Chapter 1

Introductory concepts

It's all a lot of simple tricks and nonsense.
—Han Solo, *Star Wars*

One of the most important things to realize in computation—the thing that makes it intellectually exciting—is that methods which are mathematically equivalent may be vastly different in terms of computational performance. It often happens that the most obvious, direct, or well-known method for a particular problem is unsuitable in computational practice. Hence in this book the emphasis is on the study of methods: how to find them, study their properties, and choose among them.

1.1 Mathematical background

Broadly speaking, you should be familiar with basic calculus (typically three college semesters) and linear algebra (half or one semester). For the chapters on differential equations, an introductory course is recommended.

From linear algebra you should be comfortable with matrices, vectors, and simple algebra with them (addition and multiplication). Hopefully you are familiar with their connection to linear systems of the form $A\mathbf{x} = \mathbf{b}$, and how to solve these systematically via Gaussian elimination.

From calculus we will use ideas such as limits, continuity and differentiability freely but mostly without rigor. The most important calculus topic you are likely to need refreshment on is the **Taylor series**. For a function $f(x)$ near $x = a$, we have the two useful forms

$$f(x) = f(a) + (x-a)f'(a) + \frac{1}{2}f''(a)(x-a)^2 + \cdots + \frac{1}{n!}f^{(n)}(a)(x-a)^n + \cdots \tag{1.1a}$$

$$f(a+h) = f(a) + hf'(a) + \frac{h^2}{2}f''(a) + \cdots + \frac{h^n}{n!}f^{(n)}(a) + \cdots . \tag{1.1b}$$

(The connection between the two forms is just $x = a + h$.) Such a series is always valid at least for x lying in an interval $(a - R, a + R)$, where R may be zero or infinity. For values of x close to a (i.e.,

$|h|$ small), we can truncate the series after a few terms and get a good approximating polynomial—polynomials as a class being much more agreeable than general functions! The effects of truncation can sometimes be analyzed using the **remainder form** of the Taylor series,

$$f(a+h) = f(a) + hf'(a) + \cdots + \frac{1}{n!}f^{(n)}(a)h^n + \frac{1}{(n+1)!}f^{(n+1)}(\xi)h^{n+1}, \tag{1.2}$$

where $\xi = \xi(h)$ is an unknowable number somewhere in the interval $(0, h)$.

One of the most important Taylor series is the **geometric series**,

$$\frac{1}{1-x} = 1 + x + x^2 + x^3 + \cdots, \tag{1.3}$$

valid for all $|x| < 1$ only. It is the limit of the finite geometric sum

$$\sum_{k=0}^{n} x^k = \frac{1-x^{n+1}}{1-x}. \tag{1.4}$$

The geometric series can be generalized to the **binomial series**,

$$(1+x)^\alpha = 1 + \alpha x + \frac{\alpha(\alpha-1)}{2}x^2 + \cdots + \frac{\alpha\cdots(\alpha-n+1)}{n!}x^n + \cdots. \tag{1.5}$$

It also converges for $|x| < 1$, and it becomes a finite sum if α is a nonnegative integer. Another fundamental series is the **exponential series**,

$$e^x = 1 + x + \frac{1}{2}x^2 + \cdots \frac{1}{n!}x^n + \cdots, \tag{1.6}$$

which is valid for all x.

Problems

1.1.1. Derive the first four nonzero terms of the Taylor series of the following functions about the given points.

(a) $\cos(x)$, $a = 0$
(b) $\cos(x)$, $a = \pi/2$
(c) $\cosh(x)$, $a = 0$
(d) e^{-x^2}, $a = 0$ (Hint: Use (1.6).)
(e) $\sin(x)$, $a = 0$
(f) $\sin(x)$, $a = \pi/2$
(g) $\sinh(x)$, $a = 0$
(h) $\sqrt{x}$, $a = 1$

1.1.2. (a) Derive

$$\log(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \cdots + (-1)^{n-1}\frac{x^n}{n} + \cdots, \tag{1.7}$$

valid for $|x| < 1$.

(b) Derive

$$\log\left(\frac{1+y}{1-y}\right) = 2y + \frac{2}{3}y^3 + \frac{2}{5}y^5 + \frac{2}{7}y^7 + \cdots, \tag{1.8}$$

valid for $|y| < 1$.

1.2 Uses and types of computation

The topic of this book is computation for mathematically formulated problems. Our discussion excludes other uses of computation, such as communications, data retrieval and mining, and visualization, though these of course play important roles in science and engineering. Mathematical computation is needed because the tools of mathematical analysis employed by humans, while able to give many insights, cannot nearly answer all the questions we like to pose. Computers, able to perform arithmetic and logical operations trillions of times faster than we do, can be used to produce additional insights.

One of the very first decisions you need to make before turning to a computer is whether you will use so-called **symbolic** or **numerical** methods. Symbolic computational environments (Maple, Mathematica, and Mathcad, for example) emphasize two particular abilities: understanding of abstract variables and expressions, such as $\int x^n\,dx$; and the use of exact or arbitrarily accurate numbers, such as π or $\sqrt{2}$. These systems do mathematical manipulation much as you would, only much faster and more encyclopedically. Numerical environments (MATLAB, Fortran, C, and most other compiled languages) operate natively on numerical values. The distinguishing feature of numerical computation is that noninteger real numbers are represented only approximately, and as a result arithmetic can be done only imprecisely.

It might seem puzzling that anyone would ever choose a system of imprecise arithmetic over an exact one. However, exact arithmetic demands much larger burdens of computer resources and time, and in many contexts there is no payoff for the extra costs.

As an example, consider the problem of finding the mth root of a number, $A^{1/m}$. In principle solving this problem takes an infinite number of the fundamental arithmetic operations $+$, $-$, $\times$, $\div$. There is a simple and ancient geometric algorithm for the case of square roots, $m = 2$. Suppose $A > 0$ is given and we have a guess x for $\sqrt{A}$. We can imagine x as one side of a rectangle whose area is A. Hence the other dimension of the rectangle is A/x. If these two lengths are identical (i. e., the rectangle is in fact a square), then their shared value is the square root, and we are done. Otherwise, we can easily show that the square root lies between x and A/x, so it seems reasonable to use the average of them as an improved guess. We represent this process with the formula

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{A}{x_n}\right), \qquad n = 0, 1, 2, \ldots. \tag{1.9}$$

This formula represents an **iteration**. Given an initial guess x_0 for $\sqrt{A}$, we set $n = 0$ and get a (better?) guess x_1. Then we set $n = 1$ and get x_2, etc., until we are satisfied or give up.

