

---

# FUNDAMENTALS OF NUMERICAL COMPUTATION

**Tobin A. Driscoll**

**Richard J. Braun**

26<sup>th</sup> Nov, 2024

created in  Curvenote

---

## **1 Preface**

## 1.1 Preface to the Julia edition

The invention of MATLAB introduced a new paradigm within research of numerical computation. Those concerned primarily with prototyping and perfecting algorithms, particularly those involving lots of linear algebra, optimization, and differential equations, were happy to adopt MATLAB as a primary computing environment. It offered concise syntax for such problems, freedom from variable types and declarations, compiling, and linking programs, convenient tools for analyzing results, and cross-platform uniformity. There were drawbacks, however, when it came to performance, scalability, and language features beyond the manipulation of vectors and matrices. While MATLAB has steadily made serious progress on closing the performance gap and introducing new language features, there remain computing tasks for which it is not ideally suited.

The landscape changed when the SciPy, NumPy, and Matplotlib packages for scientific computing in Python became stable and polished. These enabled Python to offer a fully featured MATLAB alternative that is free, open, and tightly integrated with a much larger world of computing. Among these valuable advances, though, a serious compromise lurked: performance got worse, often by orders of magnitude. While there are notable efforts to overcome the bottlenecks for many use cases, Python continues to face deep and steep performance challenges as a general-purpose scientific computing ecosystem.

Julia was designed from its inception to prioritize numerical scientific computing. It has reaped the benefits of learning from decisions and adaptations made in MATLAB and Python, borrowing the best parts from them and tackling their deficiencies. Julia's older cousins enjoy a big head start, so it's impossible to know what the size of Julia's niche will ultimately be, but interest has continued to build.

Why teach using Julia? The immediate benefits of Julia over MATLAB for the material in this text include:

- Julia allows Unicode characters, such as Greek letters, subscripts, and symbols, as variable names and operators, which makes code look more like mathematics.
- Julia makes it effortless to define functions inside of scripts as well as other functions.
- Julia's broadcasting syntax clarifies how to apply functions elementwise to arrays.
- Comprehensions are convenient and concise ways to construct vectors and matrices.
- Julia makes it easier to define keyword and optional function arguments.

There are also differences that cut both ways; for instance, Julia is often stricter about data types and sizes, which makes it more verbose and more prone to error, but arguably less likely to finish with unexpected results.

Moreover, there are some tradeoffs. MATLAB ships with and installs everything needed for this text, while Julia requires a small installation effort to get started. MATLAB's documentation is superior, and it's easier to get accurate help on the Internet. MATLAB's integrated desktop, particularly the debugger, are not yet fully matched in Julia.

There is also a wider context to consider. Julia skills are more likely to be directly applicable in, or more easily transferrable to, high-performance applications. Julia interoperates easily with Python, R, C, and even MATLAB. Julia is native to the widely used Jupyter notebook system that currently dominates data science. Not least, as a free and open-source environment, Julia enables fully reproducible computing, which is increasingly appreciated as essential to long-term progress in research.

### 1.1.1 What to expect from Julia

Unlike MATLAB and Python, Julia is just-in-time (JIT) compiled, not interpreted. As a result, large packages, including several supporting the code in this book, can take a few seconds to load.<sup>1</sup> Furthermore, if you make a change to one of your own functions, or apply it to new types of function arguments, Julia may hesitate a moment while it compiles the necessary code. On slower hardware, or with frequent revision, the lag can become irritating.

---

<sup>1</sup>It is strongly recommended that you use at least version 1.6 of Julia. In earlier versions, the wait to compile and load packages can become long.

### 1.1.2 What to expect from this book

\*Supplemental material, including animations, downloadable code and examples, suggested projects, and more, can be found at <https://bookstore.siam.org/ot177/bonus>.

We do not attempt to teach how to become a great Julia programmer. That goal is too ambitious when stacked alongside the mathematical ones. Instead we hope to exhibit decent style and avoid promoting bad habits. But when there is a conflict, clarity and simplicity usually override performance concerns.<sup>2</sup> A virtue of Julia is that one can start with a working straightforward code that can be adapted and improved to meet performance demands; we are introducing just the first stage of this process.

One choice advanced users might question is that we address vectors and matrices as starting from index 1, rather than using more general constructs such as `eachindex`, `begin`, and `first`. Our mathematics makes those specific references too, and in most languages, one must learn to deal directly with the difference between, say, 1-indexing and 0-indexing.

Another notable choice we have made is the use of the popular `Plots` package for graphics. There are many other fine choices, including and not limited to `PyPlot`, `Makie`, and `PlotlyJS`, but we needed to be concrete.

Beyond that, we touch briefly on available packages that offer advanced functionality for the problem types we study. Our hope is that the student will not only learn fundamentals by working with simple codes in the book, but also learn about the existence and syntax of some power tools for serious applications.

Finally, we have made no mention of one of Julia's defining features, *multiple dispatch*. Using its power wisely edges into advanced software design and usually requires a bird's-eye view of problems that beginners lack. We want to let students expend as much of their cognitive budget as possible on the mathematical principles that have universal application.

---

<sup>2</sup>In a fast-changing language like Julia, yesterday's performance roadblocks can disappear anyway.

I've developed an obscene interest in computation, and I'll be returning to the United States a better and impurer man.

Figure 1: \*  
John von Neumann

## 1.2 Preface to the original edition

It might seem that computing should simply be a matter of translating formulas from the page to the machine. But even when such formulas are known, applying them in a numerical fashion requires care. For instance, rounding off numbers at the sixteenth significant digit can lay low such stalwarts as the quadratic formula! Fortunately, the consequences of applying a numerical method to a mathematical problem are quite understandable from the right perspective. In fact, it is our mastery of what can go wrong in some approaches that gives us confidence in the rest of them.

If mathematical modeling is the process of turning real phenomena into mathematical abstractions, then numerical computation is largely about the transformation from abstract mathematics to concrete reality. Many science and engineering disciplines have long benefited from the tremendous value of the correspondence between quantitative information and mathematical manipulation. Other fields, from biology to history to political science, are rapidly catching up. In our opinion, a young mathematician who is ignorant of numerical computation in the 21st century has much in common with one who was ignorant of calculus in the 18th century.

### 1.2.1 To the student

Welcome! We expect that you have had lessons in manipulating power series, solving linear systems of equations, calculating eigenvalues of matrices, and obtaining solutions of differential equations. We also expect that you have written computer programs that take a nontrivial number of steps to perform a well-defined task, such as sorting a list. Even if you have rarely seen how these isolated mathematical and computational tasks interact with one another, or what they have to do with practical realities, you are in the audience for this book.

Based on our experiences teaching this subject, our guess is that some rough seas may lie ahead of you. Probably you do not remember learning all parts of calculus, linear algebra, differential equations, and computer science with equal skill and fondness. This book draws from all of these areas at times, so your weaknesses are going to surface once in a while. Furthermore, this may be the first course you have taken that does not fit neatly within a major discipline. Von Neumann's use of "impurer" in the quote above is a telling one: numerical computation is about solving problems, and the search for solution methods that work well can take us across intellectual disciplinary boundaries. This mindset may be unfamiliar and disorienting at times.

