2x\big(\cos^2(x^2) - \sin^2(x^2)\big) \\ &= 2x\cos(2x^2)\,. \end{aligned}$$

The two terms produced by the product rule merged into one via the identity $\cos^2\theta - \sin^2\theta = \cos 2\theta$. The result is clean and compact. Now turn the question around:

Given that $f'(x) = 2x\cos(2x^2)$, what is $f(x)$?

A very clever student might try the substitution $u = 2x^2$ and recover $\frac{1}{2}\sin(2x^2) + C$, which is correct — but looks nothing like $\sin(x^2)\cos(x^2)$. A typical student, even a strong one, staring at $2x\cos(2x^2)$ would not spontaneously write down $\sin(x^2)\cos(x^2)$ as the answer. The cancellation that made differentiation easy is precisely what makes the reverse step hard: the intermediate structure was erased.

The example reveals a deeper asymmetry. To state it precisely, we need to distinguish two levels.

At the *function* level — functions $\mathbb{R} \to \mathbb{R}$ — both operations are total given mild conditions. Every differentiable f has a well-defined derivative f'. And by the Cauchy construction from Chapter 8, every continuous f has a well-defined antiderivative (the accumulation function), unique up to an additive constant. No asymmetry here.

The asymmetry appears at the *representation* level — the level of expression trees. In Chapter 5 you implemented a function

$$\texttt{derivative} : \mathsf{AST} \to \mathsf{AST}$$

that takes an expression tree and returns an expression tree for its derivative. This function is *total*: it handles every node type with a small, finite set of tree transformations, and it always terminates with a valid AST. The homework is direct evidence that such an algorithm exists.

The surprise is that no analogous total function

$$\texttt{antiderivative} : \mathsf{AST} \to \mathsf{AST}$$

exists. This is not a gap in our current knowledge — it is a theorem (due to Liouville) that the class of elementary expressions is closed under differentiation but not under antidifferentiation. The antiderivative of e^{-x^2} exists and is a perfectly well-defined function $\mathbb{R} \to \mathbb{R}$; it simply cannot be written as any finite combination of polynomials, exponentials, logarithms, and trigonometric functions. There is no AST for it.

As a CS student, this result gives you a cleaner way to understand why integration is hard. The difficulty is not that the integral fails to exist — it always does. The difficulty is that the integral does not always have a *constructor*: there is no total $\mathsf{AST} \to \mathsf{AST}$ function that builds a representation of the antiderivative from a representation of the integrand. Existence and constructibility are different things, and integration is hard precisely because it falls on the wrong side of that line.

Figure 13.1 makes this asymmetry visual.

What we *can* do is identify important families of integrands for which a pattern-matching strategy reliably finds a closed form. This chapter and the next develop three such families — u-substitution and a special trigonometric substitution here, then partial fractions and integration by parts in Chapter 14. Together they handle the integrals that arise most often in engineering and science, and they are the building blocks underlying the symbolic integration algorithms in computer algebra systems such as SymPy or Mathematica.

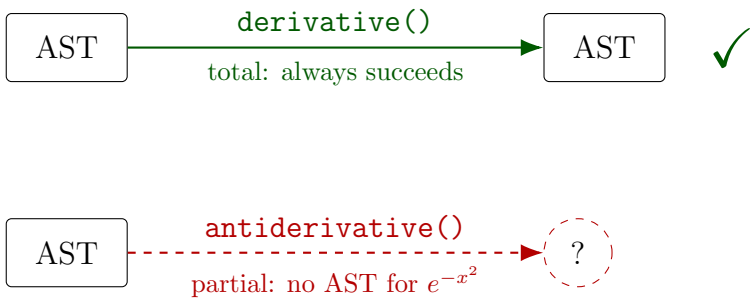


Figure 13.1: `derivative()` is a total function on the AST: it always succeeds. `antiderivative()` is only partial: it fails for expressions like e^{-x^2}, which have no AST for their antiderivative even though the antiderivative exists as a function.

13.1 U-Substitution

U-substitution is the chain rule running backwards. Recall that the chain rule gives

$$\frac{d}{dx}[F(g(x))] = F'(g(x)) \cdot g'(x)\,.$$

Reading this as an antiderivative statement:

$$\int F'(g(x)) \cdot g'(x)\,dx = F(g(x)) + C\,.$$

If we write $u = g(x)$, then $du = g'(x)\,dx$, and the integral becomes $\int F'(u)\,du = F(u) + C$. The substitution trades a complicated integral in x for a simpler one in u.

Theorem 13.1.1 (*U-substitution*)

Let g be differentiable and f continuous. If $u = g(x)$ and $du = g'(x)\,dx$, then

$$\int f(g(x))\,g'(x)\,dx = \int f(u)\,du\,.$$

For a definite integral the bounds change with u:

$$\int_a^b f(g(x))\,g'(x)\,dx = \int_{g(a)}^{g(b)} f(u)\,du\,.$$

How to apply it. Look for an inner function $g(x)$ whose derivative $g'(x)$ also appears as a factor in the integrand. Set $u = g(x)$, replace $g'(x)\,dx$ with du, and replace every remaining x with the expression for x in terms of u. If the substitution does not clear all x's from the integrand, a different choice of u is needed.

Example 13.1.2 (*U-substitution with a trig function*)

Compute $\displaystyle\int 2x\cos(x^2)\,dx$.

Set $u = x^2$, so $du = 2x\,dx$. The factor $2x\,dx$ appears exactly in the integrand:

$$\int 2x\cos(x^2)\,dx = \int \cos(u)\,du = \sin(u) + C = \sin(x^2) + C\,.$$

Example 13.1.3 (*U-substitution yielding a logarithm*)

Compute $\displaystyle\int \frac{3x^2+1}{x^3+x}\,dx$.

Set $u = x^3 + x$, so $du = (3x^2+1)\,dx$. The numerator is exactly

du:

$$\int \frac{3x^2+1}{x^3+x}\,dx = \int \frac{du}{u} = \ln|u| + C = \ln|x^3+x| + C\,.$$

Example 13.1.4 (*Definite integral with changed bounds*)

Compute $\displaystyle\int_0^1 2x\,e^{x^2}\,dx$.

Set $u = x^2$, so $du = 2x\,dx$. When $x = 0$, $u = 0$; when $x = 1$, $u = 1$. The bounds are unchanged here, but that is a coincidence:

$$\int_0^1 2x\,e^{x^2}\,dx = \int_0^1 e^u\,du = \Big[e^u\Big]_0^1 = e - 1\,.$$

13.2 Special Substitution

Consider two integrals that look almost identical:

$$\int \frac{dx}{\sqrt{1-x^2}} \qquad \text{and} \qquad \int \sqrt{1-x^2}\,dx\,.$$

One of them we can already dispatch with tools we have; the other resists every tool so far. Working out why — and what it takes to close the gap — is the point of this section.

13.2.1 Two Integrals, No Substitution Yet

Example 13.2.1 (*Table lookup for* $1/\sqrt{1-x^2}$)

Figure 2.1 lists $\mathcal{D}_x \arcsin x = \frac{1}{\sqrt{1-x^2}}$. Reading the table right-to-left,

$$\int \frac{dx}{\sqrt{1-x^2}} = \arcsin x + C\,.$$

No substitution needed — this integral is already an entry in the table.

The second integral has no matching row: the table has an entry for $\frac{1}{\sqrt{1-x^2}}$, not for $\sqrt{1-x^2}$ itself. U-substitution does not help either — there is no inner function whose derivative also appears as a factor. So we fall back on the recurring intuition from Section 8.6 of Chapter 8: read the integral as an area, and see whether that area is a shape we can compute geometrically.

The function $f(t) = \sqrt{1-t^2}$ is the upper half of the unit circle $t^2 + f^2 = 1$. Define the accumulation function

$$F(x) = \int_0^x \sqrt{1-t^2}\, dt\,,$$

which by Proposition 8.7.1 renames the integration variable to t so that x, the point where F is evaluated, is not overloaded with a second meaning. $F(x)$ is the area under the quarter circle from $t = 0$ to $t = x$.

Figure 13.2 shows this area split along the radius OP, where $P = (x, \sqrt{1-x^2})$, into two pieces.

The triangle. Right triangle OQP has legs x and $\sqrt{1-x^2}$, so its area is $\frac{1}{2}\, x\sqrt{1-x^2}$.

The sector. Label the two angles at O: $\alpha = \angle AOP$ (between radii OA and OP — this is the sector's central angle) and $\beta = \angle POQ$ (between OP and the horizontal). Since OA and OQ are perpendicular, α and β make up that right angle between them:

$$\alpha + \beta = \frac{\pi}{2}\,.$$

Triangle OQP has hypotenuse $OP = 1$ and adjacent side $OQ = x$, so

$$\cos\beta = \frac{OQ}{OP} = \frac{x}{1} = x\,, \qquad \beta = \arccos x\,.$$

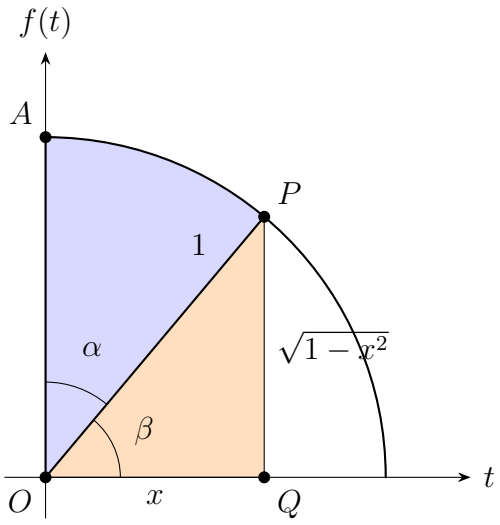


Figure 13.2: The area $F(x) = \int_0^x \sqrt{1-t^2}\,dt$ splits along the radius OP into the blue circular sector OAP (central angle $\alpha = \angle AOP$) and the orange right triangle OQP (legs x and $\sqrt{1-x^2}$, with $\beta = \angle POQ$ at O). Since $OA \perp OQ$, $\alpha + \beta = \pi/2$.