For mth roots we can generalize this reasoning to an m-dimensional box of volume A whose sides are $x, x, \ldots, A/x^{m-1}$. In this case the averaging leads to

$$x_{n+1} = \frac{1}{m}\left((m-1)x_n + \frac{A}{x_n^{m-1}}\right), \qquad n = 0, 1, 2, \ldots. \tag{1.10}$$

```
> Digits:= 15:
> A:=2;
                                   A := 2
> x:=0.9;
                                   x := 0.9
> for n from 1 to 5 do x:= (4*x+A/x^4) / 5; end;
                            x := 1.32966316110349
                            x := 1.19169619004928
                            x := 1.15169123382098
                            x := 1.14871386970513
                            x := 1.14869835541612
> A^0.2;
                              1.14869835499704
```

Figure 1.1: Numerical results for the mth-root iteration (1.10).

Figure 1.1 shows the result of this iteration with $m = 5$ and $A = 2$. Although we are using Maple, we have effectively put it into "numerical mode" by giving x an initial value with a decimal point in it. Hence all the results are rounded off to 15 significant digits. The iteration seems to behave quite nicely. After five iterations it has found 10 digits accurately.

Now we let Maple run in "symbolic mode," in which the initial (and hence subsequent) values of x are represented exactly as rational numbers. Figure 1.2 shows what happens. It's not hard to see that the number of digits needed to represent x goes up by a factor of about five for each iteration. That is, the amount of storage and time needed for each iteration grows *exponentially*. Indeed, the result of the fifth iteration (not shown) requires 6768 decimal digits to represent it. But it's no more accurate than the result shown in Figure 1.1—which means that 6758 (99.8%) of those exact digits are wasted!

This example is not meant to disparage symbolic computing. To Maple, the mth root of A is simply $A^{1/m}$, a number with certain exact properties, and being able to manipulate it subject to those properties can be useful. But the motivation of the example is important: there is nothing to be gained by exactly representing results that are approximate. Most problems in science and engineering are too difficult to solve exactly, and the errors involved in approximating the solution (or measuring physical parameters, or creating a mathematical model) are usually many orders of magnitude larger than rounding errors. Thus, while symbolic capabilities are indispensable in some types of computations, the bulk of mathematical computation is done via numerical algorithms. Those are the exclusive focus of this book.

1.3 Algorithms

An **algorithm** is, in the most abstract sense, a map or function operating on data to produce results. The key reason for the existence of numerical analysis is that algorithms that are mathematically equivalent may differ in other important ways.

```
> Digits:= 15:
> A:=2;
```

$$A := 2$$

```
> x:=9/10;
```

$$x := \frac{9}{10}$$

```
> for n from 1 to 4 do x:= (4*x+A/x^4) / 5; end;
```

$$x := \frac{218098}{164025}$$

$$x := \frac{1105661883737230646819561561}{92780516793588592766336100 0}$$

$$x := 19961455911753333063529345289110016450534871922539507248083678449375982$$
$$84128332262646103840263648655387306832643308866462326492060703801173322\$$
$$98211150635212586901126169793232808974008205971397853163544868494999711$$
$$81278814684050370694556238213120158330720148323526470012 50$$

$$x := 79027252203407341897083925397418777275392266650353954758093730438139780$$
$$67354131722649629772203364481453087220154718314766467925791215241119619$$
$$23737293165965573151766581612914118999508658475835910853846256616657201$$
$$26661199036088500501956539724956408415179507913306620636333170717604637$$
$$70888932561691295912954915906849413079434909995631850519075706003390608$$
$$63677268461638210521200706732703660842443403760823665908624232607583838$$
$$78704603214666601044151843660476468223979225250351991189355067602654803$$
$$39492150675222359052084626690895436662724905396499965352922149276174027$$
$$58788201808825528966236394982617737794473576994625919310689425204322844$$
$$26582993447441433452854833800268796289735487883650324055594215467791971$$
$$53274100542283821161353398779256751914133584287380925685841075440866145$$
$$57069191544491551637385650709872256130911423446109466159747134714608051$$
$$57777035770877339010276461244220233125102854834633408573642278647641467$$
$$56643399178942049518984535528174131139384299798035243604499492211438513$$
$$27250121556883507289971662843715754023570121319166903717181127271538507$$
$$67617366171485452681555160477215666087615055630685780385813589076552914$$
$$64659719619705325357416310671712227441273257912271799373178722065562411$$
$$97190979926499074517116326770069948928020830298730631922266871943869268$$
$$78662395620349941301382377478910605634673095731984716500 3125$$

```
> evalf((4*x+A/x^4) / 5);
```

$$1.14869835541612$$

```
> A^0.2;
```

$$1.14869835499704$$

Figure 1.2: Symbolic results for the mth-root iteration (1.10).

Example 1.1. Suppose we want to find an algorithm that maps a given x to the value of $p(x) = 4x^3 + 3x^2 + 2x + 1$. We can find x^2 and then x^3 at the cost of two multiplications. We can then apply all the coefficients (three more multiplications) and add the terms (three additions), for a total of 8 arithmetic operations.

There is a more efficient way. Organize the polynomial according to **Horner's rule**:

$$p(x) = 1 + x\big(2 + x(3 + 4x)\big).$$

From this form you can see that evaluation takes only 3 additions and 3 multiplications. The savings, while small in an absolute sense, represent 25% of the original computational effort.

The precise notion of an algorithm is usually kept informal. Equation (1.10), for example, implies an algorithm in the form of an **iteration**, a common device. One could define the data to be the values A and m, and the result to be the limit of the sequence arising from the iteration. However, while this iteration could in principle be carried on forever, in practice we use algorithms that stop after a finite number of steps. That goal may be accomplished by terminating the iteration according to some predetermined criteria—say, a bound on the number of iterations. While this is an undeniably important change in the algorithm, we often hide or ignore the distinction between the infinite form (which is often easier to analyze in some ways) and the form that must be used on a real computer.

