Chapter 2
How to Obtain and Estimate Accuracy

I always think I used computers for what
God had intended them for, to do arithmetic.
—Cleve Moler

2.1 Basic Concepts in Error Estimation

The main purpose of numerical analysis and scientific computing is to develop efficient and accurate methods to compute approximations to quantities that are difficult or impossible to obtain by analytic means. It has been convincingly argued (Trefethen [357]) that controlling rounding errors is just a small part of this, and that the main business of computing is the development of algorithms that converge rapidly. Even if we acknowledge the truth of this statement, it is still necessary to be able to control different sources of errors, including roundoff errors, so that these will not interfere with the computed results.

2.1.1 Sources of Error

Numerical results are affected by many types of errors. Some sources of error are difficult to influence; others can be reduced or even eliminated by, for example, rewriting formulas or making other changes in the computational sequence. Errors are propagated from their sources to quantities computed later, sometimes with a considerable amplification or damping. It is important to distinguish between the new error produced at the computation of a quantity (a source error), and the error inherited (propagated) from the data that the quantity depends on.

A. *Errors in Given Input Data.*
Input data can be the result of measurements which have been contaminated by different types of errors. In general one should be careful to distinguish between **systematic errors** and **random errors**. A systematic error can, for example, be produced by insufficiencies in the construction of an instrument of measurement;

such an error is the same in each trial. Random errors depend on the variation in the experimental environment which cannot be controlled.

B. *Rounding Errors During the Computations.*
A **rounding error** occurs whenever an irrational number, for example π, is shortened ("rounded off") to a fixed number of digits, or when a decimal fraction is converted to the binary form used in the computer. The limitation of floating-point numbers in a computer leads at times to a loss of information that, depending on the context, may or may not be important. Two typical cases are

(i) If the computer cannot handle numbers which have more than, say, s digits, then the exact product of two s-digit numbers (which contains $2s$ or $2s-1$ digits) cannot be used in subsequent calculations; the product must be rounded off.

(ii) In a floating-point computation, if a relatively small term b is added to a, then some digits of b are "shifted out" (see Example 2.3.1), and they will not have any effect on future quantities that depend on the value of $a+b$.

The effect of such rounding can be quite noticeable in an extensive calculation, or in an algorithm which is numerically unstable.

C. *Truncation Errors.*
These are errors committed when a limiting process is truncated (broken off) before one has come to the limiting value. A **truncation error** occurs, for example, when an infinite series is broken off after a finite number of terms, or when a derivative is approximated with a difference quotient (although in this case the term **discretization error** is better). Another example is when a nonlinear function is approximated with a linear function, as in Newton's method. Observe the distinction between truncation error and rounding error.

D. *Simplifications in the Mathematical Model.*
In most of the applications of mathematics, one makes idealizations. In a mechanical problem one might assume that a string in a pendulum has zero mass. In many other types of problems it is advantageous to consider a given body to be homogeneously filled with matter, instead of being built of atoms. For a calculation in economics, one might assume that the rate of interest is constant over a given period of time. The effects of such sources of error are usually more difficult to estimate than the types named in A, B, and C.

E. *"Human" Errors and Machine Errors.*
In all numerical work, one must expect that clerical errors, errors in hand calculation, and misunderstandings will occur. One should even be aware that textbooks (!), tables, and formulas may contain errors. When one uses computers, one can expect errors in the program itself, typing errors in entering the data, operator errors, and (less frequently) pure machine errors.

Errors which are purely machine errors are responsible for only a very small part of the strange results which (occasionally with great publicity) are produced by computers. Most of the errors depend on the so-called human factor. As a rule, the effect of this type of error source cannot be analyzed with the help of the theoretical considerations of this chapter! We take up these sources of error in order to emphasize that both the person who carries out a calculation and the person who guides the work of others can plan so that such sources of error are not damaging. One can reduce the risk of such errors by suitable adjustments in working conditions and routines. Stress and fatigue are common causes of such errors.

Intermediate results that may reveal errors in a computation are not visible when using a computer. Hence the user must be able to verify the correctness of his results or be able to prove that his process cannot fail! Therefore, one should carefully consider what kind of checks can be made, either in the final result or in certain stages of the work, to prevent the necessity of redoing a whole project just because a small error has been made in an early stage. One can often discover whether calculated values are of the wrong order of magnitude or are not sufficiently regular, for example, using difference checks (see Sec. 3.3.1).

Occasionally one can check the credibility of several results at the same time by checking that certain relations are true. In linear problems, one often has the possibility of sum checks. In physical problems, one can check to see whether energy is conserved, although because of error sources A to D one cannot expect that it will be *exactly* conserved. In some situations, it can be best to treat a problem in two independent ways, although one can usually (as intimated above) check a result with less work than this.

Errors of type E do occur, sometimes with serious consequences. The first American Venus probe was lost due to a program fault caused by the inadvertent substitution of a statement in a Fortran program of the form `DO 3 I = 1.3` for one of the form `DO 3 I = 1,3`. Erroneously replacing the comma "," with a dot "." converts the intended loop statement into an assignment statement! A hardware error that got much publicity surfaced in 1994, when it was found that the INTEL Pentium processor gave wrong results for division with floating-point numbers of certain patterns. This was discovered during research on prime numbers (see Edelman [103]) and later fixed.

From a different point of view, one may distinguish between controllable and uncontrollable (or unavoidable) error sources. Errors of type A and D are usually considered to be uncontrollable in the numerical treatment (although feedback to the constructor of the mathematical model may sometimes be useful). Errors of type C are usually controllable. For example, the number of iterations in the solution of an algebraic equation, or the step size in a simulation, can be chosen either directly or by setting a tolerance.

The rounding error in the individual arithmetic operation (type B) is, in a computer, controllable only to a limited extent, mainly through the choice between single and double precision. A very important fact is, however, that it can often be controlled by appropriate rewriting of formulas or by other changes of the algorithm; see Example 2.3.3.

If it doesn't cost too much, a controllable error source should be controlled so that its effects are evidently negligible compared to the effects of the uncontrollable sources. A reasonable interpretation of "full accuracy" is that the controllable error sources should not increase the error of a result by more than about 20%. Sometimes, "full accuracy" may be expensive, for example, in terms of computing time, memory space, or programming efforts. Then it becomes important to estimate the relation between accuracy and these cost factors. One goal of the rest of this chapter is to introduce concepts and techniques useful for this purpose.

Many real-world problems contain some nonstandard features, where understanding the general principles of numerical methods can save much time in the preparation of a program as well as in the computer runs. Nevertheless, we strongly encourage the reader to use quality library programs whenever possible, since a lot of experience and profound theoretical analysis has often been built into these (sometimes far beyond the scope of this text). It is not practical to "reinvent the wheel."

2.1.2 Absolute and Relative Errors

Approximation is a central concept in almost all the uses of mathematics. One must often be satisfied with approximate values of the quantities with which one works. Another type of approximation occurs when one ignores some quantities which are small compared to others. Such approximations are often necessary to ensure that the mathematical and numerical treatment of a problem does not become hopelessly complicated.

We make the following definition.

Definition 2.1.1.

Let $\tilde{x}$ be an approximate value whose exact value is x. Then the **absolute error** *in $\tilde{x}$ is*

$$\Delta x = |\tilde{x} - x|,$$

and if $x \neq 0$ the **relative error** *is*

$$\Delta x / x = |(\tilde{x} - x)/x|.$$

In some books the error is defined with the opposite sign to what we use here. It makes almost no difference which convention one uses, as long as one is consistent. Note that $x - \tilde{x}$ is *the correction* which should be *added* to $\tilde{x}$ to get rid of the error. The correction and the absolute error then have the same magnitude but may have different signs.

In many situations one wants to compute a strict or approximate **bound** for the absolute or relative error. Since it is sometimes rather hard to obtain an error bound that is both strict and sharp, one sometimes prefers to use less strict but often realistic **error estimates**. These can be based on the first neglected term in some expansion, or on some other asymptotic considerations.

The notation $x = \tilde{x} \pm \epsilon$ means, in this book, $|\tilde{x} - x| \leq \epsilon$. For example, if $x = 0.5876 \pm 0.0014$, then $0.5862 \leq x \leq 0.5890$, and $|\tilde{x} - x| \leq 0.0014$. In other texts, the same plus–minus notation is sometimes used for the "standard error" (see Sec. 2.3.3) or some other measure of deviation of a statistical nature. If x is a vector $\|\cdot\|$, then the error bound and the relative error bound may be defined as bounds for

$$\|\tilde{x} - x\| \quad \text{and} \quad \|\tilde{x} - x\|/\|x\|,$$

respectively, where $\|\cdot\|$ denotes some vector norm (see Sec. A.3.3 in Online Appendix A). Then a bound $\|\tilde{x} - x\|/\|x\| \leq 1/2 \cdot 10^{-p}$ implies that components $\tilde{x}_i$ with $|\tilde{x}_i| \approx \|x\|$ have about p significant digits, but this is not true for components of smaller absolute value. An alternative is to use componentwise relative errors,

$$\max_i |\tilde{x}_i - x_i|/|x_i|, \tag{2.1.1}$$

but this assumes that $x_i \neq 0$ for all i.

We will distinguish between the terms accuracy and precision. By **accuracy** we mean the absolute or relative error of an approximate quantity. The term **precision** will be reserved for the accuracy with which the basic arithmetic operations $+$, $-$, $*$, $/$ are performed. For floating-point operations this is given by the unit roundoff; see (2.2.8).

Numerical results which are not followed by any error estimations should often, though not always, be considered as having an uncertainty of $\frac{1}{2}$ of a unit in the last decimal place. In presenting numerical results, it is a good habit, if one does not want to go through the difficulty of presenting an error estimate with each result, to give explanatory remarks such as

- "All the digits given are thought to be significant."
- "The data have an uncertainty of at most three units in the last digit."
- "For an ideal two-atom gas, $c_P/c_V = 1.4$ (exactly)."

We shall also introduce some notations, useful in practice, though their definitions are not exact in a mathematical sense:

$a \ll b$ ($a \gg b$) is read "a is much smaller (much greater) than b." What is meant by "much smaller"(or "much greater") depends on the context—among other things, on the desired precision.

$a \approx b$ is read "a is approximately equal to b" and means the same as $|a - b| \ll c$, where c is chosen appropriate to the context. We *cannot generally* say, for example, that $10^{-6} \approx 0$.

$a \lessapprox b$ (or $b \gtrapprox a$) is read "a is less than or approximately equal to b" and means the same as "$a \le b$ or $a \approx b$."

Occasionally we shall have use for the following more precisely defined mathematical concepts:

$f(x) = O(g(x))$, $x \to a$, means that $|f(x)/g(x)|$ is bounded as $x \to a$ (a can be finite, $+\infty$, or $-\infty$).

$f(x) = o(g(x))$, $x \to a$, means that $\lim_{x\to a} f(x)/g(x) = 0$.

$f(x) \sim g(x)$, $x \to a$, means that $\lim_{x\to a} f(x)/g(x) = 1$.

2.1.3 Rounding and Chopping

When one counts the *number of digits* in a numerical value one should not include zeros in the beginning of the number, as these zeros only help to denote where the decimal point should be. For example, the number 0.00147 has five decimals but is given with three digits. The number 12.34 has two decimals but is given with four digits.

If the magnitude of the error in a given numerical value $\tilde{a}$ does not exceed $\frac{1}{2} \cdot 10^{-t}$, then $\tilde{a}$ is said to have t **correct decimals**. The *digits* in $\tilde{a}$ which occupy positions where the unit is greater than or equal to 10^{-t} are then called **significant digits** (any initial zeros are not counted). Thus, the number 0.001234 ± 0.000004 has five correct decimals and three significant digits, while 0.001234 ± 0.000006 has four correct decimals and two significant digits. The number of correct decimals gives one an idea of the magnitude of the *absolute error*, while the number of significant digits gives a rough idea of the magnitude of the *relative error*.

We distinguish here between two ways of rounding off a number x to a given number t of decimals. In **chopping** (or round toward zero) one simply leaves off all the decimals to the right of the tth. That way is generally *not recommended* since the rounding error has, systematically, the opposite sign of the number itself. Also, the magnitude of the error can be as large as 10^{-t}.

In **rounding to nearest** (sometimes called "correct" or "optimal" rounding), one chooses a number with s decimals which is *nearest* to x. Hence if p is the part of the number which stands to the right of the sth decimal, one leaves the tth decimal unchanged if and only if $|p| < 0.5 \cdot 10^{-s}$. Otherwise one raises the sth decimal by 1. In the case of a tie, when x is equidistant to two s decimal digit numbers, then one raises the sth decimal if it is odd or leaves it unchanged if it is even (round to even). In this way, the error is positive or negative about equally often. The error in rounding a decimal number to s decimals will always lie in the interval $[-\frac{1}{2}10^{-s}, \frac{1}{2}10^{-s}]$.

Example 2.1.1.

Shortening to three decimals,

0.2397	rounds to 0.240	(is chopped to 0.239),
−0.2397	rounds to −0.240	(is chopped to −0.239),
0.23750	rounds to 0.238	(is chopped to 0.237),
0.23650	rounds to 0.236	(is chopped to 0.236),
0.23652	rounds to 0.237	(is chopped to 0.236).

Observe that when one rounds off a numerical value one produces an error; thus it is occasionally wise to give more decimals than those which are correct. Take $a = 0.1237 \pm 0.0004$, which has three correct decimals according to the definition given previously. If one rounds to three decimals, one gets 0.124; here the third decimal may not be correct, since the least possible value for a is 0.1233.

Suppose that you are tabulating a transcendental function and that a particular entry has been evaluated as 1.2845 correct to the digits given. You want to round the value to three decimals. Should the final digit be 4 or 5? The answer depends on whether there is a nonzero trailing digit. You compute the entry more accurately and find 1.28450, then 1.284500, then 1.2845000, etc. Since the function is transcendental, there clearly is no bound on the number of digits that has to be computed before distinguishing if to round to 1.284 or 1.285. This is called the **tablemaker's dilemma**.[30]

Example 2.1.2.

The difference between chopping and rounding can be important, as is illustrated by the following story. The index of the Vancouver Stock Exchange, founded at the initial value

[30]This can be used to advantage in order to protect mathematical tables from illegal copying by rounding a few entries incorrectly, where the error in doing so is insignificant due to several trailing zeros. An illegal copy could then be exposed simply by looking up these entries!

1000.000 in 1982, was hitting lows in the 500s at the end of 1983 even though the exchange apparently performed well. It was discovered (*The Wall Street Journal*, Nov. 8, 1983, p. 37) that the discrepancy was caused by a computer program which updated the index thousands of times a day and used chopping instead of rounding to nearest! The rounded calculation gave a value of 1098.892.

Review Questions

2.1.1 Clarify with examples the various types of error sources which occur in numerical work.

2.1.2 (a) Define "absolute error" and "relative error" for an approximation $\bar{x}$ to a scalar quantity x. What is meant by an error bound?

(b) Generalize the definitions in (a) to a vector x.

2.1.3 How is "rounding to nearest" performed?

2.1.4 Give π to four decimals using (a) chopping; (b) rounding.

2.1.5 What is meant by the "tablemaker's dilemma"?

2.2 Computer Number Systems

2.2.1 The Position System

In order to represent numbers in daily life, we use a **position system** with base 10 (the decimal system). Thus, to represent the numbers we use ten different characters, and the magnitude with which the digit a contributes to the value of a number depends on the digit's position in the number. If the digit stands n steps to the right of the decimal point, the value contributed is $a \cdot 10^{-n}$. For example, the sequence of digits 4711.303 means

$$4 \cdot 10^3 + 7 \cdot 10^2 + 1 \cdot 10^1 + 1 \cdot 10^0 + 3 \cdot 10^{-1} + 0 \cdot 10^{-2} + 3 \cdot 10^{-3}.$$

Every real number has a unique representation in the above way, except for the possibility of infinite sequences of nines—for example, the infinite decimal fraction 0.3199999... represents the same number as 0.32.

One can very well consider other position systems with bases other than 10. Any integer $\beta \geq 2$ (or $\beta \leq -2$) can be used as a base. One can show that every positive real number a has, with exceptions analogous to the nines sequences mentioned above, a unique representation of the form

$$a = d_n\beta^n + d_{n-1}\beta^{n-1} + \cdots + d_1\beta^1 + d_0\beta^0 + d_{-1}\beta^{-1} + d_{-2}\beta^{-2} + \cdots$$

or, more compactly, $a = (d_n d_{n-1} \dots d_0.d_{-1}d_{-2}\dots)_\beta$, where the coefficients d_i, the "digits" in the system with base β, are positive integers d_i such that $0 \leq d_i \leq \beta - 1$.

One of the greatest advantages of the position system is that one can give simple, general rules for the arithmetic operations. The smaller the base is, the simpler these rules become. This is just one reason why most computers operate in base 2, the **binary number system**. The addition and multiplication tables then take the following simple form:

$$0+0=0, \qquad 0+1=1+0=1, \qquad 1+1=10,$$
$$0\cdot 0=0, \qquad 0\cdot 1=1\cdot 0=0, \qquad 1\cdot 1=1.$$

In the binary system the number seventeen becomes 10001, since $1\cdot 2^4+0\cdot 2^3+0\cdot 2^2+0\cdot 2^1+1\cdot 2^0=$ sixteen + one = seventeen. Put another way $(10001)_2=(17)_{10}$, where the index (in decimal representation) denotes the base of the number system. The numbers become longer written in the binary system; large integers become about 3.3 times as long, since N binary digits suffice to represent integers less than $2^N=10^{N\log_{10}2}\approx 10^{N/3.3}$.

Occasionally one groups together the binary digits in subsequences of three or four, which is equivalent to using 2^3 and 2^4, respectively, as the base. These systems are called the **octal** and **hexadecimal** number systems, respectively. The octal system uses the digits from 0 to 7; in the hexadecimal system the digits 0 through 9 and the letters A, B, C, D, E, F ("ten" through "fifteen") are used.

Example 2.2.1.

$$(17)_{10}=(10001)_2=(21)_8=(11)_{16},$$
$$(13.25)_{10}=(1101.01)_2=(15.2)_8=(D.4)_{16},$$
$$(0.1)_{10}=(0.000110011001\ldots)_2=(0.199999\ldots)_{16}.$$

Note that *the finite decimal fraction* 0.1 *cannot be represented exactly by a finite fraction in the binary number system*! (For this reason some pocket calculators use base 10.)

Example 2.2.2.

In 1991 a Patriot missile in Saudi Arabia failed to track and interrupt an incoming Scud missile due to a precision problem. The Scud then hit an army barrack and killed 28 Americans. The computer used to control the Patriot missile was based on a design dating from the 1970s using 24-bit arithmetic. For the tracking computations, time was recorded by the system clock in tenths of a second but converted to a 24-bit floating-point number. Rounding errors in the time conversions caused an error in the tracking. After 100 hours of consecutive operations the calculated time in seconds was 359,999.6567 instead of the correct value 360,000, an error of 0.3433 seconds leading to an error in the calculated range of 687 meters; see Skeel [326]. Modified software was later installed.

In the binary system the "point" used to separate the integer and fractional part of a number (corresponding to the decimal point) is called the binary point. The digits in the binary system are called **bits** (**b**inary dig**its**).

We are so accustomed to the position system that we forget that it is built upon an ingenious idea. The reader can puzzle over how the rules for arithmetic operations would look if one used Roman numerals, a number system without the position principle described above.

Recall that rational numbers are precisely those real numbers which can be expressed as a quotient between two integers. Equivalently, rational numbers are those whose representation in a position system have a finite number of digits or whose digits are repeating.

We now consider the problem of conversion between two number systems with different bases. Since almost all computers use a binary system this problem arises as soon as one wants to input data in decimal form or print results in decimal form.

Algorithm 2.1. *Conversion between Number Systems.*

Let a be an integer given in number systems with base α. We want to determine its representation in a number system with base β:

$$a = b_n\beta^m + b_{m-1}\beta^{n-1} + \cdots + b_0, \quad 0 \le b_i < \beta. \tag{2.2.1}$$

The computations are to be done in the system with base α, and thus β also is expressed in this representation. The conversion is done by successive divisions of a with β: Set $q_0 = a$, and

$$q_k/\beta = q_{k+1} + b_k/\beta, \quad k = 0, 1, 2, \dots \tag{2.2.2}$$

(q_{k+1} is the quotient and b_k is the remainder in the division).

If a is not an integer, we write $a = b + c$, where b is the integer part and

$$c = c_{-1}\beta^{-1} + c_{-2}\beta^{-2} + c_{-3}\beta^{-3} + \cdots \tag{2.2.3}$$

is the fractional part, where $c_{-1}, c_{-2}, \dots$ are to be determined. These digits are obtained as the integer parts when successively multiplying c with β: Set $p_{-1} = c$, and

$$p_k \cdot \beta = c_k\beta + p_{k-1}, \quad k = -1, -2, -3, \dots. \tag{2.2.4}$$

Since a finite fraction in a number system with base α usually does not correspond to a finite fraction in the number system with base β, rounding of the result is usually needed.

When converting by hand between the decimal system and the binary system, all computations are made in the decimal system ($\alpha = 10$ and $\beta = 2$). It is then more convenient to convert the decimal number first to octal or hexadecimal, from which the binary representation easily follows. If, on the other hand, the conversion is carried out on a binary computer, the computations are made in the binary system ($\alpha = 2$ and $\beta = 10$).

Example 2.2.3.

Convert the decimal number 176.524 to ternary form (base $\beta = 3$). For the integer part we get $176/3 = 58$ with remainder 2; $58/3 = 19$ with remainder 1; $19/3 = 6$ with remainder 1; $6/3 = 2$ with remainder 0; $2/3 = 0$ with remainder 2. It follows that $(176)_{10} = (20112)_3$.

For the fractional part we compute $.524 \cdot 3 = 1.572$, $.572 \cdot 3 = 1.716$, $.716 \cdot 3 = 2.148, \dots$. Continuing in this way we obtain $(.524)_{10} = (.112010222\dots)_3$. The finite decimal fraction does not correspond to a finite fraction in the ternary number system.

2.2.2 Fixed- and Floating-Point Representation

A computer is, in general, built to handle pieces of information of a fixed size called a **word**. The number of digits in a word (usually binary) is called the **word-length** of the computer.

Typical word-lengths are 32 and 64 bits. A real or integer number is usually stored in a word. Integers can be exactly represented, provided that the word-length suffices to store all the digits in its representation.

In the first generation of computers calculations were made in a **fixed-point** number system; i.e., real numbers were represented with a fixed number of t binary digits in the fractional part. If the word-length of the computer is $s + 1$ bits (including the sign bit), then only numbers in the interval $I = [-2^{s-t}, 2^{s-t}]$ are permitted. Some common fixed-point conventions are $t = s$ (fraction convention) and $t = 0$ (integer convention). This limitation causes difficulties, since even when $x \in I$, $y \in I$, we can have $x - y \notin I$ or $x/y \notin I$.

In a fixed-point number system one must see to it that all numbers, even intermediate results, remain within I. This can be attained by multiplying the variables by appropriate **scale factors**, and then transforming the equations accordingly. This is a tedious process. Moreover it is complicated by the risk that if the scale factors are chosen carelessly, certain intermediate results can have many leading zeros which can lead to poor accuracy in the final results. As a consequence, current numerical analysis literature rarely deals with other than floating-point arithmetic. In scientific computing fixed-point representation is mainly limited to computations with integers, as in subscript expressions for vectors and matrices.

