1

Introduction

1.1. Basic Notation

In this course we will refer frequently to *matrices*, *vectors*, and *scalars*. A matrix will be denoted by an upper case letter such as A, and its (i, j)th element will be denoted by a_{ij}. If the matrix is given by an expression such as $A + B$, we will write $(A + B)_{ij}$. In detailed algorithmic descriptions we will sometimes write $A(i, j)$ or use the Matlab [182] notation $A(i : j, k : l)$ to denote the submatrix of A lying in rows i through j and columns k through l. A lower-case letter like x will denote a vector, and its ith element will be written x_i. Vectors will almost always be column vectors, which are the same as matrices with one column. Lower-case Greek letters (and occasionally lower-case letters) will denote scalars. $\mathbb{R}$ will denote the set of real numbers; $\mathbb{R}^n$, the set of n-dimensional real vectors; and $\mathbb{R}^{m\times n}$, the set of m-by-n real matrices. $\mathbb{C}$, $\mathbb{C}^n$, and $\mathbb{C}^{m\times n}$ denote complex numbers, vectors, and matrices, respectively. Occasionally we will use the shorthand $A^{m\times n}$ to indicate that A is an m-by-n matrix. A^T will denote the *transpose* of the matrix A: $(A^T)_{ij} = a_{ji}$. For complex matrices we will also use the *conjugate transpose* A^*: $(A^*)_{ij} = \bar{a}_{ji}$. $\Re z$ and $\Im z$ will denote the real and imaginary parts of the complex number z, respectively. If A is m-by-n, then $|A|$ is the m-by-n matrix of absolute values of entries of A: $(|A|)_{ij} = |a_{ij}|$. Inequalities like $|A| \leq |B|$ are meant componentwise: $|a_{ij}| \leq |b_{ij}|$ for all i and j. We will also use this absolute value notation for vectors: $(|x|)_i = |x_i|$. Ends of proofs will be marked by $\square$, and ends of examples by $\diamond$. Other notation will be introduced as needed.

1.2. Standard Problems of Numerical Linear Algebra

We will consider the following standard problems:

- *Linear systems of equations*: Solve $Ax = b$. Here A is a given n-by-n nonsingular real or complex matrix, b is a given column vector with n entries, and x is a column vector with n entries that we wish to compute.

- *Least squares problems*: Compute the x that minimizes $\|Ax - b\|_2$. Here A is m-by-n, b is m-by-1, x is n-by-1, and $\|y\|_2 \equiv \sqrt{\sum |y_i|^2}$ is called the two-*norm* of the vector y. If $m > n$ so that we have more equations than unknowns, the system is called *overdetermined*. In this case we cannot generally solve $Ax = b$ exactly. If $m < n$, the system is called *underdetermined*, and we will have infinitely many solutions.

- *Eigenvalue problems*: Given an n-by-n matrix A, find an n-by-1 nonzero vector x and a scalar λ so that $Ax = \lambda x$.

- **Singular value problems**: Given an m-by-n matrix A, find an n-by-1 nonzero vector x and scalar λ so that $A^T Ax = \lambda x$. We will see that this special kind of eigenvalue problem is important enough to merit separate consideration and algorithms.

We choose to emphasize these standard problems because they arise so often in engineering and scientific practice. We will illustrate them throughout the book with simple examples drawn from engineering, statistics, and other fields. There are also many variations of these standard problems that we will consider, such as generalized eigenvalue problems $Ax = \lambda Bx$ (section 4.5) and "rank-deficient" least squares problems $\min_x \|Ax - b\|_2$, whose solutions are nonunique because the columns of A are linearly dependent (section 3.5).

We will learn the importance of exploiting any *special structure* our problem may have. For example, solving an n-by-n linear system costs $2/3n^3$ floating point operations if we use the most general form of Gaussian elimination. If we add the information that the system is symmetric and positive definite, we can save half the work by using another algorithm called Cholesky. If we further know the matrix is *banded* with *semibandwidth* $\sqrt{n}$ (i.e., $a_{ij} = 0$ if $|i-j| > \sqrt{n}$), then we can reduce the cost further to $O(n^2)$ by using band Cholesky. If we say quite explicitly that we are trying to solve Poisson's equation on a square using a 5-point difference approximation, which determines the matrix nearly uniquely, then by using the multigrid algorithm we can reduce the cost to $O(n)$, which is nearly as fast as possible, in the sense that we use just a constant amount of work per solution component (section 6.4).

1.3. General Techniques

There are several general concepts and techniques that we will use repeatedly:

1. matrix factorizations;

2. perturbation theory and condition numbers;

3. effects of roundoff error on algorithms, including properties of floating point arithmetic;

4. analysis of the speed of an algorithm;
5. engineering numerical software.

We discuss each of these briefly below.

1.3.1. Matrix Factorizations

A *factorization* of the matrix A is a representation of A as a product of several "simpler" matrices, which make the problem at hand easier to solve. We give two examples.

EXAMPLE 1.1. Suppose that we want to solve $Ax = b$. If A is a lower triangular matrix,

$$\begin{bmatrix} a_{11} & & & \\ a_{21} & a_{22} & & \\ \vdots & \vdots & \ddots & \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

is easy to solve using *forward substitution*:

for $i = 1$ to n
  $x_i = (b_i - \sum_{k=1}^{i-1} a_{ik}x_k)/a_{ii}$
end for

An analogous idea, *back substitution*, works if A is upper triangular. To use this to solve a general system $Ax = b$ we need the following matrix factorization, which is just a restatement of Gaussian elimination.

THEOREM 1.1. *If the n-by-n matrix A is nonsingular, there exists a permutation matrix P (the identity matrix with its rows permuted), a nonsingular lower triangular matrix L, and a nonsingular upper triangular matrix U such that $A = P \cdot L \cdot U$. To solve $Ax = b$, we solve the equivalent system $PLUx = b$ as follows:*

$$\begin{aligned} LUx &= P^{-1}b = P^Tb && \textit{(permute entries of } b\textit{)}, \\ Ux &= L^{-1}(P^Tb) && \textit{(forward substitution)}, \\ x &= U^{-1}(L^{-1}P^Tb) && \textit{(back substitution)}. \end{aligned}$$

We will prove this theorem in section 2.3. ◇

EXAMPLE 1.2. The *Jordan canonical factorization* $A = VJV^{-1}$ exhibits the eigenvalues and eigenvectors of A. Here V is a nonsingular matrix, whose columns include the eigenvectors, and J is the *Jordan canonical form* of A, a special triangular matrix with the eigenvalues of A on its diagonal. We will learn that it is numerically superior to compute the *Schur factorization* $A = UTU^*$, where U is a unitary matrix (i.e., U's columns are orthonormal), and T is upper triangular with A's eigenvalues on its diagonal. The Schur form T can be computed faster and more accurately than the Jordan form J. We discuss the Jordan and Schur factorizations in section 4.2. ◇

1.3.2. Perturbation Theory and Condition Numbers

The answers produced by numerical algorithms are seldom exactly correct. There are two sources of error. First, there may be errors in the input data to the algorithm, caused by prior calculations or perhaps measurement errors. Second, there are errors caused by the algorithm itself, due to approximations made within the algorithm. In order to estimate the errors in the computed answers from both these sources, we need to understand how much the solution of a problem is changed (or *perturbed*) if the input data is slightly perturbed.

EXAMPLE 1.3. Let $f(x)$ be a real-valued continuous function of a real variable x. We want to compute $f(x)$, but we do not know x exactly. Suppose instead that we are given $x+\delta x$ and a bound on δx. The best that we can do (without more information) is to compute $f(x+\delta x)$ and to try to bound the absolute error $|f(x+\delta x)-f(x)|$. We may use a simple linear approximation to f to get the error bound $f(x+\delta x) \approx f(x)+\delta x f'(x)$, and so the error is $|f(x+\delta x)-f(x)| \approx |\delta x| \cdot |f'(x)|$. We call $|f'(x)|$ the *absolute condition number* of f at x. If $|f'(x)|$ is large enough, then the error may be large even if δx is small; in this case we call f *ill-conditioned* at x. $\diamond$

We say *absolute* condition number because it provides a bound on the absolute error $|f(x+\delta x)-f(x)|$ given a bound on the absolute change $|\delta x|$ in the input. We will also often use the following essentially equivalent expression to bound the error:

$$\frac{|f(x+\delta x)-f(x)|}{|f(x)|} \approx \frac{|\delta x|}{|x|} \cdot \frac{|f'(x)| \cdot |x|}{|f(x)|}.$$

This expression bounds the *relative error* $|f(x+\delta x)-f(x)|/|f(x)|$ as a multiple of the *relative* change $|\delta x|/|x|$ in the input. The multiplier, $|f'(x)| \cdot |x|/|f(x)|$, is called the *relative condition number*, or often just *condition number* for short.

The condition number is all that we need to understand how error in the input data affects the computed answer: we simply multiply the condition number by a bound on the input error to bound the error in the computed solution.

For each problem we consider, we will derive its corresponding condition number.

1.3.3. Effects of Roundoff Error on Algorithms

To continue our analysis of the error caused by the algorithm itself, we need to study the effect of roundoff error in the arithmetic, or simply roundoff for short. We will do so by using a property possessed by most good algorithms: *backward stability*. We define it as follows.

> If alg(x) is our algorithm for $f(x)$, including the effects of roundoff, we call alg(x) a *backward stable algorithm* for $f(x)$ if for all x there

> is a "small" δx such that $\text{alg}(x) = f(x + \delta x)$. δx is called the *backward error*. Informally, we say that we get the exact answer ($f(x + \delta x)$) for a slightly wrong problem ($x + \delta x$).

This implies that we may bound the error as

$$\text{error} = |\text{alg}(x) - f(x)| = |f(x + \delta x) - f(x)| \approx |f'(x)| \cdot |\delta x|,$$

the product of the absolute condition number $|f'(x)|$ and the magnitude of the backward error $|\delta x|$. Thus, if $\text{alg}(\cdot)$ is backward stable, $|\delta x|$ is always small, so the error will be small unless the absolute condition number is large. Thus, backward stability is a desirable property for an algorithm, and most of the algorithms that we present will be backward stable. Combined with the corresponding condition numbers, we will have error bounds for all our computed solutions.