An algorithm may be expressed as a set of instructions written in a combination of words and mathematical or computer-science notation, loosely called **pseudocode**. A pseudocode description of an algorithm should contain enough information so that all correct implementations of it behave in exactly the same way. Pseudocode for a terminating square-root algorithm based on (1.9) might look like:

Input A, ϵ, *maxiter*

Set $x := A/2$, $n := 1$

Repeat

  Set $y := x$

  Set $x := \frac{1}{2}(x + A/x)$

  Set $n := n + 1$

until $(|x - y| < \epsilon)$ or $(n > \mathit{maxiter})$

Return x

Pseudocode is useful as a compact means of discussing an algorithm without referring to a specific computer language (which some day will go out of favor) and without needing to explain details that your audience takes for granted. Because there is little for us to take for granted at this point, and because you will benefit by seeing algorithms run on particular test cases, in this book we eschew pseudocode in favor of MATLAB functions. For the most part, this choice does not hurt readability much. A functional form of the above pseudocode is shown in Function 1.1.

Function 1.1 Iterative approximation of square roots.

```
1  function x = sqroot(A,tol)
2  % SQROOT  Iterative approximation to the square root.
3  % Input:
4  %   A     Value to be rooted (scalar)
5  %   tol   Desired upper bound for the error in the result (scalar)
6  % Output:
7  %   x     An aproximation to sqrt(A) (scalar)
8
9  maxiter = 16;
10 x = A/2;  y = Inf;
11 n = 1;
12 while abs(x-y) > tol
13   y = x;
14   x = (x+A/x)/2;
15   n = n+1;
16   if n > maxiter, warning('Maximum iterations exceeded'), break, end
17 end
18
```

Function 1.1 is structured slightly differently from its pseudocode form due to conventions in the MATLAB language. The opening block of comments serves as **documentation** for the function. There is no universal standard for such documentation, but you are encouraged to choose a consistent, thorough format. The reason for setting `y=Inf` in line 10 is to ensure that the `while` condition is initially true, so the loop body is executed at least once. Line 16 shows how functions can display warnings to the user but still return output (after `break` exits the loop); an `error` command would also display a message but halt execution and return no output as well. Appropriate warnings and errors are very good programming practice, but we do not often include them in this book in order to maintain our mathematical focus.

As another example, Function 1.2 implements an algorithm that applies Horner's rule,

$$p(x) = c_1 x^n + c_2 x^{n-1} + \cdots + c_n x + c_{n+1} = \Big(\cdots \big((c_1 x + c_2) x + c_3 \big) x + \cdots + c_n \Big) x + c_{n+1}. \quad (1.11)$$

In MATLAB, it is conventional to represent a polynomial by a vector of its coefficients in descending order of degree. As you can see from Function 1.2, MATLAB deals with vectors very naturally: you can query the vector to find its length, and you can access the kth element of a vector c by typing `c(k)`. Like Function 1.1, Function 1.2 features an iteration. In this case, however, the number of iterations is known before the iteration begins, and consequently we use a `for` construction rather than `while` as in Function 1.1.

The MATLAB codes in this section are not necessarily what an experienced MATLAB programmer would write. MATLAB's language has a number of features that can make certain tasks get expressed more compactly or generally than when using a more obvious syntax. In this book, however, we try to err on the side of clarity, using only a limited range of MATLAB's power. As you will see, our MATLAB functions are still quite different (and simpler) than the

Function 1.2 Evaluation of a polynomial by Horner's rule.

```
function p = horner(c,x)
% HORNER  Evaluate polynomial using Horner's rule.
% Input:
%   c     Coefficients of polynomial, in descending order (vector)
%   x     Evaluation point (scalar)
% Output:
%   p     Value of the polynomial at x (scalar)

n = length(c);
p = c(1);
for k = 2:n+1
  p = x*p + c(k);
end
```

equivalent functions you would write in a lower-level language like C or FORTRAN, or a more object-oriented language like Java.

MATLAB has a built-in function `polyval` that does essentially the same thing as our `horner`, so for good form we use `polyval` in the example below and the rest of the book.

Example 1.2. Taylor expansion (see problem 1.1.2) implies that

$$\log(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \cdots + (-1)^{n-1}\frac{x^n}{n} + \cdots , \tag{1.12}$$

which is valid for $|x| < 1$ and $x = 1$. Summing the series implies an infinite algorithm, but we can write a MATLAB function that returns the polynomial resulting by truncation to the first n terms of this series.

```
function f = logseries1(x,n)

c(1) = 0;                               % constant term (x^0)
for k = 1:n
  c(k+1) = (-1)^(k+1)/k;                % coeff of x^k
end
c = c(end:-1:1);                        % convert to descending order
f = polyval(c,x);
```

In this function we encounter a common problem: the way we want to subscript a vector mathematically does not agree with MATLAB conventions. In line 5, for example, we must conform to the fact that vectors are indexed starting at 1, never at 0. And in line 7, we compensate for MATLAB's preference for descending order of degree in polynomial coefficients.

Applying our function for $x = 1/2$, we get

```
>> format long  % to show all digits
>> log(1.5)     % exact value

ans =
   0.40546510810816
```

```
>> logseries1(0.5,5)

ans =
   0.40729166666667

>> logseries1(0.5,50)

ans =
   0.40546510810816
```

In this case, 50 terms are sufficient to get a valid approximation for all available digits. In general practice it might make more sense to select the truncation value of n automatically, based on x.

Constructing algorithms often involves balancing tradeoffs. Factors desired in an algorithm are

Stability If the inputs are changed by tiny amounts, the outputs should not be dramatically different (unless the problem itself has this nature).

Reliability The algorithm should do what it claims and state its limitations.

Robustness The algorithm should be applicable to as broad a range of inputs as possible and detect run-time failures.

Accuracy Errors should be as small as possible. Ideally one should be able to choose in advance an error level that is then guaranteed by the method; more often we have to settle for asymptotic bounds on or estimates of the error.