Don't panic! There is a great deal to be gained by working your way through this book. It goes almost without saying that you can acquire computing skills that are in much demand for present and future use in science, engineering, and mathematics—and increasingly, in business, social science, and humanities, too. There are less tangible benefits as well. Having a new context to wrestle with concepts like Taylor series and matrices may shed new light on why they are important enough to learn. It can be exhilarating to apply skills to solve relatable problems. Finally, the computational way of thought is one that complements other methods of analysis and can serve you well.

### 1.2.2 To the instructor

The plausibly important introductory material on numerical computation for the majority of undergraduate students easily exceeds the capacity of two semesters—and of one textbook. As instructors and as authors, we face difficult choices as a result. We set aside the goal of creating an agreeable canon. Instead we hope for students to experience an echo of that "obscene interest" that von Neumann so gleefully described and pursued. For while there are excellent practical reasons to learn about numerical computing, it also stands as a subject of intellectual and even emotional relevance. We have seen students excited and motivated by applications of their newly found abilities to problems in mechanics, biology, networks, finance, and more—problems that are of unmistakable

importance in the jungle beyond university textbooks, yet largely impenetrable using only the techniques learned within our well-tended gardens.

In writing this book, we have not attempted to be encyclopedic. We're sorry if some of your favorite topics don't appear or are minimized in the book. (It happened to us too; many painful cuts were made from prior drafts.) But in an information-saturated world, the usefulness of a textbook lies with teaching process, not content. We have aimed not for a cookbook but for an introduction to the principles of cooking.

Still, there *are* lots of recipes in the book—it's hard to imagine how one could become a great chef without spending time in the kitchen! Our language for these recipes is MATLAB for a number of reasons: it is precise, it is executable, it is as readable as could be hoped for our purposes, it rewards thinking at the vector and matrix level, and (at this writing) it is widespread and worth knowing. There are 46 standalone functions and over 150 example scripts, all of them downloadable exactly as seen in the text. Some of our codes are quite close to production quality, some are serviceable but lacking, and others still are meant for demonstration only. Ultimately our codes are mainly meant to be clear, not ideal. We try to at least be explicit about the shortcomings of our implementations.

Just as good coding and performance optimization are secondary objectives of the book, we cut some corners in the mathematics as well. We state and in some cases prove the most essential and accessible theorems, but this is not a theorem-oriented book, and in some cases we are content with less precise arguments. We have tried to make clear where solid proof ends and where approximation, estimation, heuristics, and other indispensable tools of numerical analysis begin.

The examples and exercises are meant to show and encourage a numerical mode of thought. As such they often focus on issues of convergence, experimentation leading to abstraction, and attempts to build on or refine presented ideas. Some exercises follow the premise that an algorithm's most interesting mode of operation is failure. We expect that any learning resulting from the book is likely to take place mostly from careful study of the examples and working through the problems.

**Contents** Chapter 1 explains how computers represent numbers and arithmetic, and what doing so means for mathematical expressions. Chapter 2 discusses the solution of square systems of linear equations and, more broadly, basic numerical linear algebra. Chapter 3 extends the linear algebra to linear least squares. These topics are the bedrock of scientific computing, because “everything” has multiple dimensions, and while “everything” is also nonlinear, our preferred starting point is to linearize.

Chapters 4 through 6 introduce what we take to be the rest of the most common problem types in scientific computing: roots and minimization of algebraic functions, piecewise approximation of functions, numerical analogs of differentiation and integration, and initial-value problems for ordinary differential equations. We also explain some of the most familiar and reliable ways to solve these problems, effective up to a certain point of size and/or difficulty. Chapters 1 through 6 can serve for a single-semester survey course. If desired, Chapter 6 could be left out in favor of one of Chapters 7, 8, or 9.

The remaining chapters are intended for a second course in the material. They go into more sophisticated types of problems (eigenvalues and singular values, boundary value problems, and partial differential equations), as well as more advanced techniques for problems from the first half (Krylov subspace methods, spectral approximation, stiff problems, boundary conditions, and tensor-product discretizations).

You must unlearn what you have learned.

Figure 2: Yoda, *The Empire Strikes Back*

## 2 Introduction

Our first step is to discretize the real numbers—specifically, to replace them with a finite surrogate set of numbers. This step keeps the time and storage requirements for operating with each number at constant levels, but virtually every data set and arithmetic operation is perturbed slightly away from its idealized mathematical value. We can easily keep the individual roundoff errors very small, so small that simple random accumulation is unlikely to bother us. However, some problems are extremely sensitive to these perturbations, a trait we quantify using a *condition number*. Problems with large condition numbers are difficult to solve accurately using finite precision. Furthermore, even when the condition number of a problem is not large, some algorithms for solving it allow errors to grow enormously. We call these algorithms *unstable*. In this chapter we discuss these ideas in simple settings before moving on to the more realistic problems in the rest of the book.

### Software

Instructions for obtaining Julia and the codes used throughout the text can be found at

<https://github.com/fncbook/FundamentalsNumericalComputation.jl>

The installation process, which can take 5-10 minutes, only needs to be performed once.

## 2.1 Floating-point numbers

The real number set  $\mathbb{R}$  is infinite in two ways: it is unbounded and continuous. In most practical computing, the latter kind of infiniteness is much more consequential than the former, so we turn our attention there first.

**Definition 2.1** (Floating-point numbers). *The set  $\mathbb{F}$  of **floating-point numbers** consists of zero and all numbers of the form*

$$\pm(1 + f) \times 2^n, \quad (1)$$

where  $n$  is an integer called the **exponent**, and  $1 + f$  is the **mantissa** or **significand**, in which

$$f = \sum_{i=1}^d b_i 2^{-i}, \quad b_i \in \{0, 1\}, \quad (2)$$

for a fixed integer  $d$  called the binary **precision**.

Equation (2) represents the significand as a number in  $[1, 2)$  in base-2 form. Equivalently,

$$f = 2^{-d} \sum_{i=1}^d b_i 2^{d-i} = 2^{-d} z \quad (3)$$

for an integer  $z$  in the set  $\{0, 1, \dots, 2^d - 1\}$ . Consequently, starting at  $2^n$  and ending just before  $2^{n+1}$  there are exactly  $2^d$  evenly spaced numbers belonging to  $\mathbb{F}$ .

**Example 2.1.** Suppose  $d = 2$ . Taking  $n = 0$  in (1), we enumerate

$$1 + \frac{0}{4}, 1 + \frac{1}{4}, 1 + \frac{2}{4}, 1 + \frac{3}{4}. \quad (4)$$

These are the only members of  $\mathbb{F}$  in the semi-closed interval  $[1, 2)$ , and they are separated by spacing  $\frac{1}{4}$ .

Taking  $n = 1$  doubles each of the values in the list above, and  $n = -1$  halves them. These give the floating-point numbers in  $[2, 4)$  and  $[1/2, 1)$ , respectively. The spacing between them also is doubled and halved, respectively.

