Numerical Computing with IEEE Floating Point Arithmetic

Michael L. Overton

With one Theorem, one Rule of Thumb,
and one hundred and one Exercises

Draft of Second Edition

With extensive updated text, two new chapters and new exercises

January 8, 2024

Contents

Accurate reckoning: The entrance into knowledge
of all existing things and all obscure secrets

— A'HMOSÈ, *The Rhind Mathematical Papyrus*, c. 1650 B.C.E.

I am a HAL Nine Thousand computer Production Number 3. I became
operational at the Hal Plant in Urbana, Illinois, on January 12, 1997.
The quick brown fox jumps over the lazy dog.
The rain in Spain is mainly in the plain.
Dave—are you still there?
Did you know that the square root of 10 is 3.162277660168379?
Log 10 to the base e is 0.434294481903252 . . .
correction, that is, log e to the base 10 . . .
The reciprocal of 3 is 0.33333333333333333333 . . .
2 times 2 is . . . 2 times 2 is . . .
approximately 4.10101010101010101010 . . .
I seem to be having difficulty . . .

—HAL, in *2001: A Space Odyssey*

Chapter 1

Introduction

Numerical computing means *computing with numbers*, and the subject is almost as old as civilization itself. Ancient peoples knew techniques to carry out many numerical tasks. Among the oldest computational records that we have is the Egyptian Rhind Papyrus from about 1650 B.C.E. [Cha79], quoted on the previous page. Counting stones and counting rods have been used for calculation for thousands of years; the abacus originated as a flat surface with counting stones and was used extensively in the ancient world long before it evolved into the device with beads on wires that was common in Asia until recently. The abacus was the basis of calculation in Europe until the introduction of our familiar positional decimal notation from the Middle East, beginning in the 13th century. By the end of the 16th century, positional decimal notation was in standard use throughout Europe, as it became widely recognized for its computational convenience.

The next key development was the invention and tabulation of logarithms by John Napier at the beginning of the 17th century; his idea was that time-consuming multiplication and especially division may be avoided by adding or subtracting logarithms, using tabulated values. Isaac Newton laid the foundations of modern numerical computing later in the 17th century, developing numerical techniques for the solution of many mathematical problems and inventing calculus along the way. Several of Newton's computational methods still bear his name. In Newton's footsteps followed Euler, Lagrange, Gauss, and many other great mathematicians of the 18th and 19th centuries.

The idea of using physical devices as an aid to calculation is an old one. The abacus has already been mentioned. The slide rule was invented soon after Napier's discovery of logarithms, although it was not commonly used until the middle of the 19th century. Numbers are represented on a slide rule explicitly in a logarithmic scale, and its moving rule and cursor allow multiplication and division to be carried out easily, accurate to about three decimal digits. This simple, inexpensive device was used by many generations of engineers and remained in common use until about 1975, when it was made obsolete by cheap electronic calculators. Mechanical calculating machines were devised by Schickard, Pascal, and Leibnitz in the 17th century; their descendants also remained in use until about 1975. The idea of a programmable machine that would operate without human intervention was developed in great depth by Charles Babbage in the 19th century, but his ideas were way ahead of his time and were mostly ignored. During World War II, scientific laboratories had rooms full of people doing different parts of a complicated calculation using pencil and paper, slide rules, and mechanical calculators. At that time, the word *computer* referred to a person, and those group calculations may be viewed as the early steps of parallel computing.

The Computer Age

The machine often described as the world's first operating computer was the Z3, built by the engineer Konrad Zuse in Germany in 1939–1941. The Z3 used electromechanical switching devices and computed with binary floating point numbers, a concept to be described in detail in subsequent chapters.[1] Slightly later, and in great secrecy, the British government developed a powerful electronic code-breaking machine, the Colossus. The first general-purpose operational electronic computer[2] is usually said to be the ENIAC (Electronic Numerical Integrator And Computer), a decimal machine with 18,000 vacuum tubes that was built by Eckert and Mauchly at the University of Pennsylvania in 1943–1945. Eckert was the electronics expert and Mauchly had the experience with extensive numerical computations. Two intellectual giants who strongly influenced postwar computer development in England and the United States respectively were Alan Turing, one of the founders of theoretical computer science, and John von Neumann, the Hungarian mathematician at Princeton. Important ideas that were advocated by von Neumann included the storage of instructions in the memory of the computer and the use of binary rather than decimal storage and arithmetic. Other key leaders included Maurice Wilkes and James Wilkinson in England and Herman Goldstine in the U.S. In the late 1940s and early 1950s, it was feared that the rounding errors inherent in floating point computing would make nontrivial calculations too inaccurate to be useful. Wilkinson demonstrated conclusively that this was not the case with his extensive computational experiments and innovative analysis of rounding errors accumulated in the course of a computation. Wilkinson's analysis was inspired by the work of von Neumann and Goldstine and of Turing [Wil63].[3] For more on the early history of computers, see [Wil85] as well as the more recent works [Dys12] and [Bha22, Ch. 5] for the roles of Turing and von Neumann, respectively. For a remarkable collection of essays by a cast of stars from the early days of computing, see [MHR80].

During the 1950s, the primary use of computers was for numerical computing in scientific applications. In the 1960s, computers became widely used by large businesses, but their purpose was not primarily numerical; instead, the principal use of computers became the processing of large quantities of information. Nonnumerical information, such as character strings, was represented in the computer using binary numbers, but the primary business applications were not numerical in nature. During subsequent decades, computers became ever more widespread, becoming available to medium-sized businesses in the 1970s, to many millions of small businesses and individuals during the personal computer revolution of the 1980s and 1990s, and to billions of people all around the world with the advent of inexpensive mobile phones in the 2000s. The vast majority of these computer users do not see computing with numbers as their primary interest; instead, they are interested in the processing of information, such as text, images, and sound. However, manipulation of images and sound requires extensive numerical computing. Most recently, the 2010s and early 2020s have seen the astonishingly rapid rise of machine learning, which involves massive amounts

[1]Ideas that seem to originate with Zuse include the hidden significand bit [Knu98, p. 227], to be discussed in Chapter 3, the use of ∞ and NaN [Kah96b], to be discussed in Chapter 7, the main ideas of algorithmic programming languages [Wil85, p. 225], and perhaps the concept of a stored program [Zus93, p. 44]. His autobiography [Zus93] gives an amazing account of his successful efforts at computer design and construction amid the chaos of World War II.

[2]A much more limited machine was developed a little earlier in Iowa.

[3]In his 1970 SIAM von Neumann lecture, Wilkinson gave a historical survey of rounding error analysis stating that the results of von Neumann and Goldstine compare much more favorably with later work than is generally supposed [Wil71]. He also said that it was only when he had done a good deal of error analysis himself that he came to appreciate the pioneering work of Wallace Givens.

of numerical computation and has huge impact on many aspects of our lives.

Science Today

In scientific disciplines, numerical computing is essential. Physicists use computers to solve complicated equations modeling everything from the expansion of the universe to the microstructure of the atom, and to test their theories against experimental data. Chemists and biologists use computers to determine the molecular structure of proteins. Medical researchers use computers for imaging techniques and for the statistical analysis of experimental and clinical observations. Atmospheric scientists use numerical computing to process huge quantities of data and to solve equations to predict the weather. Electronics engineers design ever faster, smaller, and more reliable computers using numerical simulation of electronic circuits. Design of cars, airplanes and spacecraft involves extensive modeling using numerical computing.

In brief, all fields of science and engineering rely heavily on numerical computing. The traditional two branches of science are theoretical science and experimental science. *Computational science* has long been described as a third branch, having a status that is essentially equal to, perhaps even eclipsing, that of its two older siblings. The availability of greatly improved computational techniques and immensely faster computers allows the routine solution of complicated problems that would have seemed impossible just a generation ago.

Chapter 2

The Real Numbers

The *real* numbers can be represented conveniently by a line. Every point on the line corresponds to a real number, but only a few are marked in Figure 2.1. The line stretches infinitely far in both directions, toward ∞ and $-\infty$, which are not themselves numbers in the conventional sense but are included among the *extended real* numbers. The *integers* are the numbers $0, 1, -1, 2, -2, 3, -3, \ldots$. We say that there is an *infinite* but *countable* number of integers; by this we mean that every integer would eventually appear in the list if we count for long enough, even though we can never count all of them. The *rational* numbers are those that consist of a ratio of two integers, e.g., 1/2, 2/3, 6/3; some of these, e.g., 6/3, are integers. To see that the number of rational numbers is countable, imagine the nonzero rationals listed in an infinite two-dimensional array as in Figure 2.2. Listing the first line and then the second, and so on, does not work, since the first line never terminates. Instead, we generate a list of all rational numbers diagonal by diagonal: first 0, then $\pm 1/1$; then $\pm 2/1, \pm 1/2$; then $\pm 3/1, \pm 2/2, \pm 1/3$; then $\pm 4/1, \pm 3/2, \pm 2/3, \pm 1/4$; etc. In this way, every rational number (including every integer) is eventually generated. In fact, every nonzero rational number is generated many times (e.g., 1/2 and 2/4 are the same number). However, every rational number does have a unique representation in lowest terms, achieved by canceling any common factor in the numerator and denominator (thus 2/4 reduces to 1/2).

The *irrational* numbers are the real numbers that are not rational. Familiar examples of irrational numbers are $\sqrt{2}$, π, and e. The numbers $\sqrt{2}$ and π have been studied for more than two thousand years. The number e, mentioned in the quote from HAL on page vi, is the limit of

$$\left(1 + \frac{1}{n}\right)^n$$

as $n \to \infty$. Investigations leading to the definition of e began in the 17th century.

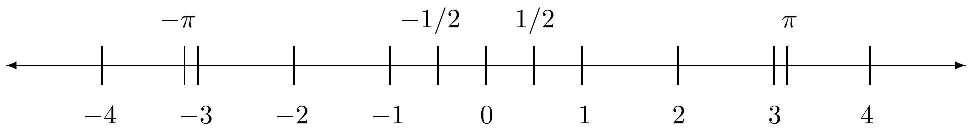

Figure 2.1: The Real Line

	1	2	3	4	...
1	$\pm 1/1$	$\pm 1/2$	$\pm 1/3$	$\pm 1/4$	...
2	$\pm 2/1$	$\pm 2/2$	$\pm 2/3$	$\pm 2/4$	...
3	$\pm 3/1$	$\pm 3/2$	$\pm 3/3$	$\pm 3/4$	...
4	$\pm 4/1$	$\pm 4/2$	$\pm 4/3$	$\pm 4/4$	...
...	...	...	...	...	...