Proving that an algorithm is backward stable requires knowledge of the roundoff error of the basic floating point operations of the machine and how these errors propagate through an algorithm. This is discussed in section 1.5.

1.3.4. Analyzing the Speed of Algorithms

In choosing an algorithm to solve a problem, one must of course consider its speed (which is also called performance) as well as its backward stability. There are several ways to estimate speed. Given a particular problem instance, a particular implementation of an algorithm, and a particular computer, one can of course simply run the algorithm and see how long it takes. This may be difficult or time consuming, so we often want simpler estimates. Indeed, we typically want to estimate how long a particular algorithm would take *before* implementing it.

The traditional way to estimate the time an algorithm takes is to count the *flops*, or *floating point operations*, that it performs. We will do this for all the algorithms we present. However, this is often a misleading time estimate on modern computer architectures, because it can take significantly more time to move the data inside the computer to the place where it is to be multiplied, say, than it does to actually perform the multiplication. This is especially true on parallel computers but also is true on conventional machines such as workstations and PCs. For example, matrix multiplication on the IBM RS6000/590 workstation can be sped up from 65 Mflops (millions of floating point operations per second) to 240 Mflops, nearly four times faster, by judiciously reordering the operations of the standard algorithm (and using the correct compiler optimizations). We discuss this further in section 2.6.

If an algorithm is *iterative*, i.e., produces a series of approximations converging to the answer rather than stopping after a fixed number of steps, then we must ask how many steps are needed to decrease the error to a tolerable level. To do this, we need to decide if the convergence is *linear* (i.e.,

the error decreases by a constant factor $0 < c < 1$ at each step so that $|\text{error}_i| \leq c \cdot |\text{error}_{i-1}|$ or faster, such as *quadratic* ($|\text{error}_i| \leq c \cdot |\text{error}_{i-1}|^2$). If two algorithms are both linear, we can ask which has the smaller constant c. Iterative linear equation solvers and their convergence analysis are the subject of Chapter 6.

1.3.5. Engineering Numerical Software

Three main issues in designing or choosing a piece of numerical software are *ease of use*, *reliability*, and *speed*. Most of the algorithms covered in this course have already been carefully programmed with these three issues in mind. If some of this existing software can solve your problem, its ease of use may well outweigh any other considerations such as speed. Indeed, if you need only to solve your problem once or a few times, it is often easier to use general purpose software written by experts than to write your own more specialized program.

There are three programming paradigms for exploiting other experts' software. The first paradigm is the traditional software library, consisting of a collection of subroutines for solving a fixed set of problems, such as solving linear systems, finding eigenvalues, and so on. In particular, we will discuss the LAPACK library [10], a state-of-the-art collection of routines available in Fortran and C. This library, and many others like it, are freely available in the public domain; see NETLIB on the World Wide Web.[1] LAPACK provides reliability and high speed (for example, making careful use of matrix multiplication, as described above) but requires careful attention to data structures and calling sequences on the part of the user. We will provide pointers to such software throughout the text.

The second programming paradigm provides a much easier-to-use environment than libraries like LAPACK, but at the cost of some performance. This paradigm is provided by the commercial system Matlab [182], among others. Matlab provides a simple interactive programming environment where all variables represent matrices (scalars are just 1-by-1 matrices), and most linear algebra operations are available as built-in functions. For example, "$C = A * B$" stores the product of matrices A and B in C, and "$A = \text{inv}(B)$" stores the inverse of matrix B in A. It is easy to quickly prototype algorithms in Matlab and to see how they work. But since Matlab makes a number of algorithmic decisions automatically for the user, it may perform more slowly than a carefully chosen library routine.

The third programming paradigm is that of *templates*, or recipes for assembling complicated algorithms out of simpler building blocks. Templates are useful when there are a large number of ways to construct an algorithm but no simple rule for choosing the best construction for a particular input problem; therefore, much of the construction must be left to the user. An example of this may be found in *Templates for the Solution of Linear Systems: Building*

[1] Recall that we abbreviate the URL prefix http://www.netlib.org to NETLIB in the text.

Blocks for Iterative Methods [24]; a similar set of templates for eigenproblems is currently under construction.

1.4. Example: Polynomial Evaluation

We illustrate the ideas of perturbation theory, condition numbers, backward stability, and roundoff error analysis with the example of *polynomial evaluation*:

$$p(x) = \sum_{i=0}^{d} a_i x^i.$$

Horner's rule for polynomial evaluation is

$p = a_d$
for $i = d - 1$ down to 0
    $p = x * p + a_i$
end for

Let us apply this to $p(x) = (x-2)^9 = x^9 - 18x^8 + 144x^7 - 672x^6 + 2016x^5 - 4032x^4 + 5376x^3 - 4608x^2 + 2304x - 512$. In the bottom of Figure 1.1, we see that near the zero $x = 2$ the value of $p(x)$ computed by Horner's rule is quite unpredictable and may justifiably be called "noise." The top of Figure 1.1 shows an accurate plot.

To understand the implications of this figure, let us see what would happen if we tried to find a zero of $p(x)$ using a simple zero finder based on Bisection, shown below in Algorithm 1.1.

Bisection starts with an interval $[x_{low}, x_{high}]$ in which $p(x)$ changes sign ($p(x_{low}) \cdot p(x_{high}) < 0$) so that $p(x)$ must have a zero in the interval. Then the algorithm computes $p(x_{mid})$ at the interval midpoint $x_{mid} = (x_{low} + x_{high})/2$ and asks whether $p(x)$ changes sign in the bottom half interval $[x_{low}, x_{mid}]$ or top half interval $[x_{mid}, x_{high}]$. Either way, we find an interval of half the original length containing a zero of $p(x)$. We can continue bisecting until the interval is as short as desired.

So the decision between choosing the top half interval or bottom half interval depends on the sign of $p(x_{mid})$. Examining the graph of $p(x)$ in the bottom half of Figure 1.1, we see that this sign varies rapidly from plus to minus as x varies. So changing x_{low} or x_{high} just slightly could completely change the sequence of sign decisions and also the final interval. Indeed, depending on the initial choices of x_{low} and x_{high}, the algorithm could converge *anywhere* inside the "noisy region" from 1.95 to 2.05 (see Question 1.21).

To explain this fully, we return to properties of floating point arithmetic.

ALGORITHM 1.1. *Finding zeros of $p(x)$ using Bisection.*

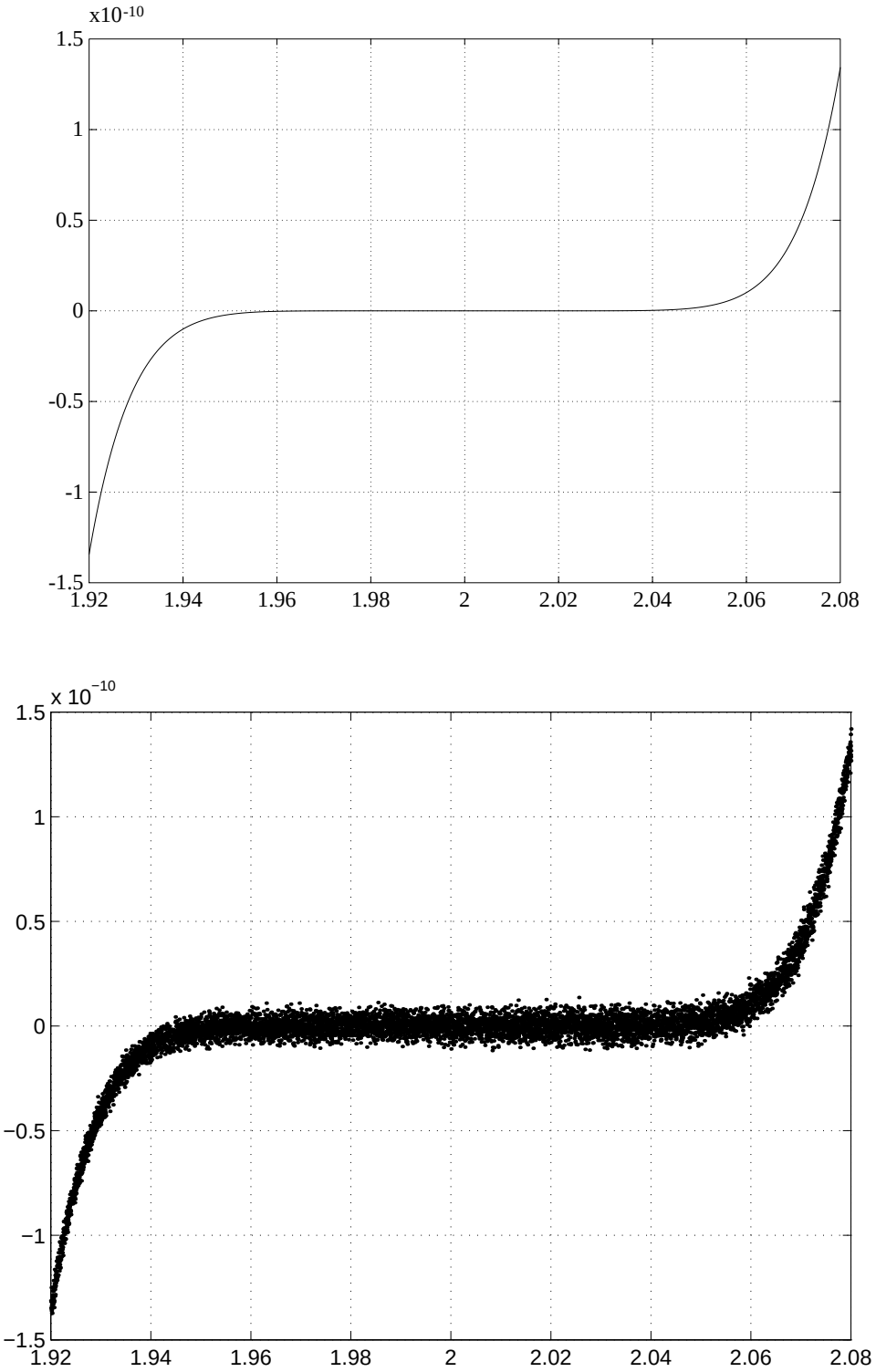
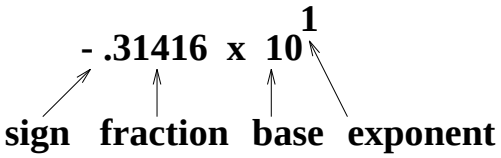