Efficiency The algorithm should minimize the effort needed (say, execution time) to achieve a given level of accuracy.

Storage The algorithm should require minimal memory resources.

Simplicity The algorithm should be as easy as possible to describe and program.

Also, remember that every algorithm solves only a mathematical problem. It cannot offer guidance on how well that problem models the true situation of interest, how well physical parameters have been measured, and so on. Always approach computer-generated results with healthy skepticism.

Problems

1.3.1. Modify Function 1.1 to find the mth root of a number using (1.10), where m is another input parameter for the function. Be sure to test your function on a few cases.

1.3.2. There is an iteration to find the inverse of a nonzero value a using only multiplication and subtraction:

$$x_{n+1} = x_n(2 - ax_n).$$

(a) Show that if $x_{n+1} = x_n = r$ (we say that r is a fixed point of the iteration), then either $r = 0$ or $r = 1/a$.

(b) Write a MATLAB function to implement this iteration. You will have to specify how to stop the algorithm when $|x_n - a^{-1}| \leq \epsilon$, for an arbitrary ϵ, *without* doing any divisions or inversions.

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

$$s^2 = \frac{1}{n-1} \sum_{i=1}^{n} (x_i - \overline{x})^2, \qquad \overline{x} = \frac{1}{n} \sum_{i=1}^{n} x_i. \tag{1.13}$$

Write a MATLAB function `s2=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 some data and compare it to MATLAB's built-in `var`.

1.4 Errors

In the two preceding sections we have seen that we will have to expect errors as a cost of doing numerical business. We have a few different ways of measuring and talking about errors.

Suppose that x is an approximation to an exact result x^*. Three ways to measure the error in x are

Error	$x^* - x$	**Relative error**	$\dfrac{\lvert x^* - x \rvert}{\lvert x^* \rvert}$
Absolute error	$\lvert x^* - x \rvert$	**Accurate digits**	$\left\lfloor -\log_{10}\left(\dfrac{\lvert x^* - x \rvert}{\lvert x^* \rvert}\right) \right\rfloor$

The terms "error" and "absolute error" are often used interchangeably. To define accurate digits we have used the floor function $\lfloor t \rfloor$, which gives the largest integer no greater than t. Digits correspond to the notion of significant digits in science, and are also linked to scientific notation; e. g., 6.02×10^{-23} is given to 25 decimal places but just three digits.

We have assumed in our definitions that x and x^* are numbers. If they are vectors, functions, etc., we need a definition of magnitude or **norm** to replace the absolute values. (Sometimes norms are written using the same symbol.)

The two previous sections have introduced two unavoidable sources of error for correctly implemented numerical algorithms: **roundoff error**, due to approximating infinitely long numbers, and **truncation error**, due to approximating infinitely long algorithms. Almost all of the time, truncation error is far more important than roundoff error.

Roundoff error

We cannot represent the entire abstract continuum of real numbers exactly using data of fixed length. The error we incur by choosing a finite subset of real numbers that can be represented exactly is usually called **roundoff error**. It is determined by the computer's internal representation of numbers.

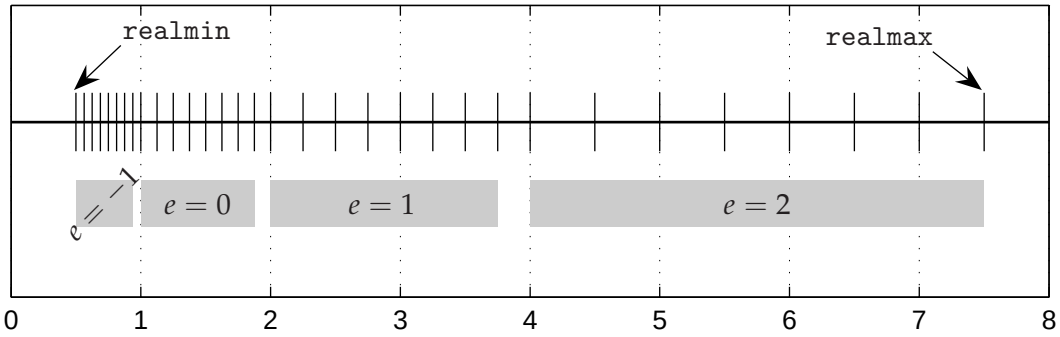

Figure 1.3: Floating point values as defined by (1.14) with 8 choices for f and 4 choices for e (only positive ones shown). There are 8 values inside $[1, 2)$, and these are scaled by factors 2^{-1}, 2^1, and 2^2 to complete the set. In this system the smallest positive value is $\frac{1}{2}$ and the largest is $4 \times (15/8) = 7.5$. In 64-bit double precision there are $2^{52} \approx 4.5 \times 10^{15}$ numbers in $[1, 2)$, and they are scaled using exponents between -1022 and 1023.

Virtually all PCs and workstations today adhere to the **IEEE 754** standard, which essentially represents values in base-2 scientific notation:

$$\pm(1 + f) \times 2^e. \tag{1.14}$$

Such values are called **floating-point numbers**. We will consider only **double precision** numbers, for which f is a number in $[0, 1)$ represented using 52 binary digits, e is an integer between -1022 and 1023, requiring 11 binary digits, and the sign of the number requires one more binary bit, leading to 64 bits total.[1] As a consequence, the smallest positive floating point number, known as `realmin`, is 2^{-1022}, and the largest, known as `realmax`, is 2^{1024}.

The floating point numbers are *not* distributed evenly in a visual sense along the real line. Figure 1.3 indicates how the positive values are distributed for the equivalent of 6-bit numbers. The main consequence of the standard is that every number between `realmin` and `realmax` is represented with a *relative* error no greater than $\epsilon_M = 2^{-52} \approx 2.2 \times 10^{-16}$. This number is called **machine precision** or **unit roundoff**, and it corresponds to about 16 significant digits in the decimal system. This is quite a respectable amount of precision, especially compared to most measurements of the physical world. In MATLAB, the machine precision is automatically assigned to the name `eps`.[2]

We will no longer be concerned with the limitations implied by `realmin` and `realmax`, which arise infrequently in practice. Without going into any details, we will also assume that the results of basic arithmetic operations have a relative error bounded by ϵ_M. For example, if x and y are floating-point numbers, then the computed value of xy is a floating-point value $xy(1 + \delta)$, for a number δ satisfying $|\delta| < \epsilon_M$. This is an idealization, but not much of one.