Figure 2.2: The Nonzero Rational Numbers

Every irrational number can be defined as the limit of a sequence of rational numbers, but there is no way of listing all the irrational numbers—the set of irrational numbers is said to be *uncountable*.

Positional Number Systems

The idea of representing numbers using powers of 10 was used by many ancient peoples, e.g., the Hebrews, the Greeks, the Romans, and the Chinese, but the positional number system we use today was not. The Romans used a system where each power of 10 required a different symbol: X for 10, C for 100 = 10^2, M for 1000 = 10^3, etc., and repetition, together with additional symbols for quinary groupings, was used to indicate how many of each power of 10 were present. For example, MDCCCCLXXXV means 1000 + 500 + 400 + 50 + 30 + 5 = 1985. The familiar abbreviations such as IV for 4 were not used by the Romans. The Chinese system, which is still in use, is similar except that instead of repetition, symbols for the numbers 1 through 9 are used to modify each power of 10. These systems allowed easy transcription of numbers to an abacus for calculation, although they are not convenient for calculation with pencil and paper.

Large numbers cannot be conveniently represented by such systems. The positional notation used worldwide today requires a key idea: the representation of zero by a symbol. As far as we know, this was first used by the Babylonians about 300 B.C.E. Our decimal positional system was developed in India around 600 C.E. and was used for centuries by the Arabs in the Middle East before being passed on to Europe during the period 1200–1600—hence the name "Arabic numerals." This decimal, or base 10, system requires 10 symbols, representing the numbers 0 through 9. The system is called *positional* (or place-value) because the meaning of the number is understood from the position of the symbols, or *digits*, of the number. Zero is needed, for example, to distinguish 601 from 61. The reason for the decimal choice is the simple biological fact that humans have 10 fingers and thumbs. Indeed, the word *digit* derives from the Latin word for finger. Other positional systems developed by ancient peoples include the base 60 system used by the Babylonians, the vestiges of which are still seen today in our division of the hour into 60 minutes and the minute into 60 seconds, and the base 20 system developed by the Mayans, which was used for astronomical calculations. The Mayans are the only people known to have invented the positional number system, with its crucial use of a symbol for zero, independently of the Babylonians.

Decimal notation was initially used only for integers and was not used much for fractions until the 17th century. A reluctance to use decimal fractions is still evident in the use of quarters, eighths, sixteenths, etc., for machine tool sizes in the United States (and, until 2001, for stock market prices).

Although decimal representation is convenient for people, it is not particularly convenient for use on computers. The binary, or base 2, system is much more useful: in this, every number is represented as a string of *bits*, each of which is either 0 or 1.

The word *bit* is an abbreviation for *binary digit*; a *bitstring* is a string of bits. Each bit corresponds to a different power of 2, just as each digit of a decimal number corresponds to a different power of 10. Computer storage devices are all based on binary representation: the basic unit is also called a bit, which may be viewed as a single physical entity that is either "off" or "on." Bits in computer storage are organized in groups of 8, each called a *byte.* A byte can represent any of $256 = 2^8$ (2 to the power 8) different bitstrings, which may be viewed as representing the integers from 0 to 255. Alternatively, we may think of these 256 different bitstrings as representing 256 different *characters.*[1] A *word* is 4 consecutive bytes of computer storage (i.e., 32 bits), and a *double word* is 8 consecutive bytes (64 bits). Because of the importance of powers of two in organizing computer storage, traditionally a kilobyte means 2^{10} (1024) bytes, a megabyte 2^{20} bytes, a gigabyte 2^{30} bytes, a terabyte 2^{40} bytes, a petabyte 2^{50} bytes, and an exabyte 2^{60} bytes. However, in other scientific contexts, the prefixes *kilo, mega, giga, tera, peta* and *exa* generally mean $10^3, 10^6, 10^9, 10^{12}, 10^{15}$ and 10^{18}, respectively. Although 2^{10} is only 2.4% bigger than 10^3, the number 2^{60} is more than 15% bigger than 10^{18}, so the difference between the power of 2 and the power of 10 conventions is becoming more significant as the numbers grow bigger. Consequently, the terms *kibibyte*, *exbibyte*, etc., have been introduced to distinguish 2^{10} from 10^3 bytes, 2^{60} from 10^{18} bytes, etc. The next prefixes in the traditional sequence are *zetta* and *yotta*, but they are not in common use — yet.

Although the binary system was not in wide use before the computer age, the idea of representing numbers as sums of powers of 2 is far from new. It was used as the basis for a multiplication algorithm described in the *Rhind Mathematical Papyrus* [Cha79], written nearly four millennia ago (see p. vi).

Binary and Decimal Representation

Every real number has a decimal representation and a binary representation (and, indeed, a representation in a base equal to any integer greater than 1). Instead of representation, we sometimes use the word *expansion.* The representation of integers is straightforward, requiring an expansion in nonnegative powers of the base. For example, consider the number

$$(71)_{10} = 7 \times 10 \; + \; 1$$

and its binary equivalent

$$(1000111)_2 = 1 \times 64 \; + \; 0 \times 32 \; + \; 0 \times 16 \; + \; 0 \times 8 \; + \; 1 \times 4 \; + \; 1 \times 2 \; + \; 1 \times 1. \quad (2.1)$$

Nonintegral real numbers have digits (or bits) to the right of the decimal (or binary) point; these expansions may be finite or nonterminating. For example, 11/2 has the expansions

$$\frac{11}{2} = (5.5)_{10} = 5 \times 1 \; + \; 5 \times \frac{1}{10}$$

and

$$\frac{11}{2} = (101.1)_2 = 1 \times 4 \; + \; 0 \times 2 \; + \; 1 \times 1 \; + \; 1 \times \frac{1}{2}. \quad (2.2)$$

Both of these expansions terminate. However, the number 1/10, which obviously has the finite decimal representation $(0.1)_{10}$, does not have a finite binary representation.

[1]The ASCII encoding scheme defines standard character interpretations for the first 128 of these bitstrings; Unicode is an extension that initially allowed up to 2^{16} two-byte characters, but has now been extended beyond this limitation, allowing encoding of virtually all written languages in the world — as well as many emojis.

Instead, it has the nonterminating expansion

$$\frac{1}{10} = (0.0001100110011\ldots)_2 = \frac{1}{16} + \frac{1}{32} + \frac{0}{64} + \frac{0}{128} + \frac{1}{256} + \frac{1}{512} + \frac{0}{1024} + \cdots. \quad (2.3)$$

Note that this representation, although nonterminating, is *repeating.* The fraction 1/3 has nonterminating expansions in both binary and decimal:

$$\frac{1}{3} = (0.333\ldots)_{10} = (0.010101\ldots)_2.$$

Rational numbers always have either finite or repeating expansions. For example,

$$\frac{1}{7} = (0.142857142857\ldots)_{10}.$$

In fact, any finite expansion can also be expressed as a repeating expansion. For example, 1/10 can be expressed as

$$\frac{1}{10} = (0.0999\ldots)_{10}.$$

However, we will use the finite expansion when it exists.

Irrational numbers always have nonterminating, nonrepeating expansions. For example,

$$\sqrt{2} = (1.414213\ldots)_{10},\ \pi = (3.141592\ldots)_{10},\ e = (2.71828182845\ldots)_{10}.$$

The first 10 digits of e may suggest that its representation is repeating, but it is not.

Exercise 2.1 *Conversion of integers from binary representation to decimal is straightforward, because we are so familiar with the decimal representations of the powers of* 2*. Devise (or look up) a systematic method to convert the decimal representation of an integer to binary. Which do you find more convenient: determining the bits from left to right, or from right to left? Both methods are acceptable, but once you get the idea, one of them is easier to use systematically than the other. Test your choice on some examples and convert the binary results back to decimal as a check. Does your method extend to convert a finite decimal representation of a nonintegral rational number, such as* 0.1*, to its binary representation?*

When working with binary numbers, it's convenient to use the more compact hexadecimal notation (base 16), with the symbols 0,...,9,A,...,F representing the bitstrings 0000 through 1111.

Chapter 3

Computer Representation of Numbers

What is the best way to represent numbers on the computer? Let us start by considering integers. Typically, integers are stored using a 32-bit word, so we confine our attention to this case. If we were concerned only with nonnegative integers, the representation would be easy: a bitstring specifying the binary representation of the integer. For example, the integer 71 (see (2.1)) would be stored as

$$\boxed{00000000000000000000000001000111}\,.$$

The nonnegative integers that we can represent in this way range from 0 (a bitstring of 32 zeros) to $2^{32}-1$ (a bitstring of 32 ones). The number 2^{32} is too big, since its binary representation consists of a one followed by 32 zeros.

Signed Integers via 2's Complement

In fact, we need to be able to represent negative integers in addition to positive integers and 0. The most obvious idea is *sign-and-modulus*: use one of the 32 bits to represent the sign, and use the remaining 31 bits to store the magnitude of the integer, which may then range from 0 to $2^{31}-1$. However, nearly all machines use a more clever representation called 2's *complement*.[1] A nonnegative integer x, where $0 \le x \le 2^{31}-1$, is stored as the binary representation of x, but a negative integer $-y$, where $1 \le y \le 2^{31}$, is stored as the binary representation of the positive integer

$$2^{32} - y. \tag{3.1}$$

For example, the integer -71 is stored as

$$\boxed{11111111111111111111111110111001}\,.$$

In order to see that this is correct, let us add the 2's complement representations for 71 and -71 together:

$$\begin{array}{rl} & (\ \ 00000000000000000000000001000111\ \ )_2 \\ + & (\ \ 11111111111111111111111110111001\ \ )_2 \\ = & (\,100000000000000000000000000000000\ \ )_2. \end{array}$$

[1]There is a third system called 1's complement, where a negative integer $-y$ is stored as the binary representation of $2^{32}-y-1$. This system was used by some supercomputers in the 1960s and 1970s but is now obsolete.