On the other hand, fixed-point computations can be much faster than floating-point, especially since modern microprocessors have superscalar architectures with several fixed-point units but only one floating-point unit. In computer graphics, fixed-point is used almost exclusively once the geometry is transformed and clipped to the visible window. Fixed-point square roots and trigonometric functions are also pretty quick and are easy to write.

By a **normalized floating-point representation** of a real number a, we mean a representation in the form

$$a = \pm m \cdot \beta^e, \quad \beta^{-1} \le m < 1, \quad e \text{ an integer}. \tag{2.2.5}$$

Such a representation is possible for all real numbers a and is unique if $a \neq 0$. (The number zero is treated as a special case.) Here the fraction part m is called the **mantissa**[31] or **significand**), e is the **exponent**, and β the **base** (also called the **radix**).

In a computer, the number of digits for e and m is limited by the word-length. Suppose that t digits are used to represent m. Then we can only represent floating-point numbers of the form

$$\bar{a} = \pm \overline{m} \cdot \beta^e, \quad \overline{m} = (0.d_1 d_2 \cdots d_t)_\beta, \quad 0 \le d_i < \beta, \tag{2.2.6}$$

where $\overline{m}$ is the mantissa m rounded to t digits, and the exponent is limited to a finite range

$$e_{\min} \le e \le e_{\max}. \tag{2.2.7}$$

A floating-point number system F is characterized by the base β, the precision t, and the numbers $e_{\min}$, $e_{\max}$. Only a finite set F of rational numbers can be represented in the form (2.2.6). The numbers in this set are called **floating-point numbers**. Since $d_1 \neq 0$ this set contains, including the number zero, precisely

$$2(\beta - 1)\beta^{t-1}(e_{\max} - e_{\min} + 1) + 1$$

[31] Strictly speaking, "mantissa" refers to the decimal part of a logarithm.

numbers. (Show this!) The limited number of digits in the exponent implies that a is limited in magnitude to an interval which is called the **range** of the floating-point system. If a is larger in magnitude than the largest number in the set F, then a cannot be represented at all (**exponent spill**). The same is true, in a sense, of numbers smaller than the smallest nonzero number in F.

Example 2.2.4.

Consider the floating-point number system for $\beta = 2, t = 3, e_{\min} = -1$, and $e_{\max} = 2$. The positive normalized numbers in the corresponding set F are shown in Figure 2.2.1. The set F contains exactly $2 \cdot 16 + 1 = 33$ numbers. In this example the nonzero numbers of smallest magnitude that can be represented are $(0.100)_2 \cdot 2^{-1} = \frac{1}{4}$ and the largest is $(0.111)_2 \cdot 2^2 = \frac{7}{2}$.

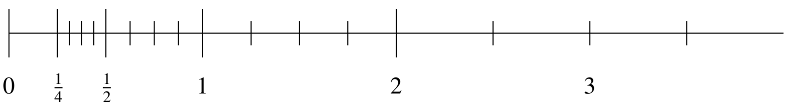

Figure 2.2.1. *Positive normalized numbers when $\beta = 2$, $t = 3$, and $-1 \le e \le 2$.*

Notice that floating-point numbers are not equally spaced; the spacing jumps by a factor of β at each power of β. This wobbling is smallest for $\beta = 2$.

Definition 2.2.1.

The spacing of floating-point numbers is characterized by the **machine epsilon**, *which is the distance ϵ_M from* 1.0 *to the next larger floating-point number.*

The leading significant digit of numbers represented in a number system with base β has been observed to closely fit a logarithmic distribution; i.e., the proportion of numbers with leading digit equal to n is $\ln_\beta(1 + 1/n)$ $(n = 0, 1, \dots, \beta - 1)$. It has been shown that, under this assumption, taking the base equal to 2 will minimize the mean square representation error. A discussion of this intriguing fact, with historic references, is found in Higham [199, Sec. 2.7].

Even if the operands in an arithmetic operation are floating-point numbers in F, the *exact* result of the operation may not be in F. For example, the exact product of two floating-point t-digit numbers has $2t$ or $2t - 1$ digits.

If a real number a is in the range of the floating-point system, the obvious thing to do is to represent a by $\bar{a} = fl\,(a)$, where $fl\,(a)$ denotes a number in F which is nearest to a. This corresponds to rounding of the mantissa m, and according to Sec. 2.1.3, we have

$$|\overline{m} - m| \le \frac{1}{2}\beta^{-t}.$$

(There is one exception. If $|m|$ after rounding should be raised to 1, then $|\overline{m}|$ is set equal to 0.1 and e is raised by 1.) Since $m \ge 0.1$ this means that the magnitude of the relative error in $\bar{a}$ is at most equal to

$$\frac{\frac{1}{2}\beta^{-t}\cdot\beta^{e}}{m\cdot\beta^{e}} \le \frac{1}{2}\beta^{-t+1}.$$

Even with the exception mentioned above this relative bound still holds. (If chopping is used, this doubles the error bound above.) This proves the following theorem.

Theorem 2.2.2.

In a floating-point number system $F = F(\beta, t, e_{\min}, e_{\max})$ every real number in the floating-point range of F can be represented with a relative error, which does not exceed the **unit roundoff** *u, which is defined by*

$$u = \begin{cases} \frac{1}{2}\beta^{-t+1} & \textit{if rounding is used,} \\ \beta^{-t+1} & \textit{if chopping is used.} \end{cases} \tag{2.2.8}$$

Note that in a floating-point system both large and small numbers are represented with nearly *the same relative precision*. The quantity u is, in many contexts, a natural unit for relative changes and relative errors. For example, termination criteria in iterative methods usually depend on the unit roundoff.

To measure the difference between a floating-point number and the real number it approximates we shall occasionally use "**unit in last place**" or **ulp**. We shall often say that "the quantity is perturbed by a few ulps." For example, if in a decimal floating-point system the number 3.14159 is represented as $0.3142 \cdot 10^1$, this has an error of 0.41 ulps.

Example 2.2.5.

Sometimes it is useful to be able to approximately determine the unit roundoff in a program at run time. This may be done using the observation that $u \approx \mu$, where μ is *the smallest floating-point number x for which $fl\,(1 + x) > 1$*. The following program computes a number μ which differs from the unit roundoff u by at most a factor of 2:

```
$x := 1$;
while $1 + x > 1$    $x := x/2$; end;
$\mu := x$;
```

One reason why u does not exactly equal μ is that so-called double rounding may occur. This is when a result is first rounded to extended format and then to the target precision.

A floating-point number system can be extended by including **denormalized numbers** (also called subnormal numbers). These are numbers with the minimum exponent and with the most significant digit equal to zero. The three numbers

$$(.001)_2 2^{-1} = 1/16, \qquad (.010)_2 2^{-1} = 2/16, \qquad (.011)_2 2^{-1} = 3/16$$

can then also be represented (see Figure 2.2.2). Because the representations of denormalized numbers have initial zero digits these have fewer digits of precision than normalized numbers.

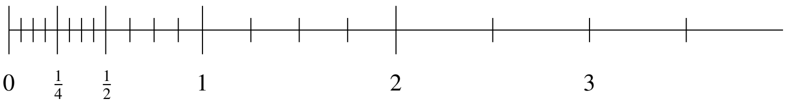

Figure 2.2.2. *Positive normalized and denormalized numbers when $\beta = 2$, $t = 3$, and $-1 \le e \le 2$.*

2.2.3 IEEE Floating-Point Standard

Actual computer implementations of floating-point representations may differ in detail from the one given above. Although some pocket calculators use a floating-point number system with base $\beta = 10$, almost all modern computers use base $\beta = 2$. Most current computers now conform to the IEEE 754 standard for binary floating-point arithmetic.[32] This standard from 1985 (see [205]), which is the result of several years' work by a subcommittee of the IEEE, is now implemented on almost all chips used for personal computers and workstations. There is also a standard IEEE 854 for radix independent floating-point arithmetic [206]. This is used with base 10 by several hand calculators.

The IEEE 754 standard specifies basic and extended formats for floating-point numbers, elementary operations and rounding rules available, conversion between different number formats, and binary–decimal conversion. The handling of exceptional cases like exponent overflow or underflow and division by zero are also specified.

Two main basic formats, single and double precision, are defined using 32 and 64 bits, respectively. In **single precision** a floating-point number a is stored as the sign s (one bit), the exponent e (8 bits), and the mantissa m (23 bits). In **double precision** 11 of the 64 bits are used for the exponent, and 52 bits are used for the mantissa. The value v of a is in the normal case

$$v = (-1)^s (1.m)_2 2^e, \quad -e_{\min} \le e \le e_{\max}. \tag{2.2.9}$$

Note that the digit before the binary point is always 1 for a normalized number. Thus the normalization of the mantissa is different from that in (2.2.6). This bit is not stored (the hidden bit). In that way one bit is gained for the mantissa. A biased exponent is stored and no sign bit used for the exponent. In single precision $e_{\min} = -126$ and $e_{\max} = 127$, and $e + 127$ is stored.

The unit roundoff equals

$$u = \begin{cases} 2^{-24} \approx 5.96 \cdot 10^{-8} & \text{in single precision,} \\ 2^{-53} \approx 1.11 \cdot 10^{-16} & \text{in double precision.} \end{cases}$$

(The machine epsilon is twice as large.) The largest number that can be represented is approximately $2.0 \cdot 2^{127} \approx 3.4028 \times 10^{38}$ in single precision and $2.0 \cdot 2^{1023} \approx 1.7977 \times 10^{308}$ in double precision. The smallest normalized number is $1.0 \cdot 2^{-126} \approx 1.1755 \times 10^{-38}$ in single precision and $1.0 \cdot 2^{-1022} \approx 2.2251 \times 10^{-308}$ in double precision.

An exponent $e = e_{\min} - 1$ and $m \neq 0$ signifies the denormalized number

$$v = (-1)^s (0.m)_2 2^{e_{\min}}.$$

[32] W. Kahan, University of California, Berkeley, was given the Turing Award by the Association of Computing Machinery for his contribution to this standard.

The smallest denormalized number that can be represented is $2^{-126-23} \approx 1.4013 \cdot 10^{-45}$ in single precision and $2^{-1022-52} \approx 4.9407 \cdot 10^{-324}$ in double precision.

There are distinct representations for $+0$ and -0. ± 0 is represented by a sign bit, the exponent $e_{\min} - 1$, and a zero mantissa. Comparisons are defined so that $+0 = -0$. One use of a signed zero is to distinguish between positive and negative underflowed numbers. Another use occurs in the computation of complex elementary functions; see Sec. 2.2.4.

Infinity is also signed and $\pm\infty$ is represented by the exponent $e_{\max} + 1$ and a zero mantissa. When overflow occurs the result is set to $\pm\infty$. This is safer than simply returning the largest representable number, which may be nowhere near the correct answer. The result $\pm\infty$ is also obtained from the illegal operations $a/0$, where $a \neq 0$. The infinity symbol obeys the usual mathematical conventions such as $\infty + \infty = \infty$, $(-1) \times \infty = -\infty$, $a/\infty = 0$.

The IEEE standard also includes two extended precision formats that offer extra precision and exponent range. The standard only specifies a lower bound on how many extra bits it provides.[33] Most modern processors use 80-bit registers for processing real numbers and store results as 64-bit numbers according to the IEEE double precision standard. Extended formats simplify tasks such as computing elementary functions accurately in single or double precision. Extended precision formats are used also by hand calculators. These will often display 10 decimal digits but use 13 digits internally—"*the calculator knows more than it shows.*"

The characteristics of the IEEE formats are summarized in Table 2.2.1. (The hidden bit in the mantissa accounts for the $+1$ in the table. Note that double precision satisfies the requirements for the single extended format, so three different precisions suffice.)

Table 2.2.1. *IEEE floating-point formats.*

	Format	t	e	$e_{\min}$	$e_{\max}$
Single	32 bits	$23 + 1$	8 bits	-126	127
Single extended	$\geq$ 43 bits	≥ 32	$\geq$ 11 bits	≤ -1022	≥ 1023
Double	64 bits	$52 + 1$	11 bits	-1022	1023
Double extended	$\geq$ 79 bits	≥ 64	$\geq$ 15 bits	$\leq -16{,}382$	$\geq 16{,}383$

Example 2.2.6.

Although the exponent range of the floating-point formats seems reassuringly large, even simple programs can quickly give exponent spill. If $x_0 = 2$, $x_{n+1} = x_n^2$, then already $x_{10} = 2^{1024}$ is larger than what IEEE double precision permits. One should also be careful in computations with factorials, e.g., $171! \approx 1.24 \cdot 10^{309}$ is larger than the largest double precision number.

Four rounding modes are supported by the standard. The default rounding mode is round to the nearest representable number, with round to even in the case of a tie. (Some computers, in case of a tie, round away from zero, i.e., raise the absolute value of the

[33]Hardware implementation of extended precision normally does not use a hidden bit, so the double extended format uses 80 bits rather than 79.

number, because this is easier to realize technically.) Chopping is also supported as well as directed rounding to ∞ and to $-\infty$. The latter mode simplifies the implementation of interval arithmetic; see Sec. 2.5.3.

The standard specifies that all arithmetic operations should be performed as if they were first calculated to infinite precision and then rounded to a floating-point number according to one of the four modes mentioned above. This also includes the square root and conversion between an integer and a floating-point. The standard also requires that the conversion between internal formats and decimal be correctly rounded. This can be achieved using extra **guard digits** in the intermediate result of the operation before normalization and rounding. Using a single guard digit, however, will not always ensure the desired result. However, by introducing a second guard digit and a third sticky bit (the logical OR of all succeeding bits) the rounded exact result can be computed at only a slightly higher cost (Goldberg [158]). One reason for specifying precisely the results of arithmetic operations is to improve the portability of software. If a program is moved between two computers, both supporting the IEEE standard, intermediate results should be the same.

IEEE arithmetic is a closed system; that is, every operation, even mathematically invalid operations, such as $0/0$ or $\sqrt{-1}$, produces a result. To handle exceptional situations without aborting the computations some bit patterns (see Table 2.2.2) are reserved for special quantities like NaN ("Not a Number") and ∞. NaNs (there are more than one NaN) are represented by $e = e_{\max} + 1$ and $m \neq 0$.

Table 2.2.2. *IEEE* 754 *representation.*

Exponent	Mantissa	Represents
$e = e_{\min} - 1$	$m = 0$	± 0
$e = e_{\min} - 1$	$m \neq 0$	$\pm 0.m \cdot 2^{e_{\min}}$
$e_{\min} < e < e_{\max}$		$\pm 1.m \cdot 2^{e}$
$e = e_{\max} + 1$	$m = 0$	$\pm\infty$
$e = e_{\max} + 1$	$m \neq 0$	NaN

Note that the gap between zero and the smallest normalized number is $1.0 \times 2^{e_{\min}}$. This is much larger than for the spacing $2^{-t+1} \times 2^{e_{\min}}$ for the normalized numbers for numbers just larger than the underflow threshold; compare Figure 2.2.1. With denormalized numbers the spacing becomes more regular and permits what is called **gradual underflow**. This makes many algorithms well behaved close to the underflow threshold also. Another advantage of having gradual underflow is that it makes it possible to preserve the property

$$x = y \quad \Leftrightarrow \quad x - y = 0$$

as well as other useful relations. Several examples of how denormalized numbers make writing reliable floating-point code easier are analyzed by Demmel [94].

One illustration of the use of extended precision is in converting between IEEE 754 single precision and decimal. The converted single precision number should ideally be converted with enough digits so that when it is converted back the binary single precision number is recovered. It might be expected that since $2^{24} < 10^8$, eight decimal digits in the converted number would suffice. But it can be shown that nine decimal digits are needed to recover the binary number uniquely (see Goldberg [158, Theorem 15] and Problem 2.2.4). When converting back to binary form a rounding error as small as one ulp will give the wrong answer. To do this conversion efficiently, extended single precision is needed.[34]

A NaN is generated by operations such as $0/0$, $+\infty + (-\infty)$, $0 \times \infty$, and $\sqrt{-1}$. A NaN compares unequal with everything including itself. (Note that $x \neq x$ is a simple way to test if x equals a NaN.) When a NaN and an ordinary floating-point number are combined the result is the same as the NaN operand. A NaN is also often used for uninitialized or missing data.

Exceptional operations also raise a flag. The default is to set a flag and continue, but it is also possible to pass control to a trap handler. The flags are "sticky" in that they remain set until explicitly cleared. This implies that without a log file everything before the last setting is lost, which is why it is always wise to use a trap handler. There is one flag for each of the following five exceptions: underflow, overflow, division by zero, invalid operation, and inexact. For example, by testing the flags, it is possible to test if an overflow is genuine or the result of division by zero.

Because of cheaper hardware and increasing problem sizes, double precision is used more and more in scientific computing. With increasing speed and memory becoming available, bigger and bigger problems are being solved and actual problems may soon require more than IEEE double precision! When the IEEE 754 standard was defined no one expected computers able to execute more than 10^{12} floating-point operations per second.

2.2.4 Elementary Functions

Although the square root is included, the IEEE 754 standard does not deal with the implementation of other familiar elementary functions, such as the exponential function exp, the natural logarithm log, and the trigonometric and hyperbolic functions sin, cos, tan, sinh, cosh, tanh, and their inverse functions. With the IEEE 754 standard more accurate implementations are possible which in many cases give almost correctly rounded exact results. To always guarantee correctly rounded exact results sometimes requires computing many more digits than the target accuracy (cf. the tablemaker's dilemma) and therefore is in general too costly. It is also important to preserve monotonicity, e.g., $0 \leq x \leq y \leq \pi/2 \Rightarrow \sin x \leq \sin y$, and range restrictions, e.g., $\sin x \leq 1$, but these demands may conflict with rounded exact results!

The first step in computing an elementary function is to perform a **range reduction**. To compute trigonometric functions, for example, $\sin x$, an additive range reduction is first performed, in which a reduced argument x^*, $-\pi/4 \leq x^* \leq \pi/4$, is computed by finding an integer k such that

$$x^* = x - k\pi/2 \quad (\pi/2 = 1.57079\,63267\,94896\,61923\ldots).$$

(Quantities such as $\pi/2$, $\log(2)$, that are often used in standard subroutines, are listed in decimal form to 30 digits and octal form to 40 digits in Hart et al. [187, Appendix C] and to

[34]It should be noted that some computer languages do not include input/output routines, but these are developed separately. This can lead to double rounding, which spoils the carefully designed accuracy in the IEEE 754 standard. (Some banks use separate routines with chopping even today—you may guess why!)

40 and 44 digits in Knuth [230, Appendix A].) Then $\sin x = \pm \sin x^*$ or $\sin x = \pm \cos x^*$, depending on if k mod 4 equals 0, 1, 2, or 3. Hence approximation for $\sin x$ and $\cos x$ need only be provided for $0 \le x \le \pi/4$. If the argument x is very large, then cancellation in the range reduction can lead to poor accuracy; see Example 2.3.7.

To compute $\log x$, $x > 0$, a multiplicative range reduction is used. If an integer k is determined such that

$$x^* = x/2^k, \quad x^* \in [1/2, 1],$$

then $\log x = \log x^* + k \cdot \log 2$.

To compute the exponential function $\exp(x)$ an integer k is determined such that

$$x^* = x - k \log 2, \quad x^* \in [0, \log 2] \quad (\log 2 = 0.69314\,71805\,59945\,30942\ldots).$$

It then holds that $\exp(x) = \exp(x^*) \cdot 2^k$ and hence we only need an approximation of $\exp(x)$ for the range $x \in [0, \log 2]$.

Coefficients of polynomial and rational approximations suitable for software implementations are tabulated in Hart et al. [187] and Cody and Waite [75]. But approximation of functions can now be simply obtained using software such as Maple [65]. For example, in Maple the commands

```
Digits = 40; minimax(exp(x), x = 0..1, [i,k],1,'err')
```

mean that we are looking for the coefficients of the minimax approximation of the exponential function on [0, 1] by a rational function with numerator of degree i and denominator of degree k with weight function 1, and that the variable `err` should be equal to the approximation error. The coefficients are to be computed to 40 decimal digits. A trend now is for elementary functions to be increasingly implemented in hardware. Hardware implementations are discussed by Muller [272]. Carefully implemented algorithms for elementary functions are available from `www.netlib.org/fdlibm` in the library package fdlibm (Freely Distributable Math. Library) developed by Sun Microsystems and used by MATLAB.

Example 2.2.7.

On a computer using IEEE double precision arithmetic the roundoff unit is $u = 2^{-53} \approx 1.1 \cdot 10^{-16}$. One wishes to compute $\sinh x$ with good *relative* accuracy, both for small and large $|x|$, at least moderately large. Assume that e^x is computed with a relative error less than u in the given interval. The formula $(e^x - e^{-x})/2$ for $\sinh x$ is sufficiently accurate except when $|x|$ is very small and cancellation occurs. Hence for $|x| \ll 1$, e^x and e^{-x} and hence $(e^x - e^{-x})/2$ can have *absolute* errors of order of magnitude (say) u. Then the *relative* error in $(e^x - e^{-x})/2$ can have magnitude $\approx u/|x|$; for example, this is more than 100% for $x \approx 10^{-16}$.

For $|x| \ll 1$ one can instead use (say) two terms in the series expansion for $\sinh x$,

$$\sinh x = x + x^3/3! + x^5/5! + \ldots.$$

Then one gets an absolute truncation error which is about $x^5/120$, and a roundoff error of the order of $2u|x|$. Thus the formula $x + x^3/6$ is better than $(e^x - e^{-x})/2$ if

$$|x|^5/120 + 2u|x| < u.$$

If $2u|x| \ll u$, we have $|x|^5 < 120u = 15 \cdot 2^{-50}$, or $|x| < 15^{1/5} \cdot 2^{-10} \approx 0.00168$ (which shows that $2u|x|$ really could be ignored in this rough calculation). Thus, if one switches from $(e^x - e^{-x})/2$ to $x + x^3/6$ for $|x| < 0.00168$, the relative error will nowhere exceed $u/0.00168 \approx 0.66 \cdot 10^{-13}$. If one needs higher accuracy, one can take more terms in the series, so that the switch can occur at a larger value of $|x|$.

For very large values of $|x|$ one must expect a relative error of order of magnitude $|xu|$ because of roundoff error in the argument x. Compare the discussion of range reduction in Sec. 2.2.4 and Problem 2.2.13.

For complex arguments the elementary functions have discontinuous jumps across when the argument crosses certain branch cuts in the complex plane. They are represented by functions which are single-valued excepts for certain straight lines called **branch cuts**. Where to put these branch cuts and the role of IEEE arithmetic in making these choices are discussed by Kahan [217].

Example 2.2.8.

The function $\sqrt{x}$ is multivalued and there is no way to select the values so the function is continuous over the whole complex plane. If a branch cut is made by excluding all real negative numbers from consideration the square root becomes continuous. Signed zero provides a way to distinguish numbers of the form $x + i(+0)$ and $x + i(-0)$ and to select one or the other side of the cut.

