A Programmer's Introduction to Mathematics

Jeremy Kun

First edition, 2018.

pimbook.org

To my wife, Erin.

My unbounded, uncountable thanks goes out to the many people who read drafts at various stages of roughness and gave feedback, including (in alphabetical order by first name), Aaron Shifman, Adam Lelkes, Alex Walchli, Ali Fathalian, Arun Koshy, Ben Fish, Craig Stuntz, Devin Ivy, Erin Kelly, Fred Ross, Ian Sharkey, Jasper Slusallek, Jean-Gabriel Young, João Rico, John Granata, Julian Leonardo Cuevas Rozo, Kevin Finn, Landon Kavlie, Louis Maddox, Matthijs Hollemans, Olivia Simpson, Pablo González de Aledo, Paige Bailey, Patrick Regan, Patrick Stein, Rodrigo Zhou, Stephanie Labasan, Temple Keller, Trent McCormick.

Special thanks to Devin Ivy for a thorough technical review of two key chapters.

Contents

Our Goal

This book has a straightforward goal: to teach you how to engage with mathematics.

Let's unpack this. By "mathematics," I mean the universe of books, papers, talks, and blog posts that contain the meat of mathematics: formal definitions, theorems, proofs, conjectures, and algorithms. By "engage" I mean that for any mathematical topic, you have the cognitive tools to actively progress toward understanding that topic. I will "teach" you by introducing you to—or having you revisit—a broad foundation of topics and techniques that support the rest of mathematics. I say "with" because mathematics requires active participation.

We will define and study many basic objects of mathematics, such as polynomials, graphs, and matrices. More importantly, I'll explain *how to think* about those objects as seasoned mathematicians do. We will examine the hierarchies of mathematical abstraction, along with many of the softer skills and insights that constitute "mathematical intuition." Along the way we'll hear the voices of mathematicians—both famous historical figures and my friends and colleagues—to paint a picture of mathematics as both a messy amalgam of competing ideas and preferences, and a story with delightfully surprising twists and connections. In the end, I will show you how mathematicians think about mathematics.

So why would someone like you[1] want to engage with mathematics? Many software engineers, especially the sort who like to push the limits of what can be done with programs, eventually come to realize a deep truth: mathematics unlocks a *lot* of cool new programs. These are truly novel programs. They would simply be impossible to write (if not inconceivable!) without mathematics. That includes programs in this book about cryptography, data science, and art, but also to many revolutionary technologies in industry, such as signal processing, compression, ranking, optimization, and artificial intelligence. As importantly, a wealth of opportunity makes programming more fun! To quote Randall Munroe in his XKCD comic *Forgot Algebra,* "The only things you HAVE to know are how to make enough of a living to stay alive and how to get your taxes done. All the fun parts of life are optional." If you want your career to grow beyond shuffling data around to meet arbitrary business goals, you should learn the tools that enable you to write programs that captivate and delight you. Mathematics is one of those tools.

Programmers are in a privileged position to engage with mathematics. As a program-

[1] Hopefully you're a programmer; otherwise, the title of this book must have surely caused a panic attack.

mer, you eat paradigms for breakfast and reshape them into new ones for lunch. Your comfort with functions, logic, and protocols gives you an intuitive familiarity with basic topics such as boolean algebra, recursion, and abstraction. You can rely on this to make mathematics less foreign, progressing all the faster to more nuanced and stimulating topics. Contrast this to most educational math content aimed at students with no background and focusing on rote exercises and passing tests. As a bonus, programming allows me to provide immediate applications that ground the abstract ideas in code. In each chapter of this book, we'll fashion our mathematical designs into a program you couldn't have written before, to dazzling effect. The code is available on Github,[2] with a directory for each chapter.