Observe that the smallest element of  $\mathbb{F}$  that is greater than 1 is  $1 + 2^{-d}$ , and we call the difference *machine epsilon*.<sup>3</sup>

**Definition 2.2** (Machine epsilon). *For a floating-point set with  $d$  binary digits of precision, **machine epsilon** (or **machine precision**) is  $\epsilon_{\text{mach}} = 2^{-d}$ .*

We define the rounding function  $\text{fl}(x)$  as the map from real number  $x$  to the nearest member of  $\mathbb{F}$ . The distance between the floating-point numbers in  $[2^n, 2^{n+1})$  is  $2^n \epsilon_{\text{mach}} = 2^{n-d}$ . As a result, every real  $x \in [2^n, 2^{n+1})$  is no farther than  $2^{n-d-1}$  away from a member of  $\mathbb{F}$ . Therefore we conclude that  $|\text{fl}(x) - x| \leq \frac{1}{2}(2^{n-d})$ , which leads to the bound

$$\frac{|\text{fl}(x) - x|}{|x|} \leq \frac{2^{n-d-1}}{2^n} \leq \frac{1}{2} \epsilon_{\text{mach}}. \quad (5)$$

In words, every real number is represented with a uniformly bounded relative precision. Inequality (5) holds true for negative  $x$  as well. In [Exercise 2](#) you are asked to show that an equivalent statement is that

$$\text{fl}(x) = x(1 + \epsilon) \quad \text{for some } |\epsilon| \leq \frac{1}{2} \epsilon_{\text{mach}}. \quad (6)$$

The value of  $\epsilon$  depends on  $x$ , but this dependence is not usually shown explicitly.

<sup>3</sup>The terms machine epsilon, machine precision, and unit roundoff aren't used consistently across references, but the differences are not consequential for our purposes.

### 2.1.1 Precision and accuracy

It may help to recast (1) and (2) in terms of base 10:

$$\pm \left( b_0 + \sum_{i=1}^d b_i 10^{-i} \right) \times 10^n = \pm (b_0.b_1b_2 \cdots b_d) \times 10^n, \quad (7)$$

where each  $b_i$  is in  $\{0, 1, \dots, 9\}$  and  $b_0 \neq 0$ . This is simply scientific notation with  $d + 1$  significant digits. For example, Planck's constant is  $6.626068 \times 10^{-34} \text{ m}^2 \cdot \text{kg}/\text{sec}$  to seven digits. If we alter just the last digit from 8 to 9, the relative change is

$$\frac{0.000001 \times 10^{-34}}{6.626068 \times 10^{-34}} \approx 1.51 \times 10^{-7}. \quad (8)$$

We therefore say that the constant is given with 7 decimal digits of precision. That's in contrast to noting that the value is given to 40 decimal *places*. A major advantage of floating point is that the relative precision does not depend on the choice of physical units. For instance, when expressed in eV·sec, Planck's constant is  $4.135668 \times 10^{-15}$ , which still has 7 digits but only 21 decimal places.

Floating-point precision functions the same way, except that computers prefer base 2 to base 10. The **precision** of a floating-point number is always  $d$  binary digits, implying a resolution of the real numbers according to (5).

It can be easy to confuse precision with **accuracy**, especially when looking at the result of a calculation on the computer. Every result is computed and represented using  $d$  binary digits, but not all of those digits may accurately represent an intended value. Suppose  $x$  is a number of interest and  $\tilde{x}$  is an approximation to it. The **absolute accuracy** of  $\tilde{x}$  is

$$|\tilde{x} - x|, \quad (9)$$

while the **relative accuracy** is

$$\frac{|\tilde{x} - x|}{|x|}. \quad (10)$$

Absolute accuracy has the same units as  $x$ , while relative accuracy is dimensionless. We can also express the relative accuracy as the **number of accurate digits**, computed in base 10 as

$$-\log_{10} \left| \frac{\tilde{x} - x}{x} \right|. \quad (11)$$

We often round this value down to an integer, but it does make sense to speak of “almost seven digits” or “ten and a half digits.”

**Example 2.2.**

### 2.1.2 Double precision

Most numerical computing today is done in the **IEEE 754** standard. This defines **single precision** with  $d = 23$  binary digits for the fractional part  $f$ , and the more commonly used **double precision** with  $d = 52$ . In double precision,

$$\epsilon_{\text{mach}} = 2^{-52} \approx 2.2 \times 10^{-16}. \quad (12)$$

We often speak of double-precision floating-point numbers as having about 16 decimal digits. The 52-bit significand is paired with a sign bit and 11 binary bits to represent the exponent  $n$  in (1), for a total of 64 binary bits per floating-point number.



**Example 2.3.**

Our theoretical description of  $\mathbb{F}$  did not place limits on the exponent, but in double precision its range is limited to  $-1022 \leq n \leq 1023$ . Thus, the largest number is just short of  $2^{1024} \approx 2 \times 10^{308}$ , which is enough in most applications. Results that should be larger are said to *overflow* and will actually result in the value `Inf`. Similarly, the smallest positive number is  $2^{-1022} \approx 2 \times 10^{-308}$ , and smaller values are said to *underflow* to zero.<sup>4</sup>

Note the crucial difference between  $\epsilon_{\text{mach}} = 2^{-52}$ , which is the distance between 1 and the next larger double-precision number, and  $2^{-1022}$ , which is the smallest positive double-precision number. The former has to do with relative precision, while the latter is about absolute precision. Getting close to zero always requires a shift in thinking to absolute precision because any finite error is infinite relative to zero.

One more double-precision value is worthy of note: **NaN**, which stands for **Not a Number**. It is the result of an undefined arithmetic operation such as  $0/0$ .

**2.1.3 Floating-point arithmetic**

Computer arithmetic is performed on floating-point numbers and returns floating-point results. We assume the existence of machine-analog operations for real functions such as  $+$ ,  $-$ ,  $\times$ ,  $/$ ,  $\sqrt{\phantom{x}}$ , and so on. Without getting into the details, we will suppose that each elementary machine operation creates a floating-point result whose relative error is bounded by  $\epsilon_{\text{mach}}$ . For example, if  $x$  and  $y$  are in  $\mathbb{F}$ , then for machine addition  $\oplus$  we have the bound

$$\frac{|(x \oplus y) - (x + y)|}{|x + y|} \leq \epsilon_{\text{mach}}. \quad (13)$$



Hence the relative error in arithmetic is essentially the same as for the floating-point representation itself. However, playing by these rules can lead to disturbing results.

**Example 2.4.**

There are two ways to look at Example 2.4. On one hand, its two versions of the result differ by less than  $1.2 \times 10^{-16}$ , which is very small — not just in everyday terms, but with respect to the operands, which are all close to 1 in absolute value. On the other hand, the difference is as large as the exact result itself! We formalize and generalize this observation in the next section. In the meantime, keep in mind that exactness cannot be taken for granted in floating-point computation.