Fig. 1.1. *Plot of* $y = (x-2)^9 = x^9 - 18x^8 + 144x^7 - 672x^6 + 2016x^5 - 4032x^4 + 5376x^3 - 4608x^2 + 2304x - 512$ *evaluated at* 8000 *equispaced points, using* $y = (x-2)^9$ *(top) and using Horner's rule (bottom).*

```
proc bisect (p, x_low, x_high, tol)
/* find a root of p(x) = 0 in [x_low, x_high]
     assuming p(x_low) · p(x_high) < 0 */
/* stop if zero found to within ±tol */
p_low = p(x_low)
p_high = p(x_high)
while x_high − x_low > 2 · tol
     x_mid = (x_low + x_high)/2
     p_mid = p(x_mid)
     if p_low · p_mid < 0 then /* there is a root in [x_low, x_mid] */
          x_high = x_mid
          p_high = p_mid
     else if p_mid · p_high < 0 then /* there is a root in [x_mid, x_high] */
          x_low = x_mid
          p_low = p_mid
     else /* x_mid is a root */
          x_low = x_mid
          x_high = x_mid
     end if
end while
root = (x_low + x_high)/2
```

1.5. Floating Point Arithmetic

The number -3.1416 may be expressed in *scientific notation* as follows:



Computers use a similar representation called *floating point*, but generally the base is 2 (with exceptions, such as 16 for IBM 370 and 10 for some spreadsheets and most calculators). For example, $.10101_2 \times 2^3 = 5.25_{10}$.

A floating point number is called *normalized* if the leading digit of the fraction is nonzero. For example, $.10101_2 \times 2^3$ is normalized, but $.010101_2 \times 2^4$ is not. Floating point numbers are usually normalized, which has two advantages: each nonzero floating point value has a unique representation as a bit string, and in binary the leading 1 in the fraction need not be stored explicitly (because it is always 1), leaving one extra bit for a longer, more accurate fraction.

The most important parameters describing floating point numbers are the base; the number of digits (bits) in the fraction, which determines the precision; and the number of digits (bits) in the exponent, which determines the exponent range and thus the largest and smallest representable numbers. Different

floating point arithmetics also differ in how they round computed results, what they do about numbers that are too near zero (underflow) or too big (overflow), whether $\pm\infty$ is allowed, and whether useful nonnumbers are provided (sometimes called NaNs, indefinites, or reserved operands) are provided. We discuss each of these below.

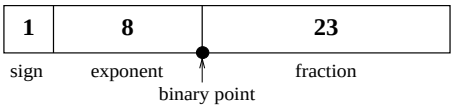
First we consider the precision with which numbers can be represented. For example, $.31416 \times 10^1$ has five decimal digits, so any information less than $.5 \times 10^{-4}$ may have been lost. This means that if x is a real number whose best five-digit approximation is $.31416 \times 10^1$, then the *relative representation error* in $.31416 \times 10^1$ is

$$\frac{|x - .31416 \times 10^1|}{.31416 \times 10^1} \leq \frac{.5 \times 10^{-4}}{.31416 \times 10^1} \approx .16 \times 10^{-4}.$$

The maximum relative representation error in a normalized number occurs for $.10000 \times 10^1$, which is the most accurate five-digit approximation of all numbers in the interval from .999995 to 1.00005. Its relative error is therefore bounded by $.5 \cdot 10^{-4}$. More generally, the *maximum relative representation error* in a floating point arithmetic with p digits and base β is $.5 \times \beta^{1-p}$. This is also half the distance between 1 and the next larger floating point number, $1 + \beta^{1-p}$.

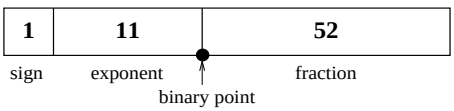
Computers have historically used many different choices of base, number of digits, and range, but fortunately the *IEEE standard for binary arithmetic* is now most common. It is used on SUN, DEC, HP, and IBM workstations and all PCs. IEEE arithmetic includes two kinds of floating point numbers: *single precision* (32 bits long) and *double precision* (64 bits long).

IEEE single precision



If s, e, and $f < 1$ are the 1-bit sign, 8-bit exponent, and 23-bit fraction in the IEEE single precision format, respectively, then the number represented is $(-1)^s \cdot 2^{e-127} \cdot (1 + f)$. The maximum relative representation error is $2^{-24} \approx 6 \cdot 10^{-8}$, and the range of positive normalized numbers is from 2^{-126} (the *underflow threshold*) to $2^{127} \cdot (2 - 2^{-23}) \approx 2^{128}$ (the *overflow threshold*), or about 10^{-38} to 10^{38}. The positions of these floating point numbers on the real number line are shown in Figure 1.2 (where we use a 3-bit fraction for ease of presentation).

IEEE double precision



If s, e, and $f < 1$ are the 1-bit sign, 11-bit exponent, and 52-bit fraction in IEEE double precision format, respectively, then the number represented is $(-1)^s \cdot 2^{e-1023} \cdot (1 + f)$. The maximum relative representation error is $2^{-53} \approx 10^{-16}$, and the exponent range is 2^{-1022} (the *underflow threshold*) to $2^{1023} \cdot$

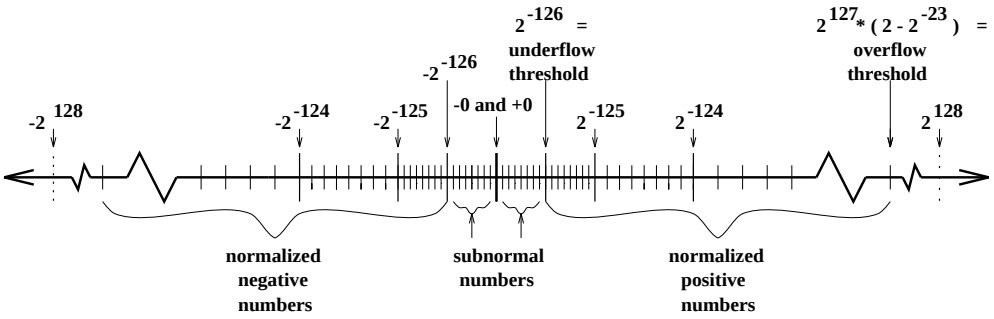


Fig. 1.2. *Real number line with floating point numbers indicated by solid tick marks. The range shown is correct for IEEE single precision, but a* 3*-bit fraction is assumed for ease of presentation so that there are only* $2^3 - 1 = 7$ *floating point numbers between consecutive powers of* 2*, not* $2^{23} - 1$*. The distance between consecutive tick marks is constant between powers of* 2 *and doubles/halves across powers of* 2 *(among the normalized floating point numbers).* $+2^{128}$ *and* -2^{128}*, which are one unit in the last place larger in magnitude than the overflow threshold (the largest finite floating point number,* $2^{127} \cdot (2 - 2^{-23})$*), are shown as dotted tick marks. The figure is symmetric about* 0*;* $+0$ *and* -0 *are distinct floating point bit strings but compare as numerically equal. Division by zero is the only binary operation that gives different results,* $+\infty$ *and* $-\infty$*, for different signed zero arguments.*

$(2 - 2^{-52}) \approx 2^{1024}$ (the *overflow threshold*), or about 10^{-308} to 10^{308}.

When the true value of a computation $a \odot b$ (where $\odot$ is one of the four binary operations $+$, $-$, $*$, and $/$) cannot be represented exactly as a floating point number, it must be approximated by a nearby floating point number before it can be stored in memory or a register. We denote this approximation by $\mathrm{fl}(a \odot b)$. The difference $(a \odot b) - \mathrm{fl}(a \odot b)$ is called the *roundoff error*. If $\mathrm{fl}(a \odot b)$ is a nearest floating point number to $a \odot b$, we say that the arithmetic *rounds correctly* (or just *rounds*). IEEE arithmetic has this attractive property. (IEEE arithmetic breaks ties, when $a \odot b$ is exactly halfway between two adjacent floating point numbers, by choosing $\mathrm{fl}(a \odot b)$ to have its least significant bit zero; this is called *rounding to nearest even*.) When rounding correctly, if $a \odot b$ is within the exponent range (otherwise we get *overflow* or *underflow*), then we can write

$$\mathrm{fl}(a \odot b) = (a \odot b)(1 + \delta), \tag{1.1}$$

where $|\delta|$ is bounded by ε, which is called variously *machine epsilon*, *machine precision*, or *macheps*. Since we are rounding as accurately as possible, ε is equal to the maximum relative representation error $.5 \cdot \beta^{1-p}$. IEEE arithmetic also guarantees that $\mathrm{fl}(\sqrt{a}) = \sqrt{a}(1 + \delta)$, with $|\delta| \leq \varepsilon$. This is the most common model for roundoff error analysis and the one we will use in this book. A nearly identical formula applies to complex floating point arithmetic; see Question 1.12. However, formula (1.1) does ignore some interesting details.

IEEE arithmetic also includes *subnormal numbers*, i.e., unnormalized float-

ing point numbers with the minimum possible exponent. These represent tiny numbers between zero and the smallest normalized floating point number; see Figure 1.2. Their presence means that a difference $\mathrm{fl}(x-y)$ can never be zero because of underflow, yielding the attractive property that the predicate $x = y$ is true if and only if $\mathrm{fl}(x-y) = 0$. To incorporate errors caused by underflow into formula (1.1) one would change it to

$$\mathrm{fl}(a \odot b) = (a \odot b)(1+\delta) + \eta,$$

where $|\delta| \leq \varepsilon$ as before, and $|\eta|$ is bounded by a tiny number equal to the largest error caused by underflow ($2^{-150} \approx 10^{-45}$ in IEEE single precision and $2^{-1075} \approx 10^{-324}$ in IEEE double precision).