Adding in binary by hand is like adding in decimal. Proceed bitwise right to left; when 1 and 1 are added together, the result is 10 (base 2), so the resulting bit is set to 0 and the 1 is carried over to the next bit to the left. The sum of the representations for 71 and for -71 is thus the bitstring for 2^{32}, as required by the definition (3.1). The bit in the leftmost position of the sum cannot be stored in the 32-bit word and is called an *overflow bit*. If it is discarded, the result is 0—exactly what we want for the result of $71 + (-71)$. This is the motivation for the 2's complement representation.

Exercise 3.1 *Using a* 32*-bit word, how many different integers can be represented by* (a) *sign and modulus;* (b) 2*'s complement? Express the answer using powers of* 2*. For which of these two systems is the representation for zero unique?*

Exercise 3.2 *Suppose we wish to store integers using only a* 16*-bit half-word (*2 *bytes). This is called a short integer format. What is the range of integers that can be stored using* 2*'s complement? Express the answer using powers of* 2 *and also translate the numbers into decimal notation.*

Exercise 3.3 *Using an* 8*-bit format for simplicity, give the* 2*'s complement representation for the following integers:* 1*,* 10*,* 100*,* -1*,* -10*, and* -100*. Verify that addition of a negative number to its positive counterpart yields zero, as required, when the overflow bit is discarded.*

Exercise 3.4 *Show that if an integer x between* -2^{31} *and* $2^{31}-1$ *is represented using* 2*'s complement in a* 32*-bit word, the leftmost bit is* 1 *if x is negative and* 0 *if x is positive or* 0*.*

Exercise 3.5 *An easy way to convert the representation of a nonnegative integer x to the* 2*'s complement representation for $-x$ begins by changing all* 0 *bits to* 1*s and all* 1 *bits to* 0*s. One more step is necessary to complete the process; what is it, and why?*

All computers provide hardware instructions for adding integers. If two positive integers are added together, the result may give an integer greater than or equal to 2^{31}. In this case, we say that *integer overflow* occurs. One would hope that this leads to an informative error message for the user, but whether or not this happens depends on the programming language and compiler being used. In some cases, the overflow bits may be discarded and the programmer must be careful to prevent this from happening.[2] The same problem may occur if two negative integers are added together, giving a negative integer with magnitude greater than 2^{31}.

On the other hand, if two integers with opposite sign are added together, integer overflow cannot occur, although an overflow bit may arise when the 2's complement bitstrings are added together. Consider the operation

$$x + (-y),$$

where

$$0 \le x \le 2^{31} - 1 \quad \text{and} \quad 1 \le y \le 2^{31}.$$

Clearly, it is possible to store the desired result $x - y$ without integer overflow. The result may be positive, negative, or zero, depending on whether $x > y$, $x = y$, or $x < y$. Now let us see what happens if we add the 2's complement representations for

[2]The IEEE floating point standard, to be introduced in the next chapter, says nothing about requirements for integer arithmetic.

x and $-y$, i.e., the bitstrings for the nonnegative numbers x and $2^{32} - y$. We obtain the bitstring for

$$x + (2^{32} - y) = 2^{32} + (x - y) = 2^{32} - (y - x).$$

If $x \geq y$, the leftmost bit of the result is an overflow bit, corresponding to the power 2^{32}, but this bit can be discarded, giving the correct result $x - y$. If $x < y$, the result fits in 32 bits with no overflow bit, so we have the desired result in this case too, as it represents the negative value $-(y - x)$ in 2's complement.

This demonstrates an important property of 2's complement representation: no special hardware is needed for integer subtraction. The addition hardware can be used once the negative number $-y$ has been represented using 2's complement.

Exercise 3.6 *Show the details for the integer sums* $50 + (-100)$, $100 + (-50)$, *and* $50 + 50$, *using an* 8*-bit format.*

Besides addition, the other standard hardware operations on integer operands are multiplication and division. Consider two positive integers x and y, both of which are less than 2^{32}, so both can be stored in a 32-bit format. The product $x \times y$ must be less than 2^{64}, so it could always be stored in a 64-bit double word, but we normally want to store the result in the same 32-bit format, and this is possible only if $x \times y$ is less than 2^{32}, i.e., the leading 32 bits of the 64-bit product $x \times y$ are zero. If not, integer overflow occurs. On the other hand, again assuming x and y are positive 32-bit integers, the integer division operation x/y always yields an integer quotient that fits in the same format, provided that we discard any remainder. Integer division by zero normally leads to program termination and an error message for the user.

Exercise 3.7 *Given an example of two 3-bit binary numbers x and y whose product is 6 bits with no leading zeros.*

Exercise 3.8 *(D. Goldberg) Besides division by zero, is there any other division operation that could result in integer overflow?*

Fixed Point

Now let us turn to the representation of nonintegral real numbers. Rational numbers could be represented by pairs of integers, the numerator and denominator. This has the advantage of accuracy but the disadvantage of being very inconvenient for arithmetic. Instead, for most numerical computing purposes, real numbers, whether rational or irrational, are approximately stored using the binary representation of the number. There are two standard methods, called fixed point and floating point.

In *fixed point* representation, the computer word may be viewed as divided into three fields: one 1-bit field for the sign of the number, one field of bits for the binary representation of the number before the binary point, and one field of bits for the binary representation after the binary point. For example, in a 32-bit word with field widths of 15 and 16, respectively, the number 11/2 (see (2.2)) would be stored as

0	000000000000101	1000000000000000

,

while the number 1/10 (see (2.3)) would be approximately stored as

0	000000000000000	0001100110011001

.

The fixed point system is severely limited by the size of the numbers it can store. In the example just given, only numbers ranging in size from (exactly) 2^{-16} to (slightly less than) 2^{15} could be stored. This limitation motivates the use of floating point representation.

Floating Point

Floating point representation is based on *exponential* (or *scientific*) notation. In exponential notation, a nonzero real number x is expressed in decimal as

$$x = \pm S \times 10^E, \qquad \text{where } 1 \le S < 10,$$

and E is an integer. The numbers S and E are called the *significand*[3] and the *exponent*, respectively. For example, the exponential representation of 365.25 is 3.6525×10^2, and the exponential representation of 0.00036525 is 3.6525×10^{-4}. It is always possible to satisfy the requirement that $1 \le S < 10$, as S can be obtained from x by repeatedly multiplying or dividing by 10, decrementing or incrementing the exponent E accordingly. We can imagine that the *decimal point floats* to the position immediately after the first nonzero digit in the decimal expansion of the number—hence the name floating point.

For representation on the computer, we prefer base 2 to base 10, so we write a nonzero number x in the form

$$x = \pm S \times 2^E, \qquad \text{where } 1 \le S < 2. \tag{3.2}$$

Consequently, the binary expansion of the significand is

$$S = (b_0.b_1b_2b_3\ldots)_2 \quad \text{with } \; b_0 = 1. \tag{3.3}$$

For example, the number 11/2 is expressed as

$$\frac{11}{2} = (1.011)_2 \times 2^2. \tag{3.4}$$

Now it is the *binary point* that *floats* to the position after the first nonzero bit in the binary expansion of x, changing the exponent E accordingly. Of course, this is not possible if the number x is zero, but at present we are considering only the nonzero case. Since b_0 is 1, we may write

$$S = (1.b_1b_2b_3\ldots)_2.$$

The bits following the binary point are called the *fractional* part of the significand. We say that (3.2), (3.3) is the *normalized* representation of x, and the process of obtaining it is called *normalization.*

To store normalized numbers, we divide the computer word into three fields to represent the sign, the exponent E, and the significand S, respectively. A 32-bit word could be divided into fields as follows: 1 bit for the sign, 8 bits for the exponent, and 23 bits for the significand. The sign bit is 0 for positive numbers and 1 for negative numbers. Since the exponent field is 8 bits, it can be used to represent exponents E between -128 and 127 (for example, using 2's complement, though this is not the way it is normally done). The 23 significand bits can be used to store the first 23 bits after the binary point in the binary expansion of S, namely, $b_1, \ldots, b_{23}$. It is not necessary to store b_0, since we know it has the value 1: we say that b_0 is a *hidden bit.* Of course, it might not be possible to store the number x with such a scheme, either because E is outside the permissible range -128 to 127 or because the bits $b_{24}, b_{25}, \ldots$ in the binary expansion of S are not all zero. A real number is called a *floating point number* if it can be stored *exactly* on the computer using the given floating point representation

[3]An older term for significand, but one that is still often used, is *mantissa.*

scheme. If a number x is not a floating point number, it must be *rounded* before it can be stored on the computer. This will be discussed later.

Using this idea, the number 11/2 (see (3.4)) would be stored as

$$\begin{array}{|c|c|c|}\hline 0 & \text{ebits}(2) & 01100000000000000000000 \\ \hline\end{array},$$

and the number

$$71 = (1.000111)_2 \times 2^6$$

would be stored as

$$\begin{array}{|c|c|c|}\hline 0 & \text{ebits}(6) & 00011100000000000000000 \\ \hline\end{array}.$$

For now, the bits in the exponent field are not shown explicitly, but written in the functional form "ebits(E)"; we will give details in the next chapter. Since the bitstring stored in the significand field is actually the *fractional part* of the significand, we also refer to this field as the *fraction field*. Given a string of bits in the fraction field, it is necessary to imagine that "1." appears in front of the string to obtain the significand.

In this scheme, if x is exactly a power of 2, so that the significand is the number 1.0, the bits stored in the fraction field are all 0 (since b_0 is not stored). For example,

$$1 = (1.000\ldots)_2 \times 2^0$$

would be stored as

$$\begin{array}{|c|c|c|}\hline 0 & \text{ebits}(0) & 00000000000000000000000 \\ \hline\end{array}$$

and the number

$$1024 = (1.000\ldots)_2 \times 2^{10}$$

would be stored as

$$\begin{array}{|c|c|c|}\hline 0 & \text{ebits}(10) & 00000000000000000000000 \\ \hline\end{array}.$$

Now consider the much larger number

$$2^{71} = (1.000\ldots)_2 \times 2^{71}.$$

This integer is much too large to store in a 32-bit word using the integer format discussed earlier. However, there is no difficulty representing it in floating point, using the representation

$$\begin{array}{|c|c|c|}\hline 0 & \text{ebits}(71) & 00000000000000000000000 \\ \hline\end{array}.$$

Exercise 3.9 *What is the largest floating point number in this system, assuming the significand field can store only the bits $b_1 \ldots b_{23}$ and the exponent is limited by $-128 \le E \le 127$? Don't forget that the hidden bit, b_0, is* 1.