**Observation 2.1.** *We should not expect that two mathematically equivalent results will be equal when computed in floating point, only that they be relatively close together.*

**2.1.4 Exercises****Note**

Exercises marked with  are intended to be done by hand or with the aid of a simple calculator. Exercises marked with  are intended to be solved using a computer.

- Consider a floating-point set  $\mathbb{F}$  defined by (1) and (2) with  $d = 4$ .
  - How many elements of  $\mathbb{F}$  are there in the real interval  $[1/2, 4]$ , including the endpoints?
  - What is the element of  $\mathbb{F}$  closest to the real number  $1/10$ ? (Hint: Find the interval  $[2^n, 2^{n+1})$  that contains  $1/10$ , then enumerate all the candidates in  $\mathbb{F}$ .)
  - What is the smallest positive integer not in  $\mathbb{F}$ ? (Hint: For what value of the exponent does the spacing between floating-point numbers become larger than 1?)
- Prove that (5) is equivalent to (6). This means showing first that (5) implies (6), and then separately that (6) implies (5).

<sup>4</sup>Actually, there are some still-smaller *denormalized* numbers that have less precision, but we won't use that level of detail.

3. There are much better rational approximations to  $\pi$  than  $22/7$  as used in Example 2.2. For each one below, find its absolute and relative accuracy, and (rounding down to an integer) the number of accurate digits.
  - (a)  $355/113$
  - (b)  $103638/32989$
4. IEEE 754 **single precision** specifies that 23 binary bits are used for the value  $f$  in the significand  $1 + f$  in (2). Because they need less storage space and can be operated on more quickly than double-precision values, single-precision values can be useful in low-precision applications. (They are supported as type `Float32` in Julia.)
  - (a) In base-10 terms, what is the first single-precision number greater than 1 in this system?
  - (b) What is the smallest positive integer that is not a single-precision number? (See the hint to Exercise 1.)
5. Julia defines a function `nextfloat` that gives the next-larger floating-point value of a given number. What is the next float past `floatmax()`? What is the next float past `-Inf`?

## 2.2 Problems and conditioning

Let's think a bit about what must be the easiest math problem you've dealt with in quite some time: adding 1 to a number. Formally, we describe this problem as a function  $f(x) = x + 1$ , where  $x$  is any real number.

On a computer,  $x$  will be represented by its floating-point counterpart,  $\text{fl}(x)$ . Given the property (5), we have  $\text{fl}(x) = x(1 + \epsilon)$  for some  $\epsilon$  satisfying  $|\epsilon| < \epsilon_{\text{mach}}/2$ . There is no error in representing the value 1.

Let's suppose that we are fortunate and that the addition proceeds exactly, with no additional errors. Then the machine result is just

$$y = x(1 + \epsilon) + 1. \quad (14)$$

We can derive the relative error in this result:

$$\frac{|y - f(x)|}{|f(x)|} = \frac{|(x + \epsilon x + 1) - (x + 1)|}{|x + 1|} = \frac{|\epsilon x|}{|x + 1|}. \quad (15)$$

This error could be quite large if the denominator is small. In fact, we can make the relative error as large as we please by taking  $x$  very close to -1. This is essentially what happened in Example 2.4.

You may have encountered this situation before when using significant digits for scientific calculations. Suppose we round all results to five decimal digits, and we add -1.0012 to 1.0000. The result is -0.0012, or  $-1.2 \times 10^{-3}$  in scientific notation. Notice that even though both operands are specified to five digits, it makes no sense to write more than two digits in the answer because there is no information in the problem beyond their decimal places.

This phenomenon is known as **subtractive cancellation**, or loss of significance. We may say that three digits were "lost" in the mapping from -1.0012 to -0.0012. There's no way the loss could be avoided, *regardless of the algorithm*, once we decided to round off everything to a fixed number of digits.

**Observation 2.2** (Subtractive cancellation). *Subtractive cancellation is a loss of accuracy that occurs when two numbers add or subtract to give a result that is much smaller in magnitude. It is one of the most common mechanisms introducing dramatic growth of errors in floating-point computation.*

In double precision, all the values are represented to about 16 significant decimal digits of precision, but it's understood that subtractive cancellation may render some of those digits essentially meaningless.

### 2.2.1 Condition numbers

Now we consider problems more generally. As above, we represent a problem as a function  $f$  that maps a real data value  $x$  to a real result  $f(x)$ . We abbreviate this situation by the notation  $f : \mathbb{R} \mapsto \mathbb{R}$ , where  $\mathbb{R}$  represents the real number set.

When the problem  $f$  is approximated in floating point on a computer, the data  $x$  is represented as a floating-point value  $\tilde{x} = \text{fl}(x)$ . Ignoring all other sources of error, we define the quantitative measure

$$\frac{\frac{|f(x) - f(\tilde{x})|}{|f(x)|}}{\frac{|x - \tilde{x}|}{|x|}}, \quad (16)$$

which is the ratio of the relative changes in result and data. We make this expression more convenient if we recall that floating-point arithmetic gives  $\tilde{x} = x(1 + \epsilon)$  for some value  $|\epsilon| \leq \epsilon_{\text{mach}}/2$ . Hence

$$\frac{|f(x) - f(x + \epsilon x)|}{|\epsilon f(x)|}. \quad (17)$$

Finally, we idealize what happens in a perfect computer by taking a limit as  $\epsilon_{\text{mach}} \rightarrow 0$ .

**Definition 2.3** (Condition number (scalar function)). *The relative **condition number** of a scalar function  $f(x)$  is*

$$\kappa_f(x) = \lim_{\epsilon \rightarrow 0} \frac{|f(x) - f(x(1 + \epsilon))|}{|\epsilon f(x)|}. \quad (18)$$

The condition number is a ratio of the relative error of the output to the relative error of the input. It depends only on the problem and the data, not the computer or the algorithm.

Assuming that  $f$  has at least one continuous derivative, we can simplify the expression (18) through some straightforward manipulations:

$$\begin{aligned} \kappa_f(x) &= \lim_{\epsilon \rightarrow 0} \left| \frac{f(x + \epsilon x) - f(x)}{\epsilon f(x)} \right| \\ &= \lim_{\epsilon \rightarrow 0} \left| \frac{f(x + \epsilon x) - f(x)}{\epsilon x} \cdot \frac{x}{f(x)} \right| \\ &= \left| \frac{x f'(x)}{f(x)} \right|. \end{aligned} \quad (19)$$

In retrospect, it should come as no surprise that the change in values of  $f(x)$  due to small changes in  $x$  involves the derivative of  $f$ . In fact, if we were making measurements of changes in absolute rather than relative terms, the condition number would be simply  $|f'(x)|$ .