Therefore

$$\alpha = \frac{\pi}{2} - \beta = \frac{\pi}{2} - \arccos x = \arcsin x\,.$$

A sector of radius 1 and central angle α has area $\frac{1}{2}\alpha = \frac{1}{2}\arcsin x$.

Proposition 13.2.2 (*Area Under a Semicircle*)

For $-1 \le x \le 1$,

$$\int_0^x \sqrt{1-t^2}\,dt = \frac{1}{2}\Big(x\sqrt{1-x^2} + \arcsin x\Big)\,,$$

and therefore

$$\int \sqrt{1-x^2}\,dx = \frac{1}{2}\Big(x\sqrt{1-x^2} + \arcsin x\Big) + C\,.$$

No algebra was needed — only the areas of a triangle and a sector. The rest of this section arrives at the same formula a second way, by substitution, and explains why the two routes agree.

13.2.2 The Substitution $x = \sin\theta$

Both integrals involve $\sqrt{1-x^2}$, and $\sqrt{1-x^2}$ only makes sense (as a real number) for $-1 \le x \le 1$. This is exactly the range of $\sin\theta$, which suggests trying $x = \sin\theta$.

Theorem 13.2.3 (*The Substitution* $x = \sin\theta$)

For $-1 \le x \le 1$, set $x = \sin\theta$ with $\theta \in \left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$, so $dx = \cos\theta\, d\theta$. On this range $\cos\theta \ge 0$, so

$$\sqrt{1-x^2} = \sqrt{1-\sin^2\theta} = \sqrt{\cos^2\theta} = \cos\theta\,,$$

with no absolute value or branch choice needed.

The restriction to $\theta \in [-\pi/2, \pi/2]$ is not just bookkeeping. Outside that range $\cos\theta$ can be negative, and $\sqrt{\cos^2\theta}$ would equal $-\cos\theta$ instead of $\cos\theta$ — the substitution would still be legal, but the clean cancellation above would fail. Restricting to $[-\pi/2, \pi/2]$ is precisely what keeps $\theta = \arcsin x$ a genuine function (one output per input) and keeps $\sqrt{1-x^2} = \cos\theta$ valid without cases.

A remark on velocity. As θ sweeps once through $[-\pi/2, \pi/2]$, $x = \sin\theta$ sweeps once through $[-1, 1]$ — the substitution does not change which x-values are reached, only how they are parametrized. The rate of that sweep is $dx/d\theta = \cos\theta$, which is not constant: it equals 1 at $\theta = 0$ ($x = 0$, the middle of the interval) and shrinks to 0 as $\theta \to \pm\pi/2$ ($x \to \pm 1$, the ends). Equal steps in θ crowd together near $x = 0$ and spread out near $x = \pm 1$. This non-uniform "speed" is exactly the Jacobian factor $dx = \cos\theta\, d\theta$ that makes the substitution

work — it is not an algebraic accident.

13.2.3 Applying the Substitution to Both Integrals

Example 13.2.4 ($x = \sin\theta$ ***applied to*** $1/\sqrt{1-x^2}$)

With $x = \sin\theta$, $dx = \cos\theta\, d\theta$, and $\sqrt{1-x^2} = \cos\theta$, the $\cos\theta$ in dx cancels the $\cos\theta$ in the denominator:

$$\int \frac{dx}{\sqrt{1-x^2}} = \int \frac{\cos\theta\, d\theta}{\cos\theta} = \int d\theta = \theta + C = \arcsin x + C\,.$$

This matches Example 13.2.1 exactly.

A second proof of $\mathcal{D}_x \arcsin x$. Example 13.2.4 is more than a consistency check — it is an independent proof of Theorem 7.5.7 (Chapter 7), reached by a completely different route than implicit differentiation. Define $G(x) = \int_0^x \frac{dt}{\sqrt{1-t^2}}$. By FTC Part 1 (Theorem 10.2.1), $G'(x) = \frac{1}{\sqrt{1-x^2}}$ automatically, with no substitution involved. Example 13.2.4 shows separately that $G(x) = \arcsin x$ — and that computation never uses $\mathcal{D}_x \arcsin x$ at all, only $\mathcal{D}_x \sin\theta = \cos\theta$ and the definition of arcsin as the inverse of sin on $[-\pi/2, \pi/2]$. Combining the two facts,

$$\mathcal{D}_x \arcsin x = G'(x) = \frac{1}{\sqrt{1-x^2}}\,,$$

the same conclusion as Theorem 7.5.7, by a route that never mentions implicit differentiation.

Example 13.2.5 ($x = \sin\theta$ ***applied to*** $\sqrt{1-x^2}$)

This time $\cos\theta$ does not cancel — it multiplies the $\cos\theta$ from dx,

since $\sqrt{1-x^2} = \cos\theta$ is now in the numerator:

$$\int \sqrt{1-x^2}\,dx = \int \cos\theta \cdot \cos\theta\,d\theta = \int \cos^2\theta\,d\theta\,.$$

The power-reduction identity $\cos^2\theta = \frac{1}{2}(1+\cos 2\theta)$ gives

$$\int \cos^2\theta\,d\theta = \frac{1}{2}\big(\theta + \sin\theta\cos\theta\big) + C\,.$$

Substituting back $\theta = \arcsin x$, $\sin\theta = x$, $\cos\theta = \sqrt{1-x^2}$:

$$\int \sqrt{1-x^2}\,dx = \frac{1}{2}\Big(\arcsin x + x\sqrt{1-x^2}\Big) + C\,,$$

matching Proposition 13.2.2 exactly.

The two integrals use the identical substitution, one line of algebra apart, yet one collapses immediately and the other needs an extra identity. The difference is entirely where $\cos\theta$ lands: in the denominator it cancels; in the numerator it multiplies. And for $\int \sqrt{1-x^2}\,dx$, that extra work reproduces — term for term — the triangle-plus-sector computation behind Proposition 13.2.2: $\arcsin x$ is the sector's angle, and $x\sqrt{1-x^2} = \sin\theta\cos\theta$ is twice the triangle's area. The substitution is the geometric picture, in algebraic disguise.

? Open Questions

- The introduction to this chapter observes that `derivative`: AST $\to$ AST is a *total* function (Chapter 5), but no analogous total `antiderivative` function exists — some elementary functions provably have no elementary antiderivative. Given that, is the question "does *this specific* elementary function have an elementary antiderivative?" even *decidable* — is there an algorithm that always answers yes or no in finite time, the way a halting-problem-style question might not be? (It is decidable — the Risch algorithm mentioned in Chapter 16 decides exactly this question — which is itself a little surprising given how close the question sits to classic undecidable ones.)

Chapter 14

Standard Techniques

Chapter Objectives

- Decompose a rational function into partial fractions and integrate each term.
- Recognize and apply the three standard integral forms: $\ln|x+b|$ (linear denominator), $\ln|x^2+b|$ (quadratic denominator, numerator x), and $\arctan$ (irreducible quadratic denominator).
- Apply integration by parts to products of functions.

Chapter 13 covered u-substitution and the trigonometric substitution $x = \sin\theta$. This chapter develops two more families of technique for integrals that neither substitution nor a direct table lookup handles: partial fractions, and — optionally — integration by parts. Despite the name, neither is a rare or specialized trick: partial fractions in particular is one of the most commonly applied integration techniques in practice, precisely because rational functions show up constantly in engineering and applied science.

14.1 Partial Fractions

In this section we discuss the integral of a quotient of polynomials.

$$\int \frac{a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n}{b_0 + b_1 x + b_2 x^2 + \cdots + b_m x^m} \tag{14.1}$$

14.1.1 Setting Up the Decomposition

In Equation (14.1), we can assume that $m > n$ (the denominator has higher degree than the numerator) because if $n \leq m$, recall from the Algebra-1course that a so-called rational function of the form $\frac{N(x)}{D(x)}$ can be rewritten. Using the Euclidian division algorithm we can determine polynomials $Q(x)$ and $R(x)$ such that the polynomial in Equation (14.1) can be decomposed into $\frac{N(x)}{D(x)} = Q(x) + \frac{R(x)}{D(x)}$ where the degree of $R(x)$ is strictly less than the degree of $D(x)$. If the degree of $N(x)$ is already less than the degree of $D(x)$, then simply $Q(x) = 0$ in the decomposition.

Therefore

$$\int \frac{N(x)}{D(x)} dx = \int Q(x)\, dx + \int \frac{R(x)}{D(x)} dx\,,$$

where $\int Q(x)\, dx$ can be computed easily (this is simply integrating a polynomial using linearity and the Power rule), and we are left with the task of computing $\int \frac{R(x)}{D(x)} dx$ with the degree of $R(x)$ strictly less than the degree of $D(x)$.