We do not attempt the herculean task of tracking all the individual errors due to roundoff in a computation! For almost all calculations today, the random accumulation of roundoff errors

[1] IEEE 754 also defines a few special "denormalized" numbers that we do not deal with in this book.

[2] As with any name in MATLAB, you can legally assign `eps` a new value. But that will not change the unit roundoff for future computations! That value is determined by the computer's hardware.

is of no concern. In part this is due to the **random walk** effect whereby positive and negative errors tend to partially cancel. Statistically speaking, after N operations a random accumulation of roundoff error amplifies the unit roundoff by a factor of $\sqrt{N}$ on average. So, for instance, even after a trillion operations, we still should expect ten accurate digits.

The real danger is not from randomly accumulated errors but from systematic errors. Sometimes, a single operation causes an intolerable and avoidable amount of roundoff. In other cases, a method allows random errors to be organized and reinforced, much as a trumpet uses resonance to turn white noise at the mouthpiece into a loud single pitch at the bell. These problems are loosely referred to as **instability**, which we discuss further in section 1.6 and throughout the rest of the book.

Truncation error

Some problems, such as sorting a list of words, can be solved exactly in a finite number of operations. Problems of this type arising in linear algebra are the subjects of Chapter 2 and Chapter 3. Most nonlinear problems and problems related to continuous variables, such as integration and differential equations, cannot be solved in a finite number of operations, even in principle. In those problems we incur a **truncation error** resulting from the termination of an infinite process. Usually we can, at least in theory, find an approximate solution with a truncation error as small as we wish, a notion called **convergence**. However, in a given algorithm, reducing the error comes at the cost of more operations.

We already encountered truncation error in the square root algorithm written as Function 1.1. We decided to terminate that iteration when successive approximations to $\sqrt{A}$ differed by less than a given tolerance. This is pretty reliable in practice but lacks mathematical rigor. A more rigorous approach is to observe that for $x > 0$,

$$|x - \sqrt{A}| = \left| \frac{x^2 - A}{x + \sqrt{A}} \right| < \frac{|x^2 - A|}{x} = \left| x - \frac{A}{x} \right|.$$

The left-hand side is the error in an estimate x to $\sqrt{A}$. We cannot compute this exactly for a given x unless we already know the result. However, the quantity on the right can be computed just from x and the input A; in fact, it is the difference in the two rectangle side lengths that are averaged in the iteration itself. Hence we have a computable, rigorous truncation error bound. Sadly, this is something of a rarity among interesting applications. Still, we can often make bounds or estimates that tell us quite a bit about how a truncated algorithm converges. Often that insight is more valuable than the estimate itself.

Example 1.3. Consider what happens when we truncate the Taylor series in (1.12), valid for $|x| < 1$, at degree n—after a little creative factorization:

$$\log(1+x) = x - \frac{x^2}{2} + \cdots + (-1)^{n-1}\frac{x^n}{n} + (-1)^n \frac{x^{n+1}}{n+1}\left(1 - \frac{n+1}{n+2}x + \frac{n+1}{n+3}x^2 - \cdots\right). \tag{1.15}$$

The truncation error is itself an infinite series. Taking the absolute value of each term in the error series, we see that the error is bounded above by

$$\frac{|x|^{n+1}}{n+1}\left(1+\frac{n+1}{n+2}|x|+\frac{n+1}{n+3}|x|^2+\cdots\right) < \frac{|x|^{n+1}}{n+1}\left(1+|x|+|x|^2+\cdots\right) = \frac{|x|^{n+1}}{n+1}\left(\frac{1}{1-|x|}\right). \quad (1.16)$$

This error bound tells us we can expect larger truncation errors as $x \to 1$, so truncation of this series is not a good idea to compute the logarithm near 2.

If instead we set $1+x = (1+y)/(1-y)$, or equivalently $y = x/(x+2)$, then we can use the other series from problem 1.1.2:

$$\log\left(\frac{1+y}{1-y}\right) = 2y + \frac{2}{3}y^3 + \frac{2}{5}y^5 + \frac{2}{7}y^7 + \cdots. \quad (1.17)$$

Similar manipulations as before (see problem 1.4.2) give the error bound

$$\frac{2}{2m+1}|y|^{2m+1}\left(\frac{1}{1-|y|^2}\right) \quad (1.18)$$

after truncation of (1.17) at degree $n = (2m-1)$. Since $x \to 1$ corresponds to $y \to \frac{1}{3}$, this bound still decreases rapidly as $n \to \infty$ for computing $\log(2)$. In fact, $y \to 1$ only as $x \to \infty$, so this series is appropriate for many real values of x.

Very often one can find an "obvious" method that has a fairly large truncation error—that is, one that decreases slowly with increased effort. Improving the rate of decay in truncation error, while not creating instability, is a major goal in numerical analysis.

There is a third, ill-defined (and ever-changing) category of finite problems whose exact solutions are so time-consuming that they may as well be infinite. One example of this situation is chess, which does after all have a finite number of possible board configurations, like tic-tac-toe. However, the number of possibilities in chess is so large that humans, and, for the forseeable future, computers, have to play using inexact, truncated strategies.

Problems

1.4.1. In this problem you will experiment with the random walk. Write a function `randwalk(N)` that finds the sum of a random sequence of values chosen from $\{1, -1\}$ with equal probability. (You can use the built-in `rand` or `randn` for this.) Pick a value of N between 10 and 100, and run `randwalk` 10,000 times, as suggested here:

```
for n=1:10000, s(n)=randwalk(N); end
mean(s)
mean( abs(s) )
```

Verify that the first result is near zero and that the second is approximately $\sqrt{N}$.

1.4.2. (Continuation of Example 1.3.)

(a) Using manipulations similar to (1.15) and (1.16), derive the bound

$$\frac{2}{2m+1}|y|^{2m+1}\left(\frac{1}{1-|y|^2}\right)$$

on the absolute error resulting from truncation of (1.17) at degree $n = (2m-1)$.