IEEE arithmetic includes the symbols $\pm\infty$ and NaN (*Not a Number*).$\pm\infty$ is returned when an operation overflows, and behaves according to the following arithmetic rules: $x/\pm\infty = 0$ for any finite floating point number x, $x/0 = \pm\infty$ for any nonzero floating point number x, $+\infty + \infty = +\infty$, etc. An NaN is returned by any operation with no well-defined finite or infinite result, such as $\infty - \infty$, $\frac{\infty}{\infty}$, $\frac{0}{0}$, $\sqrt{-1}$, NaN $\odot\, x$, etc.

Whenever an arithmetic operation is invalid and so produces an NaN, or overflows or divides by zero to produce $\pm\infty$, or underflows, an *exception flag* is set and can later be tested by the user's program. These features permit one to write both more reliable programs (because the program can detect and correct its own exceptions, instead of simply aborting execution) and faster programs (by avoiding "paranoid" programming with many tests and branches to avoid possible but unlikely exceptions). For examples, see Question 1.19, the comments following Lemma 5.3, and [80].

The most expensive error known to have been caused by an improperly handled floating point exception is the crash of the Ariane 5 rocket of the European Space Agency on June 4, 1996. See HOME/ariane5rep.html for details.

Not all machines use IEEE arithmetic or round carefully, although nearly all do. The most important modern exceptions are those machines produced by Cray Research,[2] although future generations of Cray machines may use IEEE arithmetic.[3] Since the difference between $\mathrm{fl}(a \odot b)$ computed on a Cray and $\mathrm{fl}(a \odot b)$ computed on an IEEE machine usually lies in the 14th decimal place or beyond, the reader may wonder whether the difference is important. Indeed, most algorithms in numerical linear algebra are insensitive to details in the way roundoff is handled. But it turns out that some algorithms are easier to design, or more reliable, when rounding is done properly. Here are two examples.

[2]We include machines such as the NEC SX-4, which has a "Cray mode" in which it performs arithmetic the same way. We exclude the Cray T3D and T3E, which are parallel computers built from DEC Alpha processors, which use IEEE arithmetic very nearly (underflows are flushed to zero for speed's sake).

[3]Cray Research was purchased by Silicon Graphics in 1996.

When the Cray C90 subtracts 1 from the next smaller floating point number, it gets -2^{-47}, which is twice the correct answer, -2^{-48}. Getting even tiny differences to high relative accuracy is essential for the correctness of the divide-and-conquer algorithm for finding eigenvalues and eigenvectors of symmetric matrices, currently the fastest algorithm available for the problem. This algorithm requires a rather nonintuitive modification to guarantee correctness on Cray machines (see section 5.3.3).

The Cray may also yield an error when computing $\arccos(x/\sqrt{x^2+y^2})$ because excessive roundoff causes the argument of arccos to be larger than 1. This cannot happen in IEEE arithmetic (see Question 1.17).

To accommodate error analysis on a Cray C90 or other Cray machines we may instead use the model $\mathrm{fl}(a \pm b) = a(1+\delta_1) \pm b(1+\delta_2)$, $\mathrm{fl}(a*b) = (a*b)(1+\delta_3)$, and $\mathrm{fl}(a/b) = (a/b)(1+\delta_3)$, with $|\delta_i| \le \varepsilon$, where ε is a small multiple of the maximum relative representation error.

Briefly, we can say that correct rounding and other features of IEEE arithmetic are designed to preserve as many mathematical relationships used to derive formulas as possible. It is easier to design algorithms knowing that (barring over/underflow) $\mathrm{fl}(a-b)$ is computed with a small relative error (otherwise divide-and-conquer can fail), and that $-1 \le c \equiv \mathrm{fl}(x/\sqrt{x^2+y^2}) \le 1$ (otherwise $\arccos(c)$ can fail). There are many other such mathematical relationships that one relies on (often unwittingly) to design algorithms. For more details about IEEE arithmetic and its relationship to numerical analysis, see [157, 156, 80].

Given the variability in floating point across machines, how does one write portable software that depends on the arithmetic? For example, iterative algorithms that we will study in later chapters frequently have loops such as

repeat
    ...
    update e
until "e is negligible compared to f,"

where $e \ge 0$ is some error measure, and $f > 0$ is some comparison value (see section 4.4.5 for an example). By negligible we mean "is $e \le c \cdot \varepsilon \cdot f$?," where $c \ge 1$ is some modest constant, chosen to trade off accuracy and speed of convergence. Since this test requires the machine-dependent constant ε, this test has in the past often been replaced by the *apparently* machine-independent test "is $e + f = f$?" The idea here is that adding e to f and rounding will yield f again if $e < \varepsilon f$ or perhaps a little smaller. But this test can fail (by requiring e to be *much* smaller than necessary, or than attainable), depending on the machine and compiler used (see the next paragraph). So the best test indeed uses ε explicitly. It turns out that with sufficient care one can compute ε in a machine-independent way, and software for this is available in the LAPACK subroutines `slamch` (for single precision) and `dlamch` (for double precision). These routines also compute or estimate the overflow

threshold (without overflowing!), the underflow threshold, and other parameters. Another portable program that uses these explicit machine parameters is discussed in Question 1.19.

Sometimes one needs higher precision than is available from IEEE single or double precision. For example, higher precision is of use in algorithms such as iterative refinement for improving the accuracy of a computed solution of $Ax = b$ (see section 2.5.1). So IEEE defines another, higher precision called *double extended*. For example, *all* arithmetic operations on an Intel Pentium (or its predecessors going back to the Intel 8086/8087) are performed in 80-bit double extended registers, providing 64-bit fractions and 15-bit exponents. Unfortunately, not all languages and compilers permit one to declare and compute with double-extended precision variables.

Few machines offer anything beyond double-extended arithmetic in hardware, but there are several ways in which more accurate arithmetic may be simulated in software. Some compilers on DEC Vax and DEC Alpha, SUN Sparc, and IBM RS6000 machines permit the user to declare *quadruple precision* (or *real*16* or *double double precision*) variables and to perform computations with them. Since this arithmetic is simulated using shorter precision, it may run several times slower than double. Cray's single precision is similar in precision to IEEE double, and so Cray double precision is about twice IEEE double; it too is simulated in software and runs relatively slowly. There are also algorithms and packages available for simulating much higher precision floating point arithmetic, using either integer arithmetic [20, 21] or the underlying floating point (see Question 1.18) [202, 216].

Finally, we mention *interval arithmetic*, a style of computation that automatically provides guaranteed error bounds. Each variable in an interval computation is represented by a pair of floating point numbers, one a lower bound and one an upper bound. Computation proceeds by rounding in such a way that lower bounds and upper bounds are propagated in a guaranteed fashion. For example, to add the intervals $a = [a_l, a_u]$ and $b = [b_l, b_u]$, one rounds $a_l + b_l$ *down* to the nearest floating point number, c_l, and rounds $a_u + b_u$ *up* to the nearest floating point number, c_u. This guarantees that the interval $c = [c_l, c_u]$ contains the sum of any pair of variables from a and from b. Unfortunately, if one naively takes a program and converts all floating point variables and operations to interval variables and operations, it is most likely that the intervals computed by the program will quickly grow so wide (such as $[-\infty, +\infty]$) that they provide no useful information at all. (A simple example is to repeatedly compute $x = x - x$ when x is an interval; instead of getting $x = 0$, the width $x_u - x_l$ of x doubles at each subtraction.) It is possible to modify old algorithms or design new ones that do provide useful guaranteed error bounds [4, 138, 160, 188], but these are often several times as expensive as the algorithms discussed in this book. The error bounds that we present in this book are not guaranteed in the same mathematical sense that interval bounds are, but they are reliable enough in almost all situations. (We discuss

this in more detail later.) We will not discuss interval arithmetic further in this book.

1.6. Polynomial Evaluation Revisited

Let us now apply roundoff model (1.1) to evaluating a polynomial with Horner's rule. We take the original program,

$p = a_d$
for $i = d - 1$ down to 0
    $p = x \cdot p + a_i$
end for

Then we add subscripts to the intermediate results so that we have a unique symbol for each one (p_0 is the final result):

$p_d = a_d$
for $i = d - 1$ down to 0
    $p_i = x \cdot p_{i+1} + a_i$
end for

Then we insert a roundoff term $(1 + \delta_i)$ at each floating point operation to get

$p_d = a_d$
for $i = d - 1$ down to 0
    $p_i = ((x \cdot p_{i+1})(1 + \delta_i) + a_i)(1 + \delta'_i)$, where $|\delta_i|, |\delta'_i| \leq \varepsilon$
end for

Expanding, we get the following expression for the final computed value of the polynomial:

$$p_0 = \sum_{i=0}^{d-1} \left[(1 + \delta'_i) \prod_{j=0}^{i-1} (1 + \delta_j)(1 + \delta'_j) \right] a_i x^i + \left[\prod_{j=0}^{d-1} (1 + \delta_j)(1 + \delta'_j) \right] a_d x^d \, .$$

This is messy, a typical result when we try to keep track of every rounding error in an algorithm. We simplify it using the following upper and lower bounds:

$$\begin{aligned} (1 + \delta_1) \cdots (1 + \delta_j) &\leq (1 + \varepsilon)^j \leq \frac{1}{1 - j\varepsilon} = 1 + j\varepsilon + O(\varepsilon^2), \\ (1 + \delta_1) \cdots (1 + \delta_j) &\geq (1 - \varepsilon)^j \geq 1 - j\varepsilon. \end{aligned}$$

These bounds are correct, provided that $j\varepsilon < 1$. Typically, we make the reasonable assumption that $j\varepsilon \ll 1$ ($j \ll 10^7$ in IEEE single precision) and make the approximations

$$1 - j\varepsilon \leq (1 + \delta_1) \cdots (1 + \delta_j) \leq 1 + j\varepsilon.$$

This lets us write

$$\begin{aligned} p_0 &= \sum_{i=0}^{d} (1+\overline{\delta}_i) a_i x^i, \quad \text{where } |\overline{\delta}_i| \le 2d\varepsilon \\ &= \sum_{i=0}^{d} \overline{a}_i x^i \end{aligned}$$

So the computed value p_0 of $p(x)$ is the exact value of a slightly different polynomial with coefficients $\overline{a}_i$. This means that evaluating $p(x)$ is "backward stable," and the "backward error" is $2d\varepsilon$ measured as the maximum relation change of any coefficient of $p(x)$.