Example 14.1.1 (*Reducing* $N(x)/D(x)$ *by Long Division*)

Suppose we wisht to evaluate the following intgral:

$$\int \frac{2x^3 + 3x^2 - x + 5}{x^2 + 1} dx$$

which is a case in which the degree of the numerator is larger than the degree of the denominator: $3 > 2$.

Let $N(x) = 2x^3 + 3x^2 - x + 5$, of degree 3, and $D(x) = x^2 + 1$, of degree 2, so the degree of $N(x)$ is *not* already less than the degree of $D(x)$: we are in exactly the situation Equation (14.1) excludes, and must first find $Q(x)$ and $R(x)$ before partial fractions can even begin. Long division of $N(x)$ by $D(x)$ proceeds one term of the quotient at a time, exactly as with integers:

$$\begin{array}{rr|l} & & 2x+3 \\ \hline x^2+1 & 2x^3+3x^2-x+5 & \\ & -2x^3 \quad\quad -2x & \\ \hline & 3x^2-3x+5 & \\ & -3x^2 \quad\quad -3 & \\ \hline & -3x+2 & \end{array}$$

Reading off the quotient above the bar and the final line as the remainder, $Q(x) = 2x + 3$ and $R(x) = -3x + 2$. As a check,

$$(2x+3)(x^2+1) + (-3x+2) = 2x^3 + 3x^2 - x + 5 = N(x)\,,$$

confirming

$$\frac{2x^3+3x^2-x+5}{x^2+1} = 2x + 3 + \frac{-3x+2}{x^2+1}\,,$$

with $R(x) = -3x + 2$ of degree 1, strictly less than the degree of $D(x)$, as required.

We now rewrite the original integral:

$$\begin{aligned}\int \frac{2x^3+3x^2-x+5}{x^2+1}dx &= \int\left(2x+3+\frac{-3x+2}{x^2+1}\right)dx \\ &= \int(2x+3)dx+\int\frac{-3x+2}{x^2+1}dx \\ &= x^2+3x+\int\frac{-3x+2}{x^2+1}dx\end{aligned}$$

The remaining task of computing $\displaystyle\int\frac{-3x+2}{x^2+1}\,dx$ is exactly the kind of integral the rest of this section addresses.

It is true, although we did not show this in the Algebra-1 course, that any polynomial with rational coefficients can be factorized uniquely (except for order) into factors each of which is of one of the forms $(x-r)$ or (x^2+ax+b), each of which may be repeated (multiplicity) — a consequence of the Fundamental Theorem of Algebra (every degree-n polynomial has n complex roots, counted with multiplicity) together with the Complex Conjugate Root Theorem (the non-real roots of a real-coefficient polynomial occur in conjugate pairs, so pairing each conjugate pair back into a factor $(x-r)(x-\bar{r})=x^2+ax+b$ leaves only real linear factors and irreducible real quadratic factors). In this chapter, we handle linear factors $(x-r)$ of multiplicity at most 2, and irreducible quadratic factors (x^2+ax+b) of multiplicity 1.

The fact that $D(x)$ can be factored into one or more of this small variety of terms means that a rational function such as $\frac{R(x)}{D(x)}$ can always be written as a sum of simpler quotients, each of one of the following forms:

- $\displaystyle\frac{A}{x-r}$ — a simple linear denominator (Section 14.1.3)

- $\dfrac{A}{x-r} + \dfrac{B}{(x-r)^2}$ — a repeated linear factor; both numerators are constants (Section 14.1.4)
- $\dfrac{Bx+C}{x^2+ax+b}$ — an irreducible quadratic denominator with a linear numerator (Section 14.1.5, optional)

Example 14.1.2 (*From partial fractions to rational function*)

The opposite direction — from partial fractions to a single rational function — is straightforward algebra. Starting from the sum of simple quotients:

$$\frac{R(x)}{D(x)} = \frac{1}{x+1} + \frac{2}{x-1} - \frac{1}{(x-1)^2} \tag{14.2}$$

$$= \frac{1}{(x+1)}\frac{(x-1)^2}{(x-1)^2} + \frac{2}{(x-1)}\frac{(x+1)(x-1)}{(x+1)(x-1)} - \frac{1}{(x-1)^2}\frac{(x+1)}{(x+1)}$$

$$= \frac{(x-1)^2 + 2(x+1)(x-1) - (x+1)}{(x-1)^2(x+1)}$$

$$= \frac{(x^2-2x+1) + (2x^2-2) - (x+1)}{(x^2-2x+1)(x+1)}$$

$$= \frac{3x^2-3x-2}{x^3-x^2-x+1} \tag{14.3}$$

The algebra problem we have to solve when asked to integrate $\frac{3x^2-3x-2}{x^3-x^2-x+1}$ is how to start with Equation (14.3) and derive Equation (14.2).

The difficulty of integrating by partial fractions expansion is to tediously determine these coefficients A and B for each of the terms in the sum. Once the coefficients have been determined, the integrals can be computed by invoking Propositions 14.1.3, 14.1.4, 14.1.5, and 14.1.9.

Throughout this section the constant of integration is written K rather than C, to avoid confusion with the coefficient C used in the partial fraction decompositions.

One honest question remains: why do rational functions arise in practice? The homework exercise in Chapter 11 provides one concrete example — a data link whose transfer rate is modelled by a rational function of time. More generally, rational functions appear wherever two competing processes interact in a way that produces a ratio, and they are central to the theory of signals and systems you will encounter in later courses. For this course it is enough to know that the algebra you already command makes the technique work.

14.1.2 Three Integrals to Remember

Every partial fraction decomposition reduces to combinations of three standard forms. Each is derived by a short u-substitution from entries already in the derivative table.

The simplest form is a pure linear denominator. Reading the derivative table (Figure 2.1, page 71) right-to-left gives $\int du/u = \ln|u| + K$; shifting the variable by b changes nothing.

Proposition 14.1.3 (*Denominator* $x + b$)

$$\int \frac{dx}{x+b} = \ln|x+b| + K \tag{14.4}$$

Set $u = x + b$. Then $du = dx$, so the integral becomes $\int du/u$.

Proof.

$$\int \frac{dx}{x+b} = \int \frac{du}{u} = \ln|u| + K = \ln|x+b| + K.$$

The second form arises when the denominator is quadratic and the numerator is exactly x. The factor x is the telltale sign: it is, up to a constant factor, the derivative of the denominator $x^2 + b$.

Proposition 14.1.4 (*Denominator* $x^2 + b$*, numerator* x)

$$\int \frac{x\,dx}{x^2 + b} = \tfrac{1}{2} \ln |x^2 + b| + K \tag{14.5}$$

This form requires $b > 0$ in the partial fractions context. If $b < 0$, then $x^2 + b = (x - \sqrt{|b|})(x + \sqrt{|b|})$ is reducible and would already have been split into two linear terms of the form shown in Equation (14.4).

Set $u = x^2 + b$. Then $du = 2x\,dx$, so $x\,dx = \frac{1}{2}\,du$ and the x in the numerator is absorbed exactly.

Proof.

$$\begin{aligned} \int \frac{x\,dx}{x^2 + b} &= \frac{1}{2} \int \frac{du}{u} \\ &= \frac{1}{2} \ln |u| + K \\ &= \frac{1}{2} \ln |x^2 + b| + K\,. \end{aligned}$$

The third form handles a denominator that is a sum of squares. In the partial fractions context, b^2 is the positive constant that remains after completing the square on an irreducible quadratic; it is always positive because a negative value would mean the factor is reducible.

Proposition 14.1.5 (*Denominator* $x^2 + b^2$)

$$\int \frac{dx}{x^2 + b^2} = \frac{1}{b} \arctan \frac{x}{b} + K \tag{14.6}$$

Set $u = x/b$, so $x = bu$ and $dx = b\,du$. This normalizes the denominator to $b^2(u^2 + 1)$, and then $\int du/(u^2 + 1) = \arctan(u) + K$ follows from the derivative table (Figure 2.1), $(\arctan)' = 1/(1 + u^2)$.

Proof.

$$\begin{aligned} \int \frac{dx}{x^2 + b^2} &= \int \frac{b\,du}{b^2u^2 + b^2} \\ &= \frac{1}{b} \int \frac{du}{u^2 + 1} \\ &= \frac{1}{b} \arctan(u) + K \\ &= \frac{1}{b} \arctan \frac{x}{b} + K\,. \end{aligned}$$

14.1.3 Simple Linear Denominators

When the denominator factors into *distinct* linear factors $(x - r_1)(x - r_2) \cdots (x - r_k)$, the decomposition places one unknown constant over each factor:

$$\frac{p(x)}{(x - r_1)(x - r_2) \cdots (x - r_k)} = \frac{A_1}{x - r_1} + \frac{A_2}{x - r_2} + \cdots + \frac{A_k}{x - r_k}\,.$$

Multiply both sides by the denominator, then find each A_i by substituting $x = r_i$ (which annihilates every term except the one containing

A_i). Each piece integrates to $A_i \ln|x - r_i|$.

Example 14.1.6 (*Simple linear factors*)

Compute $\displaystyle\int \frac{1}{x^2-1}\,dx$.