All told, this book is *not* a textbook. I won't drill you with exercises, though drills have their place. We won't build up any particular field of mathematics from scratch. Though we'll visit calculus, linear algebra, and many other topics, this book is far too short to cover everything a mathematician ought to know about these topics. Moreover, while much of the book is appropriately rigorous, I will occasionally and judiciously loosen rigor when it facilitates a better understanding and relieves tedium. I will note when this occurs, and we'll discuss the role of rigor in mathematics more broadly.

Indeed, rather than read an encyclopedic reference, you want to become *comfortable* with the process of learning mathematics. In part that means becoming comfortable with discomfort, with the struggle of understanding a new concept, and the techniques that mathematicians use to remain productive and sane. Many people find calculus difficult, or squeaked by a linear algebra course without grokking it. After this book you should have a core nugget of understanding of these subjects, along with the cognitive tools that will enable you dive as deeply as you like.

As a necessary consequence, in this book you'll learn how to read and write proofs. The simplest and broadest truth about mathematics is that it revolves around proofs. Proofs are both the primary vehicle of insight and the fundamental measure of judgment. They are the law, the currency, and the fine art of mathematics. Most of what makes mathematics mysterious and opaque—the rigorous definitions, the notation, the overloading of terminology, the mountains of theory, and the unspoken obligations on the reader—is due to the centrality of proofs. A dominant obstacle to learning math is an unfamiliarity with this culture. In this book I'll show you why proofs are so important, cover the basic methods, and display examples of proofs in each chapter. To be sure, you don't have to understand every proof to finish this book, and you will probably be confounded by a few. Embrace your humility. I hope to convince you that each proof contains layers of insight that are genuinely worthwhile, and that no single person can see the complete picture in a single sitting. As you grow into mathematics, the act of reading even previously understood proofs provides both renewed and increaseed wisdom. So long as you identify the value gained by your struggle, your time is well spent.

I'll also teach you how to read between the mathematical lines of a text, and understand the implicit directions and cultural cues that litter textbooks and papers. As we proceed

[2] `pimbook.org`

through the chapters, we'll gradually become more terse, and you'll have many opportunities to practice parsing, interpreting, and understanding math. All of the topics in this book are explained by hundreds of other sources, and each chapter's exercises include explorations of concepts beyond these pages. In addition, I'll discuss how mathematicians approach problems, and how their process influences the culture of math.

You will not learn everything you want to know in this book, nor will you learn everything this book has to offer in one sitting. Those already familiar with math may find early chapters offensively slow and detailed. Those genuinely new to math may find the later chapters offensively fast. This is by design. I want you to be exposed to as much mathematics as possible, to learn the definitions of central mathematicl ideas, to be introduced to notations, conventions, and attitudes, and to have ample opportunity to explore topics that pique your interest.

A number of topics are conspicuously missing from this book, my negligence of which approaches criminal. Except for a few informal cameos, we ignore complex numbers, probability and statistics, differential equations, and formal logic. In my humble opinion, none of these topics is as fundamental for mathematical computer science as those I've chosen to cover. After becoming comfortable with the topics in this book, for example, probability will be very accessible. The chapter on eigenvalues will include a miniature introduction to differential equations. The chapter on groups will briefly summarize complex numbers. Probability will echo in your brain when we discuss random graphs and machine learning. Moreover, many topics in this book are prerequisites for these other areas. And, of course, as a single human self-publishing this book on nights and weekends, I have only so much time.

The first step on our journey is to confirm that mathematics has a culture worth becoming acquainted with. We'll do this with a comparative tour of the culture of software that we understand so well.

Chapter 1

Like Programming, Mathematics has a Culture

Mathematics knows no races or geographic boundaries; for mathematics, the cultural world is one country.

–David Hilbert

Do you remember when you started to really *learn* programming? I do. I spent two years in high school programming games in Java. Those two years easily contain the worst and most embarrassing code I have ever written. My code absolutely reeked. Hundred-line functions and thousand-line classes, magic numbers, unreachable blocks of code, ridiculous comments, a complete disregard for sensible object orientation, and type-coercion that would make your skin crawl. The code worked, but it was filled with bugs and mishandled edge-cases. I broke every rule in the book, and for all my shortcomings I considered myself a hot-shot (at least, among my classmates!). I didn't know how to design programs, or what made a program "good," other than that it ran and I could impress my friends with a zombie shooting game.