Using this backward error bound, we bound the error in the computed polynomial:

$$\begin{aligned} |p_0 - p(x)| &= \left| \sum_{i=0}^{d} (1+\overline{\delta_i}) a_i x^i - \sum_{i=0}^{d} a_i x^i \right| \\ &= \left| \sum_{i=0}^{d} \overline{\delta}_i a_i x^i \right| \le \sum_{i=0}^{d} \varepsilon 2d |a_i \cdot x^i| \\ &\le 2d\varepsilon \sum_{i=0}^{d} |a_i \cdot x^i|. \end{aligned}$$

Note that $\sum_i |a_i x^i|$ bounds the largest value that we could compute if there were no cancellation from adding positive and negative numbers, and the error bound is $2d\varepsilon$ times smaller. This is also the case for computing dot products and many other polynomial-like expressions.

By choosing $\overline{\delta}_i = \varepsilon \cdot \text{sign}(a_i x^i)$, we see that the error bound is attainable to within the modest factor $2d$. This means that we may use

$$\frac{\sum_{i=0}^{d} |a_i x^i|}{|\sum_{i=0}^{d} a_i x^i|}$$

as the *relative condition number* for polynomial evaluation.

We can easily compute this error bound, at the cost of doubling the number of operations:

$p = a_d$, $bp = |a_d|$
for $i = d-1$ down to 0
    $p = x \cdot p + a_i$
    $bp = |x| \cdot bp + |a_i|$
end for
error bound $= bp = 2d \cdot \varepsilon \cdot bp$

so the true value of the polynomial is in the interval $[p - bp, p + bp]$, and the number of guaranteed correct decimal digits is $-\log_{10}(|\frac{bp}{p}|)$. These bounds are

plotted in the top of Figure 1.3 for the polynomial discussed earlier, $(x-2)^9$. (The reader may wonder whether roundoff errors could make this computed error bound inaccurate. This turns out not to be a problem and is left to the reader as an exercise.)

The graph of $-\log_{10}|\frac{bp}{p}|$ in the bottom of Figure 1.3, a lower bound on the number of correct decimal digits, indicates that we expect difficulty computing $p(x)$ to high relative accuracy when $p(x)$ is near 0. What is special about $p(x) = 0$? An arbitrarily small error ε in computing $p(x) = 0$ causes an infinite relative error $\frac{\varepsilon}{p(x)} = \frac{\varepsilon}{0}$. In other words, our relative error bound $2d\varepsilon \ \sum_{i=0}^{d} |a_i x^i| / |\sum_{i=0}^{d} a_i x^i|$ is infinite.

DEFINITION 1.1. *A problem whose condition number is infinite is called* ill-posed. *Otherwise it is called* well-posed.[4]

There is a simple geometric interpretation of the condition number: it tells us how far $p(x)$ is from a polynomial which is ill-posed.

DEFINITION 1.2. *Let* $p(z) = \sum_{i=0}^{d} a_i z^i$ *and* $q(z) = \sum_{i=0}^{d} b_i z^i$. *Define the* relative distance $d(p,q)$ from p to q *as the smallest value satisfying* $|a_i - b_i| \leq d(p,q) \cdot |a_i|$ *for* $0 \leq i \leq d$. *(If all* $a_i = 0$, *then we can more simply write* $d(p,q) = \max_{0 \leq i \leq d} |\frac{a_i - b_i}{a_i}|$.)

Note that if $a_i = 0$, then b_i must also be zero for $d(p,q)$ to be finite.

THEOREM 1.2. *Suppose that* $p(z) = \sum_{i=0}^{d} a_i z^i$ *is not identically zero.*

$$\min\{d(p,q) \text{ such that } q(x) = 0\} = \frac{|\sum_{i=0}^{d} a_i x^i|}{\sum_{i=0}^{d} |a_i x^i|} .$$

In other words, the distance from p *to the nearest polynomial* q *whose condition number at* x *is infinite is the reciprocal of the condition number of* $p(x)$.

Proof. Write $q(z) = \sum b_i z^i = \sum (1+\varepsilon_i) a_i z^i$ so that $d(p,q) = \max_i |\varepsilon_i|$. Then $q(x) = 0$ implies $|p(x)| = |q(x) - p(x)| = |\sum_{i=0}^{d} \varepsilon_i a_i x^i| \leq \sum_{i=0}^{d} |\varepsilon_i a_i x^i| \leq \max_i |\varepsilon_i| \ \sum_i |a_i x^i|$, which in turn implies $d(p,q) = \max |\varepsilon_i| \geq |p(x)| / \sum_i |a_i x^i|$. To see that there is a q this close to p, choose

$$\varepsilon_i = \frac{-p(x)}{\sum |a_i x^i|} \cdot \text{sign}(a_i x^i). \quad \square$$

[4]This definition is slightly nonstandard, because ill-posed problems include those whose solutions are continuous as long as they are nondifferentiable. Examples include multiple roots of polynomials and multiple eigenvalues of matrices (section 4.3). Another way to describe an ill-posed problem is one in which the number of correct digits in the solution is not always within a constant of the number of digits used in the arithmetic in the solution. For example, multiple roots of polynomials tend to lose *half* or more of the precision of the arithmetic.

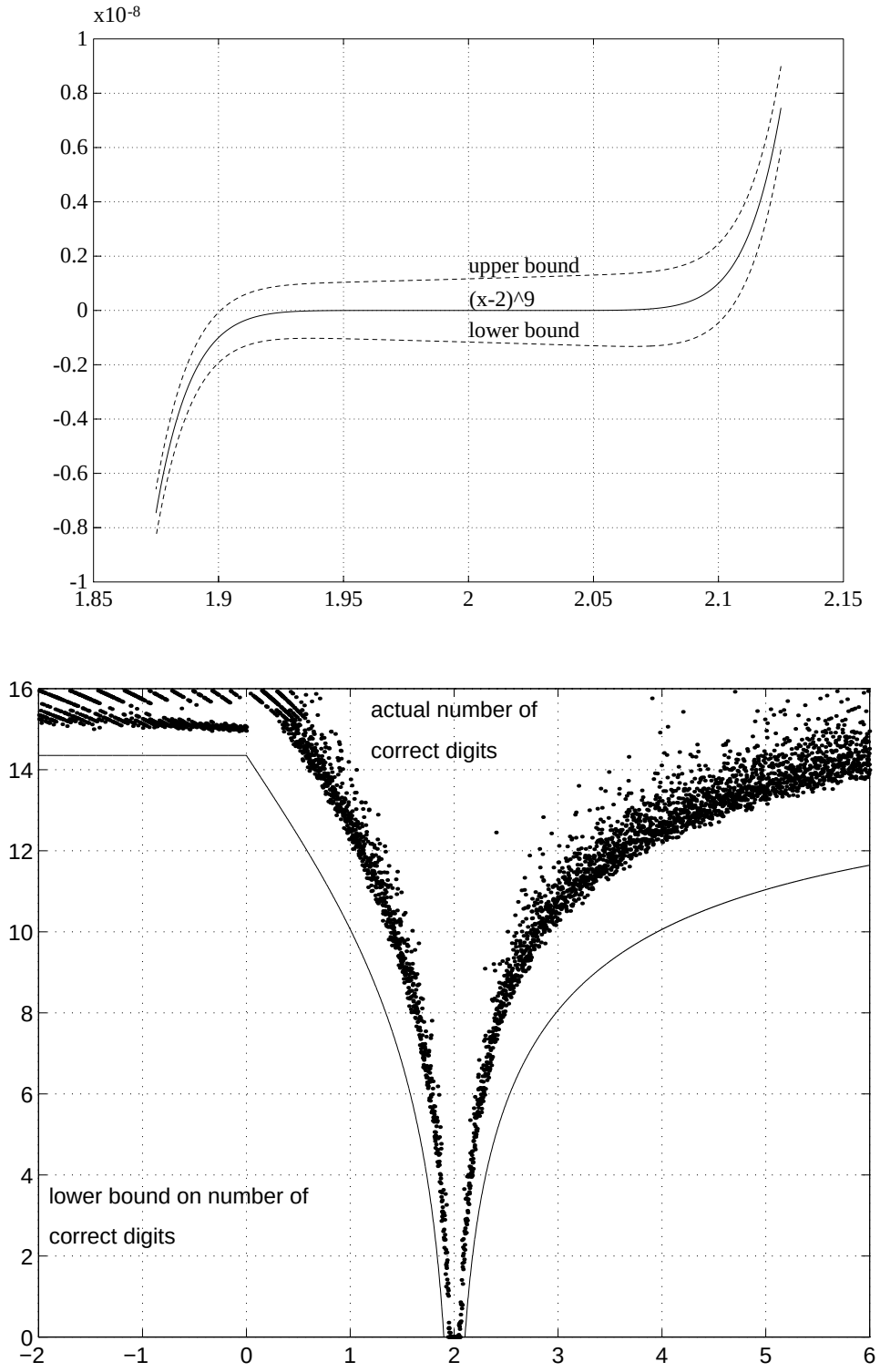


Fig. 1.3. *Plot of error bounds on the value of* $y = (x-2)^9$ *evaluated using Horner's rule.*

This simple reciprocal relationship between condition number and distance to the nearest ill-posed problem is very common in numerical analysis, and we shall encounter it again later.

At the beginning of the introduction we said that we would use canonical forms of matrices to help solve linear algebra problems. For example, knowing the exact Jordan canonical form makes computing exact eigenvalues trivial. There is an analogous canonical form for polynomials, which makes accurate polynomial evaluation easy: $p(x) = a_d \prod_{i=1}^{d}(x - r_i)$. In other words, we represent the polynomial by its leading coefficient a_d and its roots $r_1, \ldots r_n$. To evaluate $p(x)$ we use the obvious algorithm

$p = a_d$
for $i = 1$ to d
    $p = p \cdot (x - r_i)$
end for

It is easy to show the computed $p = p(x) \cdot (1 + \delta)$, where $|\delta| \leq 2d\varepsilon$; i.e., we always get $p(x)$ with high relative accuracy. But we need the roots of the polynomial to do this!

1.7. Vector and Matrix Norms

Norms are used to measure errors in matrix computations, so we need to understand how to compute and manipulate them.