Exercise 3.10 *What is the smallest positive floating point number in this system? Don't forget that the hidden bit, b_0, is* 1.

Exercise 3.11 *What is the smallest positive integer that is not exactly representable as a floating point number in this system?*

Exercise 3.12 *Suppose we change* (3.2) *so that the bounds on the significand are* $\frac{1}{2} \le S < 1$, *change* (3.3) *to*

$$S = (0.b_1b_2b_3b_4\ldots)_2, \qquad \textit{with} \;\; b_1 = 1,$$

and change our floating point system so that the significand field stores only the bits $b_2, \ldots, b_{24}$, *with the exponent limited by* $-128 \le E \le 127$ *as before. What is the largest floating point number in this system? What is the smallest positive floating point number in this system (remember that* $b_1 = 1$*)? What is the smallest positive integer that is not exactly representable as a floating point number in this system?*

If a number x does not have a finite binary expansion, we must terminate its expansion somewhere. For example, consider the number

$$1/10 = (0.0001100110011\ldots)_2.$$

If we truncate this to 23 bits after the binary point, we obtain

$$(0.00011001100110011001100)_2.$$

However, if we then normalize this to obtain

$$(1.1001100110011001100)_2 \times 2^{-4},$$

so that there is a 1 before the binary point, we find that we now have only 19 correct bits after the binary point. This leads to the unnecessarily inaccurate representation

0	ebits(−4)	10011001100110011000000

with the last 4 bits set to 0000 instead of 1100. Clearly, this is not a good idea. It is preferable to *first* normalize and *then* truncate, so that we retain 23 correct bits after the binary point:

0	ebits(−4)	10011001100110011001100

This way all the available bits are used. Note that it might be better to round the final bit up to 1. We will discuss this later.

Precision, Machine Epsilon, and Ulp

The *precision* of the floating point system is the number of bits in the significand (including the hidden bit). We denote the precision by p. In the system just described, $p = 24$ (23 stored bits in the fractional part of the significand and 1 leading hidden bit). Any normalized floating point number with precision p can be expressed as

$$x = \pm(1.b_1b_2\ldots b_{p-2}b_{p-1})_2 \times 2^E. \tag{3.5}$$

The smallest such x that is greater than 1 is

$$(1.00\ldots01)_2 = 1 + 2^{-(p-1)}.$$

We give a special name, *machine epsilon*,[4] to the gap between this number and the number 1, and we write this as

$$\epsilon_{\text{mch}} = (0.00\ldots01)_2 = 2^{-(p-1)}. \tag{3.6}$$

[4]We follow Higham [Hig02] in our definitions of ϵ_{mch} and ulp. Higham also uses the term *unit roundoff* to mean $\epsilon_{\text{mch}}/2$, that is, half the gap between 1 and the next larger floating point number; some authors call this the machine epsilon.

More generally, for a floating point number x given by (3.5) we define

$$\mathrm{ulp}(x) = (0.00\ldots01)_2 \times 2^E = 2^{-(p-1)} \times 2^E = \epsilon_{\mathrm{mch}} \times 2^E. \tag{3.7}$$

Ulp is short for *unit in the last place.* If $x > 0$, then $\mathrm{ulp}(x)$ is the gap between x and the next larger floating point number. If $x < 0$, $\mathrm{ulp}(x)$ is the gap between x and the next smaller floating point number (larger in absolute value). Note that $\mathrm{ulp}(1) = \epsilon_{\mathrm{mch}}$.

Exercise 3.13 *Let the precision* $p = 24$, *so* $\epsilon_{\mathrm{mch}} = 2^{-23}$. *Determine* $\mathrm{ulp}(x)$ *for* x *having the following values:* 0.25, 2, 3, 4, 10, 100, 1030. *Give your answer as a power of* 2*; do not convert this to decimal.*

The Special Number Zero

So far, we have discussed only nonzero numbers. The number *zero* is special. It cannot be normalized, since all the bits in its representation are zero. Thus, it cannot be represented using the scheme described so far. A pattern of all zeros in the significand field represents the significand 1.0, not 0.0, since the bit b_0= 1 is hidden. There are two ways to address this difficulty. The first, which was used by most floating point implementations until about 1975, is to give up the idea of a hidden bit and instead insist that the leading bit b_0 in the binary representation of a nonzero number must be stored explicitly, even though it is always 1. In this way, the number zero can be represented by a significand that has all zero bits. This approach effectively reduces the precision of the system by one bit because, to make room for b_0, we must give up storing the final bit (b_{23} in the system described above). The second approach is to use a special string in the exponent field to signal that the number is zero. This reduces by one the number of possible exponents E that are allowed for representing nonzero numbers. This is the approach taken by the IEEE standard, to be discussed in the next chapter. In either case, there is the question of what to do about the sign of zero. Traditionally, this was ignored, but we shall see a different approach in the next chapter.

The Toy Number System

It is quite instructive to suppose that the computer word size is much smaller than 32 bits and work out in detail what all the possible floating point numbers are in such a case. Suppose that all numbers have the form

$$\pm(b_0.b_1b_2)_2 \times 2^E,$$

with b_0 stored explicitly and all nonzero numbers required to be normalized. Thus, b_0 is allowed to be zero only if b_1 and b_2 are also zero, indicating that the number represented is zero. Suppose also that the only possible values for the exponent E are -1, 0, and 1. We shall call this system the *toy floating point number system.* The set of toy floating point numbers is shown in Figure 3.1.

The precision of the toy system is $p = 3$. The largest number is $(1.11)_2 \times 2^1 = (3.5)_{10}$, and the smallest positive number is $(1.00)_2 \times 2^{-1} = (0.5)_{10}$. Since the next floating point number bigger than 1 is 1.25, machine epsilon for the toy system is $\epsilon_{\mathrm{mch}} = 0.25$. Note that the gap between consecutive floating point numbers becomes *smaller* as the magnitudes of the numbers themselves get smaller, and *bigger* as the magnitudes get bigger. Specifically, consider the positive floating point numbers with $E = 0$: these are the numbers 1, 1.25, 1.5, and 1.75. For each of these numbers, say x, the gap between x and the next larger floating point number, i.e., $\mathrm{ulp}(x)$, is machine epsilon, $\epsilon_{\mathrm{mch}} = 0.25$. For the positive floating point numbers x with $E = 1$, the gap is twice as big, i.e., $\text{ulp}(x) = 2\epsilon_{\text{mch}}$, and for those x with $E = -1$, the gap is $\text{ulp}(x) = \frac{1}{2}\epsilon_{\text{mch}}$. Summarizing, the gap between a positive toy floating point number $x = (b_0.b_1b_2)_2 \times 2^E$ and the next bigger toy floating point number is

$$\text{ulp}(x) = \epsilon_{\text{mch}} \times 2^E,$$

as already noted in (3.7).

Another important observation to make about Figure 3.1 is that the gaps between zero and ± 0.5 are much greater than the gaps between numbers ranging from ± 0.5 to ± 1. We shall show in the next chapter how these gaps can be filled in with the introduction of *subnormal* numbers.

Exercise 3.14 *Suppose we add another bit to the toy number system, allowing significands of the form $b_0.b_1b_2b_3$, with b_0 stored explicitly as before and all nonzero numbers required to be normalized. The restrictions on the exponent are unchanged. Mark the new numbers on a copy of Figure* 3.1*.*

Fixed Point versus Floating Point

Some of the early computers used fixed point representation and some used floating point. Von Neumann was initially skeptical of floating point and promoted the use of fixed point representation. He was well aware that the range limitations of fixed point would be too severe to be practical, but he believed that the necessary scaling by a power of 2 should be done by the programmer, not the machine; he argued that bits were too precious to be wasted on storing an exponent when they could be used to extend the precision of the significand [Kah97]. Wilkinson experimented extensively with a compromise system called block floating point, where an automatic scale factor is maintained for a vector, i.e., for a block of many numbers, instead of one scale factor per number. This means that only the largest number (in absolute value) in the vector is sure to be normalized; if a vector contains numbers with widely varying magnitudes, those with smaller magnitudes are stored much less accurately. By the late 1950s it was apparent that the floating point system is far more versatile and efficient than fixed point or block floating point.[5]

Knuth [Knu98, pp. 196, 225] attributes the origins of floating point notation to the Babylonians. In their base 60 number system, zero was never used at the end of a number, and hence a power of 60 was always implicit. The Babylonians, like von Neumann, did not explicitly store their exponents.

[5]However, technology is always subject to change. In Higham's forward to the 2023 sixtieth anniversary reprint of Wilkinson's 1963 classic book [Wil23], he writes "Although floating-point arithmetic dominates today's computational landscape, fixed-point arithmetic is widely used in digital signal processing and block floating-point arithmetic is enjoying renewed interest in machine learning."

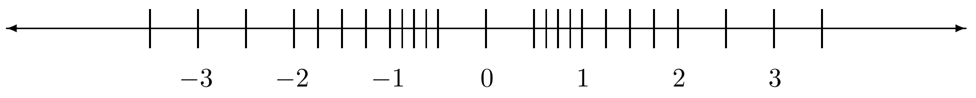

Figure 3.1: The Toy Floating Point Numbers

Chapter 4

IEEE Floating Point Representation

Floating point computation was in standard use by the mid 1950s. During the subsequent two decades, each computer manufacturer developed its own floating point system, leading to much inconsistency in how one program might behave on different machines. For example, although most machines developed during this period used binary floating point systems roughly similar to the one described in the previous chapter, the IBM 360/370 series, which dominated computing during the 1960s and 1970s, used a hexadecimal system (base 16). On these machines, the significand is stored using 24 bits, to be interpreted as 6 hexadecimal digits, leaving 1 bit for the sign and 7 bits for the exponent (representing a power of 16). Normalization requires only that the first hexadecimal digit be nonzero; consequently, the significand could have up to 3 leading zero bits. Therefore, the accuracy of the significands ranges from 21 to 24 bits; some numbers (such as 1/10; see (2.3)) are represented less accurately than on a binary machine. One motivation for this design was to reduce the bit shifting required during floating point add and subtract operations. Another benefit is that the hexadecimal base allows a much greater range of normalized floating point numbers than a binary system permits.