To test the implementation of elementary functions, a Fortran package ELEFUNT has been developed by Cody [73]. This checks the quality using identities like $\cos x = \cos(x/3)(4\cos^2(x/3) - 1)$. For complex elementary functions a package CELEFUNT serves the same purpose; see Cody [74].

2.2.5 Multiple Precision Arithmetic

Hardly any quantity in the physical world is known to an accuracy beyond IEEE double precision. A value of π correct to 20 decimal digits would suffice to calculate the circumference of a circle around the sun at the orbit of the Earth to within the width of an atom. There seems to be little need for multiple precision calculations. Occasionally, however, one may want to perform some calculations, for example, the evaluation of some mathematical constant (such as π and Euler's constant γ) or elementary functions, to very high precision.[35] Extremely high precision is sometimes needed in experimental mathematics when trying to discover new mathematical identities. Algorithms which may be used for these purposes include power series, continued fractions, solutions of equations with Newton's method, or other superlinearly convergent methods.

For performing such tasks it is convenient to use arrays to represent numbers in a floating-point form with a large base and a long mantissa, and to have routines for performing floating-point operations on such numbers. In this way it is possible to *simulate arithmetic of arbitrarily high precision* using standard floating-point arithmetic.

[35] In October 1995 Yasumasa Kanada of the University of Tokyo computed π to 6,442,458,938 decimals on a Hitachi supercomputer; see [11].

Brent [46, 45] developed the first major such multiple precision package in Fortran 66. His package represents multiple precision numbers as arrays of integers and operates on them with integer arithmetic. It includes subroutines for multiple precision evaluation of elementary functions. A more recent package called MPFUN, written in Fortran 77 code, is that of Bailey [9]. In MPFUN a multiple precision number is represented as a vector of single precision floating-point numbers with base 2^{24}. Complex multiprecision numbers are also supported. There is also a Fortran 90 version of this package [10], which is easy to use.

A package of MATLAB m-files called **Mulprec** for computations in, principally, unlimited precision floating-point, has been developed by the first-named author. Documentation of Mulprec and the m-files can be downloaded from the homepage of this book at `www.siam.org/books/ot103` together with some examples of its use.

Fortran routines for high precision computation are also provided in Press et al. [294, Sec. 20.6], and are also supported by symbolic manipulation systems such as Maple [65] and Mathematica [382]; see Online Appendix C.

Review Questions

2.2.1 What base β is used in the binary, octal, and hexadecimal number systems?

2.2.2 Show that any *finite* decimal fraction corresponds to a binary fraction that eventually is periodic.

2.2.3 What is meant by a normalized floating-point representation of a real number?

2.2.4 (a) How large can the maximum relative error be in representation of a real number a in the floating-point system $F = F(\beta, p, e_{\min}, e_{\max})$? It is assumed that a is in the range of F.

(b) How are the quantities "machine epsilon" and "unit roundoff" defined?

2.2.5 What are the characteristics of the IEEE single and double precision formats?

2.2.6 What are the advantages of including denormalized numbers in the IEEE standard?

2.2.7 Give examples of operations that give NaN as their result.

Problems and Computer Exercises

2.2.1 Which rational numbers can be expressed with a finite number of binary digits to the right of the binary point?

2.2.2 (a) Prove the algorithm for conversion between number systems given in Sec. 2.2.1.

(b) Give the hexadecimal form of the decimal numbers 0.1 and 0.3. What error is incurred in rounding these numbers to IEEE 754 single and double precision?

(c) What is the result of the computation $0.3/0.1$ in IEEE 754 single and double precision?

2.2.3 (Kahan) An (over)estimate of u can be obtained for almost any computer by evaluating $|3 \times (4/3 - 1) - 1|$ using rounded floating-point for every operation. Test this on a calculator or computer available to you.

2.2.4 (Goldberg [158]) The binary single precision numbers in the half-open interval $[10^3, 1024)$ have 10 bits to the left and 14 bits to the right of the binary point. Show that there are $(2^{10} - 10^3) \cdot 2^{14} = 393{,}216$ such numbers, but only $(2^{10} - 10^3) \cdot 10^4 = 240{,}000$ decimal numbers with eight decimal digits in the same interval. Conclude that eight decimal digits are not enough to uniquely represent single precision binary numbers in the IEEE 754 standard.

2.2.5 Suppose one wants to compute the power A^n of a square matrix A, where n is a positive integer. To compute $A^{k+1} = A \cdot A^k$ for $k = 1 : n - 1$ requires $n - 1$ matrix multiplications. Show that the number of multiplications can be reduced to less than $2\lfloor \log_2 n \rfloor$ by converting n into binary form and successively squaring $A^{2k} = (A^k)^2$, $k = 1 : \lfloor \log_2 n \rfloor$.

2.2.6 Give in decimal representation: (a) $(10000)_2$; (b) $(100)_8$; (c) $(64)_{16}$; (d) $(FF)_{16}$; (e) $(0.11)_8$; (f) the largest positive integer which can be written with 31 binary digits (answer with one significant digit).

2.2.7 (a) Show how the following numbers are stored in the basic single precision format of the IEEE 754 standard: 1.0; −0.0625; 250.25; 0.1.

(b) Give in decimal notation the largest and smallest positive numbers which can be stored in this format.

2.2.8 (Goldberg [158, Theorem 7]) When $\beta = 2$, if m and n are integers with $m < 2^{p-1}$ (p is the number of bits in the mantissa) and n has the special form $n = 2^i + 2^j$, then $fl((m/n) \times n) = m$ provided that floating-point operations are exactly rounded to nearest. The sequence of possible values of n starts with 1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 16, 17. Test the theorem on your computer for these numbers.

2.2.9 Let pi be the closest floating-point number to π in double precision IEEE 754 standard. Find a sufficiently accurate approximation to π from a table and show that $\pi - pi \approx 1.2246 \cdot 10^{-16}$. What value do you get on your computer for $\sin \pi$?

2.2.10 (Edelman) Let x, $1 \le x < 2$, be a floating-point number in IEEE double precision arithmetic. Show that $fl(x \cdot fl(1/x))$ is either 1 or $1 - \epsilon_M/2$, where $\epsilon_M = 2^{-52}$ (the machine epsilon).

2.2.11 (N. J. Higham) Let a and b be floating-point numbers with $a \le b$. Show that the inequalities $a \le fl((a + b)/2) \le b$ can be violated in base 10 arithmetic. Show that $a \le fl(a + (b - a)/2) \le b$ in base β arithmetic.

2.2.12 (Muller) A rational approximation of $\tan x$ in $[-\pi/4, \pi/4]$ is

$$r(x) = \frac{(0.99999\,99328 - 0.09587\,5045x^2)x}{1 - (0.42920\,9672 + 0.00974\,3234x^2)x^2}.$$

Determine the approximate maximum error of this approximation by comparing with the function on your system on 100 equidistant points in $[0, \pi/4]$.

2.2.13 (a) Show how on a binary computer the exponential function can be approximated by first performing a range reduction based on the relation $e^x = 2^y$, $y = x/\log 2$, and then approximating 2^y on $y \in [0, 1/2]$.

(b) Show that since 2^y satisfies $2^{-y} = 1/2^y$ a rational function $r(y)$ approximating 2^y should have the form

$$r(y) = \frac{q(y^2) + ys(y^2)}{q(y^2) - ys(y^2)},$$

where q and s are polynomials.

(c) Suppose the $r(y)$ in (b) is used for approximating 2^y with

$$q(y) = 20.81892\,37930\,062 + y,$$
$$s(y) = 7.21528\,91511\,493 + 0.05769\,00723\,731y.$$

How many additions, multiplications, and divisions are needed in this case to evaluate $r(y)$? Investigate the accuracy achieved for $y \in [0, 1/2]$.

2.3 Accuracy and Rounding Errors

2.3.1 Floating-Point Arithmetic

It is useful to have a model of how the basic floating-point operations are carried out. If x and y are two floating-point numbers, we denote by

$$fl\,(x + y), \quad fl\,(x - y), \quad fl\,(x \times y), \quad fl\,(x/y)$$

the results of floating addition, subtraction, multiplication, and division, which the machine stores in memory (after rounding or chopping). In what follows we will assume that underflow or overflow does not occur, and that the following **standard model** for the arithmetic holds.

Definition 2.3.1.

Assume that $x, y \in F$. Then in the **standard model** *it holds that*

$$fl\,(x \text{ op } y) = (x \text{ op } y)(1 + \delta), \quad |\delta| \le u, \tag{2.3.1}$$

where u is the unit roundoff and "op" stands for one of the four elementary operations: $+$, $-$, $\times$, and $/$.

The standard model holds with the default rounding mode for computers implementing the IEEE 754 standard. In this case we also have

$$fl\,(\sqrt{x}) = \sqrt{x}(1 + \delta), \quad |\delta| \le u. \tag{2.3.2}$$

If a guard digit is lacking, then instead of (2.3.1) only the weaker model

$$fl\,(x \text{ op } y) = x(1 + \delta_1) \text{ op } y(1 + \delta_2), \quad |\delta_i| \le u, \tag{2.3.3}$$

holds for addition/subtraction. The lack of a guard digit is a serious drawback and can lead to damaging inaccuracy caused by cancellation. Many algorithms can be proved to work satisfactorily only if the standard model (2.3.1) holds. We remark that on current computers multiplication is as fast as addition/subtraction. Division usually is five to ten times slower than a multiply, and a square root about twice as slow as division.

Some earlier computers lack a guard digit in addition/subtraction. Notable examples are several pre-1995 models of Cray computers (Cray 1,2, X-MP,Y-MP, and C90), which were designed to have the highest possible floating-point performance. The IBM 360, which used a hexadecimal system, lacked a (hexadecimal) guard digit between 1964 and 1967. The consequences turned out to be so intolerable that a guard digit had to be retrofitted.

Sometimes the floating-point computation is more precise than what the standard model assumes. An obvious example is that when the exact value x op y can be represented as a floating-point number there is no rounding error at all.

Some computers can perform a *fused multiply–add* operation, i.e., an expression of the form $a \times x + y$ can be evaluated with just one instruction and there is only *one rounding error* at the end:

$$fl\,(a \times x + y) = (a \times x + y)(1 + \delta), \quad |\delta| \le u.$$

Fused multiply–add can be used to advantage in many algorithms. For example, Horner's rule to evaluate the polynomial $p(x) = a_0x^n + a_1x^{n-1} + \cdots + a_{n-1}x + a_n$, which uses the recurrence relation $b_0 = a_0$, $b_i = b_{i-1} \cdot x + a_i$, $i = 1 : n$, needs only n fused multiply–add operations.

It is important to realize that floating-point operations have, to some degree, other properties than the exact arithmetic operations. Floating-point addition and multiplication are commutative but not associative, and the distributive law also fails for them. This makes the analysis of floating-point computations quite difficult.

Example 2.3.1.

To show that associativity does not, in general, hold for floating addition, consider adding the three numbers

$$a = 0.1234567 \cdot 10^0, \qquad b = 0.4711325 \cdot 10^4, \qquad c = -b$$

in a decimal floating-point system with $t = 7$ digits in the mantissa. The following scheme indicates how floating-point addition is performed:

$$fl\,(b + c) = 0, \qquad fl\,(a + fl\,(b + c)) = a = 0.1234567 \cdot 10^0$$

$$\begin{array}{rcr|r} a &=& 0.0000123 & 4567 \cdot 10^4 \\ +b &=& 0.4711325 & \cdot 10^4 \\ \hline fl\,(a+b) &=& 0.4711448 & \cdot 10^4 \\ c &=& -0.4711325 & \cdot 10^4 \end{array}.$$

The last four digits to the right of the vertical line are lost by **outshifting**, and

$$fl\,(fl\,(a + b) + c) = 0.0000123 \cdot 10^4 = 0.1230000 \cdot 10^0 \ne fl\,(a + fl\,(b + c)).$$

An interesting fact is that, assuming a guard digit is used, *floating-point subtraction of two sufficiently close numbers is always exact.*

Lemma 2.3.2 (*Sterbenz* [333]).
Let the floating-point numbers x and y satisfy

$$y/2 \le x \le 2y.$$

Then $fl(x - y) = x - y$, unless $x - y$ underflows.

Proof. By the assumption the exponent of x and y in the floating-point representations of x and y can differ by at most one unit. If the exponent is the same, then the exact result will be computed. Therefore, assume the exponents differ by one. After scaling and, if necessary, interchanging x and y, it holds that $x/2 \le y \le x < 2$ and the exact difference $z = x - y$ is of the form

$$\begin{array}{rcl} x & = & x_1.x_2 \dots x_t \\ y & = & \underline{0\,.y_1 \dots y_{t-1}y_t} \quad . \\ z & = & z_1.z_2 \dots z_t z_{t+1} \end{array}$$

But from the assumption, $x/2 - y \le 0$ or $x - y \le y$. Hence we must have $z_1 = 0$, and thus after shifting the exact result is also obtained in this case. $\square$

With gradual underflow, as in the IEEE 754 standard, the condition that $x - y$ does not underflow can be dropped.

Example 2.3.2.

A corresponding result holds for any base β. For example, using four-digit floating decimal arithmetic we get with guard digit

$$fl\,(0.1000 \cdot 10^1 - 0.9999) = 0.0001 = 1.000 \cdot 10^{-4}$$

(exact), but without guard digit

$$fl\,(0.1000 \cdot 10^1 - 0.9999) = (0.1000 - 0.0999)10^1 = 0.0001 \cdot 10^1 = 1.000 \cdot 10^{-3}.$$

The last result satisfies (2.3.3) with $|\delta_i| \le 0.5 \cdot 10^{-3}$ since $0.10005 \cdot 10^1 - 0.9995 = 10^{-3}$.

Outshiftings are common causes of loss of information that may lead to **catastrophic cancellation** later, in the computations of a quantity that one would have liked to obtain with good relative accuracy.

Example 2.3.3.

An example where the result of Lemma 2.3.2 can be used to advantage is in computing compounded interest. Consider depositing the amount c every day in an account with an interest rate i compounded daily. With the accumulated capital, the total at the end of the year equals

$$c[(1 + x)^n - 1]/x, \quad x = i/n \ll 1,$$

and $n = 365$. Using this formula does not give accurate results. The reason is that a rounding error occurs in computing $fl(1 + x) = 1 + \bar{x}$ and low order bits of x are lost. For example, if $i = 0.06$, then $i/n = 0.0001643836$; in decimal arithmetic using six digits when this is added to one we get $fl(1 + i/n) = 1.000164$, and thus four low order digits are lost.

The problem then is to accurately compute $(1+x)^n = \exp(n \log(1 + x))$. The formula

$$\log(1 + x) = \begin{cases} x & \text{if } fl\,(1 + x) = 1, \\ x\dfrac{\log(1 + x)}{(1 + x) - 1} & \text{otherwise} \end{cases} \tag{2.3.4}$$

can be shown to yield accurate results when $x \in [0, 3/4]$ and the computed value of $\log(1 + x)$ equals the exact result rounded; see Goldberg [158, p. 12].

To check this formula we recall that the base e of the natural logarithm can be defined by the limit

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

In Figure 2.3.1 we show computed values, using double precision floating-point arithmetic, of the sequence $|(1 + 1/n)^n - e|$ for $n = 10^p$, $p = 1 : 14$. More precisely, the expression was computed as

$$|\exp(n \log(1 + 1/n)) - \exp(1)|.$$

The smallest difference $3 \cdot 10^{-8}$ occurs for $n = 10^8$, for which about half the number of bits in $x = 1/n$ are lost. For larger values of n rounding errors destroy the convergence. But using (2.3.4) we obtain correct results for all values of n! (The Maclaurin series

$$\log(1 + x) = x - x^2/2 + x^3/3 - x^4/4 + \cdots$$

will also give good results.)

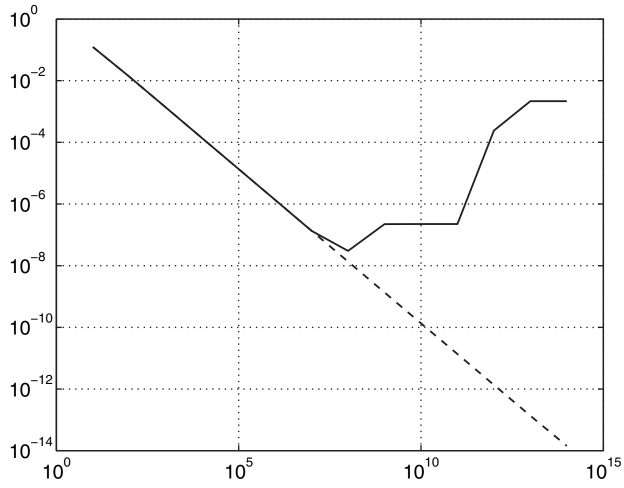

Figure 2.3.1. *Computed values for $n = 10^p$, $p = 1 : 14$, of the sequences: solid line $|(1 + 1/n)^n - e|$; dashed line $|\exp(n \log(1 + 1/n)) - e|$ using* (2.3.4).

A fundamental insight from the above examples can be expressed in the following way:

> "mathematically equivalent" formulas or algorithms are not in general "numerically equivalent."

This adds a new dimension to calculations in finite precision arithmetic and it will be a recurrent theme in the analysis of algorithms in this book.

By **mathematical equivalence** of two algorithms we mean here that the algorithms give exactly the same results from the same input data, if the computations are made without rounding error ("with infinitely many digits"). One algorithm can then, as a rule, formally be derived from the other using the rules of algebra for real numbers, and with the help of mathematical identities. Two algorithms are **numerically equivalent** if their respective floating-point results using the same input data are the same.

In error analysis for compound arithmetic expressions based on the standard model (2.3.1), one often needs an upper bound for quantities of this form:

$$\epsilon \equiv |(1+\delta_1)(1+\delta_2)\cdots(1+\delta_n) - 1|, \quad |\delta_i| \le u, \quad i = 1:n.$$

Then $\epsilon \le (1+u)^n - 1$. Assuming that $nu < 1$, an elementary calculation gives

$$\begin{aligned}(1+u)^n - 1 &= nu + \frac{n(n-1)}{2!}u^2 + \cdots + \binom{n}{k}u^k + \cdots \\ &< nu\Big(1 + \frac{nu}{2} + \cdots + \Big(\frac{nu}{2}\Big)^{k-1} + \cdots\Big) = \frac{nu}{1 - nu/2}.\end{aligned} \tag{2.3.5}$$

Similarly, it can be shown that $(1-u)^{-n} - 1 < nu/(1-nu)$, and the following useful result follows (Higham [199, Lemma 3.1]).

Lemma 2.3.3.

Let $|\delta_i| \le u$, $\rho_i = \pm 1$, i = 1:n, and set

$$\prod_{i=1}^{n}(1+\delta_i)^{\rho_i} = 1 + \theta_n. \tag{2.3.6}$$

If $nu < 1$, then $|\theta_n| < \gamma_n$, where

$$\gamma_n = nu/(1-nu). \tag{2.3.7}$$

Complex arithmetic can be reduced to real arithmetic. Let $x = a + ib$ and $y = c + id$ be two complex numbers. Then we have

$$\begin{aligned} x \pm y &= a \pm c + i(b \pm d), \\ x \times y &= (ac - bd) + i(ad + bc), \\ x/y &= \frac{ac + bd}{c^2 + d^2} + i\frac{bc - ad}{c^2 + d^2}. \end{aligned} \tag{2.3.8}$$

Using the above formula, complex addition (subtraction) needs two real additions, and multiplying two complex numbers requires four real multiplications.

Lemma 2.3.4.

Assume that the standard model (2.3.1) *for floating-point arithmetic holds. Then, provided that no overflow or underflow occurs, no denormalized numbers are produced and the complex operations computed according to* (2.3.8) *satisfy*

$$\begin{aligned} fl\,(x \pm y) &= (x \pm y)(1+\delta), \quad |\delta| \le u, \\ fl\,(x \times y) &= x \times y(1+\delta), \quad |\delta| \le \sqrt{5}u, \\ fl\,(x/y) &= x/y(1+\delta), \quad |\delta| \le \sqrt{2}\gamma_4, \end{aligned} \tag{2.3.9}$$

where δ is a complex number and γ_n is defined in (2.3.7).

Proof. See Higham [199, Sec. 3.6]. The result for complex multiplication is due to Brent et al. [48]. □

The square root of a complex number $u + iv = \sqrt{x+iy}$ is given by

$$u = \left(\frac{r+x}{2}\right)^{1/2}, \qquad v = \left(\frac{r-x}{2}\right)^{1/2}, \quad r = \sqrt{x^2+y^2}. \tag{2.3.10}$$

When $x > 0$ there will be cancellation when computing v, which can be severe if also $|x| \gg |y|$ (cf. Sec. 2.3.4). To avoid this we note that $uv = \sqrt{r^2 - x^2}/2 = y/2$, and thus v can be computed from $v = y/(2u)$. When $x < 0$ we instead compute v from (2.3.10) and set $u = y/(2v)$.

Most rounding error analyses given in this book are formulated for real arithmetic. Since the bounds in Lemma 2.3.4 are of the same form as the standard model for real arithmetic, these can simply be extended to complex arithmetic.

In some cases it may be desirable to avoid complex arithmetic when working with complex matrices. This can be achieved in a simple way by replacing the complex matrices and vectors by real ones of twice the order. Suppose that a complex matrix $A \in \mathbf{C}^{n\times n}$ and a complex vector $z \in \mathbf{C}^n$ are given, where

$$A = B + iC, \qquad z = x + iy,$$

with real B, C, x, and y. Form the real matrix $\tilde{A} \in \mathbf{R}^{2n\times 2n}$ and real vector $\tilde{z} \in \mathbf{R}^{2n}$ defined by

$$\tilde{A} = \begin{pmatrix} B & -C \\ C & B \end{pmatrix}, \qquad \tilde{z} = \begin{pmatrix} x \\ y \end{pmatrix}.$$

It is easy to verify the following rules:

$$\widetilde{(Az)} = \tilde{A}\tilde{z}, \qquad \widetilde{(AB)} = \tilde{A}\tilde{B}, \qquad \widetilde{(A^{-1})} = (\tilde{A})^{-1}.$$

Thus we can solve complex-valued matrix problems using algorithms for the real case. But this incurs a penalty in storage and arithmetic operations.

2.3.2 Basic Rounding Error Results

We now use the notation of Sec. 2.3.1 and the standard model of floating-point arithmetic (Definition 2.3.1) to carry out rounding error analysis of some basic computations. Most, but not all, results are still true if only the weaker bound (2.3.3) holds for addition and subtraction. Note that $fl\,(x \text{ op } y) = (x \text{ op } y)(1+\delta)$, $|\delta| \le u$, can be interpreted for multiplication to mean that $fl\,(x \cdot y)$ is the *exact result* of $x \cdot y(1+\delta)$ for some δ, $|\delta| \le u$. In the same way, the results using the three other operations can be interpreted as *the result of exact operations where the operands have been perturbed by a relative amount which does not exceed u*. In **backward error analysis** (see Sec. 2.4.4) one applies the above interpretation step by step backward in an algorithm.