Factor: $x^2 - 1 = (x-1)(x+1)$. Write

$$\frac{1}{(x-1)(x+1)} = \frac{A}{x-1} + \frac{B}{x+1}\,.$$

$$\begin{aligned}
\frac{1}{(x-1)(x+1)} &= \frac{A}{x-1} + \frac{B}{x+1} \\
(x+1)(x-1)\frac{1}{(x-1)(x+1)} &= \left(\frac{A}{x-1} + \frac{B}{x+1}\right)(x+1)(x-1) \\
1 &= A\,(x+1) + B\,(x-1) \\
x = 1 &\implies 1 = 2A \\
x = -1 &\implies 1 = -2B
\end{aligned}$$

So we have $A = \frac{1}{2}$ and $B = -\frac{1}{2}$. Both terms are scalar multiples of a linear denominator, so Proposition 14.1.3 applies to each:

$$\begin{aligned}
\frac{1}{(x-1)(x+1)} &= \frac{1/2}{x-1} - \frac{1/2}{x+1} \\
\int \frac{1}{(x-1)(x+1)}dx &= \int \frac{1/2}{x-1}dx - \int \frac{1/2}{x+1}dx \\
&= \tfrac{1}{2}(\ln|x-1| - \ln|x+1|) + K \\
&= \tfrac{1}{2}\ln\left|\frac{x-1}{x+1}\right| + K\,.
\end{aligned}$$

Heaviside Cover-up Method: For simple linear factors there is

a faster route which lets us find the numerator constants by inspection. To find the constant A_i over $(x - r_i)$, cover that factor in the denominator with your thumb and evaluate the rest at $x = r_i$. In the example above:

$$A = \left.\frac{1}{x+1}\right|_{x=1} = \frac{1}{2}\,, \qquad B = \left.\frac{1}{x-1}\right|_{x=-1} = \frac{1}{-2} = -\frac{1}{2}\,.$$

This is algebraically identical to multiplying through and substituting $x = r_i$, but skips the intermediate step. The method works for every simple linear factor simultaneously and in any order.

Example 14.1.7 (*Heaviside Cover-up Method*)

Factor: $x^2 - 1 = (x-1)(x+1)$. Write

$$\frac{1}{(x-1)(x+1)} = \frac{A}{x-1} + \frac{B}{x+1}\,.$$

Now we can determine A and B by inspection.

1. Cover up $(x-1)$ and all the terms on the right which lack $x-1$ in the denominator. Since $x - 1 = 0 \implies x = 1$, we evaluate the result at $x = 1$.

$$\frac{1}{\cancel{(x-1)}(x+1)} = \frac{A}{\cancel{x-1}} + \cancel{\frac{B}{x+1}}$$
$$\left.\frac{1}{x+1}\right|_{x=1} = A$$
$$A = \frac{1}{1+1} = \frac{1}{2}$$

2. Cover up $(x+1)$ and all the terms on the right which lack $x+1$ in the denominator. Since $x + 1 = 0 \implies x = -1$, we

evaluate the result at $x = -1$.

$$\frac{1}{(x-1)\cancel{(x+1)}} = \frac{\cancel{A}}{\cancel{x-1}} + \frac{B}{\cancel{x+1}}$$

$$\left.\frac{1}{x-1}\right|_{x=-1} = B$$

$$B = \frac{1}{-1-1} = -\frac{1}{2}$$

So exactly as in Example 14.1.6 we have $A = \frac{1}{2}$ and $B = -\frac{1}{2}$ and Proposition 14.1.3 applies to each term.

$$\frac{1}{(x-1)(x+1)} = \frac{1/2}{x-1} - \frac{1/2}{x+1}$$

$$\begin{aligned}\int \frac{1}{(x-1)(x+1)}dx &= \int \frac{1/2}{x-1}dx - \int \frac{1/2}{x+1}dx \\ &= \tfrac{1}{2}(\ln|x-1| - \ln|x+1|) + K \\ &= \tfrac{1}{2}\ln\left|\frac{x-1}{x+1}\right| + K\,.\end{aligned}$$

It breaks down in two situations. For a *repeated* factor $(x-r)^m$ it gives only the coefficient of the highest power $\frac{A_m}{(x-r)^m}$; the lower-power coefficients still require coefficient matching. For *irreducible quadratic* factors there is no root to substitute, so cover-up gives nothing at all.

14.1.4 Repeated Linear Factors

If a linear factor $(x-r)$ appears with multiplicity m in the denominator, the decomposition must include one term for each power up to m:

$$\frac{p(x)}{(x-r)^m} \supset \frac{A_1}{x-r} + \frac{A_2}{(x-r)^2} + \cdots + \frac{A_m}{(x-r)^m}\,.$$

The powers integrate as $\int (x-r)^{-k}\,dx = \frac{(x-r)^{1-k}}{1-k}$ for $k \geq 2$, and as $\ln|x-r|$ for $k=1$.

Example 14.1.8 (*Repeated linear factor*)

Compute $\displaystyle\int \frac{1}{x(x-1)^2}\,dx$.

Decompose:

$$\frac{1}{x(x-1)^2} = \frac{A}{x} + \frac{B}{x-1} + \frac{C}{(x-1)^2}\,.$$

Multiply through: $1 = A(x-1)^2 + Bx(x-1) + Cx$.

Substitute $x=0$: $1=A$.

Substitute $x=1$: $1=C$.

Match the x^2 coefficient: $0 = A+B$, so $B=-1$.

The A/x and $B/(x-1)$ terms both use Proposition 14.1.3. The $C/(x-1)^2$ term is a negative-integer power: $\int (x-1)^{-2}\,dx = -(x-1)^{-1}$, which is an ordinary power rule and does not match any of the three standard forms.

Therefore

$$\int \frac{dx}{x(x-1)^2} = \ln|x| - \ln|x-1| - \frac{1}{x-1} + K\,.$$

Why is the numerator over $(x-r)^k$ always a constant, and not something more general like $Cx+D$? Because a linear numerator over a repeated linear factor is never genuinely new — it always reduces to

the simpler form. Write $Cx + D$ by adding and subtracting Cr:

$$\begin{aligned}\frac{Cx+D}{(x-r)^2} &= \frac{Cx - Cr + Cr + D}{(x-r)^2} \\ &= \frac{C(x-r) + (Cr+D)}{(x-r)^2} \\ &= \frac{C}{x-r} + \frac{Cr+D}{(x-r)^2}\,.\end{aligned}$$

So $Cx + D$ splits into two terms that are already in our decomposition. The general principle: the numerator over each power of an irreducible factor must have degree *strictly less* than the degree of that factor. A linear factor has degree 1, so its numerators are degree 0 (constants). An irreducible quadratic has degree 2, which is why the next subsection allows degree-1 (linear) numerators.

14.1.5 Irreducible Quadratic Denominators (Optional)

A quadratic $x^2 + px + q$ is *irreducible* over $\mathbb{R}$ when its discriminant $p^2 - 4q < 0$, meaning it has no real roots and cannot be factored into linear factors. For each such factor in the denominator, the decomposition contributes a term $\frac{Bx+C}{x^2+px+q}$.

Proposition 14.1.9 (*Linear over Irreducible Quadratic*)

Let $x^2 + px + q$ be irreducible over $\mathbb{R}$ and set $a = \sqrt{q - p^2/4} > 0$. Then

$$\int \frac{Bx+C}{x^2+px+q}\,dx = \frac{B}{2}\ln(x^2+px+q) + \frac{C - \frac{Bp}{2}}{a}\arctan\frac{x+\frac{p}{2}}{a} + K\,.$$

Note that $\ln(x^2+px+q)$ carries no absolute value bars. Irreducibility ($p^2 - 4q < 0$) means the quadratic has no real roots, so it never

reaches zero; since its leading coefficient is positive it is strictly positive for all $x \in \mathbb{R}$, and ln of a positive number needs no bars.

Write $Bx + C = \frac{B}{2}(2x + p) + (C - \frac{Bp}{2})$ to split the numerator into a part proportional to the derivative of the denominator and a constant residue. Proposition 14.1.4 handles the first piece and Proposition 14.1.5 handles the second, after completing the square.

Proof. Split the numerator: $Bx + C = \frac{B}{2}(2x + p) + (C - \frac{Bp}{2})$. Then

$$\int \frac{Bx + C}{x^2 + px + q}\,dx = \frac{B}{2}\int \frac{(2x + p)\,dx}{x^2 + px + q} + \Big(C - \frac{Bp}{2}\Big)\int \frac{dx}{x^2 + px + q}\,.$$

First integral. Since $d(x^2 + px + q) = (2x + p)\,dx$, the substitution $u = x^2 + px + q$ gives

$$\frac{B}{2}\int du/u = \frac{B}{2}\ln(x^2 + px + q)$$

(Proposition 14.1.4 with u in place of x).

Second integral. Complete the square: $x^2 + px + q = (x + \frac{p}{2})^2 + a^2$ with $a^2 = q - p^2/4 > 0$. Setting $v = x + p/2$ gives $\int dv/(v^2 + a^2) = \frac{1}{a}\arctan(v/a)$ (Proposition 14.1.5). Multiplying by the coefficient $(C - Bp/2)$ yields the arctan term.

Example 14.1.10 (*Irreducible quadratic, logarithm case*)

Compute $\displaystyle\int \frac{1}{x(x^2 + 1)}\,dx$.

The factor $x^2 + 1$ is irreducible (discriminant $-4 < 0$). Decom-

pose:

$$\frac{1}{x(x^2+1)} = \frac{A}{x} + \frac{Bx+C}{x^2+1}\,.$$

Multiply through: $1 = A(x^2+1) + (Bx+C)x$.

Substitute $x = 0$: $A = 1$. Match x^2: $B = -1$. Match x^1: $C = 0$.