**Example 2.5.** *Let's return to our "add 1" problem and generalize it slightly to  $f(x) = x - c$  for constant  $c$ . We compute, using (19),*

$$\kappa_f(x) = \left| \frac{(x)(1)}{x - c} \right| = \left| \frac{x}{x - c} \right|. \quad (20)$$

*The result is the relative change (15) normalized by the size of the perturbation  $\epsilon$ . The condition number is large when  $|x| \gg |x - c|$ . Considering that  $c$  can be negative, this result applies to both addition and subtraction. Furthermore, the situation is symmetric in  $x$  and  $c$ ; that is, if we perturbed  $c$  and not  $x$ , the result would be  $|c|/|x - c|$ .*

**Example 2.6.** *Another elementary operation is to multiply by a constant:  $f(x) = cx$  for nonzero  $c$ . We compute*

$$\kappa_f(x) = \left| \frac{x f'(x)}{f(x)} \right| = \left| \frac{(x)(c)}{cx} \right| = 1. \quad (21)$$

*We conclude that multiplication by a real number leads to the same relative error in the result as in the data. In other words, multiplication does not have the potential for cancellation error that addition does.*

Condition numbers of the major elementary functions are given in Table 1.

As you are asked to show in Exercise 4, when two functions  $f$  and  $g$  are combined in a chain as  $h(x) = f(g(x))$ , the composite condition number is

$$\kappa_h(x) = \kappa_f(g(x)) \cdot \kappa_g(x). \quad (22)$$

### 2.2.2 Estimating errors

Refer back to the definition of  $\kappa_f$  as a limit in (18). Approximately speaking, if  $|\epsilon|$  is small, we expect

$$\left| \frac{f(x + \epsilon x) - f(x)}{f(x)} \right| \approx \kappa_f(x) |\epsilon|. \quad (23)$$

Table 1: Relative condition numbers of elementary functions

Function	Condition number
$f(x) = x + c$	$\kappa_f(x) = \frac{ x }{ x + c }$
$f(x) = cx$	$\kappa_f(x) = 1$
$f(x) = x^p$	$\kappa_f(x) =  p $
$f(x) = e^x$	$\kappa_f(x) =  x $
$f(x) = \sin(x)$	$\kappa_f(x) =  x \cot(x) $
$f(x) = \cos(x)$	$\kappa_f(x) =  x \tan(x) $
$f(x) = \log(x)$	$\kappa_f(x) = \frac{1}{ \log(x) }$

That is, whenever the data  $x$  is perturbed by a small amount, we expect that relative perturbation to be magnified by a factor of  $\kappa_f(x)$  in the result.

**Observation 2.3.** *If  $\kappa_f \approx 10^d$ , then we expect to lose up to  $d$  decimal digits of accuracy in computing  $f(x)$  from  $x$ .*

Large condition numbers signal when errors cannot be expected to remain comparable in size to roundoff error. We call a problem poorly conditioned or **ill-conditioned** when  $\kappa_f(x)$  is large, although there is no fixed threshold for the term.

If  $\kappa_f \approx 1/\epsilon_{\text{mach}}$ , then we can expect the result to have a relative error of as much as 100% simply by expressing the data  $x$  in finite precision. Such a function is essentially not computable at this machine epsilon.

**Example 2.7.** *Consider the problem  $f(x) = \cos(x)$ . By the table above,  $\kappa_f(x) = |x \tan x|$ . There are two different ways in which  $\kappa$  might become large:*

- *If  $|x|$  is very large, then perturbations that are small relative to  $x$  may still be large compared to 1. Because  $|f(x)| \leq 1$  for all  $x$ , this implies that the perturbation will be large relative to the result, too.*
- *The condition number grows without bound as  $x$  approaches an odd integer multiple of  $\pi/2$ , where  $f(x) = 0$ . A perturbation which is small relative to a nonzero  $x$  may not be small relative to  $f(x)$  in such a case.*

You may have noticed that for some functions, such as the square root, the condition number can be less than 1. This means that relative changes get *smaller* in the passage from input to output. However, every result in floating-point arithmetic is still subject to rounding error at the relative level of  $\epsilon_{\text{mach}}$ . In practice,  $\kappa_f < 1$  is no different from  $\kappa_f = 1$ .

### 2.2.3 Polynomial roots

Most problems have multiple input and output values. These introduce complications into the formal definition of the condition number. Rather than worry over those details here, we can still look at variations in only one output value with respect to one data value at a time.

**Example 2.8.** *Consider the problem of finding the roots of a quadratic polynomial; that is, the values of  $t$  for which  $at^2 + bt + c = 0$ . Here the data are the coefficients  $a$ ,  $b$ , and  $c$  that define the polynomial, and the solution to the problem are the two (maybe complex-valued) roots  $r_1$  and  $r_2$ . Formally, we might write  $f([a, b, c]) = [r_1, r_2]$  using vector notation.*

*Let's pick one root  $r_1$  and consider what happens to it as we vary just the leading coefficient  $a$ . This suggests a scalar function  $f(a) = r_1$ . Starting from  $ar_1^2 + br_1 + c = 0$ , we differentiate implicitly with respect to  $a$  while holding  $b$  and  $c$  fixed:*

$$r_1^2 + 2ar_1 \left( \frac{dr_1}{da} \right) + b \frac{dr_1}{da} = 0. \quad (24)$$

Solving for the derivative, we obtain

$$\frac{dr_1}{da} = \frac{-r_1^2}{2ar_1 + b}. \quad (25)$$

Hence the condition number for the problem  $f(a) = r_1$  is

$$\kappa_f(a) = \left| \frac{a}{r_1} \cdot \frac{dr_1}{da} \right| = \left| \frac{ar_1}{2ar_1 + b} \right| = \left| \frac{r_1}{r_1 - r_2} \right|, \quad (26)$$

where in the last step we used the quadratic formula:

$$|2ar_1 + b| = \left| \sqrt{b^2 - 4ac} \right| = |a(r_1 - r_2)|. \quad (27)$$

Based on (26), we can expect poor conditioning in the rootfinding problem if and only if  $|r_1| \gg |r_1 - r_2|$ . Similar conclusions apply for  $r_2$  and for variations with respect to the coefficients  $b$  and  $c$ .

The calculation in Example 2.8 generalizes to polynomials of higher degree.

**Observation 2.4.** *Roots of polynomials are ill-conditioned with respect to changes in the polynomial coefficients when they are much closer to each other than to the origin.*

The condition number of a root can be arbitrarily large. In the extreme case of a repeated root, the condition number is formally infinite, which implies that the ratio of changes in the root to changes in the coefficients cannot be bounded.