In addition to inconsistencies of representation, there were also many inconsistencies in the properties of floating point arithmetic. See Chapter 6 for examples of difficulties that could arise unexpectedly on some machines. Consequently, it was very difficult to write *portable software* that would work properly on all machines. Programmers needed to be aware of various difficulties that might arise on different machines and attempt to forestall them.

A Historic Collaboration: IEEE p754

In an extraordinary cooperation between academic computer scientists and microprocessor chip designers, a standard for binary floating point representation and arithmetic was developed in the late 1970s and early 1980s and, most importantly, was followed carefully by the microprocessor industry. As this was the beginning of the personal computer revolution, the impact was enormous. The scientists who wrote the standard did so under the auspices of the Institute for Electrical and Electronics Engineers in a working group known as IEEE p754.[1] The academic computer scientists

[1] IEEE is pronounced "I triple E." The IEEE official terminology was P754, but we follow the usage p754 in [Kah10, Sev98].

on the committee were led by William Kahan of the University of California at Berkeley; industrial participants included representatives from Apple, Digital Equipment Corporation (DEC), Intel, Hewlett-Packard, Motorola, and National Semiconductor. Kahan's interest in the project had been sparked originally by the efforts of John Palmer, of Intel, to ensure that Intel's new 8087 chip would have the best possible floating point arithmetic. An early document that included many of the ideas adopted by the standard was written in 1978–1979 by Kahan, Coonen, and Stone; see [Cod81]. Kahan was awarded the 1989 Turing Prize by the Association of Computing Machinery for his work in leading IEEE p754.

In [Sev98], Kahan recalled: "It was remarkable that so many hardware people there, knowing how difficult p754 would be, agreed that it should benefit the community at large. If it encouraged the production of floating-point software and eased the development of reliable software, it would help create a larger market for everyone's hardware. This degree of altruism was so astonishing that MATLAB's creator Cleve Moler used to advise foreign visitors not to miss the country's two most awesome spectacles: the Grand Canyon, and meetings of IEEE p754."

The IEEE standard for binary floating point arithmetic was published in 1985, when it became known officially as IEEE 754-1985 [IEE85]. In 1989, it received international endorsement as IEC 559, later designated ISO/IEC 60559. A second IEEE floating point standard, for radix-independent floating point arithmetic, IEEE 854-1987 [IEE87], was adopted in 1987. The second standard was motivated by the existence of decimal, rather than binary, floating point machines, particularly hand-held calculators, and set requirements for both binary and decimal floating point arithmetic in a common framework. The demands for binary arithmetic imposed by IEEE 854 are consistent with those previously established by IEEE 754. A substantial revision of IEEE 754, which discusses both binary and decimal floating point arithmetic and supersedes *both* earlier standards, was published in 2008 [IEE08]. Superficially, the 2008 revision looks very different from the 1985 version because it was completely rewritten with much new nomenclature, but the most important requirements and recommendations of the 1985 version remain unchanged.[2] Features introduced in 2008 include support for a fused multiply-add operation and the standardization of some additional floating point formats, along with new requirements and recommendations for programming language standards. Another revision was published in 2019, primarily making minor clarifications and changes [IEE19, Hou19]. Both revisions were subsequently endorsed as revisions of ISO/IEC 60559. In this book, we discuss only binary floating point, not decimal floating point, and when we write "the IEEE standard," we mean IEEE 754, sometimes distinguishing between the 1985, 2008 and 2019 versions. The term "IEEE arithmetic" is used to mean floating point arithmetic that is in compliance with the IEEE standard. As IEEE standards must be renewed every ten years to remain active, plans are already under way for the next revision of IEEE 754, expected in 2029.

In 2023, the IEEE honored the floating point standard with a milestone plaque proposed by Jerome Coonen and installed at the University of California, Berkeley. See the documentation in the milestone proposal [IEE23] as well as many other references given there for more on the history of the standard.

IEEE Floating Point Essentials

The IEEE standard has three very important requirements:

[2]All versions of the standard use the word "shall" to mean *is required to* and "should" to mean *is recommended to.*

- consistent representation of floating point numbers by all machines adopting the standard (discussed in this chapter);
- correctly rounded floating point operations, using various rounding modes (see Chapters 5 and 6);
- consistent treatment of exceptional situations such as division by zero (see Chapter 7).

In the basic IEEE formats, the leading bit of a normalized number is hidden, as described in the previous chapter. Thus, a special representation is needed for storing zero. However, zero is not the only number for which the IEEE standard has a special representation. Another special number, not used on older machines, is ∞. This allows the possibility of dividing a positive number by zero and storing a sensible mathematical result, namely ∞, instead of terminating with an overflow message. This turns out to be very useful, as we shall see later, although one must be careful about what is meant by such a result. One question that immediately arises is: what about $-\infty$? It turns out to be convenient to have representations for $-\infty$ as well as ∞, and for -0 as well as 0. We will give more details in Chapter 7, but note for now that -0 and 0 are *two different representations for the same number zero*, while $-\infty$ and ∞ represent *two very different numbers*. Another special value is NaN, which stands for "Not a Number" and is accordingly not a number at all, but an error pattern. This too will be discussed later. All of these special values, as well as others representing *subnormal* numbers, are represented through the use of two specific bit patterns (all zeros and all ones) in the exponent field.

The Single Format (binary32)

There are two widely used basic floating point formats, *single* (*binary32*) and *double* (*binary64*).[3] *Single format* numbers use a 32-bit word and their representations are summarized in Table 4.1. Let us discuss these in some detail. The first bit, denoted $\pm$, refers to the sign of the number, a zero bit being used to represent a positive sign. The next 8 bits, denoted $a_1a_2a_3\ldots a_8$, comprise the exponent field, and the last 23 bits, denoted $b_1b_2\ldots b_{23}$, give the fractional part of the significand. The first line in Table 4.1 shows that the representation for zero requires a special zero bitstring for the exponent field *as well as* a zero bitstring for the fraction field, i.e.,

$\pm$	00000000	00000000000000000000000

.

No other line in the table can be used to represent the number zero, for all lines except the first and the last represent normalized[4] numbers, with an initial bit equal to 1; this is the one that is hidden. In the case of the first line of the table, the hidden bit is 0, not 1. The 2^{-126} in the first line is confusing at first sight, but let us ignore that for the moment since $(0.000\ldots0)_2 \times 2^{-126}$ is certainly one way to write the number 0. In the case when the exponent field has a zero bitstring but the fraction field has a nonzero bitstring, the number represented is said to be *subnormal*.[5] Let us briefly

[3]The names *single format* and *double format* were used in the 1985 version of the standard; these were replaced by the names *binary32* and *binary64* in the 2008 revision. They are also often known as *single precision*, *float32* or *fp32*, and *double precision*, *float64* or *fp64*, respectively. The original names *single* and *double* remain widely used so they are the terms we mostly use.

[4]The word *normalized* was used in the 1985 version of IEEE 754. The word *normal* replaced it in IEEE 854 and in the 2008 version of IEEE 754, but we prefer *normalized*, partly because its usage long predates the IEEE standard, and partly because the word *normal* has so many meanings in other contexts.

[5]The word *denormalized* was used in the 1985 version of IEEE 754. The word *subnormal* replaced it in IEEE 854 and in the 2008 version of IEEE 754.

Table 4.1: IEEE Single Format (binary32)

$\pm$	$a_1a_2a_3\ldots a_8$	$b_1b_2b_3\ldots b_{23}$

If exponent bitstring $a_1\ldots a_8$ is	Then numerical value represented is
$(00000000)_2 = (0)_{10}$	$\pm(0.b_1b_2b_3\ldots b_{23})_2 \times 2^{-126}$
$(00000001)_2 = (1)_{10}$	$\pm(1.b_1b_2b_3\ldots b_{23})_2 \times 2^{-126}$
$(00000010)_2 = (2)_{10}$	$\pm(1.b_1b_2b_3\ldots b_{23})_2 \times 2^{-125}$
$(00000011)_2 = (3)_{10}$	$\pm(1.b_1b_2b_3\ldots b_{23})_2 \times 2^{-124}$
$\downarrow$	$\downarrow$
$(01111111)_2 = (127)_{10}$	$\pm(1.b_1b_2b_3\ldots b_{23})_2 \times 2^{0}$
$(10000000)_2 = (128)_{10}$	$\pm(1.b_1b_2b_3\ldots b_{23})_2 \times 2^{1}$
$\downarrow$	$\downarrow$
$(11111100)_2 = (252)_{10}$	$\pm(1.b_1b_2b_3\ldots b_{23})_2 \times 2^{125}$
$(11111101)_2 = (253)_{10}$	$\pm(1.b_1b_2b_3\ldots b_{23})_2 \times 2^{126}$
$(11111110)_2 = (254)_{10}$	$\pm(1.b_1b_2b_3\ldots b_{23})_2 \times 2^{127}$
$(11111111)_2 = (255)_{10}$	$\pm\infty$ if $b_1 = \cdots = b_{23} = 0$, NaN otherwise

postpone the discussion of subnormal numbers and go on to the other lines of the table.

All the lines of Table 4.1 except the first and the last refer to the normalized numbers, i.e., all the floating point numbers that are not special in some way. Note especially the relationship between the exponent bitstring $a_1a_2a_3\ldots a_8$ and the actual exponent E. We see that the exponent representation does not use either the sign-and-modulus or the 2's complement integer representation discussed in the previous chapter, but something called *biased representation* : the bitstring that is stored is the binary representation of $E + 127$. The number 127, which is added to the desired exponent E, is called the *exponent bias*. For example, the number $1 = (1.000\ldots0)_2\times2^0$ is stored as

0	01111111	00000000000000000000000

.

Here the exponent bitstring is the binary representation for $0 + 127$ and the fraction bitstring is the binary representation for 0 (the fractional part of 1.0). The number $11/2 = (1.011)_2 \times 2^2$ is stored as

0	10000001	01100000000000000000000

.