Apply Proposition 14.1.3 to A/x and Proposition 14.1.9 to $(Bx+C)/(x^2+1)$ with $p = 0$, $q = 1$, $a = 1$, $B = -1$, $C = 0$:

$$\begin{aligned}\int \frac{dx}{x(x^2+1)} &= \ln|x| + \left(\tfrac{-1}{2}\ln(x^2+1) + 0\cdot\arctan(x)\right) + K \\ &= \ln|x| - \tfrac{1}{2}\ln(x^2+1) + K\,.\end{aligned}$$

Example 14.1.11 (*Irreducible quadratic, arctangent case*)

Compute $\displaystyle\int \frac{1}{(x+1)(x^2+1)}\,dx$.

Decompose:

$$\frac{1}{(x+1)(x^2+1)} = \frac{A}{x+1} + \frac{Bx+C}{x^2+1}\,.$$

Multiply through: $1 = A(x^2+1) + (Bx+C)(x+1)$.

Substitute $x = -1$: $1 = 2A$, so $A = \frac{1}{2}$.

Match x^2: $0 = A + B$, so $B = -\frac{1}{2}$.

Match x^0: $1 = A + C$, so $C = \frac{1}{2}$.

Apply Proposition 14.1.3 to $A/(x+1)$ and Proposition 14.1.9

to $(Bx+C)/(x^2+1)$ with $p=0$, $q=1$, $a=1$, $B=-\frac{1}{2}$, $C=\frac{1}{2}$:

$$\begin{aligned}\int \frac{dx}{(x+1)(x^2+1)} &= \tfrac{1}{2}\ln|x+1| \\ &\quad + \left(\tfrac{-1/2}{2}\ln(x^2+1) + \tfrac{1/2-0}{1}\arctan x\right) + K \\ &= \tfrac{1}{2}\ln|x+1| - \tfrac{1}{4}\ln(x^2+1) + \tfrac{1}{2}\arctan(x) + K\,.\end{aligned}$$

14.2 Integration by Parts (Optional)

Integration by parts is the product rule running backwards. Recall that $(uv)' = u'v + uv'$, which integrates to $uv = \int u'v\,dx + \int uv'\,dx$. Rearranging:

Theorem 14.2.1 (*Integration by Parts*)

$$\int u\,dv = uv - \int v\,du\,.$$

The goal is to choose u and dv so that $\int v\,du$ is simpler than $\int u\,dv$. In practice you'll need to partition a given function into two parts, u and v, in such a way that you can compute the derivative of u and the integral of $v\,du$. It is not always obvious how to define u and v, but one hope is to choose u so that when you compute its derivative, it becomes simpler. For example $\mathcal{D}\,x = 1$ (simpler) or $\mathcal{D}\ln(x) = \frac{1}{x}$ (simpler).

A reliable heuristic for choosing u is the **LIATE** priority order: prefer u to be a **L**ogarithm, then an **I**nverse trigonometric function, then an **A**lgebraic (polynomial) function, then a **T**rigonometric function, then an **E**xponential. Whatever is left becomes dv. The motivation is that the derivative of a logarithm removes the logarithm and thus

is simpler than the logarithm itself, and the derivative of an inverse trigonometric function removes the inverse trigonometric function, reducing the expression to some quotient of polynomials and possibly square roots. If these derivatives are unclear, take a fresh look at Figure 2.1.

Example 14.2.2 (*Polynomial times exponential*)

Compute $\int x\,e^x\,dx$.

x is algebraic; e^x is exponential. LIATE puts algebraic before exponential, so choose $u = x$ and $dv = e^x\,dx$, giving $du = dx$ and $v = e^x$:

$$\int x\,e^x\,dx = xe^x - \int e^x\,dx = xe^x - e^x + C = (x-1)e^x + C\,.$$

Example 14.2.3 (*Logarithm alone*)

Compute $\int \ln x\,dx$.

There is no obvious second factor, but we can write $dv = dx$ so that $v = x$. LIATE puts logarithm first, so $u = \ln x$, $du = \frac{1}{x}\,dx$:

$$\int \ln x\,dx = x\ln x - \int x\cdot\frac{1}{x}\,dx = x\ln x - \int dx = x\ln x - x + C\,.$$

? Open Questions

- LIATE is explicitly called a *heuristic*, not a theorem. Can you find an integral where following LIATE's priority order literally leads to a harder integral than you started with, forcing you to swap the choice of u and dv? (Try $\int e^x \sin x \, dx$ and apply integration by parts twice, always choosing $u =$ the trig or exponential factor consistently — what happens to the integral on the right-hand side after the second application?)
- This chapter's partial-fractions treatment is explicitly scoped to "linear factors $(x-r)$ of multiplicity at most 2, and irreducible quadratic factors of multiplicity 1." What does the decomposition look like for a linear factor of multiplicity 3, such as $(x-r)^3$? Using the pattern from Section 14.1.4, how many unknown constants would you need, and what would the general term look like for multiplicity m?

Chapter 15

HW-3: Variable Data Transfer Rate

Chapter Objectives

- Evaluate integrals using u-substitution, selecting the substitution that simplifies the integrand.
- Integrate rational functions by partial fraction decomposition, including simple and repeated linear factors.
- Apply integration by parts to evaluate products of functions.
- Model a multi-phase data transfer process by integrating a piecewise rate function and compute the total data transferred.

Key relationship. If $r(t)$ is the instantaneous data transfer rate (in Megabits per second) at time t (in seconds), then the total number of Megabits transferred over an interval $[a, b]$ is

$$\int_a^b r(t)\,dt\,.$$

This is a direct application of the Fundamental Theorem of Calculus, Part 2 (Theorem 10.2.2): the integral accumulates the instantaneous rate over time.

15.1 U-Substitution Practice

For each integral, identify a substitution $u = g(x)$, rewrite the integral in terms of u, integrate, and substitute back. Use `scipy.integrate.quad` to verify your answers numerically.

1. $\displaystyle\int (2x+1)(x^2+x+3)^4\,dx$
2. $\displaystyle\int \sin(3x+1)\,dx$
3. $\displaystyle\int x\,e^{x^2}\,dx$
4. $\displaystyle\int \frac{x}{x^2+5}\,dx$
5. $\displaystyle\int \frac{\sin(\ln x)}{x}\,dx$
6. $\displaystyle\int_1^2 \frac{2x}{(x^2+1)^2}\,dx$ (definite integral; express the answer as a fraction)

15.2 A Second Proof of $\mathcal{D}_x \arccos x$ (Optional)

Section 13.2 used the substitution $t = \sin\theta$ to evaluate $\displaystyle\int \frac{dt}{\sqrt{1-t^2}}$ (Example 13.2.4), and observed that this gives a second proof of $\mathcal{D}_x \arcsin x = \frac{1}{\sqrt{1-x^2}}$ (Theorem 7.5.7) that never uses implicit differentiation. This exercise repeats that argument for arccos, using $t = \cos\theta$ in place of $t = \sin\theta$.

Define

$$H(x) = \int_1^x \frac{-1}{\sqrt{1-t^2}}\, dt\,, \qquad -1 < x \le 1\,.$$

1. Apply FTC Part 1 (Theorem 10.2.1) directly to $H(x)$ to find $H'(x)$. No substitution is needed for this part.

2. Substitute $t = \cos\theta$, choosing $\theta \in [0, \pi]$ so that $\sin\theta \ge 0$ throughout (compare Theorem 13.2.3, which restricted $\theta \in [-\pi/2, \pi/2]$ for the substitution $t = \sin\theta$). Write dt in terms of $d\theta$, and simplify $\sqrt{1-t^2}$ in terms of θ.

3. Convert the limits of integration: what value of θ corresponds to $t = 1$? What value corresponds to $t = x$? (The second answer is a familiar inverse trig expression.)

4. Rewrite $H(x)$ entirely in terms of θ, evaluate the resulting integral, and substitute back to show $H(x) = \arccos x$.

5. Combine parts 1 and 4 to conclude

$$\mathcal{D}_x \arccos x = \frac{-1}{\sqrt{1-x^2}}\,,$$

matching Theorem 7.5.8 (Chapter 7) — by a route that never mentions implicit differentiation.

6. Explain why $H(x)$ was defined with lower limit 1 rather than 0. *Hint:* what is $\arccos 1$? Compare with why $G(x) = \int_0^x \frac{dt}{\sqrt{1-t^2}}$ in the text uses lower limit 0.

15.3 Partial Fractions Practice

For each integral, decompose the integrand into partial fractions, then integrate each term using the appropriate proposition from Section 14.1.

1. $\displaystyle\int \frac{1}{x^2-1}\,dx$

   Hint: factor the denominator first.

2. $\displaystyle\int \frac{2x+3}{(x+1)(x+3)}\,dx$

3. $\displaystyle\int \frac{5}{x(x+5)}\,dx$

4. $\displaystyle\int \frac{x+4}{(x-1)^2}\,dx$

   Hint: the repeated linear form is $\dfrac{A}{x-1}+\dfrac{B}{(x-1)^2}$ (Proposition 14.1.3, Section 14.1.4).

5. $\displaystyle\int \frac{3}{(x+1)(x^2+4)}\,dx$

   Hint: one linear factor plus one irreducible quadratic (Section 14.1.5).

15.4 Integration by Parts Practice (Optional)

These problems use the integration by parts formula (Theorem 14.2.1) from Section 14.2. For each integral, identify u and dv using the LIATE priority order, then apply the formula. Use `scipy.integrate.quad` to verify your answers numerically.