Missing proofs are left as problems at the end of the chapter.

DEFINITION 1.3. *Let $\mathcal{B}$ be a real (complex) linear space $\mathbb{R}^n$* (or $\mathbb{C}^n$). *It is* normed *if there is a function $\|\cdot\| : \mathcal{B} \to \mathbb{R}$, which we call a* norm, *satisfying all of the following* :

1) $\|x\| \geq 0$, *and* $\|x\| = 0$ *if and only if* $x = 0$ (positive definiteness),
2) $\|\alpha x\| = |\alpha| \cdot \|x\|$ *for any real (or complex) scalar* α (homogeneity),
3) $\|x + y\| \leq \|x\| + \|y\|$ (the triangle inequality).

EXAMPLE 1.4. The most common norms are $\|x\|_p = (\sum_i |x_i|^p)^{1/p}$ for $1 \leq p < \infty$, which we call *p-norms*, as well as $\|x\|_\infty = \max_i |x_i|$, which we call the *$\infty$-norm* or *infinity-norm*. Also, if $\|x\|$ is any norm and C is any nonsingular matrix, then $\|Cx\|$ is also a norm. $\diamond$

We see that there are many norms that we could use to measure errors; it is important to choose an appropriate one. For example, let $x_1 = [1, 2, 3]^T$ in meters and $x_2 = [1.01, 2.01, 2.99]^T$ in meters. Then x_2 is a good approximation to x_1 because the relative error $\frac{\|x_1 - x_2\|_\infty}{\|x_1\|_\infty} \approx .0033$, and $x_3 = [10, 2.01, 2.99]^T$ is a bad approximation because $\frac{\|x_1 - x_3\|_\infty}{\|x_1\|_\infty} = 3$. But suppose the first component

is measured in kilometers instead of meters. Then in this norm $\hat{x}_1$ and $\hat{x}_3$ look close:

$$\hat{x}_1 = \begin{bmatrix} .001 \\ 2 \\ 3 \end{bmatrix}, \ \hat{x}_3 = \begin{bmatrix} .01 \\ 2.01 \\ 2.99 \end{bmatrix}, \text{ and } \frac{\|\hat{x}_1 - \hat{x}_3\|_\infty}{\|\hat{x}_1\|_\infty} \approx .0033 .$$

To compare $\hat{x}_1$ and $\hat{x}_3$, we should use

$$\|\hat{x}\|_s \equiv \left\| \begin{bmatrix} 1000 & & \\ & 1 & \\ & & 1 \end{bmatrix} \hat{x} \right\|_\infty$$

to make the units the same or so that equally important errors make the norm equally large.

Now we define *inner products*, which are a generalization of the standard *dot product* $\sum_i x_i y_i$, and arise frequently in linear algebra.

DEFINITION 1.4. *Let $\mathcal{B}$ be a real (complex) linear space. $\langle \cdot, \cdot \rangle : \mathcal{B} \times \mathcal{B} \to \mathbb{R}(\mathbb{C})$ is an* inner product *if all of the following apply :*

1) $\langle x, y \rangle = \langle y, x \rangle$ *(or* $\overline{\langle y, x \rangle}$*)*,
2) $\langle x, y + z \rangle = \langle x, y \rangle + \langle x, z \rangle$,
3) $\langle \alpha x, y \rangle = \alpha \langle x, y \rangle$ *for any real (or complex) scalar* α,
4) $\langle x, x \rangle \geq 0$, *and* $\langle x, x \rangle = 0$ *if and only if* $x = 0$.

EXAMPLE 1.5. Over $\mathbb{R}$, $\langle x, y \rangle = y^T x = \sum_i x_i y_i$, and over $\mathbb{C}$, $\langle x, y \rangle = y^* x = \sum_i x_i \bar{y}_i$ are inner products. (Recall that $y^* = \bar{y}^T$ is the conjugate transpose of y.) $\diamond$

DEFINITION 1.5. *x and y are* orthogonal *if* $\langle x, y \rangle = 0$.

The most important property of an inner product is that it satisfies the Cauchy–Schwartz inequality. This can be used in turn to show that $\sqrt{\langle x, x \rangle}$ is a norm, one that we will frequently use.

LEMMA 1.1. Cauchy–Schwartz inequality. $|\langle x, y \rangle| \leq \sqrt{\langle x, x \rangle \cdot \langle y, y \rangle}$.

LEMMA 1.2. *$\sqrt{\langle x, x \rangle}$ is a norm.*

There is a one-to-one correspondence between inner-products and *symmetric (Hermitian) positive definite matrices*, as defined below. These matrices arise frequently in applications.

DEFINITION 1.6. *A real symmetric (complex Hermitian) matrix A is* positive definite *if $x^T A x > 0$ ($x^* A x > 0$) for all $x = 0$. We abbreviate symmetric positive definite to s.p.d., and Hermitian positive to h.p.d..*

LEMMA 1.3. *Let $\mathcal{B} = \mathbb{R}^n$ (or $\mathbb{C}^n$) and $\langle \cdot, \cdot \rangle$ be an inner product. Then there is an n-by-n s.p.d. (h.p.d.) matrix A such that $\langle x, y \rangle = y^T Ax$ ($y^* Ax$). Conversely, if A is s.p.d (h.p.d.), then $y^T Ax$ ($y^* Ax$) is an inner product.*

The following two lemmas are useful in converting error bounds in terms of one norm to error bounds in terms of another.

LEMMA 1.4. *Let $\| \cdot \|_\alpha$ and $\| \cdot \|_\beta$ be two norms on $\mathbb{R}^n$ (or $\mathbb{C}^n$). There are constants $c_1, c_2 > 0$ such that, for all x, $c_1 \|x\|_\alpha \leq \|x\|_\beta \leq c_2 \|x\|_\alpha$. We also say that norms $\| \cdot \|_\alpha$ and $\| \cdot \|_\beta$ are* equivalent *with respect to constants c_1 and c_2.*

LEMMA 1.5.

$$\begin{array}{rcccl} \|x\|_2 & \leq & \|x\|_1 & \leq & \sqrt{n}\|x\|_2, \\ \|x\|_\infty & \leq & \|x\|_2 & \leq & \sqrt{n}\|x\|_\infty, \\ \|x\|_\infty & \leq & \|x\|_1 & \leq & n\|x\|_\infty. \end{array}$$

In addition to vector norms, we will also need *matrix norms* to measure errors in matrices.

DEFINITION 1.7. *$\| \cdot \|$ is a* matrix norm *on m-by-n matrices if it is a vector norm on $m \cdot n$ dimensional space:*

1) *$\|A\| \geq 0$ and $\|A\| = 0$ if and only if $A = 0$,*
2) *$\|\alpha A\| = |\alpha| \cdot \|A\|$,*
3) *$\|A + B\| \leq \|A\| + \|B\|$.*

EXAMPLE 1.6. $\max_{ij} |a_{ij}|$ is called the *max norm*, and $(\sum |a_{ij}|^2)^{1/2} = \|A\|_F$ is called the *Frobenius norm*. $\diamond$

The following definition is useful for bounding the norm of a product of matrices, something we often need to do when deriving error bounds.

DEFINITION 1.8. *Let $\| \cdot \|_{m \times n}$ be a matrix norm on m-by-n matrices, $\| \cdot \|_{n \times p}$ be a matrix norm on n-by-p matrices, and $\| \cdot \|_{m \times p}$ be a matrix norm on m-by-p matrices. These norms are called* mutually consistent *if $\|A \cdot B\|_{m \times p} \leq \|A\|_{m \times n} \cdot \|B\|_{n \times p}$, where A is m-by-n and B is n-by-p.*

DEFINITION 1.9. *Let A be m-by-n, $\| \cdot \|_{\hat{m}}$ be a vector norm on $\mathbb{R}^m$, and $\| \cdot \|_{\hat{n}}$ be a vector norm on $\mathbb{R}^n$. Then*

$$\|A\|_{\hat{m}\hat{n}} \equiv \max_{\substack{x=0 \\ x \in \mathbb{R}^n}} \frac{\|Ax\|_{\hat{m}}}{\|x\|_{\hat{n}}}$$

is called an operator norm *or* induced norm *or* subordinate matrix norm.

The next lemma provides a large source of matrix norms, ones that we will use for bounding errors.

LEMMA 1.6. *An operator norm is a matrix norm.*

Orthogonal and *unitary matrices*, defined next, are essential ingredients of nearly all our algorithms for least squares problems and eigenvalue problems.

DEFINITION 1.10. *A real square matrix Q is* orthogonal *if $Q^{-1} = Q^T$. A complex square matrix is* unitary *if $Q^{-1} = Q^*$.*

All rows (or columns) of orthogonal (or unitary) matrices have unit 2-norms and are orthogonal to one another, since $QQ^T = Q^TQ = I$ ($QQ^* = Q^*Q = I$).

The next lemma summarizes the essential properties of the norms and matrices we have introduced so far. We will use these properties later in the book.

LEMMA 1.7.

1. $\|Ax\| \leq \|A\| \cdot \|x\|$ *for a vector norm and its corresponding operator norm, or the vector two-norm and matrix Frobenius norm.*
2. $\|AB\| \leq \|A\| \cdot \|B\|$ *for any operator norm or for the Frobenius norm. In other words, any operator norm (or the Frobenius norm) is mutually consistent with itself.*
3. *The max norm and Frobenius norm are not operator norms.*
4. $\|QAZ\| = \|A\|$ *if Q and Z are orthogonal or unitary for the Frobenius norm and for the operator norm induced by $\|\cdot\|_2$. This is really just the Pythagorean theorem.*
5. $\|A\|_\infty \equiv \max_{x=0} \frac{\|Ax\|_\infty}{\|x\|_\infty} = \max_i \sum_j |a_{ij}|$ = *maximum absolute row sum.*
6. $\|A\|_1 \equiv \max_{x=0} \frac{\|Ax\|_1}{\|x\|_1} = \|A^T\|_\infty = \max_j \sum_i |a_{ij}|$ = *maximum absolute column sum.*
7. $\|A\|_2 \equiv \max_{x=0} \frac{\|Ax\|_2}{\|x\|_2} = \sqrt{\lambda_{\max}(A^*A)}$, *where $\lambda_{\max}$ denotes the largest eigenvalue.*
8. $\|A\|_2 = \|A^T\|_2$.
9. $\|A\|_2 = \max_i |\lambda_i(A)|$ *if A is* normal, *i.e., $AA^* = A^*A$.*
10. *If A is n-by-n, then $n^{-1/2}\|A\|_2 \leq \|A\|_1 \leq n^{1/2}\|A\|_2$.*
11. *If A is n-by-n, then $n^{-1/2}\|A\|_2 \leq \|A\|_\infty \leq n^{1/2}\|A\|_2$.*
12. *If A is n-by-n, then $n^{-1}\|A\|_\infty \leq \|A\|_1 \leq n\|A\|_\infty$.*
13. *If A is n-by-n, then $\|A\|_1 \leq \|A\|_F \leq n^{1/2}\|A\|_2$.*

Proof. We prove part 7 only and leave the rest to the reader.