By repeated use of the formula (2.3.1) in the case of multiplication, one can show that

$$fl\,(x_1 x_2 \cdots x_n) = x_1 x_2(1+\delta_2)x_3(1+\delta_3)\cdots x_n(1+\delta_n),$$
$$|\delta_i| \le u, \quad i = 2:n$$

holds; i.e., the computed **product** $fl\,(x_1 x_2 \cdots x_n)$ is *exactly* equal to a product of the factors

$$\tilde{x}_1 = x_1, \quad \tilde{x}_i = x_i(1+\delta_i), \quad i = 2:n.$$

Using the estimate and notation of (2.3.7) it follows from this analysis that

$$|fl\,(x_1 x_2 \cdots x_n) - x_1 x_2 \cdots x_n| < \gamma_{n-1}|x_1 x_2 \cdots x_n|, \tag{2.3.11}$$

which bounds the **forward error** of the computed result.

For a **sum** of n floating-point numbers similar results can be derived. If the sum is computed in the natural order, we have

$$\begin{aligned} &fl\,(\cdots(((x_1+x_2)+x_3)+\cdots+x_n)) \\ &\qquad = x_1(1+\delta_1)+x_2(1+\delta_2)+\cdots+x_n(1+\delta_n), \end{aligned}$$

where

$$|\delta_1| < \gamma_{n-1}, \qquad |\delta_i| < \gamma_{n+1-i}, \quad i = 2:n,$$

and thus the computed sum is *the exact sum* of the numbers $x_i(1+\delta_i)$. This also gives an estimate of the forward error

$$\begin{aligned} &|fl\,(\cdots(((x_1+x_2)+x_3)+\cdots+x_n)) - (x_1+x_2+x_3+\cdots+x_n)| \\ &\qquad < \sum_{i=1}^{n} \gamma_{n+1-i}|x_i| \le \gamma_{n-1}\sum_{i=1}^{n}|x_i|, \end{aligned} \tag{2.3.12}$$

where the last upper bound holds independent of the summation order.

Notice that to minimize the first upper bound in equation (2.3.12), the terms *should be added in increasing order of magnitude*. For large n an even better bound can be shown if the summation is done using the divide and conquer technique described in Sec. 1.2.3; see Problem 2.3.5.

Example 2.3.4.

Using a hexadecimal machine ($\beta = 16$), with $t = 6$ and chopping ($u = 16^{-5} \approx 10^{-6}$), we computed

$$\sum_{n=1}^{10,000} n^{-2} \approx 1.644834$$

in two different orders. Using the natural summation order $n = 1, 2, 3, \dots$ the error was $1.317 \cdot 10^{-3}$. Summing in the opposite order $n = 10{,}000, 9{,}999, 9{,}998, \dots$ the error was reduced to $2 \cdot 10^{-6}$. This was not unexpected. Each operation is an addition, where the partial sum s is increased by n^{-2}. Thus, in each operation, one commits an error of about $s \cdot u$, and all these errors are added. Using the first summation order, we have $1 \le s \le 2$ in every step, but using the other order of summation we have $s < 10^{-2}$ in 9,900 of the 10,000 additions.

Similar bounds for roundoff errors can easily be derived for basic vector and matrix operations; see Wilkinson [377, pp. 114–118]. For an **inner product** $x^T y$ computed in the natural order we have

$$fl\,(x^T y) = x_1 y_1(1 + \delta_1) + x_2 y_2(1 + \delta_2) + \cdots + x_n y_n(1 + \delta_n),$$

where

$$|\delta_1| < \gamma_n, \quad |\delta_r| < \gamma_{n+2-i}, \quad i = 2 : n.$$

The corresponding forward error bound becomes

$$|fl\,(x^T y) - x^T y| < \sum_{i=1}^{n} \gamma_{n+2-i} |x_i||y_i| < \gamma_n \sum_{i=1}^{n} |x_i||y_i|.$$

If we let $|x|$, $|y|$ denote vectors with elements $|x_i|$, $|y_i|$ the last estimate can be written in the simple form

$$|fl\,(x^T y) - x^T y| < \gamma_n |x^T||y|. \tag{2.3.13}$$

This bound is independent of the summation order and also holds for the weaker model (2.3.3) valid with no guard digit rounding.

The outer product of two vectors $x, y \in \mathbf{R}^n$ is the matrix $xy^T = (x_i y_j)$. In floating-point arithmetic we compute the elements $fl\,(x_i y_j) = x_i y_j(1 + \delta_{ij})$, $\delta_{ij} \le u$, and thus

$$|fl\,(xy^T) - xy^T| \le u\,|xy^T|. \tag{2.3.14}$$

This is a satisfactory result for many purposes, but the computed result is not in general a rank one matrix and it is not possible to find Δx and Δy such that $fl(xy^T) = (x + \Delta x)(x + \Delta y)^T$.

The product of two t-digit floating-point numbers can be exactly represented with at most $2t$ digits. This allows inner products to be computed in extended precision without much extra cost. If fl_e denotes computation with extended precision and u_e the corresponding unit roundoff, then the forward error bound for an inner product becomes

$$|fl\,(fl_e(x^T y)) - x^T y| < u|x^T y| + \frac{nu_e}{1 - nu_e/2}(1 + u)|x^T||y|, \tag{2.3.15}$$

where the first term comes from the final rounding. If $|x^T||y| \le u|x^T y|$, then the computed inner product is almost as accurate as the correctly rounded exact result. These accurate inner products can be used to improve accuracy by so-called *iterative refinement* in many linear algebra problems. But since computations in extended precision are machine dependent it has been difficult to make such programs portable.[36] The recent development of Extended and Mixed Precision BLAS (Basic Linear Algebra Subroutines; see [247]) may now make this more feasible.

Similar error bounds can easily be obtained for matrix multiplication. Let $A \in \mathbf{R}^{m\times n}$, $B \in \mathbf{R}^{n\times p}$, and denote by $|A|$ and $|B|$ matrices with elements $|a_{ij}|$ and $|b_{ij}|$. Then it holds that

$$|fl\,(AB) - AB| < \gamma_n |A||B|, \tag{2.3.16}$$

where the inequality is to be interpreted elementwise. Often we shall need bounds for some norm of the error matrix. From (2.3.16) it follows that

$$\|fl\,(AB) - AB\| < \gamma_n \|\,|A|\,\|\,\|\,|B|\,\|. \tag{2.3.17}$$

Hence, for the 1-norm, ∞-norm, and the Frobenius norm we have

$$\|fl\,(AB) - AB\| < \gamma_n \|A\|\,\|B\|, \tag{2.3.18}$$

but unless A and B have only nonnegative elements, we get for the 2-norm only the weaker bound

$$\|fl\,(AB) - AB\|_2 < n\gamma_n \|A\|_2\,\|B\|_2. \tag{2.3.19}$$

To reduce the effects of rounding errors in computing a sum $\sum_{i=1}^n x_i$ one can use **compensated summation**. In this algorithm the rounding error in each addition is estimated and then compensated for with a correction term. Compensated summation can be useful when a large number of small terms are to be added as in numerical quadrature. Another example is the case in the numerical solution of initial value problems for ordinary differential equations. Note that in this application the terms have to be added in the order in which they are generated.

Compensated summation is based on the possibility to *simulate double precision floating-point addition in single precision arithmetic*. To illustrate the basic idea we take, as in Example 2.3.1,

$$a = 0.1234567 \cdot 10^0, \qquad b = 0.4711325 \cdot 10^4,$$

so that $s = fl\,(a+b) = 0.4711448 \cdot 10^4$. Suppose we form

$$c = fl\,(fl\,(b-s)+a) = -0.1230000 \cdot 10^0 + 0.1234567 \cdot 10^0 = 4567000 \cdot 10^{-3}.$$

Note that the variable c is computed without error and picks up the information that was lost in the operation $fl\,(a+b)$.

[36]It was suggested that the IEEE 754 standard should require inner products to be precisely specified, but that did not happen.

Algorithm 2.2. *Compensated Summation.*

The following algorithm uses compensated summation to accurately compute the sum $\sum_{i=1}^{n} x_i$:

```
$s := x_1$;  $c := 0$;
for $i = 2 : n$
    $y := c + x_i$;
    $t := s + y$;
    $c := (s - t) + y$;
    $s := t$;
end
```

It can be proved (see Goldberg [158]) that on binary machines with a guard digit the computed sum satisfies

$$s = \sum_{i=1}^{n}(1 + \xi_i)x_i, \quad |\xi_i| < 2u + O(nu^2). \tag{2.3.20}$$

This formulation is a typical example of a backward error analysis; see Sec. 2.4.4. The first term in the error bound is independent of n.

2.3.3 Statistical Models for Rounding Errors

The bounds for the accumulated rounding error we have derived so far are estimates of the **maximal error**. These bounds ignore the sign of the errors and tend to be much too pessimistic when the number of variables is large. However, they can still give valuable insight into the behavior of a method and be used for the purpose of comparing different methods.

An alternative is a statistical analysis of rounding errors, which is based on the assumption that rounding errors are independent and have some statistical distribution. It was observed already in the 1950s that rounding errors occurring in the solution of differential equations *are not random and are often strongly correlated.* This does not in itself preclude that useful information can sometimes be obtained by modeling them by random uncorrelated variables! In many computational situations and scientific experiments, where the error can be considered to have arisen from the addition of a large number of *independent* error sources of about the same magnitude, an assumption that the errors are normally distributed is justified.

Example 2.3.5.

Figure 2.3.2 illustrates the effect of rounding errors on the evaluation of two different expressions for the polynomial $p(x) = (x - 1)^5$ for $x \in [0.999, 1.001]$, in IEEE double precision (unit roundoff $u = 1.1 \cdot 10^{-16}$). Among other things it shows that the monotonicity of a function can be lost due to rounding errors. The model of rounding errors as independent random variables works well in this example. It is obvious that it would be impossible to locate the zero of $p(x)$ to a precision better than about $(0.5 \cdot 10^{-14})^{1/6} \approx 0.0007$ using the

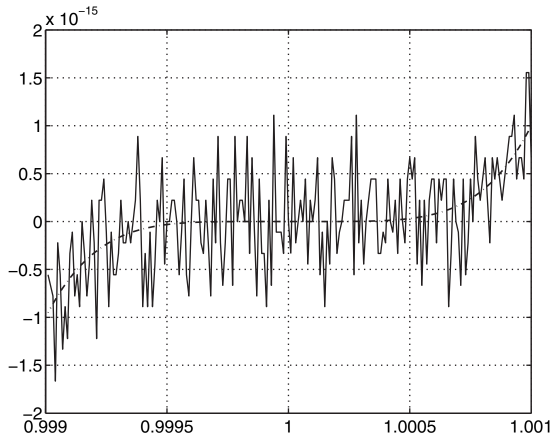

Figure 2.3.2. *Calculated values of a polynomial near a multiple root: solid line* $p(x) = x^5 - 5x^4 + 10x^3 - 10x^2 + 5x - 1 = 0$*; dashed line* $p(x) = (x-1)^5$.

expanded form of $p(x)$. But using the expression $p(x) = (1-x)^5$ function values can be evaluated with constant *relative* precision even close to $x = 1$, and the problem disappears!

This example shows that although multiple roots are in general ill-conditioned, an important exception is when the function $f(x)$ *is given in such a form that it can be computed with less absolute error as x approaches* α.

The theory of **standard error** is based on probability theory and will not be treated in detail here. The standard error of an estimate of a given quantity is the same as the *standard deviation of its sampling distribution.*

If in a sum $y = \sum_{i=1}^{n} x_i$ each x_i has error $|\Delta_i| \le \delta$, then the maximum error bound for y is $n\delta$. Thus, *the maximal error grows proportionally to* n. If n is large—for example, $n = 1000$—then it is in fact highly improbable that the real error will be anywhere near $n\delta$, since that bound is attained only when every Δx_i has the same sign and the same maximal magnitude. Observe, though, that if positive numbers are added, each of which has been abridged to t decimals by chopping, then each Δx_i has the same sign and a magnitude which is on average $\frac{1}{2}\delta$, where $\delta = 10^{-t}$. Thus, the real error is often about 500δ.

If the numbers are rounded instead of chopped, and if one can assume that the errors in the various terms are stochastically independent with standard deviation ϵ, then the standard error in y becomes (see Theorem 2.4.5)

$$(\epsilon^2 + \epsilon^2 + \cdots + \epsilon^2)^{1/2} = \epsilon\sqrt{n}.$$

Thus the standard error of the sum grows only proportionally to $\sqrt{n}$. This supports the rule of thumb, suggested by Wilkinson [376, p. 26], that *if a rounding error analysis gives a bound* $f(n)u$ *for the maximum error, then one can expect the real error to be of size* $\sqrt{f(n)}u$.

If $n \gg 1$, then the error in y is, under the assumptions made above, approximately normally distributed with standard deviation $\sigma = \epsilon\sqrt{n}$. The corresponding frequency function,

$$f(t) = \frac{1}{\sqrt{2\pi}} e^{-t^2/2},$$

is illustrated in Figure 2.3.3; the curve shown there is also called the Gauss curve. The assumption that the error is normally distributed with standard deviation σ means, for example, that the statement "the magnitude of the error is greater than 2σ" (see the shaded area of Figure 2.3.3) is true in only about 5% of all cases (the unshaded area under the curve). More generally, the assertion that the magnitude of the error is larger than σ, 2σ, and 3σ respectively, is about 32%, 5%, and 0.27%.

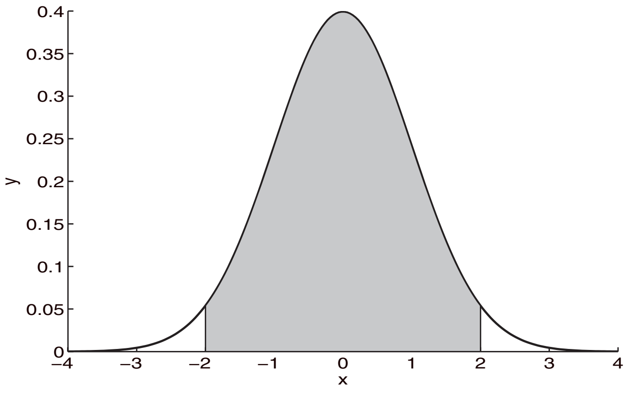

Figure 2.3.3. *The frequency function of the normal distribution for* $\sigma = 1$.

One can show that if the individual terms in a sum $y = \sum_{i=1}^{n} x_i$ have a **uniform** probability distribution in the interval $[-\frac{1}{2}\delta, \frac{1}{2}\delta]$, then the standard deviation of an individual term is $\delta/12$. Therefore, in only about 5% of cases is the error in the sum of 1000 terms greater than $2\delta\sqrt{1000/12} \approx 18\delta$, which can be compared to the maximum error 500δ. This shows that rounding can be far superior to chopping when a statistical interpretation (especially the assumption of independence) can be given to the principal sources of errors. Observe that, in the above, we have only considered the propagation of errors which were present in the original data, and have ignored the effect of possible roundoff errors in the additions themselves.

For rounding errors the formula for standard errors is used. For systematic errors, however, the formula for maximal error (2.4.5) should be used.

2.3.4 Avoiding Overflow and Cancellation

In the rare cases when input and output data are so large or small in magnitude that the range of the machine is not sufficient, one can use higher precision or else work with logarithms or some other transformation of the data. One should, however, keep in mind the risk that *intermediate results* in a calculation can produce an exponent which is too large (overflow) or too small (underflow) for the floating-point system of the machine. Different actions may be taken in such situations, as well for division by zero. Too small an exponent is usually, but not always, unproblematic. If the machine does not signal underflow, but simply sets the result equal to zero, there is a risk, however, of harmful consequences. Occasionally, "unexplainable errors" in output data are caused by underflow somewhere in the computations.

The **Pythagorean sum** $c = \sqrt{a^2 + b^2}$ occurs frequently, for example, in conversion to polar coordinates and in computing the complex modulus and complex multiplication. If the obvious algorithm is used, then damaging underflows and overflows may occur in the squaring of a and b even if a and b and the result c are well within the range of the floating-point system used. This can be avoided by using instead the algorithm: If $a = b = 0$, then $c = 0$; otherwise set $p = \max(|a|, |b|)$, $q = \min(|a|, |b|)$, and compute

$$\rho = q/p; \qquad c = p\sqrt{1+\rho^2}. \tag{2.3.21}$$

Example 2.3.6.

The formula (2.3.8) for complex division suffers from the problem that intermediate results can overflow even if the final result is well within the range of the floating-point system. This problem can be avoided by rewriting the formula as for the Pythagorean sum: If $|c| > |d|$, then compute

$$\frac{a+ib}{c+id} = \frac{a+be}{r} + i\,\frac{b-ae}{r}, \qquad e = \frac{d}{c}, \qquad r = c + de.$$

If $|d| > |c|$, then $e = c/d$ is computed and a corresponding formula used.

Similar precautions are also needed for computing the Euclidean length (norm) of a vector $\|x\|_2 = \left(\sum_{i=1}^n x_i^2\right)^{1/2}$, $x \neq 0$. We could avoid overflows by first finding $x_{max} = \max_{1\le i\le n} |x_i|$ and then forming

$$s = \sum_{i=1}^{n} (x_i/x_{max})^2, \qquad \|x\|_2 = x_{max}\sqrt{s}. \tag{2.3.22}$$

This has the drawback of needing *two passes* through the data.

Algorithm 2.3.

The following algorithm, due to S. J. Hammarling, for computing the Euclidean length of a vector requires only one pass through the data. It is used in the level-1 BLAS routine xNRM2:

```
$t = 0;\ \ s = 1;$
for $i = 1 : n$
    if $|x_i| > 0$
        if $|x_i| > t$
            $s = 1 + s(t/x_i)^2;\ \ t = |x_i|;$
        else
            $s = s + (x_i/t)^2;$
        end
    end
end
$\|x\|_2 = t\sqrt{s};$
```

On the other hand this code does not vectorize and can therefore be slower if implemented on a vector computer.

One very common reason for poor accuracy in the result of a calculation is that somewhere a subtraction has been carried out in which the difference between the operands is considerably less than either of the operands. This causes a loss of relative precision. (Note that, on the other hand, relative precision is preserved in addition of nonnegative quantities, multiplication, and division.)

Consider the computation of $y = x_1 - x_2$, where $\tilde{x}_1 = x_1 + \Delta x_1$, $\tilde{x}_2 = x_2 + \Delta x_2$ are approximations to the exact values. If the operation is carried out exactly the result is $\tilde{y} = y + \Delta y$, where $\Delta y = \Delta x_1 - \Delta x_2$. But, since the errors Δx_1 and Δx_2 can have opposite sign, the best error bound for $\tilde{y}$ is

$$|\Delta y| \le |\Delta x_1| + |\Delta x_2|. \tag{2.3.23}$$

Notice the plus sign! Hence for the relative error we have

$$\left|\frac{\Delta y}{y}\right| \le \frac{|\Delta x_1| + |\Delta x_2|}{|x_1 - x_2|}. \tag{2.3.24}$$

This shows that *there can be very poor relative accuracy in the difference between two nearly equal numbers*. This phenomenon is called **cancellation of terms**.

In Sec. 1.2.1 it was shown that when using the well-known "textbook" formula

$$r_{1,2} = \big(-b \pm \sqrt{b^2 - 4ac}\big)\big/(2a)$$

for computing the real roots of the quadratic equation $ax^2 + bx + c = 0$ $(a \ne 0)$, cancellation could cause a loss of accuracy in the root of smallest magnitude. This can be avoided by computing the root of *smaller magnitude* from the relation $r_1 r_2 = c/a$ between coefficients and roots. The following is a suitable algorithm.

Algorithm 2.4. *Solving a Quadratic Equation.*

```
$d := b^2 - 4ac$;
**if** $d \ge 0$ % real roots
    $r_1 := -\mathrm{sign}(b)\big(|b| + \sqrt{d}\big)/(2a)$;
    $r_2 := c/(a \cdot r_1)$;
**else** % complex roots $x + iy$
    $x := -b/(2a)$;
    $y := \sqrt{-d}/(2a)$;
**end**
```

Note that we define sign $(b) = 1$ if $b \ge 0$, else sign $(b) = -1$.[37] It can be proved that in IEEE arithmetic this algorithm computes *a slightly wrong solution to a slightly wrong problem.*

[37]In MATLAB sign $(0) = 0$, which can lead to failure of this algorithm.

Lemma 2.3.5.

Assume that Algorithm 2.4 *is used to compute the roots* $r_{1,2}$ *of the quadratic equation* $ax^2 + bx + c = 0$. *Denote the computed roots by* $\bar{r}_{1,2}$ *and let* $\tilde{r}_{1,2}$ *be the exact roots of the nearby equation* $ax^2 + bx + \tilde{c} = 0$, *where* $|\tilde{c} - c| \le \gamma_2 |\tilde{c}|$. *Then* $|\tilde{r}_i - \bar{r}_i| \le \gamma_5 |\tilde{r}_i|$, $i = 1, 2$.

Proof. See Kahan [216]. □

More generally, if $|\delta| \ll x$, then one should rewrite

$$\sqrt{x+\delta} - \sqrt{x} = \frac{x+\delta-x}{\sqrt{x+\delta}+\sqrt{x}} = \frac{\delta}{\sqrt{x+\delta}+\sqrt{x}}.$$

There are other exact ways of rewriting formulas which are as useful as the above; for example,

$$\cos(x+\delta) - \cos x = -2\sin(\delta/2)\sin(x+\delta/2).$$

If one cannot find an exact way of rewriting a given expression of the form $f(x+\delta) - f(x)$, it is often advantageous to use one or more terms in the Taylor series

$$f(x+\delta) - f(x) = f'(x)\delta + \frac{1}{2}f''(x)\delta^2 + \cdots$$

Example 2.3.7 (*Cody* [73]).

To compute $\sin 22$ we first find $\lfloor 22/(\pi/2) \rfloor = 14$. It follows that $\sin 22 = -\sin x^*$, where $x^* = 22 - 14(\pi/2)$. Using the correctly rounded 10-digit approximation $\pi/2 = 1.57079\,6327$ we obtain

$$x^* = 22 - 14 \cdot 1.57079\,6327 = 8.85142 \cdot 10^{-3}.$$

Here cancellation has taken place and the reduced argument has a maximal error of $7 \cdot 10^{-9}$. The actual error is slightly smaller since the correctly rounded value is $x^* = 8.85144\,8711 \cdot 10^{-3}$, which corresponds to a relative error in the computed $\sin 22$ of about $2.4 \cdot 10^{-6}$, in spite of using a 10-digit approximation to $\pi/2$.

For very large arguments the relative error can be much larger. Techniques for carrying out accurate range reductions without actually needing multiple precision calculations are discussed by Muller [272]; see also Problem 2.3.9.

In previous examples we got a warning that cancellation would occur, since x_2 was found as the difference between two nearly equal numbers each of which was, relatively, much larger than the difference itself. In practice, one does not always get such a warning, for two reasons: first, in using a computer one has no direct contact with the individual steps of calculation; second, cancellation can be spread over a great number of operations. This may occur in computing a partial sum of an infinite series. For example, in a series where the size of some terms are many orders of magnitude larger than the sum of the series, small relative errors in the computation of the large terms can then produce large errors in the result.

It has been emphasized here that calculations where cancellation occurs should be avoided. But there are cases where one has not been able to avoid it, and there is no time to wait for a better method. Situations occur in practice where (say) the first ten digits are lost, and we need a decent relative accuracy in what will be left.[38] Then, high accuracy is required in intermediate results. This is an instance where the high accuracy in IEEE double precision is needed!