The number $1/10 = (1.100110011\ldots)_2 \times 2^{-4}$ has a nonterminating binary expansion. If we truncate this to fit the significand field size, we find that 1/10 is stored as

0	01111011	10011001100110011001100

.

We shall see other rounding options in the next chapter.

The range of exponent field bitstrings (also known as the *biased exponents*) for normalized numbers is 00000001 to 11111110 (the decimal numbers 1 through 254), representing actual exponents (also known as *unbiased exponents*) from $E_{\min} = -126$ to $E_{\max} = 127$. Note that $E_{\max}$ is also the exponent bias, and that $E_{\min} = -E_{\max}+1$. The smallest positive normalized number that can be stored is represented by

0	00000001	00000000000000000000000

,

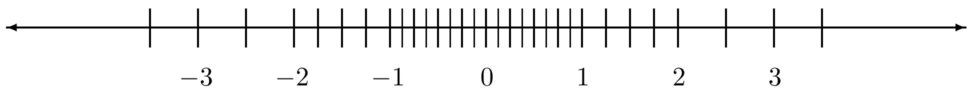

Figure 4.1: The Toy System Including Subnormal Numbers

and we denote this by

$$N_{\min} = (1.000\ldots0)_2 \times 2^{-126} = 2^{-126} \approx 1.2 \times 10^{-38}. \qquad (4.1)$$

The largest normalized number (equivalently, the largest finite number) is represented by

0	11111110	11111111111111111111111

,

and we denote this by

$$N_{\max} = (1.111\ldots1)_2 \times 2^{127} = (2 - 2^{-23}) \times 2^{127} \approx 2^{128} \approx 3.4 \times 10^{38}. \qquad (4.2)$$

The last line of Table 4.1 shows that an exponent bitstring consisting of all 1s is a special pattern used to represent $\pm\infty$ or NaN, depending on the fraction bitstring. We will discuss these in Chapter 7.

Subnormals

Finally, let us return to the first line of Table 4.1. The idea here is as follows: although 2^{-126} is the smallest normalized number that can be represented, we can use the combination of the special zero exponent bitstring and a nonzero fraction bitstring to represent smaller numbers called subnormal numbers. For example, 2^{-127}, which is the same as $(0.1)_2 \times 2^{-126}$, is represented as

0	00000000	10000000000000000000000

,

while $2^{-149} = (0.0000\ldots01)_2 \times 2^{-126}$ (with 22 zero bits after the binary point) is stored as

0	00000000	00000000000000000000001

.

This is the smallest positive number that can be stored, which we denote by $S_{\min}$. Now we see the reason for the 2^{-126} in the first line of the table. It allows us to represent numbers in the range immediately below the smallest positive normalized number. Subnormal numbers cannot be normalized, since normalization would result in an exponent that does not fit in the field.

Let us return to our example of the toy system with a tiny word size, illustrated in Figure 3.1, and see how the inclusion of subnormal numbers changes it. We get six extra numbers: $\pm(0.11)_2 \times 2^{-1} = \pm3/8$, $\pm(0.10)_2 \times 2^{-1} = \pm1/4$, and $\pm(0.01)_2 \times 2^{-1} = \pm1/8$; these are shown in Figure 4.1. Note that *the gaps between zero and* ±0.5 *are evenly filled in by the subnormal numbers*, using the same spacing as that between the numbers in the range ±0.5 to ±1.

The number of possible nonzeros in the significand field of a subnormal number depends on its magnitude. Specifically, the accuracy to which it can approximate a number drops as the size of the subnormal number decreases. Thus $(1/10) \times 2^{-123} = (0.11001100\ldots)_2 \times 2^{-126}$ is truncated to

$$\begin{array}{|c|c|c|}\hline 0 & 00000000 & 11001100110011001100110 \\ \hline \end{array},$$

while $(1/10) \times 2^{-135} = (0.11001100\ldots)_2 \times 2^{-138}$ is truncated to

$$\begin{array}{|c|c|c|}\hline 0 & 00000000 & 00000000000011001100110 \\ \hline \end{array}.$$

Exercise 4.1 *Determine the IEEE single format floating point representation for the following numbers:* 2, 30, 31, 32, 33, $23/4$, $(23/4)\times 2^{100}$, $(23/4)\times 2^{-100}$, *and* $(23/4)\times 2^{-135}$. *Truncating the significand as in the* $1/10$ *example, do the same for the numbers* $1/5 = (1/10) \times 2$, $1024/5 = (1/10) \times 2^{11}$, *and* $(1/10) \times 2^{-140}$, *using* (2.3) *to avoid decimal-to-binary conversions.*

Exercise 4.2 *What is the gap between* 2 *and the first IEEE single format number larger than* 2*? What is the gap between* 1024 *and the first IEEE single format number larger than* 1024*?*

Exercise 4.3 *Give an algorithm that, given two IEEE single format floating point numbers* x *and* y*, determines whether* x *is less than, equal to, or greater than* y*, by comparing their representations bitwise from left to right, stopping as soon as the first differing bit is encountered. Assume that neither* x *nor* y *is* ± 0, $\pm\infty$ *, or* NaN*. The fact that such a comparison can be done easily motivates biased exponent representation. It also justifies referring to the left end of the representation as the "most significant" end.*

Exercise 4.4 *This extends Exercise* 3.14*, which considered the toy number system with one additional bit in the significand. Mark the subnormal numbers in this system on the modified copy of Figure* 3.1 *that you used to answer Exercise* 3.14*.*

The Double Format (binary64)

The single format is not adequate for many applications, either because higher precision is desired or (less often) because a greater exponent range is needed. A second IEEE basic format is *double* (binary64), which uses a 64-bit double word. Details are shown in Table 4.2. The ideas are the same as before; only the field widths and exponent bias are different. Now there are 11 bits for the exponent field, the exponent bias is 1023, and the exponents range from $E_{\min} = -1022$ to $E_{\max} = 1023$, while the number of bits in the fraction field is 52. Numbers with no finite binary expansion, such as $1/10$ or π, are represented more accurately with the double format than they are with the single format. The smallest positive normalized double format number is

$$N_{\min} = 2^{-1022} \approx 2.2 \times 10^{-308} \tag{4.3}$$

and the largest is

$$N_{\max} = (2 - 2^{-52}) \times 2^{1023} \approx 1.8 \times 10^{308}. \tag{4.4}$$

Other IEEE Formats

The 1985 IEEE standard recommended support for an extended format, with, assuming that the *double* format is supported, the extended format having at least 15 bits

Table 4.2: IEEE Double Format (binary64)

$\pm$	$a_1a_2a_3\ldots a_{11}$	$b_1b_2b_3\ldots b_{52}$

If exponent bitstring is $a_1\ldots a_{11}$	Then numerical value represented is
$(00000000000)_2 = (0)_{10}$	$\pm(0.b_1b_2b_3\ldots b_{52})_2 \times 2^{-1022}$
$(00000000001)_2 = (1)_{10}$	$\pm(1.b_1b_2b_3\ldots b_{52})_2 \times 2^{-1022}$
$(00000000010)_2 = (2)_{10}$	$\pm(1.b_1b_2b_3\ldots b_{52})_2 \times 2^{-1021}$
$(00000000011)_2 = (3)_{10}$	$\pm(1.b_1b_2b_3\ldots b_{52})_2 \times 2^{-1020}$
$\downarrow$	$\downarrow$
$(01111111111)_2 = (1023)_{10}$	$\pm(1.b_1b_2b_3\ldots b_{52})_2 \times 2^{0}$
$(10000000000)_2 = (1024)_{10}$	$\pm(1.b_1b_2b_3\ldots b_{52})_2 \times 2^{1}$
$\downarrow$	$\downarrow$
$(11111111100)_2 = (2044)_{10}$	$\pm(1.b_1b_2b_3\ldots b_{52})_2 \times 2^{1021}$
$(11111111101)_2 = (2045)_{10}$	$\pm(1.b_1b_2b_3\ldots b_{52})_2 \times 2^{1022}$
$(11111111110)_2 = (2046)_{10}$	$\pm(1.b_1b_2b_3\ldots b_{52})_2 \times 2^{1023}$
$(11111111111)_2 = (2047)_{10}$	$\pm\infty$ if $b_1 = \cdots = b_{52} = 0$, NaN otherwise

available for the exponent and at least 63 bits for the fractional part of the significand. The Intel x87 microprocessors, to be discussed in Chapter 8, implement arithmetic with an extended format in hardware, using 80-bit floating point registers, with 1 bit for the sign, 15 bits for the exponent, and 64 bits for the significand. The leading bit of a normalized or subnormal number is not hidden as it is in the single and double formats but is explicitly stored. The purpose of the extended format is to allow computations to be carried out with higher precision in the registers before a result is stored to a double format location in memory.

The 2008 version of the standard introduced a third basic format, binary128, also known as quadruple precision. Its exponent field width is 15 bits and its fraction field is 112 bits, so the significand has 113 bits including the hidden bit. An implementation of the standard is required to support at least one of the three basic formats (binary32, binary64 or binary128). However, at present, very few machines support binary128 in hardware. The standard continues to recommend that an extended precision format be supported in addition to the widest basic format supported, but this advice is generally not followed by recent microprocessors, as discussed in Chapter 8.

For any format, let w denote the width of the *exponent* field in bits, namely 8 for the single format, 11 for the double format and 15 for the quadruple format. For all these formats, the largest exponent $E_{\max}$ is related to the exponent width w by a simple formula,

$$E_{\max} = 2^{w-1} - 1. \tag{4.5}$$

Thus, $E_{\max}$ is 127, 1023 or 16383, respectively, for the single, double and quadruple formats. Also, it is always the case that the exponent bias equals $E_{\max}$ and that $E_{\min} = -E_{\max} + 1 = -2^{w-1} + 2$. This then immediately implies that the smallest *biased* exponent, which is the number stored in the exponent field, is $E_{\min} + E_{\max} = 1$, and the largest biased exponent is $2E_{\max} = 2^w - 2$, with a bitstring of all ones except a zero in the last position.