1. $\displaystyle\int x \cos x \, dx$

2. $\displaystyle\int x \, e^{-x} \, dx$

3. $\displaystyle\int x \ln x \, dx$

   Hint: LIATE puts the logarithm before the algebraic factor, so choose $u = \ln x$.

4. $\displaystyle\int \arctan x \, dx$

   Hint: write $dv = dx$ so that $v = x$, and use the fact that $\mathcal{D}_x \arctan x = \dfrac{1}{1 + x^2}$.

5. $\displaystyle\int x^2 e^x \, dx$

   Hint: apply integration by parts twice.

15.5 Variable Data Transfer Rate

A data link transfers information at an instantaneous rate $r(t)$ (Megabits per second) that varies over time. The link operates in three consecutive phases. In each phase, t denotes the time elapsed since the start of that phase, measured in seconds.

15.5.1 Phase 1: Slow Start

During the first $\sqrt{3}$ seconds the connection ramps up. The transfer rate is modelled by

$$r_1(t) = \frac{t}{\sqrt{t^2+1}}, \quad t \in \left[0, \sqrt{3}\right].$$

1. Identify a substitution $u = g(t)$ that simplifies the integral $\displaystyle\int_0^{\sqrt{3}} \frac{t}{\sqrt{t^2+1}}\,dt$. Write du in terms of dt.

2. Rewrite the integral entirely in terms of u, including the new limits of integration.

3. Find the antiderivative in terms of u, then substitute back to obtain an antiderivative in terms of t.

4. Evaluate the definite integral. How many Megabits are transferred during Phase 1?

15.5.2 Phase 2: Dual Bottleneck

After the slow-start phase the connection is simultaneously limited by two independent constraints—CPU processing delay and network

latency—whose combined effect gives a rate inversely proportional to their product:

$$r_2(t) = \frac{6}{(t+1)(t+3)}, \quad t \in [0, 1].$$

1. Write $\dfrac{6}{(t+1)(t+3)} = \dfrac{A}{t+1} + \dfrac{B}{t+3}$ and determine the constants A and B.

2. Integrate each term separately, recalling that

$$\int \frac{1}{t+c}\, dt = \ln|t+c| + C\,.$$

3. Evaluate the definite integral from 0 to 1 and express your answer as a single logarithm. How many Megabits are transferred during Phase 2?

15.5.3 Phase 3: Traffic Burst

A burst of traffic arrives whose instantaneous rate follows a Lorentzian profile centered at $t = 0$:

$$r_3(t) = \frac{2}{t^2+1}, \quad t \in [0, 1].$$

This is the irreducible quadratic denominator case from Section 14.1.

1. Identify the antiderivative of $\dfrac{2}{t^2+1}$ from the derivative table in Chapter 2.

2. Evaluate the definite integral from 0 to 1 and express your answer in terms of π. How many Megabits are transferred during Phase 3?

15.5.4 Total Data Transferred

1. Write the total Megabits transferred across all three phases as a sum of three definite integrals.

2. Using the Interval Splitting Proposition (Proposition 8.7.4), interpret this sum as a single integral over the full duration of the connection.

3. Substitute your results from Phases 1–3 to obtain a closed-form expression for the total. Your answer should involve an integer, a natural logarithm, and π.

4. Use Python to obtain a numerical approximation in Megabits.

Chapter 16

Debriefing: Could You Implement `integrate()`?

Chapter Objectives

- Explain why symbolic integration has no finite algorithm analogous to the derivative rules.
- Identify which integration problems are tractable by pattern matching and which are not.
- Describe at a high level how the Risch algorithm decides whether an elementary antiderivative exists.
- Articulate why `scipy.integrate.quad` is the practical tool of choice for definite integrals in engineering and science.
- *(Enrichment)* Identify the three mathematical worlds $\mathcal{A}$, $\mathcal{E}$, $C^1(\mathbb{R})$ and the surjective differential-ring homomorphism eval: $\mathcal{A} \to \mathcal{E}$; explain why `.derivative()` is total on $\mathcal{E}$ while the antiderivative is only partial.

16.1 A Thought Exercise

You have implemented `derivative()` on the `PointFree` AST. Given an expression tree, it returns the expression tree of the derivative — mechanically, correctly, every time. The rules are finite in number, and each one maps one or two child trees to a parent tree.

You have also learned several techniques of integration: linearity, the table of known antiderivatives, u-substitution, partial fractions, and integration by parts.

The thought exercise: *Could you implement an `integrate()` method on `PointFree`? How would you approach it? Which parts are tractable? Which parts are not?*

Do not attempt to write the code. Instead, think through the design and read the discussion below.

16.2 What Would Be Tractable

Some cases do have a mechanical rule, directly mirroring the derivative table. You could handle these with a `match` / `case` block just like `derivative()`:

Expression	Antiderivative
`Numeric(c)`	`Product(Numeric(c), Identity())`
`Power(n)`	`Quotient(Power(n+1), Numeric(n+1))` $(n \neq -1)$
`Sin()`	`Product(Numeric(-1), Cos())`
`Cos()`	`Sin()`
`Exp()`	`Exp()`
`Sum(f, g)`	`Sum(integrate(f), integrate(g))`
`Difference(f, g)`	`Difference(integrate(f), integrate(g))`

The last two rows work because integration, like differentiation, is a linear operator: $\int (f \pm g) = \int f \pm \int g$. Similarly, `Product(Numeric(c), f)` reduces to `Product(Numeric(c), integrate(f))`.

16.3 Where It Breaks Down

16.3.1 Products of Two Non-Constant Functions

For `Product(f, g)` where neither f nor g is a constant, there is no general rule. Integration by parts sometimes helps — $\int f\,g\,dx = f\int g\,dx - \int f' \int g\,dx\,dx$ — but this only simplifies the problem if the remaining integral is easier than the original. Whether that is the case depends on the specific f and g in a way that is hard to decide algorithmically. There is no product rule for integration.

16.3.2 Composition

For `Compose(f, g)`, u-substitution sometimes works: if the integrand contains $g(x)$ and its derivative $g'(x)$ as a factor, the substitution $u = g(x)$ simplifies the integral. But detecting this pattern in an arbitrary AST requires comparing subtrees in ways that go well beyond simple pattern matching. Most compositions yield integrals that u-substitution cannot simplify.

16.3.3 Quotients

For `Quotient(f, g)`, partial fractions works when the numerator and denominator are polynomials and the degree of the numerator is less than the degree of the denominator. Implementing this requires polynomial factorisation — an algorithm in its own right — and only handles the rational-function case.

16.4 The Deeper Obstruction

Even setting aside the algorithmic difficulties above, there is a more fundamental problem: **some elementary functions simply have no antiderivative expressible in terms of elementary functions.**

For example:

$$\int e^{-x^2}\,dx, \qquad \int \frac{\sin x}{x}\,dx, \qquad \int \frac{1}{\ln x}\,dx$$

None of these can be written as a finite combination of polynomials, exponentials, logarithms, and trigonometric functions. The antiderivatives exist — they are perfectly well-defined real-valued functions — but they cannot be expressed in the language of our ADT. No implementation of `integrate()` working within the `PointFree` ADT could ever return a correct answer for these inputs.

16.5 The Risch Algorithm

Computer algebra systems such as SymPy, Mathematica, and Maple do implement symbolic integration. The theoretical foundation is the *Risch algorithm* (1969), which decides — for a large class of elementary functions — whether an elementary antiderivative exists, and if so, computes it. The algorithm is correct and complete for that class, but it is enormously more complex than the derivative rules: the full implementation runs to thousands of lines of carefully structured case analysis.

By contrast, `derivative()` is perhaps thirty lines.

16.6 Calculus Using Large Language Models?

You may have noticed that large language models (LLMs) such as ChatGPT or Claude handle symbolic integration surprisingly well. Ask one to compute $\int x^2 e^x \, dx$ and it will usually give you the correct answer with a clear worked solution. How?

The short answer is: **training data**. LLMs are trained on vast quantities of mathematical text — textbooks, lecture notes, solution sets, forum posts — that contain thousands of worked integration examples. When asked to integrate, the model is not executing an algorithm; it is completing a pattern it has seen many times before. This is a fundamentally different mechanism from the Risch algorithm, which reasons from first principles.

The practical consequences are telling:

- **Common integrals**: LLMs perform very well. The more standard the integral, the more training examples cover it, and the more reliable the output.

- **Unusual integrals**: reliability drops. An integral that does not closely resemble anything in the training data may produce a plausible-looking but incorrect antiderivative. The model has no way to know it has gone wrong.
- **No certificate of correctness**: the Risch algorithm either produces a provably correct antiderivative or declares that none exists. An LLM produces a confident-looking answer with no such guarantee.
- **Verification**: the right response to an LLM-produced antiderivative $F(x)$ is to differentiate it and check that $F'(x)$ equals the original integrand $f(x)$. You can do this with your `derivative()` implementation. There is a subtlety, however: $F'(x)$ and $f(x)$ will almost certainly be structurally different ASTs even when they are semantically equal. As Section 6.2 discussed, structural equality is too strict a test — the right check is *numerical*: evaluate both expressions at several sample points and confirm they agree within a small tolerance. This is exactly what `eval_string` and `evaluate` already do.