Review Questions

2.3.1 What is the standard model for floating-point arithmetic? What weaker model holds if a guard digit is lacking?

2.3.2 Give examples to show that some of the axioms for arithmetic with real numbers do not always hold for floating-point arithmetic.

2.3.3 (a) Give the results of a backward and forward error analysis for computing $fl\,(x_1 + x_2 + \cdots + x_n)$. It is assumed that the standard model holds.

(b) Describe the idea in compensated summation.

2.3.4 Explain the terms "maximum error" and "standard error." What statistical assumption about rounding errors is often made when calculating the standard error in a sum due to rounding?

2.3.5 Explain what is meant by "cancellation of terms." Give an example of how this can be avoided by rewriting a formula.

Problems and Computer Exercises

2.3.1 Rewrite the following expressions to avoid cancellation of terms:
(a) $1 - \cos x$, $|x| \ll 1$; (b) $\sin x - \cos x$, $|x| \approx \pi/4$

2.3.2 (a) The expression $x^2 - y^2$ exhibits catastrophic cancellation if $|x| \approx |y|$. Show that it is more accurate to evaluate it as $(x + y)(x - y)$.

(b) Consider using the trigonometric identity $\sin^2 x + \cos^2 x = 1$ to compute $\cos x = (1 - \sin^2 x)^{1/2}$. For which arguments in the range $0 \le x \le \pi/4$ will this formula fail to give good accuracy?

2.3.3 The polar representation of a complex number is

$$z = x + iy = r(\sin\phi + \cos\phi) \equiv r \cdot e^{i\phi}.$$

Develop accurate formulas for computing this polar representation from x and y using real operations.

2.3.4 (Kahan) Show that with the use of fused multiply–add the algorithm

$$w = fl\,(b \times c); \quad y := fl\,(a \times d - w); \quad e := fl\,(b \times c - w); \quad z = fl\,(y - e);$$

[38] G. Dahlquist has encountered just this situation in a problem of financial mathematics.

computes with high relative accuracy

$$z = \det \begin{pmatrix} a & b \\ c & d \end{pmatrix} = ad - bc.$$

2.3.5 Suppose that the sum $s = \sum_{i=1}^{n} x_i, n = 2^k$, is computed using the divide and conquer technique described in Sec. 1.2.3. Show that this summation algorithm computes an exact sum

$$\bar{s} = \sum_{i=1}^{n} x_i(1 + \delta_i), \quad |\delta_i| \le \tilde{u} \log_2 n.$$

Hence for large values of n this summation order can be much more accurate than the conventional order.

2.3.6 Show that for the evaluation of a polynomial $p(x) = \sum_{i=0}^{n} a_i x^i$ by Horner's rule the following roundoff error estimate holds:

$$|fl\,(p(x)) - p(x)| < \gamma_1 \sum_{i=0}^{n} (2i + 1)|a_i|\,|x|^i, \quad (2nu \le 0.1).$$

2.3.7 In solving linear equations by Gaussian elimination expressions of the form $s = (c - \sum_{i=1}^{n-1} a_i b_i)/d$ often occur. Show that, by a slight extension of the result in the previous problem, that the computed $\bar{s}$ satisfies

$$\left|\bar{s}d - c + \sum_{i=1}^{n-1} a_i b_i\right| \le \gamma_n \Big(|\bar{s}d| + \sum_{i=1}^{n-1} |a_i||b_i|\Big),$$

where the inequality holds independent of the summation order.

2.3.8 The zeros of the reduced cubic polynomial $z^3 + 3qz - 2r = 0$ can be found from the Cardano–Tartaglia formula:

$$z = \Big(r + \sqrt{q^3 + r^2}\Big)^{1/3} + \Big(r - \sqrt{q^3 + r^2}\Big)^{1/3},$$

where the two cubic roots are to be chosen so that their product equals $-q$. One real root is obtained if $q^3 + r^2 \ge 0$, which is the case unless all three roots are real and distinct.

The above formula can lead to cancellation. Rewrite it so that it becomes more suitable for numerical calculation and requires the calculation of only one cubic root.

2.3.9 (Eldén and Wittmeyer-Koch) In the interval reduction for computing $\sin x$ there can be a loss of accuracy through cancellation in the computation of the reduced argument $x^* = x - k \cdot \pi/2$ when k is large. A way to avoid this without reverting to higher precision has been suggested by Cody and Waite [75]). Write

$$\pi/2 = \pi_0/2 + r,$$

where $\pi_0/2$ is *exactly representable* with a few digits in the (binary) floating-point system. The reduced argument is now computed as $x^* = (x - k \cdot \pi_0/2) - kr$. Here, unless k is very large, the first term can be computed without rounding error. The rounding error in the second term is bounded by $k|r|\,u$, where u is the unit roundoff. In IEEE single precision one takes

$$\pi_0/2 = 201/128 = 1.573125 = (10.1001001)_2, \qquad r = 4.838267949 \cdot 10^{-4}.$$

Estimate the relative error in the computed reduced argument x^* when $x = 1000$ and r is represented in IEEE single precision.

2.3.10 (Kahan [218]) The area A of a triangle with sides equal to a, b, c is given by Heron's formula:

$$A = \sqrt{s(s-a)(s-b)(s-c)}, \quad s = (a+b+c)/2.$$

Show that this formula fails for needle-shaped triangles, using five-digit decimal floating arithmetic and $a = 100.01$, $b = 99.995$, $c = 0.025$.
The following formula can be proved to work if addition/subtraction satisfies (2.3.21): Order the sides so that $a \geq b \geq c$, and use

$$A = \frac{1}{4}\sqrt{(a+(b+c))(c-(a-b))(c+(a-b))(a+(b-c))}.$$

Compute a correct result for the data above using this modified formula. If a person tells you that this gives an imaginary result if $a - b > c$, what do you answer him?

2.3.11 As is well known, $f(x) = (1+x)^{1/x}$ has the limit $e = 2.71828\ 18284\ 59045\ldots$ when $x \to \infty$. Study the sequences $f(x_n)$ for $x_n = 10^{-n}$ and $x_n = 2^{-n}$, for $n = 1, 2, 3, \ldots$. Stop when $x_n < 10^{-10}$ (or when $x_n < 10^{-20}$ if you are using double precision). Give your results as a table of n, x_n, and the relative error $g_n = (f(x_n) - e)/e$. Also plot $\log(|g_n|)$ against $\log(|x_n|)$. Comment on and explain your observations.

2.3.12 (a) Compute the derivative of the exponential function e^x at $x = 0$ by approximating with the difference quotients $(e^{x+h} - e^x)/h$, for $h = 2^{-i}$, $i = 1:20$. Explain your results.

(b) Repeat (a), but approximate with the central difference approximation $(e^{x+h} - e^{x-h})/(2h)$.

2.3.13 The hyperbolic cosine is defined by $\cosh t = (e^t + e^{-t})/2$, and its inverse function $t = \text{arccosh}\,(x)$ is the solution to

$$x = (e^t + e^{-t})/2.$$

Solving the quadratic equation $(e^t)^2 - 2xe^t + 1$, we find $e^t = x \pm (x^2-1)^{1/2}$ and

$$\arccos x = \log(x \pm (x^2-1)^{1/2}).$$

(a) Show that this formula suffers from serious cancellation when the minus sign is used and x is large. Try, e.g., $x = \cosh(10)$ using double precision IEEE. (Using the plus sign will just transfer the problem to negative x.)

(b) A better formula is

$$\arccos x = 2\log\big(((x+1)/2)^{1/2} + ((x-1)/2)^{1/2}\big).$$

This also avoids the squaring of x which can lead to overflow. Derive this formula and show that it is well behaved!

2.3.14 (Gautschi) Euler's constant $\gamma = 0.57721566490153286\ldots$ is defined as the limit

$$\gamma = \lim_{n\to\infty} \gamma_n, \quad \text{where} \quad \gamma_n = 1 + 1/2 + 1/3 + \cdots + 1/n - \log n.$$

Assuming that $\gamma - \gamma_n \sim cn^{-d}$, $n \to \infty$, for some constants c and $d > 0$, try to determine c and d experimentally on your computer.

2.3.15 In the statistical treatment of data, one often needs to compute the quantities

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

If the numbers x_i are the results of statistically independent measurements of a quantity with expected value m, then $\bar{x}$ is an estimate of m, whose standard deviation is estimated by $s/\sqrt{n-1}$.

(a) The computation of $\bar{x}$ and m using the formulas above have the drawback that they require two passes through the data x_i. Let α be *a provisional mean*, chosen as an approximation to $\bar{x}$, and set $x_i' = x_i - \alpha$. Show that the formulas

$$\bar{x} = \alpha + \frac{1}{n}\sum_{i=1}^{n} x_i', \qquad s^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i')^2 - (\bar{x} - \alpha)^2$$

hold for an arbitrary α.

(b) In 16 measurements of a quantity x one got the following results:

i	x_i	i	x_i	i	x_i	i	x_i
1	546.85	5	546.81	9	546.96	13	546.84
2	546.79	6	546.82	10	546.94	14	546.86
3	546.82	7	546.88	11	546.84	15	546.84
4	546.78	8	546.89	12	546.82	16	546.84

Compute $\bar{x}$ and s^2 to two significant digits using $\alpha = 546.85$.

(c) In the computations in (b), one never needed more than three digits. If one uses the value $\alpha = 0$, how many digits are needed in $(x_i')^2$ in order to get two significant digits in s^2? If one uses five digits throughout the computations, why is the cancellation in the s^2 more fatal than the cancellation in the subtraction $x_i' - \alpha$? (One can even get negative values for s^2!)

(d) If we define

$$m_k = \frac{1}{k}\sum_{i=1}^{k} x_i, \qquad q_k = \sum_{i=1}^{k}(x_i - m_k)^2 = \sum_{i=1}^{k} x_i^2 - \frac{1}{k}\Bigl(\sum_{i=1}^{k} x_i\Bigr)^2,$$

then it holds that $\bar{x} = m_n$, and $s^2 = q_n/n$. Show the recursion formulas

$$m_1 = x_1, \qquad m_k = m_{k-1} + (x_k - m_{k-1})/k$$
$$q_1 = 0, \qquad q_k = q_{k-1} + (x_k - m_{k-1})^2(k-1)/k.$$

2.3.16 Compute the sum in Example 2.3.4 using the natural summation ordering in IEEE 754 double precision. Repeat the computations using compensated summation (Algorithm 2.3).

2.4 Error Propagation

2.4.1 Numerical Problems, Methods, and Algorithms

By a **numerical problem** we mean here a clear and unambiguous description of the *functional connection* between **input data**—that is, the "independent variables" in the problem—and **output data**—that is, the desired results. Input data and output data consist of a finite number of real (or complex) quantities and are thus representable by finite dimensional vectors. The functional connection can be expressed in either explicit or implicit form. We require for the following discussion also that *the output data should be uniquely determined and depend continuously on the input data.*

By an **algorithm**[39] for a given numerical problem we mean a *complete description of well-defined operations* through which each permissible input data vector is transformed into an output data vector. By "operations" we mean here arithmetic and logical operations, which a computer can perform, together with references to previously defined algorithms. It should be noted that, as the field of computing has developed, more and more complex functions (for example, square root, circular, and hyperbolic functions) are built into the hardware. In many programming environments operations such as matrix multiplication, solution of linear systems, etc. are considered as "elementary operations" and for the user appear as black boxes.

(The concept algorithm can be analogously defined for problems completely different from numerical problems, with other types of input data and fundamental operations—for example, inflection, merging of words, and other transformations of words in a given language.)

Example 2.4.1.

To determine the largest real root of the cubic equation

$$p(z) = a_0z^3 + a_1z^2 + a_2z + a_3 = 0,$$

with real coefficients a_0, a_1, a_2, a_3, is a numerical problem. The input data vector is (a_0, a_1, a_2, a_3). The output is the desired root x; it is an implicitly defined function of the input data.

An algorithm for this problem can be based on Newton's method, supplemented with rules for how the initial approximation should be chosen and how the iteration process is to be terminated. One could also use other iterative methods, or algorithms based on the formula by Cardano–Tartaglia for the exact solution of the cubic equation (see Problem 2.3.8). Since this uses square roots and cube roots, one needs to assume that algorithms for the computation of these functions have been specified previously.

[39]The term "algorithm" is a Latinization of the name of the Arabic ninth century mathematician Al-Khowârizmî. He also introduced the word "algebra" (Al-jabr). Western Europe became acquainted with the Hindu positional number system from a Latin translation of his book entitled *Algorithmi de Numero Indorum*.

One often begins the construction of an algorithm for a given problem by breaking down the problem into subproblems in such a way that the output data from one subproblem are the input data to the next subproblem. Thus the distinction between problem and algorithm is not always so clear-cut. The essential point is that, in the formulation of the problem, one is only concerned with the initial state and the final state. In an algorithm, however, one should clearly define each step along the way, from start to finish.

We use the term **numerical method** in this book to mean a procedure either to approximate a mathematical problem with a numerical problem or to solve a numerical problem (or at least to transform it to a simpler problem). A numerical method should be more generally applicable than an algorithm, and set lesser emphasis on the completeness of the computational details. The transformation of a differential equation problem to a system of nonlinear equations can be called a numerical method—even without instructions as to how to solve the system of nonlinear equations. Newton's method is a numerical method for determining a root of a large class of nonlinear equations. In order to call it an algorithm, conditions for starting and stopping the iteration process should be added.

For a given numerical problem one can consider many different algorithms. As we have seen in Sec. 2.3 these can, in floating-point arithmetic, give approximations of widely varying accuracy to the exact solution.

Example 2.4.2.

The problem of solving the differential equation

$$\frac{d^2y}{dx^2} = x^2 + y^2$$

with boundary conditions $y(0) = 0$, $y(5) = 1$ is not a numerical problem according to the definition stated above. This is because the output is the *function* y, which cannot in any conspicuous way be specified by a finite number of parameters. The above mathematical problem can be *approximated with a numerical problem* if one specifies the output data to be the values of y for $x = h, 2h, 3h, \dots, 5-h$. Also, the domain of variation of the unknowns must be restricted in order to show that the problem has a unique solution. This can be done, however, and there are a number of different algorithms for solving the problem approximately, which have different properties with respect to the number of arithmetic operations needed and the accuracy obtained.

Before an algorithm can be used it has to be implemented in an algorithmic program language in a reliable and efficient manner. We leave these aspects aside for the moment, but *this is far from a trivial task. Most amateur algorithm writers seem to think that an algorithm is ready at the point where a professional realizes that the hard and tedious work is just beginning* (George E. Forsythe [120]).

2.4.2 Propagation of Errors and Condition Numbers

In scientific computing the given input data are usually imprecise. The errors in the input will propagate and give rise to errors in the output. In this section we develop some general tools for studying the propagation of errors. Error-propagation formulas are also of great interest in the *planning and analysis of scientific experiments.*

Note that rounding errors from each step in a calculation are also propagated to give errors in the final result. For many algorithms a rounding error analysis can be given, which shows that the computed result always equals the exact (or slightly perturbed) result of a nearby problem, where the input data have been slightly perturbed (see, e.g, Lemma 2.3.5). The effect of rounding errors on the final result can then be estimated using the tools of this section.

We first consider two simple special cases of error propagation. For a sum of an arbitrary number of terms we get the following lemma by induction from (2.3.23).

Lemma 2.4.1.

In addition (and subtraction) a bound for the absolute errors in the result is given by the sum of the bounds for the absolute errors of the operands:

$$y = \sum_{i=1}^{n} x_i, \qquad |\Delta y| \le \sum_{i=1}^{n} |\Delta x_i|. \tag{2.4.1}$$

To obtain a corresponding result for the error propagation in multiplication and division, we start with the observations that for $y = \log x$ we have $\Delta(\log x) \approx \Delta(x)/x$. In words, *the relative error in a quantity is approximately equal to the absolute error in its natural logarithm.* This is related to the fact that displacements of the same length at different places on a logarithmic scale mean the same relative change of the value. From this we obtain the following result.

Lemma 2.4.2.

In multiplication and division, an approximate bound for the relative error is obtained by adding the relative errors of the operands. More generally, for $y = x_1^{m_1} x_2^{m_2} \cdots x_n^{m_n}$,

$$\left|\frac{\Delta y}{y}\right| \lessapprox \sum_{i=1}^{n} |m_i| \left|\frac{\Delta x_i}{x_i}\right|. \tag{2.4.2}$$

Proof. The proof follows by differentiating $\log y = m_1 \log x_1 + m_2 \log x_2 + \cdots + m_n \log x_n$. $\square$

Example 2.4.3.

In Newton's method for solving a nonlinear equation a correction is to be calculated as a quotient $\Delta x = f(x_k)/f'(x_k)$. Close to a root the relative error in the computed value of $f(x_k)$ can be quite large due to cancellation. How accurately should one compute $f'(x_k)$, assuming that the work grows as one demands higher accuracy? Since the limit for the relative error in Δx is equal to the sum of the bounds for the relative errors in $f(x_k)$ and $f'(x_k)$, there is no gain in making the relative error in $f'(x_k)$ very much less than the relative error in $f(x_k)$. This observation is of great importance, particularly in the generalization of Newton's method to *systems* of nonlinear equations.

We now study the propagation of errors in more general nonlinear expressions. Consider first the case when we want to compute a function $y = f(x)$ of a single real variable

x. How is the error in x propagated to y? Let $\tilde{x} - x = \Delta x$. Then, a natural way is to approximate $\Delta y = \tilde{y} - y$ with the differential of y. By the mean value theorem, $\Delta y = f(x + \Delta x) - f(x) = f'(\xi)\Delta x$, where ξ is a number between x and $x + \Delta x$. Suppose that $|\Delta x| \le \epsilon$. Then it follows that

$$|\Delta y| \le \max_{\xi} |f'(\xi)|\epsilon, \quad \xi \in [x - \epsilon, x + \epsilon]. \tag{2.4.3}$$

In practice, it is usually sufficient to replace ξ by the available estimate of x. *Even if high precision is needed in the value of $f(x)$, one rarely needs a high relative precision in an error bound or an error estimate.* (In the neighborhood of zeros of the first derivative $f'(x)$ one has to be more careful.)

By the implicit function theorem a similar result holds if y is an implicit function of x defined by $g(x, y) = 0$. If $g(x, y) = 0$ and $\frac{\partial g}{\partial y}(x, y) \ne 0$, then in a neighborhood of x, y there exists a unique function $y = f(x)$ such that $g(x, f(x)) = 0$ and it holds that

$$f'(x) = -\frac{\partial g}{\partial x}(x, f(x)) \Big/ \frac{\partial g}{\partial y}(x, f(x)).$$

Example 2.4.4.

The result in Lemma 2.3.5 does not say whether the computed roots of the quadratic equation are close to the exact roots r_1, r_2. To answer that question we must determine how sensitive the roots are to a relative perturbation in the coefficient c. Differentiating $ax^2 + bx + c = 0$, where $x = x(c)$ with respect to c, we obtain $(2ax + b)dx/dc + 1 = 0$, $dx/dc = -1/(2ax + b)$. With $x = r_1$ and using $r_1 + r_2 = -b/a$, $r_1 r_2 = c/a$ this can be written as

$$\frac{dr_1}{r_1} = -\frac{dc}{c}\frac{r_2}{r_1 - r_2}.$$

This shows that when $|r_1 - r_2| \ll |r_2|$ the roots can be very sensitive to small relative perturbations in c.

When $r_1 = r_2$, that is, when there is a double root, this linear analysis breaks down. Indeed, it is easy to see that the equation $(x - r)^2 - \Delta c = 0$ has roots $x = r \pm \sqrt{\Delta c}$.

To analyze error propagation in a function of several variables $f = f(x_1, \dots, x_n)$, we need the following generalization of the mean value theorem.

Theorem 2.4.3.

Assume that the real-valued function f is differentiable in a neighborhood of the point $x = (x_1, x_2, \dots, x_n)$, and let $x = x + \Delta x$ be a point in this neighborhood. Then there exists a number θ such that

$$\Delta f = f(x + \Delta x) - f(x) = \sum_{i=1}^{n} \frac{\partial f}{\partial x_i}(x + \theta \Delta x)\Delta x_i, \quad 0 \le \theta \le 1.$$

Proof. The proof follows by considering the function $F(t) = f(x + t\Delta x)$, and using the mean value theorem for functions of one variable and the chain rule. $\square$

From Theorem 2.4.3 it follows that the perturbation Δf is approximately equal to the total differential. The use of this approximation means that the function $f(x)$ is, in a neighborhood of x that contains the point $x + \Delta x$, approximated by a linear function. All the techniques of differential calculus, such as logarithmic differentiation and implicit differentiation, may be useful for the calculation of the total differential; see the examples and the problems at the end of this section.

Theorem 2.4.4 (*General Formula for Error Propagation*).

Let the real-valued function $f = f(x_1, x_2, \dots, x_n)$ be differentiable in a neighborhood of the point $x = (x_1, x_2, \dots, x_n)$ with errors $\Delta x_1, \Delta x_2, \dots, \Delta x_n$. Then it holds that

$$\Delta f \approx \sum_{i=1}^{n} \frac{\partial f}{\partial x_i} \Delta x_i. \tag{2.4.4}$$

Then for the maximal error in $f(x_1, \dots, x_n)$ we obtain the approximate upper bound

$$|\Delta f| \lessapprox \sum_{i=1}^{n} \left| \frac{\partial f}{\partial x_i} \right| |\Delta x_i|, \tag{2.4.5}$$

where the partial derivatives are evaluated at x.

In order to get a *strict* bound for $|\Delta f|$, one should use in (2.4.5) the maximum absolute values of the partial derivatives in a neighborhood of the known point x. In most practical situations it suffices to calculate $|\partial f / \partial x_i|$ at x and then add a certain marginal amount (5 to 10 percent, say) for safety. Only if the Δx_i are large or if the derivatives have a large relative variation in the neighborhood of x need the maximal values be used. (The latter situation occurs, for example, in a neighborhood of an extremal point of $f(x)$.)

The bound in Theorem 2.4.4 is the best possible, unless one knows some dependence between the errors of the terms. Sometimes it can, for various reasons, be a gross overestimate of the real error.

Example 2.4.5.

Compute error bounds for $f = x_1^2 - x_2$, where $x_1 = 1.03 \pm 0.01$, $x_2 = 0.45 \pm 0.01$. We obtain

$$\left| \frac{\partial f}{\partial x_1} \right| = |2x_1| \le 2.1, \qquad \left| \frac{\partial f}{\partial x_2} \right| = |-1| = 1,$$

and find $|\Delta f| \le 2.1 \cdot 0.01 + 1 \cdot 0.01 = 0.031$, or $f = 1.061 - 0.450 \pm 0.032 = 0.611 \pm 0.032$. The error bound has been raised 0.001 because of the rounding in the calculation of x_1^2.

One is seldom asked to give mathematically guaranteed error bounds. More often it is satisfactory to give an estimate of the *order of magnitude* of the anticipated error. The bound for $|\Delta f|$ obtained with Theorem 2.4.3 estimates the maximal error, i.e., covers the worst possible cases, where the sources of error Δx_i contribute with the same sign and magnitudes equal to the error bounds for the individual variables.