(b) Modify `logseries1` from Example 1.2 into a new function `logseries2(x,n)`. Your function should use the series (1.17) to compute the result.

(c) With a trial-and-error or systematic approach, use `logseries1` to find the smallest value of n that gives an observed absolute error less than 10^{-6} when x takes on each of the values $\pm 0.5, \pm 0.75, \pm 0.9, \pm 0.99$. Then do the same for `logseries2`, and make a table comparing the efficiency of the two methods.

1.4.3. One definition for the number e is

$$e = \lim_{n\to\infty} \left(1 + \frac{1}{n}\right)^n.$$

Perform a MATLAB experiment approximating e using this formula with values $n = 10, 10^2, \ldots, 10^{15}$. Using the `short e` format, compute the error for each value of n, and explain the results in terms of roundoff errors.

1.4.4. The exponential series (1.6) converges for all real values of x, but when $|x|$ is large, many terms are needed to make a good approximation.

(a) Using MATLAB, find out experimentally how many terms of the series are needed in order to evaluate e^{10} to a relative error of less than 10^{-4}.

(b) Let $p_n(x)$ be the nth-degree polynomial resulting from truncating the series. Show that if $|x| \leq L$, the maximum value of $|e^x - p_n(x)|$ occurs at $x = \pm L$. (Hint: First explain why $e^x - p_n(x)$ is nonnegative, then find the one place where its derivative is zero.)

(c) A well-known algorithm for exponentials is the **scaling and squaring** method, based on the fact that $e^x = (e^{x/2})^2$. One divides x by two repeatedly until $|x| < 1$, evaluates the exponential for the reduced value (for example, using the Taylor series), and then squares the result as many times as x was halved in the first step. Because the Taylor series is applied to a small number, relatively few terms are needed for convergence. Write a MATLAB function that applies the scaling and squaring algorithm. Use the result of part (b) to determine experimentally how many terms of the series are needed to get an absolute accuracy of 10^{-10}.

1.5 Conditioning

What can we reasonably expect to accomplish when we try to compute values of a function f? Because of roundoff and truncation errors, we cannot expect that the computed result $\tilde{y}$ equals the exact $y = f(x)$. Naively, we might hope to make the absolute or relative error in $\tilde{y}$ small, but, as we will see, this is not always realistic. Instead, it is more convenient to define $\tilde{x}$ such that $\tilde{y} = f(\tilde{x})$, and require that the error in $\tilde{x}$ is small. This is known as the **backward error**, and by way of contrast, error in $\tilde{y}$ is called the **forward error**.

The connection between forward error and backward error is called the **conditioning** of the problem. The factor by which input perturbations can be multiplied is the **condition number**, usually given the symbol κ. In words,

$$\kappa = \frac{\text{forward error}}{\text{backward error}} = \frac{\text{change in output}}{\text{change in input}}. \tag{1.19}$$

Because floating-point arithmetic introduces relative errors in numbers, we often define κ in terms of relative changes. If a relative condition number is $\kappa \approx 10^m$, we may expect to lose as many as

m accurate digits in finding the solution of a problem. When m is comparable to the number of digits available on the computer, we say the underlying problem is **ill conditioned**, as opposed to **well conditioned**. The transition from good to ill conditioning as a function of κ is gradual and context-dependent, so these terms remain a bit vague.

We can begin to be more precise about condition numbers when we consider simple problems in the form of a function $f(x)$ that maps real inputs to real outputs.

Example 1.4. Consider the problem of squaring a real number, as represented by the function $f(x) = x^2$. We can represent a relative perturbation of size ϵ to x by the value $x(1+\epsilon)$. The relative change in the result is

$$\frac{f\big(x(1+\epsilon)\big) - f(x)}{f(x)} = \frac{[x(1+\epsilon)]^2 - x^2}{x^2} = (1+\epsilon)^2 - 1 = 2\epsilon + \epsilon^2.$$

Thus the ratio of relative changes is

$$\frac{2\epsilon + \epsilon^2}{\epsilon} = 2 + \epsilon.$$

If we think of changes to x as being induced by roundoff error and therefore very small, it seems reasonable to neglect ϵ compared to 2. Thus we would say the relative condition number is $\kappa = 2$. This is considered very well conditioned.

We now formalize the process by making the limit $\epsilon \to 0$ part of the definition for κ. In the scalar problem $f(x)$, the absolute condition number is just

$$\kappa(x) = \lim_{\epsilon \to 0} \frac{f(x+\epsilon) - f(x)}{\epsilon} = f'(x).$$

For relative condition numbers, the computation is not much harder:

$$\kappa(x) = \frac{\text{relative change in output}}{\text{relative change in input}} = \lim_{\epsilon \to 0} \frac{\dfrac{f(x+\epsilon) - f(x)}{f(x)}}{\dfrac{(x+\epsilon) - x}{x}} \tag{1.20}$$

$$= \lim_{\epsilon \to 0} \frac{x}{f(x)} \frac{f(x+\epsilon) - f(x)}{\epsilon} \tag{1.21}$$

$$= \frac{x f'(x)}{f(x)}. \tag{1.22}$$

This clearly agrees with Example 1.4. Note that *the condition number may depend on the input*.

Example 1.5. The most fundamental example of an ill conditioned problem is the difference of two numbers. For simplicity, we let $f(x) = x - y$, where we consider y to be fixed. Using (1.22), we get

$$\kappa = \frac{x}{x - y}. \tag{1.23}$$

This is large when $|x - y| \ll |x|$, or when x and y are much closer to each other than they are to zero. The difficulty is that the result is found accurately relative to the input operands and not to itself.

For instance, consider the two 12-digit numbers 3.14159265359 and 3.14159260000. Their difference is 5.539×10^{-8}. This number is known only to four significant digits, and we have "lost" eight of the orignial twelve digits in the representation. In practice, a computer will carry and display sixteen digits for every result—but not all of them are necessarily accurate.

The phenomenon in Example 1.5 is called **subtractive cancellation**: The subtraction of nearby numbers to get a much smaller result is an ill conditioned problem.[3] Broadly speaking, whenever large inputs or intermediate quantities produce a much smaller result, you should investigate for possible cancellation error.