This raises an interesting question: if an LLM has absorbed enough mathematical writing to integrate competently, what does that tell us about the nature of the skill? One reading is that integration, for the class of functions that appear in textbooks, is largely *pattern recognition* — matching the form of the integrand to a catalog of known techniques. That is exactly what the chapters on partial fractions and u-substitution trained you to do. The LLM has learned the same catalog, from the same sources, at much larger scale.

The deeper difficulty — functions with no elementary antiderivative, or integrands that require genuinely novel reasoning — remains beyond reliable LLM reach, for the same reason it is beyond a student who has only memorised patterns without understanding them.

16.7 Why We Use `scipy.integrate.quad`

This asymmetry explains a practical choice made throughout this course. Whenever we needed to evaluate a definite integral numerically, we reached for `scipy.integrate.quad` rather than trying to find an antiderivative first. Numerical integration — approximating $\int_a^b f(x)\,dx$ by Riemann sums or more sophisticated quadrature rules — works for any continuous function, regardless of whether a closed-form antiderivative exists. Symbolic integration is powerful when it works, but it does not always work.

The existence of `scipy.integrate.quad` is not a convenience shortcut. For many functions that arise in practice, it is the only tool available.

16.8 Three Worlds: The Algebra Behind `.derivative()`

Figure 16.1 previews where this section is headed. Throughout this course you have been moving between three distinct mathematical worlds without always naming them. This section makes the structure explicit: which sets are involved, how they relate to one another, and how `.derivative()` and $\mathcal{D}_x$ relate as mathematical objects.

16.8.1 The Three Sets

- $\mathcal{A}$: the **set of all ASTs** constructible from the grammar of your `PointFree` ADT (`Numeric`, `Identity`, `Sum`, `Product`, `Compose`, `Power`, `Sin`, `Cos`, `Exp`, `Ln`, ...). Because the grammar is a context-free grammar, $\mathcal{A}$ is a *recursively enumer-*

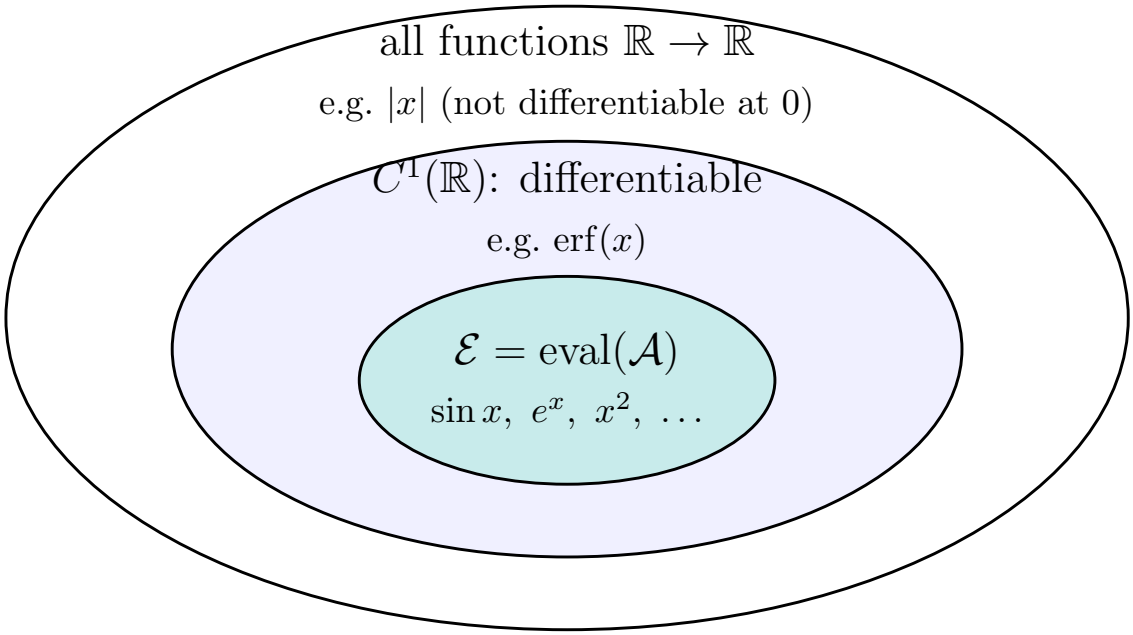


Figure 16.1: The three worlds, nested by strict inclusion. $\mathcal{E}$, the elementary functions expressible by some AST, sits inside $C^1(\mathbb{R})$, the differentiable functions — $\mathrm{erf}(x)$ is differentiable but has no AST. $C^1(\mathbb{R})$ in turn sits inside the set of all functions $\mathbb{R} \to \mathbb{R}$ — $|x|$ is a well-defined function that is not differentiable at 0.

able set — in fact it is *recursive* (decidable): you can enumerate all ASTs systematically by depth, and you can decide in finite time whether any given object is a valid AST. $\mathcal{A}$ is countably infinite.

- $\mathcal{E}$: the **set of elementary functions** — functions $\mathbb{R} \to \mathbb{R}$ expressible by at least one AST. This is the image of the evaluation map $\mathrm{eval} : \mathcal{A} \to \mathcal{E}$. Because $\mathcal{A}$ is countable, so is $\mathcal{E}$.

- $C^1(\mathbb{R})$: the **set of all differentiable functions** $\mathbb{R} \to \mathbb{R}$. This is uncountably infinite. It contains functions with no AST: the error function $\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\, dt$, for instance, is differentiable

but expressible by no finite combination of our primitives.

The inclusions are strict:

$$\mathcal{E} \subsetneq C^1(\mathbb{R}) \subsetneq \{\text{all functions } \mathbb{R} \to \mathbb{R}\}.$$

16.8.2 The Evaluation Map and the First Isomorphism Theorem

In the programming-languages literature, the function that maps a syntactic expression to its mathematical meaning is written $[\![\cdot]\!]$ ("semantic bracket" or "denotation bracket", also called Oxford brackets). Our map eval : $\mathcal{A} \to \mathcal{E}$ is exactly this: $[\![t]\!]$ is the function denoted by the AST t. We use both notations interchangeably.

The connection to the Python method you have already written is direct. `.evaluate(x)` computes the semantic function applied pointwise to a specific number:

$$t.\texttt{evaluate}(x) \;=\; [\![t]\!](x), \qquad x \in \mathbb{R}.$$

In types: $[\![\cdot]\!] : \mathcal{A} \to \mathcal{E}$ maps an AST to a function (a mathematical object), while `.evaluate` : $\mathcal{A} \times \mathbb{R} \to \mathbb{R}$ applies that function pointwise to a number. They are the curried and uncurried versions of the same operation.

The map $[\![\cdot]\!]$ has the following properties:

- **Surjective** by definition of $\mathcal{E}$.
- **Not injective**: two distinct ASTs can denote the same function. For example, `Add(Identity(), Identity())` and `Product(Numeric(2), Identity())` are syntactically different but $[\![\,\texttt{Add(...)}\,]\!] = [\![\,\texttt{Product(...)}\,]\!] = (x \mapsto 2x)$.

- **A ring homomorphism**: $[\![\cdot]\!]$ respects the ring operations. Writing $+_{\mathcal{A}}$ and $\times_{\mathcal{A}}$ for "wrap in an `Add` / `Product` node":

$$[\![s +_{\mathcal{A}} t]\!] = [\![s]\!] + [\![t]\!], \qquad [\![s \times_{\mathcal{A}} t]\!] = [\![s]\!] \cdot [\![t]\!].$$

Define the equivalence relation $s \sim t \iff [\![s]\!] = [\![t]\!]$ ("these two ASTs compute the same function"). The *first ring isomorphism theorem* then gives:

$$\mathcal{A} / \sim \;\cong\; \mathcal{E}.$$

The quotient $\mathcal{A}/\sim$ is the ring of "abstract expressions modulo algebraic identity": the object you implicitly work with whenever you say *$x + x$ is the same expression as $2x$.*

16.8.3 Differential Rings and the Commutative Diagram

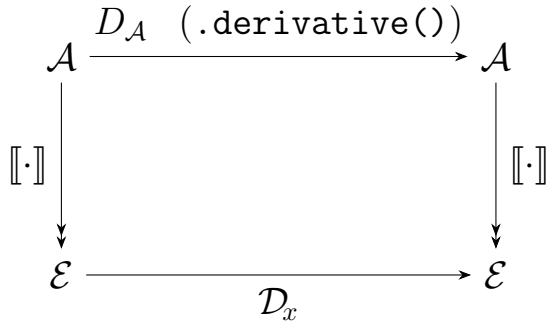
A *differential ring* is a ring R equipped with a *derivation* $\partial : R \to R$, meaning ∂ is linear and satisfies the Leibniz product rule $\partial(fg) = f\,\partial(g) + \partial(f)\,g$.

Both $\mathcal{E}$ and $C^1(\mathbb{R})$ are differential rings under $\mathcal{D}_x$, and $\mathcal{E}$ is a differential subring of $C^1(\mathbb{R})$:

$$\big(\mathcal{E},\, +,\, \cdot,\, \mathcal{D}_x\big) \;\subseteq\; \big(C^1(\mathbb{R}),\, +,\, \cdot,\, \mathcal{D}_x\big).$$

The `PointFree` ADT, equipped with `.derivative()`, forms a differential ring $(\mathcal{A},\, +_{\mathcal{A}},\, \times_{\mathcal{A}},\, D_{\mathcal{A}})$ where $D_{\mathcal{A}}$ denotes the `.derivative()` method.