Even after I started studying software in college, it was another year before I knew what a stack frame or a register was, another year before I was halfway competent with a terminal, another year before I appreciated functional programming, and to this day I *still* have an irrational fear of systems programming and networking. I built up a base of knowledge over time, with fits and starts at every step.

In a college class on C++ I was programming a Checkers game, and my task was to generate a list of legal jump-moves from a given board state. I used a depth-first search and a few recursive function calls. Once I had something I was pleased with, I compiled it and ran it on my first non-trivial example. Lo' and behold (even having followed test-driven development!), a segmentation fault smacked me in the face. Dozens of test cases and more than twenty hours of confusion later, I found the error: my recursive call passed a reference when it should have been passing a pointer. This wasn't a bug in syntax or semantics—I understood pointers and references well enough—but a design error. As most programmers can relate, the most aggravating part was that changing four characters (swapping a few ampersands with asterisks) fixed it. Twenty hours of work for four characters! Once I begrudgingly verified it worked, I promptly took the rest of the day off to play Starcraft.

Such drama is the seasoning that makes a strong programmer. One must study the topics incrementally, learn from a menagerie of mistakes, and spend hours in a befuddled stupor before becoming "experienced." This gives rise to all sorts of programmer culture, Unix jokes, urban legends, horror stories, and reverence for the masters of C that make the programming community so lovely. It's like a secret club where you know all the handshakes, but should you forget one, a crafty use of `grep` and `sed` will suffice. The struggle makes you appreciate the power of debugging tools, slick frameworks, historically enshrined hacks, and new language features that stop you from shooting your own foot.

When programmers turn to mathematics, they seem to forget these trials. The same people who invested years grokking the tools of their trade treat new mathematical tools and paradigms with surprising impatience. I can see a few reasons why. One is that they've been taking classes called "mathematics" for far longer than they've been learning to program (and mathematics was always easy!). The forced prior investment of schooling engenders a certain expectation. The problem is that the culture of mathematics and the culture of mathematics education—elementary through lower-level college courses—are completely different.

Even math majors have to reconcile this. I've had many conversations with such students, many of whom are friends, colleagues, and even family, who by their third year decided they didn't really enjoy math. The story often goes like this: a student who was good at math in high school (perhaps because of its rigid structure) reaches the point of a math major at which they must read and write proofs in earnest. It requires an earnest, open-ended exploration that they don't enjoy. Despite being a stark departure from high school math, incoming students are never warned in advance. After coming to terms with their unfortunate situation, they decide that their best option is to hold on until they can return to the comfortable setting of their prior experiences, this time in the teacher's chair.

I don't mean to insult teaching as a profession—I love teaching and understand why one would choose to do it full time. There are many excellent teachers who excel at both the math and the trickier task of engaging aloof teenagers to think critically about it. But this pattern of disenchantment among math teachers is prevalent, and it widens the conceptual gap between secondary and "college level" mathematics. Programmers often have similar feelings, that the math they were once so good at is suddenly impenetrable. It's not a feature of math, but a bug in the education system (and a negative feedback loop!) that gets blamed on math as a subject.

Another reason programmers feel impatient is because they do so many things that relate to mathematics in deep ways. They use graph theory for data structures and search. They study enough calculus to make video games. They hear about the Curry-Howard correspondence between proofs and programs. They hear that Haskell is based on a complicated math thing called category theory. They even use mathematical results in an interesting way. I worked at a "blockchain" company that implemented a Bitcoin wallet, which is based on elliptic curve cryptography. The wallet worked, but the implementer didn't understand why. They simply adapted pseudocode found on the internet. At the

risk of a dubious analogy, it's akin to a "script kiddie" who uses hacking tools as black boxes, but has little idea how they work. Mathematicians are on the other end of the spectrum, caring almost exclusively about why things work the way they do.