The connections between condition numbers and derivatives can be a useful guide when the input data has multiple variables.

Example 1.6. Consider the problem of finding the roots of a quadratic polynomial; i. e., the values of x for which $ax^2 + bx + c = 0$. Here the data are the coefficients a, b, and c that define the polynomial, and the solution to the problem is the root x. Using implicit partial differentiation with respect to each coefficient, we find

$$\begin{aligned} x^2 + 2ax\frac{\partial x}{\partial a} + b\frac{\partial x}{\partial a} = 0 \quad &\Rightarrow \quad \frac{\partial x}{\partial a} = -\frac{x^2}{2ax+b} \\ 2ax\frac{\partial x}{\partial b} + x + b\frac{\partial x}{\partial b} = 0 \quad &\Rightarrow \quad \frac{\partial x}{\partial b} = -\frac{x}{2ax+b} \\ 2ax\frac{\partial x}{\partial c} + b\frac{\partial x}{\partial c} + 1 = 0 \quad &\Rightarrow \quad \frac{\partial x}{\partial c} = -\frac{1}{2ax+b}. \end{aligned}$$

Multivariate linearization tells us that if all perturbations are infinitesimal, the total change in the root x is well approximated by the differential

$$dx = \frac{\partial x}{\partial a}da + \frac{\partial x}{\partial b}db + \frac{\partial x}{\partial c}dc = -\frac{x^2\,da + x\,db + dc}{2ax+b}.$$

Since this only applies when x is a root of $ax^2 + bx + c$, we can use the quadratic formula to remove x from the denominator:

$$|dx| = \left|\frac{x^2\,da + x\,db + dc}{(b^2 - 4ac)^{1/2}}\right|. \tag{1.24}$$

By choosing a norm for triples (a, b, c), we can turn this into a formula for the condition number. But already we can see one implication: If the discriminant $b^2 - 4ac$ is close to zero, the roots can be highly sensitive to perturbations in the coefficients. If the discriminant equals zero, the original polynomial has a double root, and changes to the roots are arbitrarily large relative to the coefficient perturbations. The condition number is then effectively infinite, and finding such roots can be called an **ill-posed problem**.

Multidimensional perturbations have a direction as well as magnitude, so a reasonable general definition of condition number has to include a maximization over all perturbation directions as well. We do not give details here.

[3] Here "small" refers to small absolute value or, more generally, norm.

Large condition numbers explain why forward errors cannot, in general, be expected to remain comparable in size to roundoff errors. Instead, an algorithm that always produces small *backward* errors is called **backward stable**. In the words of L. N. Trefethen, a backward stable algorithm gets "the right answer to nearly the right question." Subtraction of floating point numbers can be proved to be backward stable, even though cancellation can produce large forward errors. Even backward stability can be difficult to achieve, and many algorithms meet a weaker criterion known as **stability**. Rather than attempting to apply a formal definition of it, we study one case in depth in the next section.

Problems

1.5.1. Find the relative condition numbers of the following problems. Then identify all the values of x, if any, where the problem becomes ill conditioned. (For instance, "x close to zero", "large $|x|$", etc.)

(a) $f(x) = \sqrt{x}$ (b) $f(x) = x/10$ (c) $f(x) = \cos(x)$

1.5.2. Referring to Example 1.6, derive an expression for the relative condition number of a root of $ax^2 + bx + c$ due to perturbations in b only.

1.5.3. Generalize Example 1.6 to rootfinding for the nth degree polynomial $p(x) = a_n x^n + \cdots + a_1 x + a_0$, showing that

$$\frac{\partial x}{\partial a_k} = -\frac{x^k}{p'(x)}.$$

1.5.4. In Example 1.6 we found that a double root has infinite condition number with respect to coefficient perturbations. For this problem you will experiment with the phenomenon numerically. In MATLAB, let `r` be an arbitrary number between zero and one, and let `p` be the quadratic polynomial $x^2 - 2rx + r^2$ with a double root at `r`. (See the online help for `roots` for how MATLAB represents polynomials.)

(a) For each value $\epsilon = 10^{-14}, 10^{-13}, \ldots, 10^{-6}$, let `q` be a polynomial whose coefficients are those of `p` perturbed by $\pm\epsilon$ (with random sign choices). Find the roots x_1 and x_2 of `q` using `roots`, and let $d(\epsilon) = \max\{|x_1 - r|, |x_2 - r|\}$.

(b) Make a log-log plot of d versus ϵ. (See help on `loglog`.) You should see more or less a straight line.

(c) Explain why part (b) implies that $d(\epsilon) = c\epsilon^\alpha$ for constants c and α. By finding the slope of the line or by other means, determine a guess for α. You should find $0 < \alpha < 1$.

(d) Explain why the result of (c) is consistent with an infinite condition number. (For this problem the distinction between absolute and relative conditioning is irrelevant; use whichever one suits your argument.)

1.6 Stability: A case study

Whereas conditioning is a feature of a mathematical problem, and hence beyond the control of the numerical analyst, **stability** is the corresponding feature of algorithms—and it is one of the numerical analyst's primary responsibilities. No one formal definition of stability covers all types of problems. Here we try to illustrate the process of detection and correction of instability for the model problem of finding the roots of a quadratic polynomial $ax^2 + bx + c$. The conditioning

of this problem was analyzed in Example 1.6. Because we want a truly quadratic polynomial, we require $a \neq 0$. For simplicity we assume that the coefficients are real numbers, though using complex numbers does not change anything significant.

In grade school you learned

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}, \qquad x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}. \tag{1.25}$$

An obvious algorithm is to apply these two formulas.

Example 1.7. Let's apply this formula to the polynomial

$$p(x) = (x - 10^6)(x - 10^{-6}) = x^2 - (10^6 + 10^{-6})x + 1$$

in MATLAB.

```
>> format long
>> a = 1;  b = -(1e6+1e-6);  c = 1;
>> x1 = (-b + sqrt(b^2-4*a*c)) / (2*a)

x1 =

     1000000

>> x2 = (-b - sqrt(b^2-4*a*c)) / (2*a)

x2 =

     1.000007614493370e-06
```

(Some of the digits of `x2` may be different on your machine.[4]) The first value is dead on, but the second has lost accuracy in about 10 decimal digits.