The key relationship between $D_{\mathcal{A}}$ and $\mathcal{D}_x$ is the *commutative diagram*:



The double-headed arrow ($\twoheadrightarrow$) denotes a surjection. Commutativity means:

$$[\![D_{\mathcal{A}}(t)]\!] \;=\; \mathcal{D}_x\big([\![t]\!]\big) \quad \text{for every } t \in \mathcal{A}.$$

In plain language: *differentiating the AST and then evaluating gives the same function as evaluating and then differentiating.* This is precisely the soundness claim for your derivative rules.

Algebraically, $[\![\cdot]\!]$ is a surjective **differential ring homomorphism** from $(\mathcal{A}, D_{\mathcal{A}})$ to $(\mathcal{E}, \mathcal{D}_x)$. It is not an isomorphism because it is not injective.

16.8.4 Partitioning by $\mathcal{D}_x$: Equivalence Classes and the Integral Direction

The derivation $\mathcal{D}_x : \mathcal{E} \to \mathcal{E}$ is not injective. Its kernel is $\ker(\mathcal{D}_x) = \mathbb{R}$ (the constant functions). Two functions in $\mathcal{E}$ are in the same equivalence class exactly when they differ by a constant:

$$f \sim_\partial g \iff f - g = c \in \mathbb{R} \iff [f] = \{f + c \mid c \in \mathbb{R}\}.$$

These are the *antiderivative families* you wrote as $F(x) + C$.

Remark (disconnected domains). This picture assumes the domain of f is *connected*. When the domain has more than one connected component, the kernel of $\mathcal{D}_x$ is larger: a function can vanish under $\mathcal{D}_x$

while taking an *independent* value on each component. The kernel is then the space of *locally constant* functions, isomorphic to $\mathbb{R}^n$ where n is the number of connected components.

For example, $f(x) = 1/x$ has domain $\mathbb{R} \setminus \{0\} = (-\infty, 0) \cup (0, \infty)$—two components. The full antiderivative family is

$$F(x) \;=\; \ln|x| \;+\; \begin{cases} C_1 & x < 0 \\ C_2 & x > 0 \end{cases}$$

with $C_1, C_2 \in \mathbb{R}$ *independent*. The standard shorthand $\ln|x| + C$ (one constant) silently restricts to a single component. In general, the equivalence classes on a domain with n components are cosets $[f] = \{f + \phi \mid \phi \text{ locally constant on } \mathrm{dom}(f)\}$, a family parameterised by $\mathbb{R}^n$.

The quotient $\mathcal{E}/\mathbb{R}$ (elementary functions modulo constants) is isomorphic to the *image* of $\mathcal{D}_x$:

$$\mathcal{E}/\mathbb{R} \;\xrightarrow{\sim}\; \mathcal{D}_x(\mathcal{E}) \;\subsetneq\; \mathcal{E}.$$

The strict inclusion is the heart of why integration is hard: $\mathcal{D}_x(\mathcal{E}) \neq \mathcal{E}$. Not every elementary function is the derivative of another elementary function. The function e^{x^2} is in $\mathcal{E}$ but is *not* in $\mathcal{D}_x(\mathcal{E})$: Liouville proved in 1835 that it has no elementary antiderivative. There is no AST whose derivative evaluates to e^{x^2}.

The antiderivative is a right-inverse of $\mathcal{D}_x$, but only on the proper subset where it exists:

$$\mathcal{D}_x \circ \textstyle\int \;=\; \mathrm{id}_{\mathcal{D}_x(\mathcal{E})}.$$

Outside $\mathcal{D}_x(\mathcal{E})$ there is no right-inverse within $\mathcal{E}$ — only within the much larger world of all differentiable functions $C^1(\mathbb{R})$.

16.8.5 The Fundamental Asymmetry

Operator	Domain	Total?
$D_{\mathcal{A}}$ (`.derivative()`)	$\mathcal{A} \to \mathcal{A}$	Yes — every AST has a derivative AST.
$\mathcal{D}_x$	$\mathcal{E} \to \mathcal{E}$	Yes — every elementary function has an elementary derivative.
$\int$	$\mathcal{E} \to \mathcal{E}$	**No** — not every elementary function has an elementary antiderivative.

This asymmetry is structural, not accidental. Differentiation is a total operation in $\mathcal{E}$ because the derivative rules close under the grammar: the derivative of any elementary expression is again elementary. Integration is partial in $\mathcal{E}$ because the antiderivative rules do *not* close: the antiderivative of an elementary expression can escape $\mathcal{E}$ entirely. The Risch algorithm (Section 16.5) is precisely the decision procedure that tells you, for a given $f \in \mathcal{E}$, whether $f \in \mathcal{D}_x(\mathcal{E})$ — and if so, what the antiderivative is.

16.9 A Loose End from Chapter 5: `fast_power`, Revisited

Chapter 5 left an optional question dangling: why doesn't `fast_power` evaluate any faster than `naive_pow`, despite building a dramatically smaller tree (Section 5.5.1)? If you worked through that enrichment section, or are simply curious now, here is one concrete fix.

`fast_power` builds its tree by repeated squaring: `fast_power(f, 2)` constructs *one* `Product(f, f)` object, and every subsequent squaring step reuses that same object again —

`fast_power(fast_power(f, 2), n // 2)` squares the *already-constructed* tree, not a fresh copy of it. The tree is small precisely because it is built from shared references, not separate copies of `f`. `evaluate`, as written, does not know that: `case Product(f, g): return evaluate(f, x, env) * evaluate(g, x, env)` evaluates `f` and `g` as if they were unrelated, even when they are `is`-identical — so it silently throws away exactly the sharing that made the tree small in the first place.

The fix is a guard clause: check whether the two children are the *same object* (`f is g`, not merely `f == g`) before evaluating both.

```
case Product(f, g) if f is g:
    v = evaluate(f, x, env)
    return v * v
case Product(f, g):
    return evaluate(f, x, env) * evaluate(g, x, env)
```

Since `evaluate` is deterministic — the same node evaluated at the same `x` always returns the same value — this changes nothing about *what* `evaluate` returns, only how much work it does to return it. The effect is not marginal: instrumenting `evaluate` to count how many times the leaf `Sin()` is actually evaluated while computing `evaluate(fast_power(Sin(), 100), x)` gives 100 calls without the guard and just 3 with it. The guard collapses the entire squaring cascade down to roughly the recursion depth, restoring the $O(\log n)$ behavior exponentiation by squaring promised in the first place.

This guard is sound only for the four operators where both children are evaluated at the *same* point: `Sum`, `Difference`, `Product`, and `Quotient`. `Compose(f, g)` evaluates `g` at `x` and `f` at a *different* point, `evaluate(g, x, env)` — so `Compose(f, f)` means $f(f(x))$, not $f(x)$ reused. Applying the same "evaluate once, reuse" shortcut to `Compose` would silently compute the wrong answer: for `f = Sin()` at $x = 0.7$, `Compose(f, f)` correctly evaluates to $\sin(\sin(0.7)) \approx 0.601$,

not $\sin(0.7) \approx 0.644$.

This is not the first time a fix has applied to `Sum`, `Difference`, `Product`, and `Quotient` but not `Compose` — the same split governed how `Compose(op(f_1, f_2), g)` could be simplified. That is a real, recurring shape in this ADT, not a coincidence: those four operators are *pointwise* — both children see the same input — and `Compose` is the one operator that is not.

This fix is deliberately *not* built into the `evaluate` you were given in Chapter 5: doing so would have quietly answered the reflection question before your group had a chance to investigate it.

? Open Questions

- The equivalence relation $s \sim t \iff [\![s]\!] = [\![t]\!]$ is exactly the "semantic AST equality" problem that Section 6.2 in Chapter 6 raised and left unsolved for arbitrary real-valued functions. Restricted to $\mathcal{A}$ (elementary functions only, evaluated by `eval`), is $s \sim t$ decidable? Equivalently: is it decidable whether an elementary expression is identically zero? This turns out to be a genuinely subtle question in mathematics — look up *Richardson's theorem* and *Schanuel's conjecture* to see how close it sits to the edge of what is currently known.

- The chapter suggests that for textbook-style problems, integration is "largely pattern recognition," and that this is exactly what an LLM (or a diligent student) learns from enough worked examples. The Risch algorithm, by contrast, is a decision procedure with a correctness guarantee. Is there a precise line between "pattern recognition trained on enough examples" and "executing an algorithm," or does one shade into the other as the pattern library grows? Could a large enough library of patterns, combined with a verification step (differentiate the answer and check it matches), ever be made as reliable as Risch, without becoming Risch?

- This handout has posed open questions at the end of several chapters — some deferred to later chapters, some left permanently outside its scope (the Implicit Function Theorem, the Extreme Value Theorem's proof, uniform convergence, non-standard analysis, the Expression Problem, among others). Looking back, which of those have you actually resolved for yourself, and which are still open? That list is a reasonable starting point for independent study beyond this course.

Appendix A

Quick-Access QR Codes

A.1 Download the Latest Handout

Scan this code any time to download the most up-to-date version of this handout.

https://gitlab.cri.epita.fr/jim.newton/bcs-calculus-student/-/blob/main/doc/cal-handout.pdf

A.2 Download `checkout-workarea-calculus.py`

Scan this code to download the script that sets up your personal Grader workarea.

`https://gitlab.cri.epita.fr/jim.newton/bcs-calculus-student/-/blob/main/bin/checkout-workarea-calculus.py`