Since A^*A is Hermitian, there exists an eigendecomposition $A^*A = Q\Lambda Q^*$, with Q a unitary matrix (the columns are eigenvectors), and $\Lambda = \mathrm{diag}(\lambda_1, \ldots, \lambda_n)$, a diagonal matrix containing the eigenvalues, which must all be real. Note that all $\lambda_i \geq 0$ since if one, say λ, were negative, we would take q as its eigenvector and get the contradiction $0 \leq \|Aq\|_2^2 = q^T A^T A q = q^T \lambda q = \lambda \|q\|_2^2 < 0$. Therefore

$$\begin{aligned}
\|A\|_2 &= \max_{x=0} \frac{\|Ax\|_2}{\|x\|_2} = \max_{x=0} \frac{(x^*A^*Ax)^{1/2}}{\|x\|_2} = \max_{x=0} \frac{(x^*Q\Lambda Q^*x)^{1/2}}{\|x\|_2} \\
&= \max_{x=0} \frac{((Q^*x)^*\Lambda Q^*x)^{1/2}}{\|Q^*x)\|_2} = \max_{y=0} \frac{(y^*\Lambda y)^{1/2}}{\|y\|_2} = \max_{y=0} \sqrt{\frac{\sum \lambda_i y_i^2}{\sum y_i^2}} \\
&\leq \max_{y=0} \sqrt{\lambda_{\max}} \sqrt{\frac{\sum y_i^2}{\sum y_i^2}} = \sqrt{\lambda_{\max}},
\end{aligned}$$

which is attainable by choosing y to be the appropriate column of the identity matrix. □

1.8. References and Other Topics for Chapter 1

At the end of each chapter we will list the references most relevant to that chapter. They are also listed alphabetically in the bibliography at the end. In addition we will give pointers to related topics not discussed in the main text.

The most modern comprehensive work in this area is by G. Golub and C. Van Loan [119], which also has an extensive bibliography. A recent undergraduate level or beginning graduate text in this material is by D. Watkins [250]. Another good graduate text is by L. Trefethen and D. Bau [241]. A classic work that is somewhat dated but still an excellent reference is by J. Wilkinson [260]. An older but still excellent book at the same level as Watkins is by G. Stewart [233].

More detailed information on error analysis can be found in the recent book by N. Higham [147]. Older but still good general references are by J. Wilkinson [259] and W. Kahan [155].

"What every computer scientist should know about floating point arithmetic" by D. Goldberg is a good recent survey [117]. IEEE arithmetic is described formally in [11, 12, 157] as well as in the reference manuals published by computer manufacturers. Discussion of error analysis with IEEE arithmetic may be found in [53, 69, 157, 156] and the references cited therein.

A more general discussion of condition numbers and the distance to the nearest ill-posed problem is given by the author in [70] as well as in a series of papers by S. Smale and M. Shub [217, 218, 219, 220]. Vector and matrix norms are discussed at length in [119, sects. 2.2, 2.3].

1.9. Questions for Chapter 1

QUESTION 1.1. *(Easy; Z. Bai)* Let A be an orthogonal matrix. Show that $\det(A) = \pm 1$. Show that if B also is orthogonal and $\det(A) = -\det(B)$, then $A + B$ is singular.

QUESTION 1.2. *(Easy; Z. Bai)* The *rank* of a matrix is the dimension of the space spanned by its columns. Show that A has rank one if and only if $A = ab^T$ for some column vectors a and b.

QUESTION 1.3. *(Easy; Z. Bai)* Show that if a matrix is orthogonal and triangular, then it is diagonal. What are its diagonal elements?

QUESTION 1.4. *(Easy; Z. Bai)* A matrix is *strictly upper triangular* if it is upper triangular with zero diagonal elements. Show that if A is strictly upper triangular and n-by-n, then $A^n = 0$.

QUESTION 1.5. *(Easy; Z. Bai)* Let $\|\cdot\|$ be a vector norm on $\mathbb{R}^m$ and assume that $C \in \mathbb{R}^{m\times n}$. Show that if $\operatorname{rank}(A) = n$, then $\|x\|_C \equiv \|Cx\|$ is a vector norm.

QUESTION 1.6. *(Easy; Z. Bai)* Show that if $0 = s \in \mathbb{R}^n$ and $E \in \mathbb{R}^{n\times n}$, then

$$\left\| E\left(I - \frac{ss^{\mathrm{T}}}{s^T s}\right)\right\|_F^2 = \|E\|_F^2 - \frac{\|Es\|_2^2}{s^T s}.$$

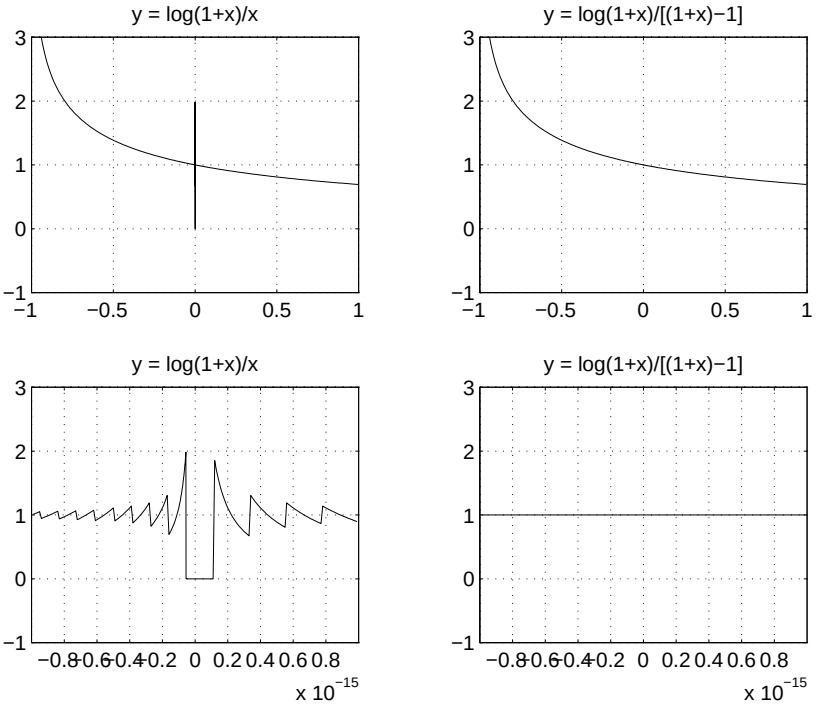
QUESTION 1.7. *(Easy; Z. Bai)* Verify that $\|xy^H\|_F = \|xy^H\|_2 = \|x\|_2\|y\|_2$ for any $x, y \in \mathbf{C}^n$.

QUESTION 1.8. *(Medium)* One can identify the degree d polynomials $p(x) = \sum_{i=0}^{d} a_i x^i$ with $\mathbb{R}^{d+1}$ via the vector of coefficients. Let x be fixed. Let S_x be the set of polynomials with an infinite relative condition number with respect to evaluating them at x (i.e., they are zero at x). In a few words, describe S_x geometrically as a subset of $\mathbb{R}^{d+1}$. Let $S_x(\kappa)$ be the set of polynomials whose relative condition number is κ or greater. Describe $S_x(\kappa)$ geometrically in a few words. Describe how $S_x(\kappa)$ changes geometrically as $\kappa \to \infty$.

QUESTION 1.9. *(Medium; from the* 1995 *final exam)* Consider the figure below. It plots the function $y = \log(1 + x)/x$ computed in two different ways. Mathematically, y is a smooth function of x near $x = 0$, equaling 1 at 0. But if we compute y using this formula, we get the plots on the left (shown in the ranges $x \in [-1, 1]$ on the top left and $x \in [-10^{-15}, 10^{-15}]$ on the bottom left). This formula is clearly unstable near $x = 0$. On the other hand, if we use the algorithm

$d = 1 + x$
if $d = 1$ then
    $y = 1$
else
    $y = \log(d)/(d-1)$
end if

we get the two plots on the right, which are correct near $x = 0$. Explain this phenomenon, proving that the second algorithm must compute an accurate answer in floating point arithmetic. Assume that the log function returns an accurate answer for any argument. (This is true of any reasonable implementation of logarithm.) Assume IEEE floating point arithmetic if that makes your argument easier. (Both algorithms can malfunction on a Cray machine.)



QUESTION 1.10. *(Medium)* Show that, barring overflow or underflow, $\mathrm{fl}(\sum_{i=1}^{d} x_i y_i) = \sum_{i=1}^{d} x_i y_i (1+\delta_i)$, where $|\delta_i| \leq d\varepsilon$. Use this to prove the following fact. Let $A^{m\times n}$ and $B^{n\times p}$ be matrices, and compute their product in the usual way. Barring overflow or underflow show that $|\mathrm{fl}(A \cdot B) - A \cdot B| \leq n \cdot \varepsilon \cdot |A| \cdot |B|$. Here the absolute value of a matrix $|A|$ means the matrix with entries $(|A|)_{ij} = |a_{ij}|$, and the inequality is meant componentwise.

The result of this question will be used in section 2.4.2, where we analyze the roundoff errors in Gaussian elimination.