The two roots are not remotely close to being a double root, and a formal application of (1.24) confirms that both are very well conditioned (see problem 2). In fact, the beloved quadratic formula is an *unstable* means of computing roots when errors are a possibility. The source of the difficulty can be made clear. Notice that

$$b^2 - 4ac \approx 10^{12} - 4 \approx 10^{12} \approx b^2,$$

so that

$$-b - \sqrt{b^2 - 4ac} \approx -b - |b| \approx 10^6 - 10^6.$$

In our example, the quadratic formula relies on subtractive cancellation! To produce a better algorithm, we must avoid this step.

[4] That's the whole point!

A subtle but important point is in order. The problem of subtracting two nearby numbers is, as was shown in Example 1.5, ill conditioned. There is nothing that can be done about it—if the numbers must be subtracted, loss of accuracy must result. However, for finding roots the subtraction of nearby numbers turns out to be avoidable. An algorithm which includes an unnecessary, ill conditioned step like subtractive cancellation will almost always be unstable.

To fix the instability, we observe that one of the roots in (1.25) will always use the addition of two positive or two negative terms in the numerator, which is numerically acceptable. The "good" root depends on b in a way that can be summarized as

$$x_j = \frac{-b - (\operatorname{sign} b)\sqrt{b^2 - 4ac}}{2a}. \tag{1.26}$$

(This also works for complex roots, and when $b = 0$ either choice of sign is fine.) A little algebra using (1.25) confirms the relationship $x_1 x_2 = c/a$. So given x_j from (1.26), we compute the other root x_{3-j} using $c/(ax_j)$, which creates no numerical problems. In fact the new method can be shown to be stable in general.

Example 1.8. Revisiting the polynomial of Example 1.7, we find

```
>> a = 1;  b = -(1e6+1e-6);  c = 1;
>> x1 = (-b - sign(b)*sqrt(b^2-4*a*c)) / (2*a)

x1 =

     1000000

>> x2 = c/(a*x1)

x2 =

     1.000000000000000e-06
```

We will revisit stability often throughout the book, in the context of different problem types. For now we can only give some generic advice.

- An algorithm should be screened for instability at least on test cases. One approach is to apply it to inputs for which the solution is known exactly. Another is to manually make relatively small input changes and note the results. Keep in mind that bad behavior may be confined to a subset of potential inputs.
- If great sensitivity is found, the next step is to ascertain whether it is due to ill conditioning.
- A closer analysis of the algorithm (perhaps assisted by debugging software) may reveal the source of instability and suggest a resolution. Sometimes it is the result of an avoidable ill conditioned step, such as subtractive cancellation.

- Accuracy (in the sense of reducing truncation error per unit of work) and stability often seem to be in opposition to one another.

In iterative algorithms, instability is often manifested as exponential growth in error and hence in the output. Ironically, this strong sort of instability is usually preferable: being quite unmistakable, it is easier to spot and correct than more gentle manifestations.

Problems

1.6.1. Explain quantitatively (using a condition number) why ten digits of accuracy were lost in Example 1.7.

1.6.2. The function

$$x = \cosh(t) = \frac{e^t + e^{-t}}{2}$$

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

$$t = \log\left(x - \sqrt{x^2 - 1}\right). \tag{1.27}$$

In MATLAB, let `t=-4:-4:-16` and `x=cosh(t)`.

(a) Find the condition number of the problem $f(x) = \operatorname{acosh}(x)$. (You can use (1.27)), or look up a formula for f' in a calculus book.) Evaluate at the entries of `x` in MATLAB. Would you consider the problem well conditioned at these inputs?

(b) Use (1.27) on `x` to approximate `t`. Record the accuracy of the answers, and explain. (Warning: You should use `format long` to get the true picture.)

(c) An alternate formula is

$$\operatorname{acosh}(x) = -2\log\left(\sqrt{\frac{x+1}{2}} + \sqrt{\frac{x-1}{2}}\right). \tag{1.28}$$

Apply (1.28) to `x` as before, and comment on the accuracy.

(d) Based on your experiments, which of the formulas (1.27) and (1.28) is unstable? What is the problem with that formula?[5]

1.6.3. (a) Find the condition number for the problem of computing $f(x) = (1 - \cos x)/\sin x$.

(b) Explain why computing f by the formula in (a) is unstable for $x \approx 0$.

(c) Using a trigonometric identity, find an alternate expression for f that is stable for $x \approx 0$. Test your formula in MATLAB and show that it is superior to the original at some inputs.

1.6.4. (a) Find the condition number for the problem of computing $\sinh(x)$. For what values of x is it large?

(b) Use the formula $\sinh(x) = \frac{1}{2}(e^x - e^{-x})$ to compute sinh in matlab at $x = 10^0, 10^{-2}, 10^{-4}, \ldots, 10^{-12}$. Compute the relative errors in the results using the built-in `sinh` function.

(c) Now approximate $\sinh(x)$ by the first four nonzero terms of its Taylor series about zero. Evaluate at the same values of x as in (b), and compute relative errors.

[5] According to a Mathworks newsletter, for a long time MATLAB used the *unstable* formula.

(d) What is responsible for the different behavior in (b) and (c)? Comment on the stability of the formulas near $x = 0$.

1.6.5. (Continuation of problem problem 1.3.3.) One problem with the formula (1.13) for sample variance is that one computes a sum for $\overline{x}$, then another sum to find s^2. Some statistics textbooks quote a "one-pass" formula,

$$s^2 = \frac{1}{n-1}\left(u - \tfrac{1}{n}v^2\right)$$
$$u = \sum_{i=1}^{n} x_i^2$$
$$v = \sum_{i=1}^{n} x_i.$$

"One-pass" means that both u and v can be computed in a single loop.[6] Try this formula for the two datasets

```
x = [ 1e6, 1+1e6, 2+1e6],     x = [ 1e9, 1+1e9, 2+1e9],
```

compare to using `var` in each case, and explain the results.

[6] Loops can be avoided altogether in MATLAB by using the `sum` command.