### Example 2.9.

#### 2.2.4 Exercises

- Use (19) to derive the relative condition numbers of the following functions appearing in Table 1.  
**(a)**  $f(x) = x^p$ ,   **(b)**  $f(x) = \log(x)$ ,   **(c)**  $f(x) = \cos(x)$ ,   **(d)**  $f(x) = e^x$ .
- Use the chain rule (22) to find the relative condition number of the given function. Then check your result by applying (19) directly.  
**(a)**  $f(x) = \sqrt{x+5}$ ,   **(b)**  $f(x) = \cos(2\pi x)$ ,   **(c)**  $f(x) = e^{-x^2}$ .
- Calculate the relative condition number of each function, and identify all values of  $x$  at which  $\kappa_f(x) \rightarrow \infty$  (including limits as  $x \rightarrow \pm\infty$ ).  
**(a)**  $f(x) = \tanh(x)$ ,   **(b)**  $f(x) = \frac{e^x - 1}{x}$ ,   **(c)**  $f(x) = \frac{1 - \cos(x)}{x}$ .
- Suppose that  $f$  and  $g$  are real-valued functions that have relative condition numbers  $\kappa_f$  and  $\kappa_g$ , respectively. Define a new function  $h(x) = f(g(x))$ . Show that for  $x$  in the domain of  $h$ , the relative condition number of  $h$  satisfies (22).
- Suppose that  $f$  is a function with relative condition number  $\kappa_f$ , and that  $f^{-1}$  is its inverse function. Show that the relative condition number of  $f^{-1}$  satisfies

$$\kappa_{f^{-1}}(x) = \frac{1}{\kappa_f(f^{-1}(x))}, \quad (28)$$

provided the denominator is nonzero.

- Referring to the derivation of (26), derive an expression for the relative condition number of a root of  $ax^2 + bx + c = 0$  due to perturbations in  $b$  only.

7. The polynomial  $x^2 - 2x + 1$  has a double root at 1. Let  $r_1(\epsilon)$  and  $r_2(\epsilon)$  be the roots of the perturbed polynomial  $x^2 - (2 + \epsilon)x + 1$ .

**(a)** / Using a computer or calculator, make a table with rows for  $\epsilon = 10^{-2}, 10^{-4}, 10^{-6}, \dots, 10^{-12}$  and columns for  $\epsilon$ ,  $r_1(\epsilon)$ ,  $r_2(\epsilon)$ ,  $|r_1(\epsilon) - 1|$ , and  $|r_2(\epsilon) - 1|$ .

**(b)** Show that the observations of part (a) satisfy

$$\max\{|r_1(\epsilon) - 1|, |r_2(\epsilon) - 1|\} \approx C\epsilon^q \quad (29)$$

for some  $0 < q < 1$ . (This supports the conclusion that  $\kappa = \infty$  at the double root.)

8. Generalize (26) to finding a root of the  $n$ th degree polynomial  $p(x) = a_n x^n + \dots + a_1 x + a_0$ , and show that the relative condition number of a root  $r$  with respect to perturbations only in  $a_k$  is

$$\kappa_r(a_k) = \left| \frac{a_k r^{k-1}}{p'(r)} \right|. \quad (30)$$

## 2.3 Algorithms

An idealized mathematical problem  $f(x)$  can usually only be approximated using a finite number of steps in finite precision. A complete set of instructions for transforming data into a result is called an **algorithm**. In most cases it is reasonable to represent an algorithm by another mathematical function, denoted here by  $\tilde{f}(x)$ .

Even simple problems can be associated with multiple algorithms.

**Example 2.10.** Suppose we want to find an algorithm that maps a given  $x$  to the value of the polynomial  $f(x) = 5x^3 + 4x^2 + 3x + 2$ . Representing  $x^2$  as  $(x)(x)$ , we can find it with one multiplication. We can then find  $x^3 = (x)(x^2)$  with one more multiplication. We can then apply all the coefficients (three more multiplications) and add all the terms (three additions), for a total of 8 arithmetic operations.

There is a more efficient algorithm, however: organize the polynomial according to **Horner's algorithm**,

$$f(x) = 2 + x(3 + x(4 + 5x)). \quad (31)$$

In this form you can see that evaluation takes only 3 additions and 3 multiplications. The savings represent 25% of the original computational effort, which could be significant if repeated billions of times.

### 2.3.1 Algorithms as code

Descriptions of algorithms may be presented as a mixture of mathematics, words, and computer-style instructions called *pseudocode*, which varies in syntax and level of formality. In this book we use pseudocode to explain the outline of an algorithm, but the specifics are usually presented as working code.

Of all the desirable traits of code, we emphasize clarity the most after correctness. We do not represent our programs as always being the shortest, fastest, or most elegant. Our primary goal is to illustrate and complement the mathematical underpinnings, while occasionally pointing out key implementation details.

As our first example, Algorithm ?? implements an algorithm that applies Horner's algorithm to a general polynomial, using the identity

$$\begin{aligned} p(x) &= c_1 + c_2x + \cdots + c_nx^{n-1} \\ &= \left( ((c_nx + c_{n-1})x + c_{n-2})x + \cdots + c_2 \right)x + c_1. \end{aligned} \quad (32)$$

#### About the code

The `length` function in line 1 returns the number of elements in vector `c`. The syntax `c[n]` accesses element `n` of a vector `c`. In Julia, the first index of a vector is 1 by default, so in line 2, the last element of `c` is accessed.

The `for / end` construct is a *loop*. The local variable `k` is assigned the value `n-1`, then the loop body is executed, then `k` is assigned `n-2`, the body is executed again, and so on until finally `k` is set to 1 and the body is executed for the last time.

The `return` statement in line 13 terminates the function and specifies one or more values to be returned to the caller. A function may have more than one `return` statement, in which case the first one encountered terminates the function; however, that coding style is mostly discouraged.

#### Example 2.11.

The quoted lines at the beginning of Algorithm ?? are a documentation string. The function itself starts off with the keyword `function`, followed by a list of its input arguments. The first of these is presumed to be a vector, whose length can be obtained and whose individual components are accessed through square bracket notation. After the computation is finished, the `return` keyword indicates which value or values are to be returned to the caller.



The Polynomials package for Julia provides its own fast methods for polynomial evaluation that supersede our simple Algorithm ?? function. This will be the case for all the codes in this book because the problems we study are well-known and important. In a more practical setting, you would take implementations of basic methods for granted and build on top of them.

### 2.3.2 Writing your own functions

Functions are a primary way of working in Julia. Any collection of statements organized around solving a type of problem should probably be wrapped in a function. Functions can be defined in text files with the extension `.jl`, at the command line (called the *REPL prompt*), or in notebooks.

As seen in Algorithm ??, one way to start a function definition is with the `function` keyword, followed by the function name and the input arguments in parentheses. For example, to represent the mathematical function  $e^{\sin x}$ , we could use

```
function myfun(x)
    s = sin(x)
    return exp(s)
end
```

The `return` statement is used to end execution of the function and return one or more (comma-separated) values to the caller of the function. If an executing function reaches its `end` statement without encountering a `return` statement, then it returns the result of the most recent statement.

For a function with a short definition like the one above, there is a more compact syntax to do the same thing:

```
myfun(x) = exp(sin(x))
```

You can also define **anonymous functions** or **lambda functions**, which are typically simple functions that are provided as inputs to other functions. This is done with an arrow notation. For example, to plot the function above (in the `Plots` package) without permanently creating it, you could enter

```
plot( x ->exp(sin(x)), 0, 6 )
```

As in most languages, input arguments and variables defined within a function have scope limited to the function itself. However, they can access values defined within an enclosing scope. For instance:

```
mycfun(x) = exp(c*sin(x))
c = 1; mycfun(3)  # returns exp(1*sin(3))
c = 2; mycfun(3)  # returns exp(2*sin(3))
```

There's a lot more to be said about functions in Julia, but this is enough to get started.