In practice, the trouble with formula (2.4.5) is that it often gives bounds which are too coarse. More realistic estimates are often obtained using the standard error introduced in

Sec. 2.3.3. Here we give without proof the result for the general case, which can be derived using probability theory and (2.4.4). (Compare with the result for the standard error of a sum given in Sec. 2.3.3.)

Theorem 2.4.5.

Assume that the errors $\Delta x_1, \Delta x_2, \dots, \Delta x_n$ are independent random variables with mean zero and standard deviations $\epsilon_1, \epsilon_2, \dots, \epsilon_n$. Then the standard error ϵ for $f(x_1, x_2, \dots, x_n)$ is given by the formula

$$\epsilon \approx \left(\sum_{i=1}^{n} \left(\frac{\partial f}{\partial x_i} \right)^2 \epsilon_i^2 \right)^{1/2}. \tag{2.4.6}$$

Analysis of error propagation is more than just a means for judging the reliability of calculated results. As remarked above, it has an equally important function as a means for *the planning of a calculation or scientific experiment*. It can help in the choice of algorithm, and in making certain decisions during a calculation. An example of such a decision is the choice of step length during a numerical integration. Increased accuracy often has to be bought at the price of more costly or complicated calculations. One can also shed some light on the degree to which it is advisable to obtain a new apparatus to improve the measurements of a given variable when the measurements of other variables are subject to error as well.

It is useful to have a measure of how sensitive the output data are to small changes in the input data. In general, if "small" changes in the input data can result in "large" changes in the output data, we call the problem **ill-conditioned**; otherwise it is called **well-conditioned**. (The definition of large may differ from problem to problem depending on the accuracy of the data and the accuracy needed in the solution.)

Definition 2.4.6.

Consider a numerical problem $y = f(x) \in R^m$, $x \in R^n$, or in component form $y_j = f_j(x_1, \dots, x_n)$, $j = 1 : m$. Let $\hat{x}$ be fixed and assume that neither $\hat{x}$ nor $\hat{y}0f(\hat{x})$ is zero. The sensitivity of y with respect to small changes in x can be measured by the relative condition number

$$\kappa(f; \hat{x}) = \lim_{\epsilon \to 0} \sup_{\|h\| = \epsilon} \left\{ \frac{\|f(x+h) - f(x)\|}{\|f(x)\|} \Big/ \frac{\|h\|}{\|x\|} \right\}. \tag{2.4.7}$$

We have used a vector norm $\|\cdot\|$ to measure the size of a vector; see Sec. A.4.3 in Online Appendix A. Common vector norms are the p-norms defined by

$$\|x\|_p = (x_1|^p + |x_2|^p + \cdots + |x_n|^p)^{1/p}, \quad 1 \le p < \infty,$$

where one usually takes $p = 1, 2$, or $p = \infty$.

The condition number (2.4.7) is a function of the input data $\hat{x}$ and also depends on the choice of norms in the data space and the solution space. It measures the maximal amount which a given relative perturbation is magnified by the function f, in the limit of infinitely small perturbations. For perturbations of sufficiently small size we have the estimate

$$\|\tilde{y} - y\| \le \kappa\epsilon\|y\| + O(\epsilon^2).$$

We can expect to have roughly $s = log_{10}\kappa$ less significant decimal digits in the solution than in the input data. However, *this may not hold for all components of the output.*

Assume that f has partial derivatives with respect to x_i, $i = 1 : n$, and let $J(x)$ be the **Jacobian matrix**

$$J_{ij}(x) = \frac{\partial f_j(x)}{\partial x_i}, \quad j = 1 : m, \quad i = 1 : n. \tag{2.4.8}$$

Then, for any matrix norm subordinate to the vector norm (see Online Appendix A.3.3), the condition number defined above can be expressed as

$$\kappa(f; \hat{x}) = \frac{\|J(\hat{x})\|\|\hat{x}\|}{\|f(\hat{x})\|}. \tag{2.4.9}$$

For a composite function $g \circ f$ the chain rule for derivatives can be used to show that

$$\kappa(g \circ f; \hat{x}) \le \kappa(g; \hat{y})\kappa(f; \hat{x}). \tag{2.4.10}$$

If the composite function is ill-conditioned we can infer from this that at least one of the functions g and f must be ill-conditioned.

If $y = f(x)$ is a linear (bounded) function $y = Mx$, where $M \in \mathbf{R}^{m\times n}$, then according to (2.4.9)

$$\kappa(M; x) = \|M\|\frac{\|x\|}{\|y\|}.$$

This inequality is sharp in the sense that for any matrix norm and for any M and x there exists a perturbation δb such that equality holds.

If M is a square and invertible matrix, then from $x = M^{-1}y$ we conclude that $\|x\| \le \|M^{-1}\| \, \|y\|$. This gives the upper bound

$$\kappa(M; x) \le \|M\|\|M^{-1}\|, \tag{2.4.11}$$

which is referred to as the condition number of M. For given x (or y), this upper bound may not be achievable for any perturbation of x. The inequality (2.4.11) motivates the following definition.

Theorem 2.4.7.

The **condition number** *for a square nonsingular matrix $M \in \mathbf{R}^{n\times n}$ equals $\kappa(M) = \|M\| \, \|M^{-1}\|$, where $\|\cdot\|$ is a subordinate matrix norm. In particular, for the Euclidean norm*

$$\kappa(M) = \kappa_2(M) = \|M\|_2 \, \|M^{-1}\|_2 = \sigma_1/\sigma_n, \tag{2.4.12}$$

where σ_1 and σ_n are the largest and smallest singular value of M.

The last expression in (2.4.12) follows by the observation that if M has singular values σ_i, $i = 1 : n$, then M^{-1} has singular values $1/\sigma_i$, $i = 1 : n$; see Theorem 1.4.4.

We note some simple properties of $\kappa(M)$. From $(\alpha M)^{-1} = M^{-1}/\alpha$ it follows that $\kappa(\alpha M) = \kappa(M)$; i.e., the condition number is invariant under multiplication of M by a scalar. Matrix norms are submultiplicative, i.e., $\|KM\| \leq \|K\|\,\|M\|$. From the definition and the identity $MM^{-1} = I$ it follows that

$$\kappa(M) = \|M\|_2\|M^{-1}\|_2 \geq \|I\| = 1,$$

i.e., the condition number κ_2 is always greater than or equal to one. The composite mapping of $z = Ky$ and $y = Mx$ is represented by the matrix product KY, and we have

$$\kappa(KM) \leq \kappa(K)\kappa(M).$$

It is important to note that the condition number is a property of the mapping $x \to y$ and does not depend on the algorithm used to evaluate y! An ill-conditioned problem is *intrinsically* difficult to solve accurately using *any* numerical algorithm. Even if the input data are exact, rounding errors made during the calculations in floating-point arithmetic may cause large perturbations in the final result. Hence, in some sense an ill-conditioned problem is not well posed.

Example 2.4.6.

If we get an inaccurate solution to an ill-conditioned problem, then often nothing can be done about the situation. (If you ask a stupid question you get a stupid answer!) But sometimes the difficulty depends on the form one has chosen to represent the input and output data of the problem.

The polynomial

$$P(x) = (x-10)^4 + 0.200(x-10)^3 + 0.0500(x-10)^2 - 0.00500(x-10) + 0.00100$$

is identical with a polynomial Q which, if the coefficients are rounded to six digits, becomes

$$\tilde{Q}(x) = x^4 - 39.8000x^3 + 594.050x^2 - 3941.00x + 9805.05.$$

One finds that $P(10.11) = 0.0015 \pm 10^{-4}$, where only three digits are needed in the computation, while $\tilde{Q}(10.11) = -0.0481 \pm \frac{1}{2} \cdot 10^{-4}$, in spite of the fact that eight digits were used in the computation. The rounding to six digits of the coefficients of Q has thus caused an error in the polynomial's value at $x = 10.11$; the erroneous value is more than 30 times larger than the correct value and has the wrong sign. When the coefficients of Q are input data, the problem of computing the value of the polynomial for $x \approx 10$ is far more ill-conditioned than when the coefficients of P are input data.

The conditioning of a problem can to some degree be illustrated geometrically. A numerical problem P means a mapping of the space X of possible input data onto the space Y of the output data. The dimensions of these spaces are usually quite large. In Figure 2.4.1 we picture a mapping in two dimensions. Since we are considering relative changes, we take the coordinate axis to be logarithmically scaled. A small circle of radius r is mapped onto an ellipse whose ratio of major to minor axis is κr, where κ is the condition number of the problem P.

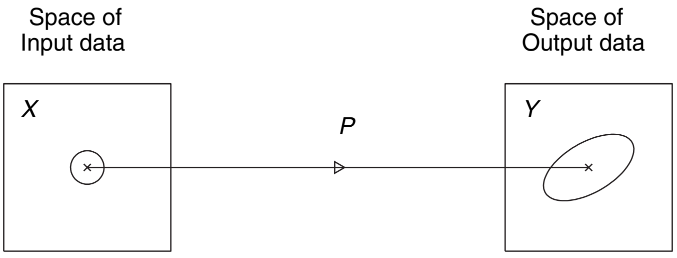

Figure 2.4.1. *Geometrical illustration of the condition number.*

2.4.3 Perturbation Analysis for Linear Systems

Consider the linear system $y = Ax$, where A is nonsingular and $y \neq 0$. From the analysis in the previous section we know that the condition number of the inverse mapping $x = A^{-1}y \neq 0$ is bounded by the condition number

$$\kappa(A^{-1}) = \kappa(A) = \|A^{-1}\|\,\|A\|.$$

Assume that the elements of the matrix A are given data and subject to perturbations δA. The perturbed solution $x + \delta x$ satisfies the linear system

$$(A + \delta A)(x + \delta x) = y.$$

Subtracting $Ax = y$ we obtain $(A + \delta A)\delta x = -\delta A x$. Assuming also that the matrix $(A + \delta A) = A(I + A^{-1}\delta A)$ is nonsingular and solving for δx yields

$$\delta x = -(I + A^{-1}\delta A)^{-1}A^{-1}\delta A x, \tag{2.4.13}$$

which is the basic identity for the analysis. Taking norms gives

$$\|\delta x\| \leq \|(I + A^{-1}\delta A)^{-1}\|\,\|A^{-1}\|\|\delta A\|\,\|x\|.$$

It can be shown (see Problem 2.4.9) that if $\|A^{-1}\delta A\| < 1$, then $A + \delta A$ is nonsingular and

$$\|(I + A^{-1}\delta A)^{-1}\| < 1/(1 - \|A^{-1}\delta A\|).$$

Neglecting second order terms,

$$\frac{\|\delta x\|}{\|x\|} \lessapprox \kappa(A)\frac{\|\delta A\|}{\|A\|}. \tag{2.4.14}$$

This shows that $\kappa(A)$ is also the condition number of $x = A^{-1}y$ with respect to perturbations in A.

For any real, orthogonal matrix Q we have

$$\kappa_2(Q) = \|Q\|_2\|Q^{-1}\|_2 = 1,$$

so Q is perfectly conditioned. By Lemma A.4.1 (see Online Appendix A) we have $\|QAP\|_2 = \|A\|_2$ for any orthogonal P and Q. It follows that

$$\kappa_2(PAQ) = \kappa_2(A),$$

i.e., *the condition number of a matrix A is invariant under orthogonal transformations*. This important fact is one reason why orthogonal transformations play a central role in numerical linear algebra.

How large may κ be before we consider the problem to be ill-conditioned? That depends on the accuracy of the data and the accuracy desired in the solution. If the data have a relative error of 10^{-7}, then we can guarantee a (normwise) relative error in the solution $\leq 10^{-3}$ if $\kappa \leq 0.5 \cdot 10^4$. But to guarantee a (normwise) relative error in the solution $\leq 10^{-6}$ we need to have $\kappa \leq 5$.

Example 2.4.7.

The Hilbert matrix H_n of order n with elements

$$H_n(i, j) = h_{ij} = 1/(i + j - 1), \quad 1 \leq i, j \leq n,$$

is a notable example of an ill-conditioned matrix. In Table 2.4.1 approximate condition numbers of Hilbert matrices of order ≤ 12, computed in IEEE double precision, are given. For $n > 12$ the Hilbert matrices are too ill-conditioned even for IEEE double precision! From a result by G. Szegö (see Gautschi [147, p. 34]) it follows that

$$\kappa_2(H_n) \approx \frac{(\sqrt{2}+1)^{4(n+1)}}{2^{15/4}\sqrt{\pi n}} \sim e^{3.5n},$$

i.e., the condition numbers grow exponentially with n. Although the severe ill-conditioning exhibited by the Hilbert matrices is rare, moderately ill-conditioned linear systems do occur regularly in many practical applications!

Table 2.4.1. *Condition numbers of Hilbert matrices of order* ≤ 12.

n	$\kappa_2(H_n)$	n	$\kappa_2(H_n)$
1	1	7	$4.753 \cdot 10^8$
2	19.281	8	$1.526 \cdot 10^{10}$
3	$5.241 \cdot 10^2$	9	$4.932 \cdot 10^{11}$
4	$1.551 \cdot 10^4$	10	$1.602 \cdot 10^{13}$
5	$4.766 \cdot 10^5$	11	$5.220 \cdot 10^{14}$
6	$1.495 \cdot 10^7$	12	$1.678 \cdot 10^{16}$

The normwise condition analysis in the previous section usually is satisfactory when the linear system is "well scaled." If this is not the case, then a **componentwise** analysis may give sharper bounds. We first introduce some notations. The absolute values $|A|$ and $|b|$ of a matrix A and vector b are interpreted componentwise

$$|A|_{ij} = (|a_{ij}|), \qquad |b|_i = (|b_i|).$$

The **partial ordering** "$\leq$" for the absolute values of matrices $|A|$, $|B|$ and vectors $|b|$, $|c|$ is to be interpreted componentwise:[40]

$$|A| \leq |B| \iff |a_{ij}| \leq |b_{ij}|, \qquad |b| \leq |c| \iff |b_i| \leq |c_i|.$$

[40]Note that $A \leq B$ in other contexts means that $B - A$ is positive semidefinite.

It follows easily that $|AB| \le |A|\,|B|$ and a similar rule holds for matrix-vector multiplication.

Taking absolute values in (2.4.13) gives componentwise error bounds for the corresponding perturbations in x,

$$|\delta x| \le |(I + A^{-1}\delta A)^{-1}|\,|A^{-1}|(|\delta A||x| + |\delta b|).$$

The matrix $(I - |A^{-1}||\delta A|)$ is guaranteed to be nonsingular if $\|\,|A^{-1}|\,|\delta A|\,\| < 1$.

Assume now that we have componentwise bounds for the perturbations in A and b, say

$$|\delta A| \le \omega|A|, \qquad |\delta b| \le \omega|b|. \tag{2.4.15}$$

Neglecting second order terms in ω and using (2.4.15) gives

$$|\delta x| \lessapprox |A^{-1}|(|\delta A||x| + |\delta b|) \le \omega|A^{-1}|(|A|\,|x| + |b|). \tag{2.4.16}$$

Taking norms in (2.4.16) we get

$$\|\delta x\| \lessapprox \omega\|\,|A^{-1}|(|A|\,|x| + |b|)\,\| + O(\omega^2). \tag{2.4.17}$$

The scalar quantity

$$\kappa_{|A|}(A) = \|\,|A^{-1}|\,|A|\,\| \tag{2.4.18}$$

is called the **Bauer–Skeel condition number** of the matrix A.

A different way to examine the sensitivity of various matrix problems is the differentiation of a parametrized matrix. Suppose that λ is a scalar and that $A(\lambda)$ is a matrix with elements $a_{ij}(\lambda)$ that are differentiable functions of λ. Then by the derivative of the matrix $A(\lambda)$ we mean the matrix

$$A'(\lambda) = \frac{d}{d\lambda}A(\lambda) = \left(\frac{da_{ij}}{d\lambda}\right). \tag{2.4.19}$$

Many of the rules for differentiation of scalar functions are easily generalized to differentiation of matrices. For differentiating a product of two matrices there holds

$$\frac{d}{d\lambda}[A(\lambda)B(\lambda)] = \frac{d}{d\lambda}[A(\lambda)]B(\lambda) + A(\lambda)\frac{d}{d\lambda}[B(\lambda)]. \tag{2.4.20}$$

Assuming that $A^{-1}(\lambda)$ exists, using this rule on the identity $A^{-1}(\lambda)A(\lambda) = I$ we obtain

$$\frac{d}{d\lambda}[A^{-1}(\lambda)]A(\lambda) + A^{-1}(\lambda)\frac{d}{d\lambda}[A(\lambda)] = 0,$$

or, solving for the derivative of the inverse,

$$\frac{d}{d\lambda}[A^{-1}(\lambda)] = -A^{-1}(\lambda)\frac{d}{d\lambda}[A(\lambda)]A^{-1}(\lambda). \tag{2.4.21}$$

2.4.4 Error Analysis and Stability of Algorithms

One common reason for poor accuracy in the computed solution is that the problem is ill-conditioned. But poor accuracy can also be caused by a poorly constructed algorithm. We say in general that an algorithm is **unstable** if it can introduce large errors in the computed solutions to a well-conditioned problem.

We consider in the following a finite algorithm with input data $(a_1, \dots, a_r)$, which by a sequence of arithmetic operations is transformed into the output data $(w_1, \dots, w_s)$, There are two basic forms of roundoff error analysis for such an algorithm, which are both useful:

(i) In **forward** error analysis, one attempts to find bounds for the errors in the solution $|\overline{w}_i - w_i|$, $i = 1 : s$, where $\overline{w}_i$ denotes the computed value of w_i. The main tool used in forward error analysis is the propagation of errors, as studied in Sec. 2.4.2.

(ii) In **backward** error analysis, one attempts to determine a modified set of data $a_i + \Delta a_i$ such that the computed solution $\overline{w}_i$ is the *exact solution*, and give bounds for $|\Delta a_i|$. There may be an infinite number of such sets; in this case we seek to minimize the size of$|\Delta a_i|$. However, it can also happen, even for very simple algorithms, that no such set exists.

Sometimes, when a pure backward error analysis cannot be achieved, one can show that the computed solution is a slightly perturbed solution to a problem with slightly modified input data. An example of such a **mixed error analysis** is the error analysis given in Lemma 2.3.5 for the solution of a quadratic equation.

In backward error analysis no reference is made to the exact solution for the original data. In practice, when the data are known only to a certain accuracy, the "exact" solution may not be well defined. Then any solution whose backward error is smaller than the domain of uncertainty of the data may be considered to be a satisfactory result.

A frequently occurring backward error problem is the following. Suppose we are given an approximate solution y to a linear system $Ax = b$. We want to find out if y is the *exact solution to a nearby perturbed system* $(A + \Delta A)y = b + \Delta b$. To this end we define the normwise backward error of y as

$$\eta(y) = \min\{\epsilon \mid (A + \Delta A)y = b + \Delta b,\ \|\Delta A\| \le \epsilon\|A\|,\ \|\Delta b\| \le \epsilon\|b\|\}. \tag{2.4.22}$$

The following theorem tells us that the normwise backward error of y is small if the residual vector $b - Ay$ is small.

Theorem 2.4.8 (*Rigal and Gaches* [305]).
The normwise backward error of y is given by

$$\eta(y) = \frac{\|r\|}{\|A\|\,\|y\| + \|b\|}, \tag{2.4.23}$$

where $r = b - Ay$, and $\|\cdot\|$ is any consistent norm.

Similarly, we define the **componentwise backward error** $\omega(y)$ of y by

$$\omega(y) = \min\{\epsilon \mid (A + \Delta A)y = b + \Delta b,\ |\Delta A| \le \epsilon\|A\|,\ |\Delta b| \le \epsilon|b|\}. \tag{2.4.24}$$

As the following theorem shows, there is a simple expression also for $\omega(y)$.

Theorem 2.4.9 (*Oettli and Prager* [277]).

Let $r = b - A\bar{x}$, E and f be nonnegative, and set

$$\omega(y) = \max_i \frac{|r_i|}{(E|\bar{x}| + f)_i}, \tag{2.4.25}$$

where $\xi/0$ is interpreted as zero if $\xi = 0$ and infinity otherwise.

By means of backward error analysis it has been shown, even for many quite complicated matrix algorithms, that the computed results which the algorithm produces under the influence of roundoff error are the *exact* output data of a problem of the same type in which the relative change in data only are of the order of the unit roundoff u.

Definition 2.4.10.

An algorithm is **backward stable** *if the computed solution $\overline{w}$ for the data a is the exact solution of a problem with slightly perturbed data $\bar{a}$ such that for some norm $\|\cdot\|$ it holds that*

$$\|\bar{a} - a\|/\|a\| < c_1 u, \tag{2.4.26}$$

where c_1 is a not too large constant and u is the unit roundoff.

We are usually satisfied if we can prove normwise forward or backward stability for some norm, for example, $\|\cdot\|_2$ or $\|\cdot\|_\infty$. Occasionally we may like the estimates to hold componentwise,

$$|\bar{a}_i - a_i|/|a_i| < c_2 u, \quad i = 1 : r. \tag{2.4.27}$$

For example, by equation (2.3.16) the usual algorithm for computing an inner product $x^T y$ is backward stable for elementwise relative perturbations.

We would like stability to hold for some *class of input data*. For example, a numerical algorithm for solving systems of linear equations $Ax = b$ is backward stable for a class of matrices $\mathcal{A}$ if for each $A \in \mathcal{A}$ and for each b the computed solution $\bar{x}$ satisfies $\bar{A}\bar{x} = \bar{b}$, where $\bar{A}$ and $\bar{b}$ are close to A and b.

To yield error bounds for $\overline{w}_i$, a backward error analysis has to be complemented with a perturbation analysis. For this the error propagation formulas in Sec. 2.4.2 can often be used. If the condition number of the problem is κ, then it follows that

$$\|\overline{w} - w\| \le c_1 u \kappa \|w\| + O(u^2). \tag{2.4.28}$$

Hence the error in the solution may still be large if the problem is ill-conditioned. But we have obtained an answer which is the exact mathematical solution to a problem with data close to the one we wanted to solve. If the perturbations $\bar{a} - a$ are within the uncertainties of the given data, *the computed solution is as good as our data warrant*!

A great advantage of backward error analysis is that, when it applies, it tends to give much sharper results than a forward error analysis. Perhaps more important, it usually also gives a better insight into the stability (or lack of it) of the algorithm.

By the definition of the condition number κ it follows that backward stability implies forward stability, but *the converse is not true*. Many important direct algorithms for solving linear systems are known to be backward stable. The following result for the Cholesky factorization is an important example.

Theorem 2.4.11 (*Wilkinson* [378]).

Let $A \in \mathbf{R}^{n\times n}$ be a symmetric positive definite matrix. Provided that

$$2n^{3/2}u\kappa(A) < 0.1, \tag{2.4.29}$$

the Cholesky factor of A can be computed without breakdown, and the computed factor $\bar{R}$ satisfies

$$\bar{R}^T\bar{R} = A + E, \quad \|E\|_2 < 2.5n^{3/2}u\|A\|_2, \tag{2.4.30}$$

and hence is the exact Cholesky factor of a matrix close to A.

For the LU factorization of matrix A the following componentwise backward error result is known.

Theorem 2.4.12.