The 2008 version of the standard also introduced specifications for several interchange formats: *binary16* (sometimes known as half precision), *binary256* (sometimes

Format	single (*binary32*)	double (*binary64*)	quadruple (*binary128*)
k	32	64	128
p	24	53	113
$\epsilon_{\text{mch}} = 2^{-(p-1)}$	$\approx 1.2 \times 10^{-7}$	$\approx 2.2 \times 10^{-16}$	$\approx 1.9 \times 10^{-34}$
$w = k - p$	8	11	15
$E_{\max} = 2^{w-1} - 1$	127	1023	16383
$E_{\min} = -2^{w-1} + 2$	-126	-1022	-16382
$bias = E_{\max}$	127	1023	16383
$N_{\max} = 2^{E_{\max}}(2 - 2^{-(p-1)})$	$\approx 3.4 \times 10^{38}$	$\approx 1.8 \times 10^{308}$	$\approx 1.2 \times 10^{4932}$
$N_{\min} = 2^{E_{\min}}$	$\approx 1.2 \times 10^{-38}$	$\approx 2.2 \times 10^{-308}$	$\approx 3.4 \times 10^{-4932}$
$S_{\min} = 2^{E_{\min}-(p-1)}$	$\approx 1.4 \times 10^{-45}$	$\approx 4.9 \times 10^{-324}$	$\approx 6.5 \times 10^{-4966}$

Table 4.3: Parameters for the three IEEE basic formats recommended by the 2008 floating point standard: total bit width k, precision p, machine epsilon ϵ_{mch}, exponent bit width w, maximum exponent, minimum exponent, exponent bias, maximum normalized number, minimum positive normalized number, minimum positive subnormal number.

known as octuple precision), and, more generally, binary-k where k is any multiple of 32 that is larger than 128. Equation (4.5) holds for these too. The exponent width w is only 5 for the half precision format, so $E_{\max} = 15$ and $E_{\min} = -14$. For the wider formats, a rather complicated formula was given for the exponent width w in terms of the total bit width k, but this formula holds only for $k \geq 64$. The purpose of the interchange formats is to standardize the widths for the exponent and significand fields in these formats, facilitating the exchange of data between machines, even if they do not support arithmetic operations using these formats. The first edition of this book, published in 2001, speculated that 256-bit floating point would become standard eventually, but this has not happened yet. In fact, the trend is now clearly in the opposite direction, with 128-bit formats being rarely used and 16-bit floating point formats becoming common, as we discuss in Chapter 15. However, other techniques for efficient high precision computation, including "double-double" and "quad-double", are described in Chapter 14.

Precision and Machine Epsilon of the IEEE Formats

Recall from the previous chapter that we use the notation p (precision) to denote the number of bits in the significand and ϵ_{mch} (machine epsilon) to mean the gap between 1 and the next larger floating point number. The precision of the IEEE single format is $p = 24$ (including the hidden bit); for the double format it is $p = 53$ (again, including the hidden bit). [6] The precision of the Intel extended format is $p = 64$, since it has no hidden bit. The first single format number larger than 1 is $1 + 2^{-23}$, and the first double format number larger than 1 is $1 + 2^{-52}$. With the Intel extended format, since there is no hidden bit, $1 + 2^{-64}$ cannot be stored exactly; the first extended format number larger than 1 is $1 + 2^{-63}$. The precision of *binary128* is $p = 113$, including the hidden bit. In all cases, the machine epsilon ϵ_{mch} is $2^{-(p-1)}$.

As earlier, let k denote the format's total bit width, that is 32, 64 or 128 respectively for the three IEEE basic formats, and let w be the number of bits in the exponent field. Then the number of stored bits in the fractional part of the significand is $k - w - 1$ (the

[6]

total number of bits minus the number needed for the exponent and the sign). Hence, we have, for the single, double and quadruple formats, but not the Intel extended format, that the precision is

$$p = (k - w - 1) + 1 = k - w,$$

where the -1 subtracts the sign bit and the $+1$ adds the hidden bit.

All the relevant parameters for the three IEEE basic formats (including *binary128* but not the Intel extended format) are summarized in Table 4.3.

Significant Digits

It is often stated that IEEE single precision floating point numbers have approximately 7 significant decimal digits and that double precision numbers have approximately 16 significant decimal digits. The rationale for this can be seen in Table 4.3: for single precision, the machine epsilon $\epsilon_{\mathrm{mch}} = 2^{-23}$ is about 10^{-7}, while for double precision, $\epsilon_{\mathrm{mch}} = 2^{-52}$ is about 10^{-16}. We deliberately use the word *approximately* here, because defining *significant digits* is problematic. For example, to how many decimal digits does

$$3.1415927$$

approximate

$$\pi = 3.141592653\ldots?$$

We might say 7, since the first 7 digits of both numbers are the same, or we might say 8, since if we round π to the nearest 8 digit decimal number, we get the same number 3.1415927. See [Hig02, Ch. 1] for a discussion of the difficulties involved in using definitions like these to define "significant digits." We will discuss this issue further in the next chapter.

Big and Little Endian

Modern computers address memory by bytes. A 32-bit word consists of 4 consecutive bytes with addresses, say, $B_1, \ldots, B_4$, where $B_4 = B_1 + 3$. Suppose we store a single format floating point number in this word. We know from Table 4.1 that a single format number has the bit format

$$\sigma a_1 a_2 a_3 \ldots a_8 b_1 b_2 b_3 \ldots b_{23},$$

where σ is the sign bit. This corresponds to 4 bytes, of which the "most significant" (see Exercise 4.3) is the byte

$$\sigma a_1 a_2 a_3 a_4 a_5 a_6 a_7.$$

Let us ask the question: is this most significant byte stored in byte B_1 or byte B_4? Surprisingly, it turns out that the answer depends on the machine. Addressing systems for which the answer is B_1 are called *Big Endian* (the first byte B_1 stores the "big end" of the floating point word). Addressing systems for which the answer is B_4 are called *Little Endian* (the first byte B_1 stores the "little end," i.e., the least significant byte, of the floating point word). Historically, IBM machines used Big Endian addressing, while Intel used Little Endian, but many modern machines can operate in either mode. The fact that different machines use different schemes means that care must be taken when passing data from one machine to another. The addressing schemes were given the names Big and Little Endian by Danny Cohen, in a whimsical reference to *Gulliver's Travels*, where the issue is which end of a boiled egg should be opened [HP95, Chapter 3.4].

Chapter 5

Rounding

We saw in the previous chapter that the finite IEEE floating point numbers can all be expressed in the form

$$\pm(b_0.b_1b_2\ldots b_{p-1})_2 \times 2^E,$$

where p is the precision of the floating point system with, for normalized numbers, $b_0 = 1$ and $E_{\min} \le E \le E_{\max}$ and, for subnormal numbers and zero, $b_0 = 0$ and $E = E_{\min}$. We denoted the largest normalized number by $N_{\max}$ and the smallest positive normalized number by $N_{\min}$. There are also two infinite floating point numbers, $\pm\infty$.

Let x be any real number. Let us define x_- to be the floating point number closest to x that is *less than or equal to* x, and define x_+ to be the floating point number closest to x that is *greater than or equal to* x. Note that if $x > N_{\max}$, then $x_+ = +\infty$ and $x_- = N_{\max}$. Also, if $0 < x < N_{\min}$, then x_+ is either a positive subnormal or $N_{\min}$ and x_- is either a positive subnormal or $+0$. Likewise, if $x < -N_{\max}$, then $x_- = -\infty$ and $x_+ = -N_{\min}$, and if $-N_{\min} < x < 0$, then x_- is either a negative subnormal or $-N_{\min}$, and x_+ is either a negative subnormal or -0. Note that, if x_- or x_+ is zero, its sign is chosen to be the same as the sign of x. Also note that, according to the definition, if x *is* a floating point number then $x_- = x_+ = x$.

We say that a real number x is in the *normalized range* of the floating point system if

$$N_{\min} \le |x| \le N_{\max}.$$

The numbers ± 0 and $\pm\infty$ and the subnormal numbers are not in the normalized range of the floating point system, although they are all valid floating point numbers. Let x be in the normalized range, with x not a floating point number, and write x in the normalized form

$$x = \pm(1.b_1b_2\ldots b_{p-1}b_pb_{p+1}\ldots)_2 \times 2^E, \tag{5.1}$$

with $E_{\min} \le E \le E_{\max}$. We first consider the case $x > 0$. Then the closest floating point number less than or equal to x is

$$x_- = (1.b_1b_2\ldots b_{p-1})_2 \times 2^E;$$

i.e., x_- is obtained by truncating the binary expansion of the significand, discarding b_p, b_{p+1}, etc. Since x is not a floating point number, at least one of the discarded bits in its expansion must be nonzero, so

$$x_+ = \big((1.b_1b_2\ldots b_{p-1})_2 + (0.00\ldots 01)_2\big) \times 2^E = \big(x_- + 2^{-(p-1)}\big) \times 2^E,$$

the next floating point number bigger than x_-, and therefore also the next one that is bigger than x. So, the gap between x_- and x_+ is

$$\text{ulp}(x) = x_+ - x_- = 2^{-(p-1)} \times 2^E = \epsilon_{\text{mch}} \times 2^E, \tag{5.2}$$

where ulp, which we introduced earlier in (3.7), means *unit in the last place,* and ϵ_{mch} is, as earlier, the machine epsilon. Finding the binary expansion of x_+ is a little more complicated, since one bit must be added to the last place of the fraction field of x_-; this may involve some bit "carries" and possibly, if all the bits in the field are 1, an increment in the exponent field. However, the largest possible exponent remains $E_{\max}$, since we assumed $x < N_{\max}$.

Now suppose $x < 0$, with x still in the normalized form (5.1) and x not a floating point number. Then the situation just described is reversed. Thus x_+ is obtained by dropping bits b_p, b_{p+1}, etc., since discarding bits of a negative number makes the number closer to zero, and therefore larger (further to the right on the real line). However, the gap between x_- and x_+ is still given by (5.2).