### 2.3.3 Exercises

1. Write a Julia function

```
function poly1(p)
```

that evaluates a polynomial  $p(x) = c_1 + c_2x + \cdots + c_nx^{n-1}$  at  $x = -1$ . You should do this directly, not by a call to or imitation of Algorithm ??. Test your function on  $r(x) = 3x^3 - x + 1$  and  $s(x) = 2x^2 - x$ .

2. In statistics, one defines the variance of sample values  $x_1, \dots, x_n$  by

$$s^2 = \frac{1}{n-1} \sum_{i=1}^n (x_i - \bar{x})^2, \quad \bar{x} = \frac{1}{n} \sum_{i=1}^n x_i. \quad (33)$$

Write a Julia function

```
function samplevar(x)
```

that takes as input a vector  $x$  of any length and returns  $s^2$  as calculated by the formula. You should test your function on the vectors `ones(100)` and `rand(200)`. If you enter using `Statistics` in Julia, then you can compare to the results of the `var` function.

3. Let  $x$  and  $y$  be vectors whose entries give the coordinates of the  $n$  vertices of a polygon, given in counterclockwise order. Write a function

```
function polygonarea(x,y)
```

that computes the area of the polygon, using this formula based on Green's theorem:

$$A = \frac{1}{2} \left| \sum_{k=1}^n x_k y_{k+1} - x_{k+1} y_k \right|. \quad (34)$$

Here  $n$  is the number of polygon vertices, and it's understood that  $x_{n+1} = x_1$  and  $y_{n+1} = y_1$ . (Note: The function `abs` computes absolute value.) Test your functions on a square and an equilateral triangle.

## 2.4 Stability

If we solve a problem using a computer algorithm and see a large error in the result, we might suspect poor conditioning in the original mathematical problem. But algorithms can also be sources of errors. When error in the result of an algorithm exceeds what conditioning can explain, we call the algorithm **unstable**.

### 2.4.1 Case study

In Example 2.8 we showed that finding the roots of a quadratic polynomial  $ax^2 + bx + c$  is poorly conditioned if and only if the roots are close to each other relative to their size. Hence, for the polynomial

$$p(x) = (x - 10^6)(x - 10^{-6}) = x^2 - (10^6 + 10^{-6})x + 1, \quad (35)$$

finding roots is a well-conditioned problem. An obvious algorithm for finding those roots is to directly apply the familiar quadratic formula,

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}, \quad x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}. \quad (36)$$

#### Example 2.12.

Example 2.12 suggests that the venerable quadratic formula is an *unstable* means of computing roots in finite precision. The roots themselves were not sensitive to the data or arithmetic—it's the specific computational path we chose that caused the huge growth in errors.

We can confirm this conclusion by finding a different path that avoids subtractive cancellation. A little algebra using (36) confirms the additional formula  $x_1 x_2 = c/a$ . So given one root  $r$ , we compute the other root using  $c/ar$ , which has only multiplication and division and therefore creates no numerical trouble.

#### Example 2.13.

The algorithms in Example 2.12 and Example 2.13 are equivalent when using real numbers and exact arithmetic. When results are perturbed by machine representation at each step, though, the effects may depend dramatically on the specific sequence of operations, thanks to the chain rule (22).

**Observation 2.5.** *The sensitivity of a problem  $f(x)$  is governed only by  $\kappa_f$ , but the sensitivity of an algorithm depends on the condition numbers of all of its individual steps.*

This situation may seem hopelessly complicated. But the elementary operations we take for granted, such as those in Table 1, are well-conditioned in most circumstances. Exceptions usually occur when  $|f(x)|$  is much smaller than  $|x|$ , although not every such case signifies trouble. The most common culprit is simple subtractive cancellation.

A practical characterization of instability is that results are much less accurate than the conditioning of the problem can explain. Typically one should apply an algorithm to test problems whose answers are well-known, or for which other programs are known to work well, in order to spot possible instabilities. In the rest of this book we will see some specific ways in which instability is manifested for different types of problems.

### 2.4.2 Backward error

In the presence of poor conditioning for a problem  $f(x)$ , even just the act of rounding the data to floating point may introduce a large change in the result. It's not realistic, then, to expect any algorithm  $\tilde{f}$  to have a small error in the sense  $\tilde{f}(x) \approx f(x)$ . There is another way to characterize the error, though, that can be a useful alternative measurement. Instead of asking, "Did you get nearly the right answer?", we ask, "Did you answer nearly the right question?"

**Definition 2.4** (Backward error). *Let  $\tilde{f}$  be an algorithm for the problem  $f$ . Let  $y = f(x)$  be an exact result and  $\tilde{y} = \tilde{f}(x)$  be its approximation by the algorithm. If there is a value  $\tilde{x}$  such that  $f(\tilde{x}) = \tilde{y}$ , then the relative **backward error** in  $\tilde{y}$  is*

$$\frac{|\tilde{x} - x|}{|x|}. \quad (37)$$

The absolute backward error is  $|\tilde{x} - x|$ .

Backward error measures the change to the original data that reproduces the result that was found by the algorithm. The situation is illustrated in Figure 3.

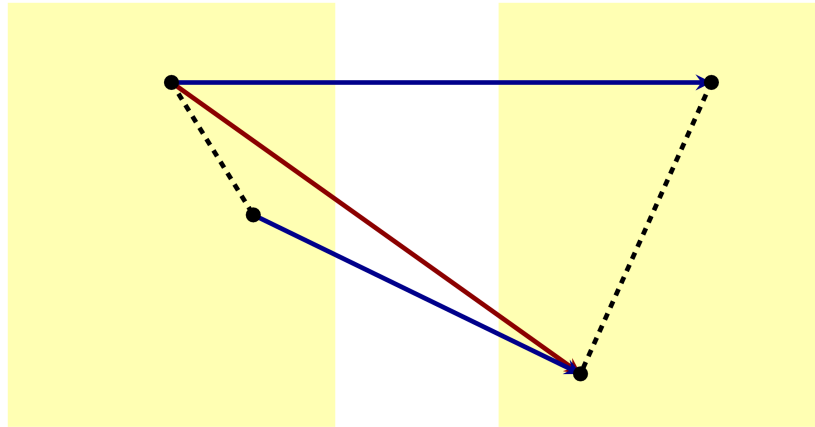


Figure 3: Backward error is the difference between the original data and the data that exactly produces the computed value.

#### Example 2.14.

Small backward error is the best we can hope for in a poorly conditioned problem. Without getting into the formal details, know that if an algorithm always produces small backward errors, then it is stable. But the converse is not always true: some stable algorithms may produce a large backward error.