If the LU factorization of the matrix $A \in \mathbf{R}^{n\times n}$ runs to completion, then the computed factors $\bar{L}$ and $\bar{U}$ satisfy

$$A + E = \bar{L}\bar{U}, \quad |E| \le \gamma_n|\bar{L}|\,|\bar{U}|, \tag{2.4.31}$$

where $\gamma_n = nu/(1-nu)$, and u is the unit roundoff.

This shows that unless the elements in the computed factors $|\bar{L}|$ and $|\bar{U}|$ become large, LU factorization is backward stable.

Example 2.4.8.

For $\epsilon = 10^{-6}$ the system

$$\begin{pmatrix} \epsilon & 1 \\ 1 & 1 \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

is well-conditioned and has the exact solution $x_1 = -x_2 = -1/(1-\epsilon) \approx -1$. In Gaussian elimination we multiply the first equation by 10^6, and subtract from the second, giving $(1 - 10^6)x_2 = -10^6$. Rounding this to $x_2 = 1$ is correct to six digits. In the back-substitution to obtain x_1, we then get $10^{-6}x_1 = 1 - 1$, or $x_1 = 0$, which is a completely wrong result. This shows that Gaussian elimination can be an unstable algorithm unless row (and/or column) interchanges are performed to limit element growth.

Some algorithms, including most iterative methods, are not backward stable. Then it is necessary to weaken the definition of stability. In practice an algorithm can be considered stable if *it produces accurate solutions for well-conditioned problems*. Such an algorithm can be called **weakly stable**. Weak stability may be sufficient for giving confidence in an algorithm.

Example 2.4.9.

In the method of normal equations for computing the solution of a linear least squares problem one first forms the matrix A^TA. This product matrix can be expressed in outer form as

$$A^TA = \sum_{i=1}^{m} a_i a_i^T,$$

where a_i^T is the ith row of A, i.e., $A^T = (\, a_1 \quad a_2 \quad \dots \quad a_m \,)$. From (2.3.14) it follows that this computation is not backward stable; i.e., it is not true that $fl(A^TA) = (A+E)^T(A+E)$ for some small error matrix E. In order to avoid loss of significant information higher precision needs to be used.

Backward stability is easier to prove when there is a sufficiently large set of input data compared to the number of output data. When computing the outer product xy^T (as in Example 2.4.9) there are $2n$ data and n^2 results. This is not a backward stable operation. When the input data are structured rather than general, backward stability often does not hold.

Example 2.4.10.

Many algorithms for solving a linear system $Ax = b$ are known to be backward stable; i.e., the computed solution is the exact solution of a system $(A+E)x = b$, where the normwise relative error $\|E\|/\|A\|$ is not much larger than the machine precision. In many applications the system matrix is **structured**. An important example is **Toeplitz matrices** T, whose entries are constant along every diagonal:

$$T = (t_{i-j})_{1\le i,j\le n} = \begin{pmatrix} t_0 & t_1 & \dots & t_{n-1} \\ t_{-1} & t_0 & \dots & t_{n-2} \\ \vdots & \vdots & \ddots & \vdots \\ t_{-n+1} & t_{-n+2} & \dots & t_0 \end{pmatrix} \in \mathbf{R}^{n\times n}. \tag{2.4.32}$$

Note that a Toeplitz matrix is completely specified by its first row and column, i.e., the $2n-1$ quantities $t = (t_{-n+1}, \dots, t_0, \dots, t_{n-1})$.

Ideally, in a strict backward error analysis, we would like to show that a solution algorithm always computes *an exact solution to a nearby Toeplitz system* defined by $T + S$, where S is small. It has been shown that no such algorithm can exist! We have to be content with algorithms that (at best) compute the exact solution of $(T + E)x = b$, where $\|E\|$ is small but E unstructured.

In the construction of an algorithm for a given problem, one often breaks down the problem into a chain of subproblems, $P_1, P_2, \dots, P_k$, for which algorithms $A_1, A_2, \dots, A_k$ are known, in such a way that the output data from P_{i-1} are the input data to P_i. *Different ways of decomposing the problem give different algorithms with, as a rule, different stability properties.* It is dangerous if the last subproblem in such a chain is ill-conditioned. On the other hand, it need not be dangerous if the first subproblem is ill-conditioned, if the problem itself is well-conditioned. *Even if the algorithms for all the subproblems are stable, we cannot conclude that the composed algorithm is stable.*

Example 2.4.11.

The problem of computing the eigenvalues λ_i of a symmetric matrix A, given its elements (a_{ij}), is always a well-conditioned numerical problem with absolute condition number equal to 1. Consider an algorithm which breaks down this problem into two subproblems:

- P_1: compute the coefficients of the characteristic polynomial of the matrix A $p(\lambda) = \det(A - \lambda I)$ of the matrix A.
- P_2: compute the roots of the polynomial $p(\lambda)$ obtained from P_1.

It is well known that the second subproblem P_2 can be very ill-conditioned. For example, for a symmetric matrix A with eigenvalues $\pm 1, \pm 2, \dots, \pm 20$ the condition number for P_2 is 10^{14} in spite of the fact that the origin lies exactly between the largest and smallest eigenvalues, so that one cannot blame the high condition number on a difficulty of the same type as that encountered in Example 2.4.7.

The important conclusion that eigenvalues should not be computed as outlined above is further discussed in Sec. 6.5.2.

On the other hand, as the next example shows, it need not be dangerous if the first subproblem of a decomposition is ill-conditioned, even if the problem itself is well-conditioned.

Example 2.4.12.

The first step in many algorithms for computing the eigenvalues λ_i of a symmetric matrix A is to use orthogonal similarity transformations to symmetric tridiagonal form,

$$Q^T A Q = T = \begin{pmatrix} \alpha_1 & \beta_2 & & & \\ \beta_2 & \alpha_2 & \beta_3 & & \\ & \ddots & \ddots & \ddots & \\ & & \beta_{n-1} & \alpha_{n-1} & \beta_n \\ & & & \beta_n & \alpha_n \end{pmatrix}.$$

In the second step the eigenvalues of T, which coincide with those of A, are computed.

Wilkinson [377, Sec. 5.28] showed that the computed tridiagonal matrix can differ significantly from the matrix corresponding to exact computation. Hence here the first subproblem is ill-conditioned. (This fact is not as well known as it should be and still alarms many users!) But the second subproblem is well-conditioned and the combined algorithm is known to be backward stable, i.e., the computed eigenvalues are the exact eigenvalues of a matrix $A+E$, where $\|E\|_2 < c(n)u\|A\|_2$. This is a more complex example of a calculation, where rounding errors cancel!

It should be stressed that *the primary purpose of a rounding error analysis is to give insight into the properties of the algorithm.* In practice we can usually expect a much smaller backward error in the computed solutions than the bounds derived in this section. It is appropriate to recall here a remark by J. H. Wilkinson:

All too often, too much attention is paid to the precise error bound that has been established. The main purpose of such an analysis is either to establish the essential numerical stability of an algorithm or to show why it is unstable and in doing so expose what sort of change is necessary to make it stable. The precise error bound is not of great importance.

The treatment in this section is geared toward matrix problems and is not very useful, for example, for time-dependent problems in ordinary and partial differential equations. In Sec. 1.5 some methods for the numerical solution of an initial value problem

$$y'' = -y, \quad y(0) = 0, \quad y'(0) = 1$$

were studied. As will be illustrated in Example 3.3.15, catastrophic error growth can occur in such processes. The notion of stability is here related to the stability of linear difference equations.

Review Questions

2.4.1 The maximal error bounds for addition and subtraction can for various reasons be a coarse overestimate of the real error. Give two reasons, preferably with examples.

2.4.2 How is the condition number $\kappa(A)$ of a matrix A defined? How does $\kappa(A)$ relate to perturbations in the solution x to a linear system $Ax = b$, when A and b are perturbed?

2.4.3 Define the condition number of a numerical problem P of computing output data $y_1, \dots, y_m$ given input data $x_1, \dots, x_n$.

2.4.4 Give examples of well-conditioned and ill-conditioned problems.

2.4.5 What is meant by (a) a forward error analysis; (b) a backward error analysis; (c) a mixed error analysis?

2.4.6 What is meant by (a) a backward stable algorithm; (b) a forward stable algorithm; (c) a mixed stable algorithm; (d) a weakly stable algorithm?

Problems and Computer Exercises

2.4.1 (a) Determine the maximum error for $y = x_1x_2^2/\sqrt{x_3}$, where $x_1 = 2.0 \pm 0.1$, $x_2 = 3.0 \pm 0.2$, and $x_3 = 1.0 \pm 0.1$. Which variable contributes most to the error?

(b) Compute the standard error using the same data as in (a), assuming that the error estimates for the x_i indicate standard deviations.

2.4.2 One wishes to compute $f = (\sqrt{2} - 1)^6$, using the approximate value 1.4 for $\sqrt{2}$. Which of the following mathematically equivalent expressions gives the best result?

$$\frac{1}{(\sqrt{2}+1)^6}; \quad (3-2\sqrt{2})^3; \quad \frac{1}{(3+2\sqrt{2})^3}; \quad 99-70\sqrt{2}; \quad \frac{1}{99+70\sqrt{2}}$$

2.4.3 Analyze the error propagation in x^α

(a) if x is exact and α in error; (b) if α is exact and x in error.

2.4.4 One is observing a satellite in order to determine its speed. At the first observation, $R = 30{,}000 \pm 10$ miles. Five seconds later, the distance has increased by $r = 125.0 \pm 0.5$ miles and the change in the angle is $\phi = 0.00750 \pm 0.00002$ radians. What is the speed of the satellite, assuming that it moves in a straight line and with constant speed in the interval?

2.4.5 One has measured two sides and the included angle of a triangle to be $a = 100.0 \pm 0.1$, $b = 101.0 \pm 0.1$, and angle $C = 1.00° \pm 0.01°$. Then the third side is given by the cosine theorem

$$c = (a^2 + b^2 - 2ab\cos C)^{1/2}.$$

(a) How accurately is it possible to determine c from the given data?

(b) How accurately does one get c if one uses the value $\cos 1° = 0.9998$, which is correct to four decimal places?

(c) Show that by rewriting the cosine theorem as

$$c = ((a-b)^2 + 4ab\sin^2(C/2))^{1/2}$$

it is possible to compute c to full accuracy using only a four-decimal table for the trigonometric functions.

2.4.6 Consider the linear system

$$\begin{pmatrix} 1 & \alpha \\ \alpha & 1 \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix},$$

where $\alpha \neq 1$. What is the relative condition number for computing x? Using Gaussian elimination and four decimal digits, compute x and y for $\alpha = 0.9950$ and compare with the exact solution $x = 1/(1-\alpha^2)$, $y = -\alpha/(1-\alpha^2)$.

2.4.7 (a) Let two vectors u and v be given with components (u_1, u_2) and (v_1, v_2). The angle ϕ between u and v is given by the formula

$$\cos\phi = \frac{u_1v_1 + u_2v_2}{(u_1^2 + u_2^2)^{1/2}(v_1^2 + v_2^2)^{1/2}}.$$

Show that computing the angle ϕ from the components of u and v is a well-conditioned problem.

Hint: Take the partial derivative of $\cos\phi$ with respect to u_1, and from this compute $\partial\phi/\partial u_1$. The other partial derivatives are obtained by symmetry.

(b) Show that the formula in (a) is *not* stable for small angles ϕ.

(c) Show that the following algorithm is stable. First normalize the vectors $\tilde{u} = u/\|u\|_2$, $\tilde{v} = v/\|v\|_2$. Then compute $\alpha = \|\tilde{u} - \tilde{v}\|_2$, $\beta = \|\tilde{u} + \tilde{v}\|_2$ and set

$$\phi = \begin{cases} 2\arctan(\alpha/\beta) & \text{if } \alpha \le \beta, \\ \pi - 2\arctan(\beta/\alpha) & \text{if } \alpha > \beta. \end{cases}$$

2.4.8 For the integral

$$I(a,b) = \int_0^1 \frac{e^{-bx}}{a + x^2}dx,$$

the physical quantities a and b have been measured to be $a = 0.4000 \pm 0.003$,

$b = 0.340 \pm 0.005$. When the integral is computed for various perturbed values of a and b, one obtains

a	b	I
0.39	0.34	1.425032
0.40	0.32	1.408845
0.40	0.34	1.398464
0.40	0.36	1.388198
0.41	0.34	1.372950

.

Estimate the uncertainty in $I(a, b)$!

2.4.9 (a) Let $B \in \mathbf{R}^{n\times n}$ be a matrix for which $\|B\| < 1$. Show that the infinite sum and product

$$(I - B)^{-1} = \begin{cases} I + B + B^2 + B^3 + B^4 \cdots, \\ (I + B)(I + B^2)(I + B^4)(I + B^8) \cdots \end{cases}$$

both converge to the indicated limit.

Hint: Use the identity $(I - B)(I + B + \cdots + B^k) = I - B^{k+1}$.

(b) Show that the matrix $(I - B)$ is nonsingular and that

$$\|(I - B)^{-1}\| \le 1/(1 - \|B\|).$$

2.4.10 Solve the linear system in Example 2.4.8 with Gaussian elimination after exchanging the two equations. Do you now get an accurate result?

2.4.11 Derive forward and backward recursion formulas for calculating the integrals

$$I_n = \int_0^1 \frac{x^n}{4x + 1}\, dx.$$

Why is one algorithm stable and the other unstable?

2.4.12 (a) Use the results in Table 2.4.1 to determine constants c and q such that $\kappa(H_n) \approx c \cdot 10^q$.

(b) Compute the Bauer–Skeel condition number $\operatorname{cond}(H_n) = \|\,|H_n^{-1}||H_n|\,\|_2$, of the Hilbert matrices for $n = 1 : 12$. Compare the result with the values of $\kappa(H_n)$ given in Table 2.4.1.

2.4.13 Vandermonde matrices are structured matrices of the form

$$V_n = \begin{pmatrix} 1 & 1 & \cdots & 1 \\ \alpha_1 & \alpha_2 & \cdots & \alpha_n \\ \vdots & \vdots & \cdots & \vdots \\ \alpha_1^{n-1} & \alpha_2^{n-1} & \cdots & \alpha_n^{n-1} \end{pmatrix}.$$

Let $\alpha_j = 1 - 2(j - 1)/(n - 1)$, $j = 1 : n$. Compute the condition numbers $\kappa_2(V_n)$ for $n = 5, 10, 15, 20, 25$. Is the growth in $\kappa_2(V_n)$ exponential in n?

2.5 Automatic Control of Accuracy and Verified Computing

2.5.1 Running Error Analysis

A different approach to rounding error analysis is to perform the analysis automatically, for *each particular computation*. This gives an *a posteriori* error analysis in contrast to the *a priori* error analysis discussed above.

A simple form of a posteriori analysis, called running error analysis, was used in the early days of computing; see Wilkinson [380]. To illustrate his idea we rewrite the basic model for floating-point arithmetic as

$$x \text{ op } y = fl\,(x \text{ op } y)(1+\epsilon).$$

This holds for most implementations of floating-point arithmetic. Then, the actual error can be estimated $|fl\,(x \text{ op } y) - x \text{ op } y| \le u|fl\,(x \text{ op } y)|$. Note that the error is now given in terms of the *computed* result and is available in the computer at the time the operation is performed. This running error analysis can often be easily implemented. We just take an existing program and modify it, so that as each arithmetic operation is performed, the absolute value of the computed results is added into the accumulating error bound.

Example 2.5.1.

The inner product $fl\,(x^T y)$ is computed by the program

```
$s = 0$;    $\eta = 0$;
for $i = 1, 2, \dots, n$
    $t = fl\,(x_i y_i)$;    $\eta = \eta + |t|$;
    $s = fl\,(s + t)$;    $\eta = \eta + |s|$;
end
```

For the final error we have the estimate $|fl\,(x^T y) - x^T y| \le \eta u$. Note that a running error analysis takes advantage of cancellations in the sum. This is in contrast to the previous estimates, which we call a priori error analysis, where the error estimate is the same for all distribution of signs of the elements x_i and y_i.

Efforts have been made to design the computational unit of a computer so that it gives, in every arithmetic operation, only those digits of the result which are judged to be significant (possibly with a fixed number of extra digits), so-called *unnormalized floating arithmetic*. This method reveals poor construction in algorithms, but in many other cases it gives a significant and unnecessary loss of accuracy. The mechanization of the rules, which a knowledgeable and experienced person would use for control of accuracy in hand calculation, is not as free from problems as one might expect. As a complement to arithmetical operations of conventional type, the above type of arithmetic is of some interest, but it is doubtful that it will ever be widely used.

A fundamental difficulty in automatic control of accuracy is that to decide how many digits are needed in a quantity to be used in later computation, one needs to consider the *entire* *context of the computations*. It can in fact occur that the errors in many operands depend on each other in such a way that they cancel each other. Such a **cancellation of error**, which is a completely different phenomenon from the previously discussed cancellation of terms, is most common in larger problems, but will be illustrated here with a simple example.

Example 2.5.2.

Suppose we want to compute $y = z_1 + z_2$, where $z_1 = \sqrt{x^2+1}$, $z_2 = 200 - x$, $x = 100 \pm 1$, with a rounding error which is negligible compared to that resulting from the errors in z_1 and z_2. The best possible error bounds in the intermediate results are $z_1 = 100 \pm 1$, $z_2 = 100 \pm 1$. It is then tempting to be satisfied with the result $y = 200 \pm 2$.

But the errors in z_1 and z_2 due to the uncertainty in x will, to a large extent, cancel each other! This becomes clear if we rewrite the expression as

$$y = 200 + \left(\sqrt{x^2+1} - x\right) = 200 + \frac{1}{\sqrt{x^2+1}+x}.$$

It follows that $y = 200 + z$, where $1/202 \lessapprox z \le 1/198$. Thus y can be computed with an absolute error less than about $2/(200)^2 = 0.5 \cdot 10^{-4}$. Therefore, using the expression $y = z_1 + z_2$ the intermediate results z_1 and z_2 should be computed to four decimals even though the last integer in these is uncertain. The result is $y = 200.0050 \pm \frac{1}{2}10^{-4}$.

In larger problems, such a cancellation of errors can occur even though one cannot easily give a way to rewrite the expressions involved. The authors have seen examples where the final result, a sum of seven terms, was obtained correctly to eight decimals even though the terms, which were complicated functions of the solution to a system of nonlinear equations with 14 unknowns, were correct only to three decimals! Another nontrivial example is given in Example 2.4.12.

2.5.2 Experimental Perturbations

In many practical problems, the functional dependence between input data and output data is so complicated that it is difficult to directly apply the general formulas for error propagation derived in Sec. 2.4.3. One can then investigate the sensitivity of the output data for perturbations in the input data by means of an **experimental perturbational calculation**. One performs the calculations many times with perturbed input data and studies the perturbations in the output data. For example, instead of using the formula for standard error of a function of many variables, given in Theorem 2.4.5, it is often easier to compute the function a number of times with randomly perturbed arguments and to use them to estimate the standard deviation of the function numerically.

Important data, such as the step length in a numerical integration or the parameter which determines when an iterative process is going to be broken off, should be varied with all the other data left unchanged. If one can easily *vary the precision of the machine* in the arithmetic operations one can get an idea of the influence of rounding errors. It is generally not necessary to make a perturbational calculation for each and every data component; one can instead *perturb many input data simultaneously*—for example, by using random numbers.

A perturbational calculation often gives not only an error estimate but also greater insight into the problem. Occasionally, it can be difficult to interpret the perturbational data correctly, since the disturbances in the output data depend not only on the mathematical problem but also on the choice of numerical method and the details in the design of the algorithm. The rounding errors during the computation are not the same for the perturbed and unperturbed problem. Thus if the output data react more sensitively than one had anticipated, it can be difficult to immediately point out the source of the error. It can then be profitable to plan a series of perturbation experiments with the help of which one can separate the effects of the various sources of error. If the dominant source of error is the method or the algorithm, then one should try another method or another algorithm. It is beyond the scope of this book to give further comments on the planning of such experiments. Imagination and the general insights regarding error analysis, which this chapter is meant to give, play a large role.

2.5.3 Interval Arithmetic

In **interval arithmetic** one assumes that all input values are given as intervals and systematically calculates an inclusion interval for each intermediate result. It is partly an automation of calculation with maximal error bounds. The importance of interval arithmetic is that it provides a tool for computing validated answers to mathematical problems.

The modern development of interval arithmetic was initiated by the work of R. E. Moore [271]. Interval arithmetic has since been developed into a useful tool for many problems in scientific computing and engineering. A noteworthy example of its use is the verification of the existence of a Lorenz attractor by W. Tucker [361]. Several extensive surveys on interval arithmetic are available; see [3, 4, 225]. Hargreaves [186] gives a short tutorial on INTLAB and also a good introduction to interval arithmetic.

The most frequently used representations for the intervals are the **infimum-supremum** representations

$$I = [a, b] := \{x \mid a \le x \le b\}, \quad (a \le b), \tag{2.5.1}$$

where x is a real number. The absolute value or the **magnitude** of an interval is defined as

$$|\,[a, b]\,| = \operatorname{mag}([a, b]) = \max\{|x| \mid x \in [a, b]\}, \tag{2.5.2}$$

and the **mignitude** of an interval is defined as

$$\operatorname{mig}([a, b]) = \min\{|x| \mid x \in [a, b]\}. \tag{2.5.3}$$

In terms of the endpoints we have

$$\begin{aligned} \operatorname{mag}([a, b]) &= \max\{|a|, |b|\}, \\ \operatorname{mig}([a, b]) &= \begin{cases} \min\{|a|, |b|\} & \text{if } 0 \notin [a, b], \\ 0 & \text{otherwise.} \end{cases} \end{aligned}$$

The result of an interval operation equals the range of the corresponding real operation. For example, the difference between the intervals $[a_1, a_2]$ and $[b_1, b_2]$ is defined as the shortest interval which contains all the numbers $x_1 - x_2$, where $x_1 \in [a_1, a_2]$, $x_2 \in [b_1, b_2]$, i.e., $[a_1, a_2] - [b_1, b_2] := [a_1 - b_2, a_2 - b_1]$. Other elementary interval arithmetic operations are similarly defined:

$$[a_1, a_2] \operatorname{op} [b_1, b_2] := \{x_1 \operatorname{op} x_2 \mid x_1 \in [a_1, a_2],\ x_2 \in [b_1, b_2]\}, \tag{2.5.4}$$

where op $\in \{+, -, \times, \text{div}\}$. The interval value of a function ϕ (for example, the elementary functions sin, cos, exp, log) evaluated on an interval is defined as

$$\phi([a, b]) = \Big[\inf_{x\in[a,b]} \phi(x),\ \sup_{x\in[a,b]} \phi(x)\Big].$$

Operational Definitions

Although (2.5.4) characterizes interval arithmetic operations, we also need **operational definitions**. We take

$$\begin{aligned}
&[a_1, a_2] + [b_1, b_2] = [a_1 + b_1, a_2 + b_2],\\
&[a_1, a_2] - [b_1, b_2] = [a_1 - b_2, a_2 - b_1],\\
&[a_1, a_2] \times [b_1, b_2] = \big[\min\{a_1b_1, a_1b_2, a_2b_1, a_2b_2\}, \max\{a_1b_1, a_1b_2, a_2b_1, a_2b_2\}\big],\\
&1/[a_1, a_2] = [1/a_2, 1/a_1] \quad \text{for} \quad a_1a_2 > 0,\\
&[a_1, a_2]/[b_1, b_2] = [a_1, a_2] \cdot (1/[b_1, b_2]).
\end{aligned} \tag{2.5.5}$$

It is easy to prove that in exact interval arithmetic the operational definitions above give the exact range (2.5.4) of the interval operations. Division by an interval containing zero is not defined and may cause an interval computation to come to a premature end.