While there's nothing inherently wrong with using mathematics as a black box, especially the sort of applied mathematics that comes with provable guarantees, many programmers *want* to understand why they work. This isn't surprising, given how much time engineers spend studying source code and the internals of brittle, technical systems. Systems that programmers rely on, such as dependency management, load balancers, search engines, alerting systems, and machine learning, all have rich mathematical foundations. We're naturally curious about how they work and how to adapt them to our needs.

Yet another hindrance to mathematics is that it has no centralized documentation. Instead it has a collection of books, papers, journals, and conferences, each with discrepancies of presentation, citing each other in a haphazard manner. A theorem presented at a computer science conference can be phrased in completely unfamiliar terms in a dynamical systems journal—even though they boil down to the same facts! In subfields like network science that straddle disciplines, one often sees "translation tables" for jargon.

Dealing with this is not easy. Students of mathematics solve these problems with knowledgeable teachers. Working mathematicians just "do it." They work out the translation details themselves with coffee and contemplation. Advanced books also lean toward terseness, despite being titled as "elementary" or an "introduction." They opt not to redefine what they think the reader must already know. The purest fields of mathematics take a sort of pretentious pride in how abstract and compact their work is (to the point where many students spend weeks or months understanding a single chapter!).

What programmers would consider "sloppy" notation is one symptom of the problem, but there there are other expectations on the reader that, for better or worse, decelerate the pace of reading. Unfortunately I have no solution here. Part of the power and expressiveness of mathematics is the ability for its practitioners to overload, redefine, and omit in a suggestive manner. Mathematicians also have thousands of years of "legacy" math that require backward compatibility. Enforcing a single specification for all of mathematics—a suggestion I frequently hear from software engineers—would be horrendously counterproductive.

Indeed, ideas we take for granted today, such as algebraic notation, drawing functions in the Euclidean plane, and summation notation, were at one point actively developed technologies. Each of these notations had a revolutionary effect, not just on science, but also, to quote Bret Victor, on our capacity to "think new thoughts." One can even draw a line from the proliferation of algebraic notation and the computational questions it raised to the invention of the computer.[1] Borrowing software terminology, algebraic notation is

[1] Leibniz, one of the inventors of calculus, dreamed of a machine that could automatically solve mathematical problems. Ada Lovelace (up to some irrelevant debate) designed the first program for computing Bernoulli numbers, which arise in algebraic formulas for computing sums of powers of integers. In the early 1900's Hilbert posed his Tenth Problem on algorithms for computing solutions to Diophantine equations, and later his Entscheidungsproblem, which was solved concurrently by Church and Turing and directly led to Turing's

among the most influential and scalable technologies humanity has ever invented. And as we'll see in Chapter 10 and Chapter 16, we can find algebraic structure hiding in exciting places. Algebraic notation helps us understand this structure not only because we can compute, but also because we can visually see the symmetries in the formulas. This makes it easier for us to identify, analyze, and encapsulate structure when it occurs.

Finally, the best mathematicians study concepts that connect decades of material, while simultaneously inventing new concepts which have no existing words to describe them. Without flexible expression, such work would be impossible. It reduces cognitive load, a theme that will follow us throughout the book. Unfortunately, it only does so for the readers who have *already* absorbed the basic concepts of discussion. By contrast, good software practice encourages code that is simple enough for anyone to understand. As such, the uninitiated programmer often has a much larger cognitive load when reading math than when reading a program.

Taken together, mathematical notation is closer to spoken language than to code. It can reduce one's mental burden via rigorous rules applied to an external representation, coupled with context and convention. All of this, the notation, the differences among subfields, the tradeoff between expressiveness and cognitive load, has grown out of hundreds of years of mathematical progress.

Equipped with this understanding, that mathematics has culturally relevant reasons for its strange practices, let's begin our journey through the mists of math with renewed openness.

Read on, and welcome to the club.

code-breaking computer.