**Example 2.15.** *One stable algorithm that is not backward stable is floating-point evaluation for our old friend,  $f(x) = x + 1$ . If  $|x| < \epsilon_{mach}/2$ , then the computed result is  $\tilde{f}(x) = 1$ , since there are no floating-point numbers between 1 and  $1 + \epsilon_{mach}$ . Hence the only possible choice for a real number  $\tilde{x}$  satisfying (37) is  $\tilde{x} = 0$ . But then  $|\tilde{x} - x|/|x| = 1$ , which indicates 100% backward error!*

### 2.4.3 Exercises

1. The formulas

$$f(x) = \frac{1 - \cos(x)}{\sin(x)}, \quad g(x) = \frac{2 \sin^2(x/2)}{\sin(x)}, \quad (38)$$

are mathematically equivalent, but they suggest evaluation algorithms that can behave quite differently in floating point.

(a) Using (19), find the relative condition number of  $f$ . (Because  $f$  and  $g$  are equivalent, the condition number of  $g$  is the same.) Show that it approaches 1 as  $x \rightarrow 0$ . (Hence it should be possible to compute the function accurately near zero.)

(b) Compute  $f(10^{-6})$  using a sequence of four elementary operations. Using Table 1, make a table like the one in Example 2.12 that shows the result of each elementary result and the numerical value of the condition number of that step.

- (c) Repeat part (b) for  $g(10^{-6})$ , which has six elementary steps.
- (d) Based on parts (b) and (c), is the numerical value of  $f(10^{-6})$  more accurate, or is  $g(10^{-6})$  more accurate?
2. Let  $f(x) = \frac{e^x - 1}{x}$ .
- (a) Find the condition number  $\kappa_f(x)$ . What is the maximum of  $\kappa_f(x)$  over  $-1 \leq x \leq 1$ ?
- (b) Use the “obvious” algorithm
- $$(\exp(x) - 1) / x$$
- to compute  $f(x)$  at  $x = 10^{-2}, 10^{-3}, 10^{-4}, \dots, 10^{-11}$ .
- (c) Create a second algorithm from the first 8 terms of the Maclaurin series, i.e.,

$$p(x) = 1 + \frac{1}{2!}x + \frac{1}{3!}x^2 + \dots + \frac{1}{8!}x^8. \quad (39)$$

Evaluate it at the same values of  $x$  as in part (b).

- (d) Make a table of the relative difference between the two algorithms as a function of  $x$ . Which algorithm is more accurate, and why?
3. The function

$$x = \cosh(y) = \frac{e^y + e^{-y}}{2} \quad (40)$$

can be inverted to yield a formula for  $\operatorname{acosh}(x)$ :

$$\operatorname{acosh}(x) = y = \log(x + \sqrt{x^2 - 1}). \quad (41)$$

For the steps below, define  $y_i = -4i$  and  $x_i = \cosh(y_i)$  for  $i = 1, \dots, 4$ . Hence  $y_i = \operatorname{acosh}(x_i)$ .

- (a) Find the relative condition number of evaluating  $f(x) = \operatorname{acosh}(x)$ . (You can use (41) or look up a formula for  $f'$  in a calculus book.) Evaluate  $\kappa_f$  at all the  $x_i$ . (You will find that the problem is well-conditioned at these inputs.)
- (b) Use (41) to approximate  $f(x_i)$  for all  $i$ . Compute the relative accuracy of the results. Why are some of the results so inaccurate?
- (c) An alternative formula is

$$y = -2 \log \left( \sqrt{\frac{x+1}{2}} + \sqrt{\frac{x-1}{2}} \right). \quad (42)$$

Apply (42) to approximate  $f(x_i)$  for all  $i$ , again computing the relative accuracy of the results.

4. (Continuation of [Exercise 1.3.2](#). Adapted from [Higham \(2002\)](#).) One drawback of the formula (33) for sample variance is that you must compute a sum for  $\bar{x}$  before beginning another sum to find  $s^2$ . Some statistics textbooks quote a single-loop formula

$$\begin{aligned} s^2 &= \frac{1}{n-1} \left( u - \frac{1}{n} v^2 \right), \\ u &= \sum_{i=1}^n x_i^2, \\ v &= \sum_{i=1}^n x_i. \end{aligned} \quad (43)$$

Try this formula for these three datasets, each of which has a variance exactly equal to 1:

```
x = [ 1e6, 1+1e6, 2+1e6 ]  
x = [ 1e7, 1+1e7, 2+1e7 ]  
x = [ 1e8, 1+1e8, 2+1e8 ]
```

Explain the results.

## 2.5 Next steps

An accessible but more advanced discussion of machine arithmetic and roundoff error can be found in Higham [Higham \(2002\)](#).

Interesting and more advanced discussion of the numerical difficulties of finding the roots of polynomials can be found in the article by Wilkinson, “The Perfidious Polynomial” [Wilkinson \(1984\)](#) and in the ripostes from Cohen, “Is the Polynomial so Perfidious?” [Cohen \(1994\)](#) and from Trefethen, “Six myths of polynomial interpolation and quadrature,” (Myth 6) [Trefethen \(2013\)](#).

### **3 Appendices**

#### **3.1 Julia codes**



## 3.2 Python codes

```
import numpy as np  
import FNC
```

## 4 Index

## Index

accuracy (relative vs. absolute), **8**  
algorithm, **16**

backward error, **19**

condition number  
  of a scalar function, **11**  
  of elementary functions, **12**

floating-point numbers, **7**, **9**

Horner's algorithm, **16**

IEEE 754, **8**

Julia  
  for, **16**  
  functions, **17**  
  indexing arrays, **16**

length, **16**  
return, **17**

machine epsilon, **7**  
  in double precision, **8**  
mantissa, *see* significand

NaN, **9**

significand, **7**  
significant digits, **8**  
stability, **19**  
subtractive cancellation, **11**, **12**, **19**

The Empire Strikes Back, **6**

unit roundoff, **7**

Yoda, **6**

## Acknowledgments

Finally, we have made no mention of one of Julia's defining features, *multiple dispatch*. Using its power wisely edges into advanced software design and usually requires a bird's-eye view of problems that beginners lack. We want to let students expend as much of their cognitive budget as possible on the mathematical principles that have universal application.

We are indebted to Qinying Chen, Hugo Diaz, Mary Gockenbach, Aidan Hamilton, Pascal Kingsley Kataboh, Lindsey Jacobs, Ross Russell, and Jerome Troy, who made this text more accurate and more readable with their sharp eyes and great suggestions. And we are deeply grateful to Paula Callaghan at SIAM, whose patience, dedication, and wisdom were crucial to seeing this through to the end.

## References

- A. Cohen. Is the Polynomial so Perfidious? *Numerische Mathematik*, 68(2):225–238, 7 1994. ISSN 0945-3245. doi:[10.1007/s002110050058](https://doi.org/10.1007/s002110050058).
- N. J. Higham. *Accuracy and Stability of Numerical Algorithms: Second Edition*. SIAM, 1 2002. ISBN 978-0-89871-802-7.
- L. N. Trefethen. *Approximation Theory and Approximation Practice*. SIAM, 1 2013. ISBN 978-1-61197-239-9.
- J. H. Wilkinson. *The Perfidious Polynomial*, volume 24, pages 1–28. Mathematical Association of America, 1984.