A degenerate interval with radius zero is called a point interval and can be identified with a single number $a \equiv [a, a]$. In this way the usual arithmetic is recovered as a special case. The intervals $0 = [0, 0]$ and $1 = [1, 1]$ are the neutral elements with respect to interval addition and interval multiplication, respectively. A nondegenerate interval has no inverse with respect to addition or multiplication. For example, we have

$$[1, 2] - [1, 2] = [-1, 1], \qquad [1, 2]/[1, 2] = [0.5, 2].$$

For interval operations the **commutative law**

$$[a_1, a_2] \operatorname{op} [b_1, b_2] = [b_1, b_2] \operatorname{op} [a_1, a_2]$$

holds. But the distributive law has to be replaced by so-called **subdistributivity**:

$$[a_1, a_2]([b_1, b_2] + [c_1, c_2]) \subseteq [a_1, a_2][b_1, b_2] + [a_1, a_2][c_1, c_2]. \tag{2.5.6}$$

This unfortunately means that expressions, which are equivalent in real arithmetic, differ in exact interval arithmetic. The simple example

$$[-1, 1]([1, 1] + [-1, -1]) = 0 \subset [-1, 1][1, 1] + [-1, 1][-1, -1] = [-2, 2]$$

shows that $-[-1, 1]$ is not the additive inverse to $[-1, 1]$ and also illustrates (2.5.6).

The operations introduced are **inclusion monotonic**, i.e.,

$$[a_1, a_2] \subseteq [c_1, c_2],\ [b_1, b_2] \subseteq [d_1, d_2] \quad \Rightarrow \quad [a_1, a_2]\,\mathrm{op}\,[b_1, b_2] \subseteq [c_1, c_2]\,\mathrm{op}\,[d_1, d_2]. \tag{2.5.7}$$

An alternative representation for an interval is the **midpoint-radius** representation, for which we use brackets,

$$\langle c, r\rangle := \{x \mid |x - c| \le r\} \quad (0 \le r), \tag{2.5.8}$$

where the midpoint and radius of the interval $[a, b]$ are defined as

$$c = \mathrm{mid}\,([a, b]) = \frac{1}{2}(a + b), \qquad r = \mathrm{rad}\,([a, b]) = \frac{1}{2}(b - a). \tag{2.5.9}$$

For intervals in the midpoint-radius representation we take as operational definitions

$$\begin{aligned}
\langle c_1, r_1\rangle + \langle c_2, r_2\rangle &= \langle c_1 + c_2, r_1 + r_2\rangle,\\
\langle c_1, r_1\rangle - \langle c_2, r_2\rangle &= \langle c_1 - c_2, r_1 + r_2\rangle,\\
\langle c_1, r_1\rangle \times \langle c_2, r_2\rangle &= \langle c_1c_2, |c_1|r_2 + r_1|c_2| + r_1r_2\rangle,\\
1/\langle c, r\rangle &= \langle c/(|c|^2 - r^2), r/(|c|^2 - r^2)\rangle, \quad (|c| > r),\\
\langle c_1, r_1\rangle/\langle c_2, r_2\rangle &= \langle c_1, r_1\rangle \times (1/\langle c_2, r_2\rangle).
\end{aligned} \tag{2.5.10}$$

For addition, subtraction, and inversion, these give the exact range, but for multiplication and division they overestimate the range (see Problem 2.5.2). For multiplication we have for any $x_1 \in \langle c_1, r_1\rangle$ and $x_2 \in \langle c_2, r_2\rangle$

$$\begin{aligned}
|x_1x_2 - c_1c_2| &= |c_1(x_2 - c_2) + c_2(x_1 - c_1) + (x_1 - c_1)(x_2 - c_2)|\\
&\le |c_1|r_2 + |c_2|r_1 + r_1r_2.
\end{aligned}$$

In implementing interval arithmetic using floating-point arithmetic, the operational interval results may not be exactly representable as floating-point numbers. Then if the lower bound is rounded down to the nearest smaller machine number and the upper bound rounded up, the exact result must be contained in the resulting interval. Recall that these rounding modes (rounding to $-\infty$ and $+\infty$) are supported by the IEEE 754 standard. For example, using five significant decimal arithmetic, we would like to get

$$[1, 1] + [-10^{-10}, 10^{-10}] = [0.99999, 1.0001]$$

or, in midpoint-radius representation,

$$\langle 1, 0\rangle + \langle 0, 10^{-10}\rangle = \langle 1, 10^{-10}\rangle.$$

Note that in the conversion between decimal and binary representation rounding the appropriate rounding mode must also be used where needed. For example, converting the point interval 0.1 to binary IEEE double precision we get an interval with radius $1.3878 \cdot 10^{-17}$. The conversion between the infimum-supremum representation is straightforward but the infimum-supremum and midpoint may not be exactly representable.

Interval arithmetic applies also to complex numbers. A complex interval in the infimum-supremum representation is

$$[z_1, z_2] = \{z = x + iy \mid x \in [x_1, x_2],\ y \in [y_1, y_2]\}.$$

This defines a closed *rectangle in the complex plane* defined by the two real intervals,

$$[z_1, z_2] = [x_1, x_2] + i[y_1, y_2], \quad x_1 \le x_2, \quad y_1 \le y_2.$$

This can be written more compactly as $[z_1, z_2] := \{z \mid z_1 \le z \le z_2\}$, where we use the partial ordering

$$z \le w \iff \Re z \le \Re w \quad \text{and} \quad \Im z \le \Im w.$$

Complex interval operations in the infimum-supremum arithmetic are defined in terms of the real intervals in the same way as the complex operations are defined for complex numbers $z = x+iy$. For addition and subtraction the result coincides with the exact range. This is not the case for complex interval multiplication, where the result is a rectangle in the complex plane, whereas the actual range is not of this shape. Therefore, for complex intervals, multiplication will result in an overestimation.

In the complex case the midpoint-radius representation is

$$\langle c, r\rangle := \{z \in \mathbf{C} \mid |z - c| \le r\}, \quad 0 \le r,$$

where the midpoint c is now a complex number. This represents a *closed circular disk in the complex plane*. The operational definitions (2.5.10) are still valid, except that some operations now are complex operations, and that inversion becomes

$$1/\langle c, r\rangle = \left\langle \bar{c}/(|c|^2 - r^2), r/(|c|^2 - r^2)\right\rangle \quad \text{for} \quad |c| > r,$$

where $\bar{c}$ is the complex conjugate of c. Complex interval midpoint-radius arithmetic is also called **circular arithmetic**. For complex multiplications it generates less overestimation than the infimum-supremum notation.

Although the midpoint-radius arithmetic seems more appropriate for complex intervals, most older implementations of interval arithmetic use infimum-supremum arithmetic. One reason for this is the overestimation also caused for real intervals by the operational definitions for midpoint-radius multiplication and division. Rump [307] has shown that the overestimation is bounded by a factor 1.5 in radius and that midpoint-radius arithmetic allows for a much faster implementation for modern vector and parallel computers.

2.5.4 Range of Functions

One use of interval arithmetic is to enclose the range of a real-valued function. This can be used, for example, for localizing and enclosing global minimizers and global minima of a real function of one or several variables in a region. It can also be used for verifying the nonexistence of a zero of $f(x)$ in a given interval.

Let $f(x)$ be a real function composed of a finite number of elementary operations and standard functions. If one replaces the variable x by an interval $[\underline{x}, \overline{x}]$ and evaluates the resulting interval expression, one gets as the result an interval denoted by $f([\underline{x}, \overline{x}])$. (It is assumed that all operations can be carried out.) A fundamental result in interval arithmetic is that this evaluation is inclusion monotonic, i.e.,

$$[\underline{x}, \overline{x}] \subseteq [\underline{y}, \overline{y}] \quad \Rightarrow \quad f([\underline{x}, \overline{x}]) \subseteq f([\underline{y}, \overline{y}]).$$

In particular this means that

$$x \subseteq [\underline{x}, \overline{x}] \;\Rightarrow\; f(x) \subseteq f([\underline{x}, \overline{x}]),$$

i.e., $f([x])$ contains the range of $f(x)$ over the interval $[\underline{x}, \overline{x}]$. A similar result holds also for functions of several variables $f(x_1, \dots, x_n)$.

An important case when interval evaluation gives the exact range of a function is when $f(x_1, \dots, x_n)$ is a rational expression, where each variable x_i occurs only once in the function.

Example 2.5.3.

In general narrow bounds cannot be guaranteed. For example, if $f(x) = x/(1-x)$, then

$$f([2,3]) = [2,3]/(1-[2,3]) = [2,3]/[-2,-1] = [-3,-1].$$

The result contains but does not coincide with the exact range $[-2, -3/2]$. But if we rewrite the expression as $f(x) = 1/(1/x - 1)$, where x only occurs once, then we get

$$f([2,3]) = 1/(1/[2,3]-1) = 1/[-2/3, -1/2] = [-2, -3/2],$$

which is the exact range.

When interval analysis is used in a naive manner as a simple technique for simulating forward error analysis, it does not usually give sharp bounds on the total computational error. To get useful results the computations in general need to be arranged so that overestimation is minimized as much as possible. Often a refined design of the algorithm is required in order to prevent the bounds for the intervals from becoming unacceptably coarse. The answer $[-\infty, \infty]$ is of course always correct but not at all useful!

The remainder term in Taylor expansions includes a variable ξ, which is known to lie in an interval $\xi \in [a, b]$. This makes it suitable to treat the remainder term with interval arithmetic.

Example 2.5.4.

Evaluate for $[x] = [2, 3]$ the polynomial

$$p(x) = 1 - x + x^2 - x^3 + x^4 - x^5.$$

Using exact interval arithmetic we find

$$p([2,3]) = [-252, 49]$$

(verify this!). This is an overestimate of the exact range, which is $[-182, -21]$. Rewriting $p(x)$ in the form $p(x) = (1-x)(1+x^2+x^4)$ we obtain the correct range. In the first example there is a **cancellation of errors** in the intermediate results (cf. Example 2.5.2), which is not revealed by the interval calculations.

Sometimes it is desired to compute a tiny interval that is guaranteed to enclose a real simple root x^* of $f(x)$. This can be done using an interval version of Newton's method. Suppose that the function $f(x)$ is continuously differentiable. Let $f'([x_0])$ denote an interval containing $f'(x)$ for all x in a finite interval $[x] := [a, b]$. Define the Newton operator $N([x])$, $[x] = [a, b]$, by

$$N([x]) := m - \frac{f(m)}{f'([x])}, \quad m = \text{mid}\,[x]. \tag{2.5.11}$$

For the properties of the **interval Newton's method**

$$[x_{k+1}] = N([x_k]), \quad k = 0, 1, 2, \dots;$$

see Sec. 6.3.3.

Another important application of interval arithmetic is to initial value problems for ordinary differential equations

$$y' = f(x, y), \quad y(x_0) = y_0, \quad y \in \mathbf{R}^n.$$

Interval techniques can be used to provide for errors in the initial values, as well as truncation and rounding errors, so that at each step intervals are computed that contain the actual solution. But it is a most demanding task to construct an interval algorithm for the initial value problem, and such algorithms tend to be significantly slower than corresponding point algorithms. One problem is that a wrapping effect occurs at each step and causes the computed interval widths to grow exponentially. This is illustrated in the following example.

Example 2.5.5.

The recursion formulas

$$x_{n+1} = (x_n - y_n)/\sqrt{2}, \qquad y_{n+1} = (x_n + y_n)/\sqrt{2}$$

mean a series of 45-degree rotations in the xy-plane (see Figure 2.5.1). By a two-dimensional interval one means a rectangle whose sides are parallel to the coordinate axes.

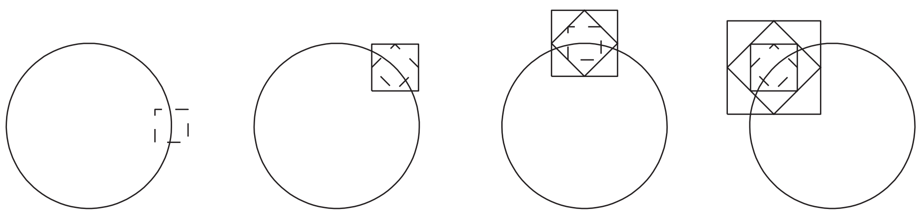

Figure 2.5.1. *Wrapping effect in interval analysis.*

If the initial value (x_0, y_0) is given as an interval $[x_0] = [1-\epsilon, 1+\epsilon]$, $[y_0] = [-\epsilon, \epsilon]$ (see the dashed square, in the leftmost portion of Figure 2.5.1), then (x_n, y_n) will, with *exact* performance of the transformations, also be a square with side 2ϵ, for all n (see the other squares in Figure 2.5.1). If the computations are made using interval arithmetic, rectangles with sides parallel to the coordinate axis will, in each step, be circumscribed about the exact image of the interval one had in the previous step. Thus the interval is multiplied by $\sqrt{2}$ in each step. After 40 steps, for example, the interval has been multiplied by $2^{20} > 10^6$. This phenomenon, intrinsic to interval computations, is called the **wrapping effect**. (Note that if one uses disks instead of rectangles, there would not be any difficulties in this example.)

2.5.5 Interval Matrix Computations

In order to treat multidimensional problems we introduce interval vectors and matrices. An interval vector is denoted by $[x]$ and has interval components $[x_i] = [\underline{x_i}, \overline{x_i}])$, $i = 1 : n$. Likewise an interval matrix $[A] = ([a_{ij}])$ has interval elements

$$\left[a_{ij}\right] = \left[\underline{a}_{ij}, \overline{a}_{ij}\right], \quad i = 1 : m, \quad j = 1 : n.$$

Operations between interval matrices and interval vectors are defined in an obvious manner. The interval matrix-vector product $[A][x]$ is the smallest interval vector, which contains the set

$$\{Ax \mid A \in [A],\ x \in [x]\}$$

but normally does not coincide with it. By the inclusion property it holds that

$$\{Ax \mid A \in [A],\ x \in [x]\} \subseteq [A][x] = \left(\sum_{j=1}^{n} \left[a_{ij}\right]\left[x_j\right]\right).$$

In general, there will be an overestimation in enclosing the image with an interval vector, caused by the fact that the image of an interval vector under a linear transformation in general is not an interval vector. This phenomenon, intrinsic to interval computations, is similar to the wrapping effect described in Example 2.5.5.

Example 2.5.6.

We have

$$A = \begin{pmatrix} 1 & 1 \\ -1 & 1 \end{pmatrix}, \quad [x] = \begin{pmatrix} [0,1] \\ [0,1] \end{pmatrix} \quad \Rightarrow \quad A[x] = \begin{pmatrix} [0,2] \\ [-1,1] \end{pmatrix}.$$

Hence $b = (\,2 \quad -1\,)^T \in A[x]$, but there is no $x \in [x]$ such that $Ax = b$. (The solution to $Ax = b$ is $x = (\,3/2 \quad 1/2\,)^T$.)

The magnitude of an interval vector or matrix is interpreted componentwise and defined by

$$|\,[x]\,| = (|\,[x_1]\,|, |\,[x_2]\,|, \dots, |\,[x_n]\,|)^T,$$

where the magnitude of the components is defined by

$$|\,[a,b]\,| = \max\{|x| \mid x \in [a,b]\}. \tag{2.5.12}$$

The ∞-norm of an interval vector or matrix is defined as the ∞-norm of its magnitude,

$$\| [x] \|_\infty = \| \, |[x]| \, \|_\infty, \qquad \| [A] \|_\infty = \| \, |[A]| \, \|_\infty. \tag{2.5.13}$$

In implementing matrix multiplication it is important to avoid case distinctions in the inner loops, because that would make it impossible to use fast vector and matrix operations. Using interval arithmetic it is possible to compute strict enclosures of the product of two matrices. Note that this is also needed in the case of the product of two point matrices since rounding errors will usually occur.

We assume that the command

$$\text{setround}(i), \quad i = -1, 0, 1,$$

sets the rounding mode to $-\infty$, to nearest, and to $+\infty$, respectively. (Recall that these rounding modes are supported by the IEEE standard.) Let A and B be point matrices and suppose we want to compute an interval matrix $[C]$ such that

$$fl(A \cdot B) \subset [C] = [C_{\text{inf}}, C_{\text{sup}}].$$

Then the following simple code, using two matrix multiplications, does that:

$$\begin{aligned} &\text{setround}(-1); \quad C_{\text{inf}} = A \cdot B; \\ &\text{setround}(1); \quad\;\; C_{\text{sup}} = A \cdot B; \end{aligned}$$

We next consider the product of a point matrix A and an interval matrix $[B] = [B_{\text{inf}}, B_{\text{sup}}]$. The following code performs this using four matrix multiplications:

$$\begin{aligned} &A_- = \min(A,\ 0); \quad A_+ = \max(A,\ 0); \\ &\text{setround}(-1); \\ &C_{\text{inf}} = A_+ \cdot B_{\text{inf}} + A_- \cdot B_{\text{sup}}; \\ &\text{setround}(1); \\ &C_{\text{sup}} = A_- \cdot B_{\text{inf}} + A_+ \cdot B_{\text{sup}}; \end{aligned}$$

(Note that the commands $A_- = \min(A,\ 0)$ and $A_+ = \max(A,\ 0)$ act componentwise.) An algorithm for computing the product of two interval matrices using eight matrix multiplications is given by Rump [308].

Fast portable codes for interval matrix computations that make use of the Basic Linear Algebra Subroutines (BLAS) and IEEE 754 standard are now available. This makes it possible to efficiently use high-performance computers for interval computation. INTLAB (INTerval LABoratory) by Rump [308, 307] is based on MATLAB, and it is particularly easy to use. It includes many useful subroutines, for example, one to compute an enclosure of the difference between the solution and an approximate solution $x_m = C\text{mid}\,[b]$. Verified solutions of linear least squares problems can also be computed.

Review Questions

2.5.1 (a) Define the magnitude and mignitude of an interval $I = [a, b]$.

(b) How is the ∞-norm of an interval vector defined?

2.5.2 Describe the two different ways of representing intervals used in real and complex interval arithmetic. Mention some of the advantages and drawbacks of each of these.

2.5.3 An important property of interval arithmetic is that the operations are inclusion monotonic. Define this term.

2.5.4 What is meant by the "wrapping effect" in interval arithmetic and what are its implications? Give some examples of where it occurs.

2.5.5 Assume that the command

$$\text{setround}(i), \quad i = -1, 0, 1,$$

sets the rounding mode to $-\infty$, to nearest, and to $+\infty$, respectively. Give a simple code that, using two matrix multiplications, computes an interval matrix $[C]$ such that for point matrices A and B

$$fl(A \cdot B) \subset [C] = [C_{\text{inf}}, C_{\text{sup}}].$$

Problems and Computer Exercises

2.5.1 Carry out the following calculations in exact interval arithmetic:

(a) $[0, 1] + [1, 2]$; (b) $[3, 3.1] - [0, 0, 2]$; (c) $[-4. - 1] \cdot [-6, 5]$;

(d) $[2, 2] \cdot [-1, 2]$; (e) $[-1, 1]/[-2, -0.5]$; (f) $[-3, 2] \cdot [-3.1, 2.1]$.

2.5.2 Show that using the operational definitions (2.5.5) the product of the disks $\langle c_1, r_1\rangle$ and $\langle c_2, r_2\rangle$ contains zero if $c_1 = c_2 = 1$ and $r_1 = r_2 = \sqrt{2} - 1$.

2.5.3 (Stoer) Using Horner's rule and exact interval arithmetic, evaluate the cubic polynomial

$$p(x) = ((x - 3)x + 3)x, \quad [x] = [0.9, 1.1].$$

Compare the result with the exact range, which can be determined by observing that $p(x) = (x - 1)^3 + 1$.

2.5.4 Treat Example 1.2.2 using interval analysis and four decimal digits. Starting with the inclusion interval $I_{10} = [0, 1/60] = [0, 0.01667]$ generate successively intervals I_k, $k = 9 : -1 : 5$, using interval arithmetic and the recursion

$$I_{n-1} = 1/(5n) - I_n/5.$$

Notes and References

Many different aspects of number systems and floating-point computations are treated in Knuth [230, Chapter 4], including the historical development of number representation. Leibniz (1703) seems to have been the first to discuss binary arithmetic. He did not advocate it for practical calculations, but stressed its importance for number-theoretic investigations. King Charles XII of Sweden came upon the idea of radix 8 arithmetic in 1717. He felt this to be more convenient than the decimal notation and considered introducing octal arithmetic into Sweden. He died in battle before decreeing such a change.

In the early days of computing, floating-point computations were not built into the hardware but implemented in software. The earliest subroutines for floating-point arithmetic were probably those developed by J. H. Wilkinson at the National Physical Laboratory, England, in 1947. A general source on floating-point computation is Sterbenz [333]. Goldberg [158] is an excellent tutorial on the IEEE 754 standard for floating-point arithmetic defined in [205]. A self-contained, accessible, and easy to read introduction with many illustrating examples is the monograph by Overton [280]. An excellent treatment on floating-point computation, rounding error analysis, and related topics is given in Higham [199, Chapter 2]. Different aspects of accuracy and reliability are discussed in [108].

The fact that thoughtless use of mathematical formulas and numerical methods can lead to disastrous results are exemplified by Stegun and Abramowitz [331] and Forsythe [122]. Numerous examples in which incorrect answers are obtained from plausible numerical methods can be found in Fox [125].

Statistical analysis of rounding errors goes back to an early paper of Goldstine and von Neumann [161]. Barlow and Bareiss [15] have investigated the distribution of rounding errors for different modes of rounding under the assumption that the mantissas of the operands are from a logarithmic distribution.

Conditioning numbers of general differentiable functions were first studied by Rice [301]. Backward error analysis was developed and popularized by J. H. Wilkinson in the 1950s and 1960s, and the classic treatise on rounding error analysis is [376]. The more recent survey [380] gives a good summary and a historical background. Kahan [216] gives an in-depth discussion of rounding error analysis with examples of how flaws in the design of hardware and software in computer systems can have undesirable effects on accuracy. The normwise analysis is natural for studying the effect of orthogonal transformations in matrix computations; see Wilkinson [376]. The componentwise approach, used in perturbation analysis for linear systems by Bauer [19], can give sharper results and has gained in popularity.

Condition numbers are often viewed pragmatically as the coefficients of the backward errors in bounds on forward errors. Wilkinson in [376] avoids a precise definition of condition numbers in order to use them more freely. The more precise limsup definition in Definition 2.4.6 is usually attributed to Rice [301].

Even in the special literature, the discussion of planning of experimental perturbations is surprisingly meager. An exception is the collection of software tools called PRECISE, developed by Chaitin-Chatelin et al.; see [63, 64]. These are designed to help the user set up computer experiments to explore the impact of the quality of convergence of numerical methods. PRECISE involves a statistical analysis of the effect on a computed solution of random perturbations in data.