QUESTION 1.11. *(Medium)* Let L be a lower triangular matrix and solve $Lx = b$ by forward substitution. Show that barring overflow or underflow, the computed solution $\hat{x}$ satisfies $(L + \delta L)\hat{x} = b$, where $|\delta l_{ij}| \leq n\varepsilon |l_{ij}|$, where ε is the machine precision. This means that forward substitution is backward stable. Argue that backward substitution for solving upper triangular systems satisfies the same bound.

The result of this question will be used in section 2.4.2, where we analyze the roundoff errors in Gaussian elimination.

QUESTION 1.12. *(Medium)* In order to analyze the effects of rounding errors, we have used the following model (see equation (1.1)):

$$fl(a \odot b) = (a \odot b)(1 + \delta),$$

where $\odot$ is one of the four basic operations $+$, $-$, $*$, and $/$, and $|\delta| \leq \varepsilon$. To show that our analyses also work for *complex* data, we need to prove an analogous formula for the four basic complex operations. Now δ will be a tiny *complex* number bounded in absolute value by a small multiple of ϵ. Prove that this is true for complex addition, subtraction, multiplication, and division. Your algorithm for complex division should successfully compute $a/a \approx 1$, where $|a|$ is either very large (larger than the square root of the overflow threshold) or very small (smaller than the square root of the underflow threshold). Is it true that both the real and imaginary parts of the complex product are always computed to high relative accuracy?

QUESTION 1.13. *(Medium)* Prove Lemma 1.3.

QUESTION 1.14. *(Medium)* Prove Lemma 1.5.

QUESTION 1.15. *(Medium)* Prove Lemma 1.6.

QUESTION 1.16. *(Medium)* Prove all parts except 7 of Lemma 1.7. Hint for part 8: Use the fact that if X and Y are both n-by-n, then XY and YX have the same eigenvalues. Hint for part 9: Use the fact that a matrix is normal if and only if it has a complete set of orthonormal eigenvectors.

QUESTION 1.17. *(Hard; W. Kahan)* We mentioned that on a Cray machine the expression $\arccos(x/\sqrt{x^2 + y^2})$ caused an error, because roundoff caused $(x/\sqrt{x^2 + y^2})$ to exceed 1. Show that this is impossible using IEEE arithmetic, barring overflow or underflow. Hint: You will need to use more than the simple model $\mathrm{fl}(a \odot b) = (a \odot b)(1 + \delta)$ with $|\delta|$ small. Think about evaluating $\sqrt{x^2}$, and show that, barring overflow or underflow, $\mathrm{fl}(\sqrt{x^2}) = x$ *exactly*; in numerical experiments done by A. Liu, this failed about 5% of the time on a Cray YMP. You might try some numerical experiments and explain them. Extra credit: Prove the same result using correctly rounded *decimal* arithmetic. (The proof is different.) This question is due to W. Kahan, who was inspired by a bug in a Cray program of J. Sethian.

QUESTION 1.18. *(Hard)* Suppose a and b are normalized IEEE double precision floating point numbers, and consider the following algorithm, running with IEEE arithmetic:

if $(|a| < |b|)$, swap a and b
$s_1 = a + b$
$s_2 = (a - s_1) + b$

Prove the following facts:

1. Barring overflow or underflow, the only roundoff error committed in running the algorithm is computing $s_1 = fl(a + b)$. In other words, both subtractions $s_1 - a$ and $(s_1 - a) - b$ are computed *exactly.*

2. $s_1 + s_2 = a + b$, *exactly.* This means that s_2 is actually the roundoff error committed when rounding the exact value of $a + b$ to get s_1.

Thus, this program in effect simulates *quadruple* precision arithmetic, representing the true sum $a + b$ as the higher-order bits (s_1) and the lower-order bits (s_2).

Using this and similar tricks in a systematic way, it is possible to efficiently simulate all four basic floating point operations in *arbitrary* precision arithmetic, using only the underlying floating point instructions and no "bit-fiddling" [202]. 128-bit arithmetic is implemented this way on the IBM RS6000 and Cray (but much less efficiently on the Cray, which does not have IEEE arithmetic).

QUESTION 1.19. *(Hard; Programming)* This question illustrates the challenges in engineering highly reliable numerical software. Your job is to write a program to compute the two-norm $s \equiv \|x\|_2 = (\sum_{i=1}^{n} x_i^2)^{1/2}$ given $x_1, ldots, x_n$. The most obvious (and inadequate) algorithm is

$s = 0$
for $i = 1$ to n
    $s = s + x_i^2$
endfor
$s = \mathrm{sqrt}(s)$

This algorithm is inadequate because it does not have the following desirable properties:

1. It must compute the answer accurately (i.e., nearly all the computed digits must be correct) unless $\|x\|_2$ is (nearly) outside the range of normalized floating point numbers.

2. It must be nearly as fast as the obvious program above in most cases.

3. It must work on any "reasonable" machine, possibly including ones not running IEEE arithmetic. This means it may not cause an error condition, unless $\|x\|_2$ is (nearly) larger than the largest floating point number.

To illustrate the difficulties, note that the obvious algorithm fails when $n = 1$ and x_1 is larger than the square root of the largest floating point number (in which case x_1^2 overflows, and the program returns $+\infty$ in IEEE arithmetic and halts in most non-IEEE arithmetics) or when $n = 1$ and x_1 is smaller than the square root of the smallest normalized floating point number (in which case x_1^2 underflows, possibly to zero, and the algorithm may return zero). Scaling the x_i by dividing them all by $\max_i |x_i|$ does not have property 2), because division is usually many times more expensive than either multiplication or addition. Multiplying by $c = 1/\max_i |x_i|$ risks overflow in computing c, even when $\max_i |x_i| > 0$.

This routine is important enough that it has been standardized as a *Basic Linear Algebra Subroutine*, or *BLAS*, which should be available on all machines [167]. We discuss the BLAS at length in section 2.6.1, and documentation and sample implementations may be found at NETLIB/blas. In particular, see NETLIB/cgi-bin/netlibget.pl/blas/snrm2.f for a sample implementation that has properties 1) and 3) but not 2). These sample implementations are intended to be starting points for implementations specialized to particular architectures (an easier problem than producing a completely portable one, as requested in this problem). Thus, when writing your own numerical software, you should think of computing $\|x\|_2$ as a building block that should be available in a numerical library on each machine.

For another careful implementation of $\|x\|_2$, see [34].

You can extract test code from NETLIB/blas/sblat1 to see if your implementation is correct; all implementations turned in must be thoroughly tested as well as timed, with times compared to the obvious algorithm above on those cases where both run. See how close to satisfying the three conditions you can come; the frequent use of the word "nearly" in conditions (1), (2) and (3) shows where you may compromise in attaining one condition in order to more nearly attain another. In particular, you might want to see how much easier the problem is if you limit yourself to machines running IEEE arithmetic.

Hint: Assume that the values of the overflow and underflow thresholds are available for your algorithm. Portable software for computing these values is available (see NETLIB/cgi-bin/netlibget.pl/lapack/util/slamch.f).

QUESTION 1.20. *(Easy; Medium)* We will use a Matlab program to illustrate how sensitive the roots of polynomial can be to small perturbations in the coefficients. The program is available[5] at HOMEPAGE/Matlab/polyplot.m.

[5]Recall that we abbreviate the URL prefix of the class homepage to HOMEPAGE in the text.

Polyplot takes an input polynomial specified by its roots r and then adds random perturbations to the polynomial coefficients, computes the perturbed roots, and plots them. The inputs are

r = vector of roots of the polynomial,
e = maximum relative perturbation to make to each coefficient of the polynomial,
m = number of random polynomials to generate, whose roots are plotted.

1. *(Easy)* The first part of your assignment is to run this program for the following inputs. In all cases choose m high enough that you get a fairly dense plot but don't have to wait too long. m = a few hundred or perhaps 1000 is enough. You may want to change the axes of the plot if the graph is too small or too large.
   - r=(1:10); e = 1e-3, 1e-4, 1e-5, 1e-6, 1e-7, 1e-8,
   - r=(1:20); e = 1e-9, 1e-11, 1e-13, 1e-15,
   - r=[2,4,8,16,..., 1024]; e=1e-1, 1e-2, 1e-3, 1e-4 (in this case, use axis([.1,1e4,-4,4]) and semilogx(real(r1),imag(r1),'.'))

   Also try your own example with complex conjugate roots. Which roots are most sensitive?

2. *(Medium)* The second part of your assignment is to modify the program to compute the condition number c(i) for each root. In other words, a relative perturbation of e in each coefficient should change root r(i) by at most about e*c(i). Modify the program to plot circles centered at r(i) with radii e*c(i), and confirm that these circles enclose the perturbed roots (at least when e is small enough that the linearization used to derive the condition number is accurate). You should turn in a few plots with circles and perturbed eigenvalues, and some explanation of what you observe.

3. *(Medium)* In the last part, notice that your formula for c(i) "blows up" if $p'(\mathrm{r(i)}) = 0$. This condition means that r(i) is a *multiple root* of $p(x) = 0$. We can still expect some accuracy in the computed value of a multiple root, however, and in this part of the question, we will ask how sensitive a multiple root can be: First, write $p(x) = q(x) \cdot (x - \mathrm{r(i)})^m$, where $q(\mathrm{r(i)}) = 0$ and m is the multiplicity of the root r(i). Then compute the m roots nearest r(i) of the slightly perturbed polynomial $p(x) - q(x)\epsilon$, and show that they differ from r(i) by $|\epsilon|^{1/m}$. So that if $m = 2$, for instance, the root r(i) is perturbed by $\epsilon^{1/2}$, which is much larger than ϵ if $|\epsilon| \ll 1$. Higher values of m yield even larger perturbations. If ϵ is around machine epsilon and represents rounding errors in computing the root, this means an m-tuple root can lose all but $1/m$-th of its significant digits.

QUESTION 1.21. *(Medium)* Apply Algorithm 1.1, Bisection, to find the roots of $p(x) = (x-2)^9 = 0$, where $p(x)$ is evaluated using Horner's rule. Use the Matlab implementation in HOMEPAGE/Matlab/bisect.m, or else write your own. Confirm that changing the input interval slightly changes the computed root drastically. Modify the algorithm to use the error bound discussed in the text to stop bisecting when the roundoff error in the computed value of $p(x)$ gets so large that its sign cannot be determined.