Exercise 5.1 *When $x > 0$ is not a floating point number, the floating point number x_+ is called the successor of x_-. Although we described finding x_+ as more complicated than finding x_-, this is actually easily done by treating the floating point encoding of x_- as a binary integer and adding 1 to it. By giving an example, explain why the following, taken from [MB+18, Sec. 7.2.1.1] is true: the possible carry propagation from the significand field to the exponent field will take care of the possible change of exponent. (See also Exercise 4.3.)*

Let us also make another definition. We say that a real number x is in the *subnormal range* of the floating point system if

$$0 < |x| < N_{\min}.$$

Let x be a positive number in the subnormal range, with x not a floating point number, and write x in the subnormal form

$$x = (0.b_1 b_2 \ldots b_{p-1} b_p b_{p+1} \ldots)_2 \times 2^{E_{\min}}. \tag{5.3}$$

It follows that the closest floating point number less than or equal to x is

$$x_- = (0.b_1 b_2 \ldots b_{p-1})_2 \times 2^{E_{\min}},$$

i.e., x_- is obtained by truncating the binary expansion of the significand, discarding b_p, b_{p+1}, etc. Note that x_- is zero if all of $b_1, \ldots, b_{p-1}$ are zero. Since x is not a floating point number, at least one of the discarded bits in its expansion is nonzero, so

$$x_+ = \big((0.b_1 b_2 \ldots b_{p-1})_2 + (0.00\ldots01)_2\big) \times 2^{E_{\min}} = \big(x_- + 2^{-(p-1)}\big) \times 2^{E_{\min}}.$$

Hence, the gap between x_- and x_+ is

$$\text{ulp}(x) = x_+ - x_- = 2^{-(p-1)} \times 2^{E_{\min}} = \epsilon_{\text{mch}} \times 2^{E_{\min}}. \tag{5.4}$$

As in the normalized case, determining the binary expansion of x_+ is a little more complicated, as it may involve bit "carries" and possibly rounding up to the normalized number $E_{\min}$. As before, all these considerations are reversed when considering negative numbers in the subnormal range, but (5.4) remains true.

Correctly Rounded Values

The IEEE standard defines the *correctly rounded value of* x, which we shall denote by $\text{round}(x)$, as follows.[1] If x is a floating point number, then $\text{round}(x) = x$. Otherwise, the correctly rounded value depends on which of the following four *rounding modes*[2] is in effect:

- *Round down*[3]
  $\text{round}(x) = x_-$.

- *Round up*[4]
  $\text{round}(x) = x_+$.

- *Round toward zero.*
  $\text{round}(x) = x_-$ if $x > 0$; $\text{round}(x) = x_+$ if $x < 0$.

- *Round to nearest.*
  $\text{round}(x)$ is either x_- or x_+, whichever is nearer to x (unless $|x| > N_{\max}$). In case of a tie, the one with its *least significant bit equal to zero* is chosen. See below for details.

If x is positive, then *round down* and *round toward zero* have the same effect. If x is negative, then *round up* and *round toward zero* have the same effect. In both cases, *round toward zero* simply requires truncating the binary expansion.

The rounding mode that is almost always used in practice is *round to nearest.* Consider x given by (5.1) again. If the first bit that cannot be stored, b_p, is 0, *round to nearest* rounds down to x_-; on the other hand, if $b_p = 1$ and at least one subsequent nonzero bit is also 1, *round to nearest* rounds up to x_+. If $b_p = 1$ and all subsequent bits are 0, there is a tie. The least significant bits, i.e., the $(p-1)$th bits after the binary point, of x_- and x_+ must be different, and the one for which this bit equals 0 is chosen to break the tie. For the motivation for this rule, see [Gol91, Theorem 5]. . When the word *round* is used without any mention of a rounding mode, it almost always means *round to nearest.* The IEEE standard requires that the default rounding mode be *round to nearest.*[5]

There is an exception to the *round to nearest* rule when $x > N_{\max}$. In this case, $\text{round}(x)$ is defined to be $+\infty$, not $N_{\max}$, unless x is so close to $N_{\max}$ that it would have rounded down to $N_{\max}$ even if the exponent range of the floating point system were increased.[6] From a strictly mathematical point of view, this is not consistent

[1] Often denoted $fl(x)$.

[2] The term *rounding mode* was used in the 1985 version of the standard. It was replaced by *rounding direction attribute* in 2008, but we prefer to use the earlier term.

[3] Called *round toward* $-\infty$ in the 1985 standard and *round toward negative* in the 2008 standard.

[4] Called *round toward* $+\infty$ in the 1985 standard and *round toward positive* in the 2008 standard.

[5] The tie-breaking rule just described for *round to nearest* has been in effect since the original 1985 standard, but starting with the 2008 revision, it is called *roundTiesToEven*, because this revision also introduced another tie-breaking option called *roundTiesToAway* which, in case of a tie, returns x_+ if $x > 0$ and x_- if $x < 0$. The primary rationale for *roundTiesToAway* concerns decimal, not binary, floating point arithmetic; see [MB+18, p. 67]. The 2019 version of the standard introduced yet another tie-breaking option called *roundTiesToZero*, with the opposite effect in case of a tie: return x_- if $x > 0$ and x_+ if $x < 0$. The rationale for *roundTiesToZero* is completely different and concerns new operations recommended in 2019; see Chapter 14. However, implementations of the standard are not required to include support for either of the newer tie-breaking rules; furthermore, *roundTiesToEven* must be the default behavior. Hence, in this book, we use *round to nearest* to mean *roundTiesToEven.*

[6] More precisely, $\text{round}(x)$ is defined to be $N_{\max}$ if $x < N_{\max} + \text{ulp}(N_{\max})/2$ and $+\infty$ otherwise. Note the use of the $<$ inequality here, not $\leq$, because in the case of a tie, the round to even rule would result in rounding up, not down.

with the usual definition, since x cannot be said to be closer to ∞ than to $N_{\max}$. From a practical point of view, however, the choice ∞ is important, since *round to nearest* is the default rounding mode and rounding to x to $N_{\max}$ may give very misleading results if x is much larger than $N_{\max}$. Similar considerations apply when $x < -N_{\max}$.

Exercise 5.2 *What are the IEEE single format binary representations for the rounded value of* $1/10$ *(see* (2.3)*), using each of the four rounding modes? What are they for* $1 + 2^{-25}$ *and* 2^{130}*?*

Exercise 5.3 *Using the IEEE single format, construct an example where x_- and x_+ are the same distance from x, and use the tie-breaking rule to define* round(x), *assuming the round-to-nearest mode is in effect.*

Exercise 5.4 *Suppose that* $0 < x < N_{\min}$, *but that x is not a subnormal floating point number. We can write*

$$x = (0.b_1 b_2 \ldots b_{p-1} b_p b_{p+1} \ldots)_2 \times 2^{E_{\min}},$$

where at least one of $b_p, b_{p+1}, \ldots$ is not zero. What is x_-? Give some examples, assuming the single format is in use ($p = 24$, $E_{\min} = -126$).

Absolute Rounding Error and Ulp

We now define rounding error.[7] Let x be a real number and define

$$\text{abserr}(x) = |\text{round}(x) - x|, \tag{5.5}$$

the *absolute rounding error* associated with x. Since round(x) is always either x_+ or x_-, and $x_- \le x \le x_+$, we have

$$|\text{round}(x) - x| < x_+ - x_-, \tag{5.6}$$

the gap between x_- and x_+; it cannot equal the gap since if x is a floating point number, abserr$(x) = 0$. So, if x is in the normalized range, with the form (5.1), we have from (5.2) that

$$\text{abserr}(x) = |\text{round}(x) - x| < x_+ - x_- = \text{ulp}(x) = 2^{-(p-1)} \times 2^E = \epsilon_{\text{mch}} \times 2^E, \tag{5.7}$$

regardless of the rounding mode in effect. Informally, we say that the absolute rounding error is less than one *ulp*, meaning ulp(x). When *round to nearest* is in effect, we can say something stronger: the absolute rounding error is *less than or equal to half the gap between x_- and x_+*, i.e.,

$$\text{abserr}(x) = |\text{round}(x) - x| \le \frac{1}{2}(x_+ - x_-) = \frac{1}{2}\text{ulp}(x) = 2^{-p} \times 2^E = \frac{1}{2}\epsilon_{\text{mch}} \times 2^E, \tag{5.8}$$

and informally, we say that the absolute rounding error is at most *half an ulp*.

Exercise 5.5 *What is* abserr$(1/10)$ *using the IEEE single format, for each of the four rounding modes? (See Exercise* 5.2*.)*

Exercise 5.6 *Suppose that* $x > N_{\max}$. *What is* abserr(x)*, for each of the four rounding modes? Look carefully at the definition of* round(x).

Exercise 5.7 *What is* abserr(x) *for x given in Exercise* 5.4*, using the rounding mode round down?*

Exercise 5.8 *Suppose x is in the subnormal range, with the form* (5.3)*. Do the bounds* (5.7) *and* (5.8) *hold with E replaced by $E_{\min}$? Why or why not?*

[7]Often called roundoff or round-off error.

Relative Rounding Error, Machine Epsilon, Significant Digits

The *relative rounding error* associated with a nonzero number x is defined by

$$\mathrm{relerr}(x) = |\delta|, \tag{5.9}$$

where

$$\delta = \frac{\mathrm{round}(x)}{x} - 1 = \frac{\mathrm{round}(x) - x}{x}. \tag{5.10}$$

Assuming that x is in the normalized range and is not a floating point number, we have, using (5.1),

$$|x| > 2^E. \tag{5.11}$$

Therefore, for all rounding modes, the relative rounding error satisfies the bound

$$\mathrm{relerr}(x) = |\delta| = \frac{|\mathrm{round}(x) - x|}{|x|} < \frac{2^{-(p-1)} \times 2^E}{2^E} = 2^{-(p-1)} = \epsilon_{\mathrm{mch}}, \tag{5.12}$$

using (5.7) and (5.11). In the case of *round to nearest*, we have

$$\mathrm{relerr}(x) = |\delta| = \frac{|\mathrm{round}(x) - x|}{|x|} < \frac{2^{-p} \times 2^E}{2^E} = 2^{-p} = \frac{1}{2}\epsilon_{\mathrm{mch}}, \tag{5.13}$$

using (5.8) and (5.11). The same inequalities hold when x *is* a floating point number in the normalized range, since then $\mathrm{relerr}(x) = \mathrm{abserr}(x) = 0$.

Exercise 5.9 *Suppose x is in the subnormal range, with the form* (5.3)*. Do the bounds* (5.12) *and* (5.13) *hold with $E = E_{\min}$? Why or why not? (See Exercise* 5.8.*)*

Let us take logarithms (base 10) on both sides of the relative error inequality (5.12) and multiply through by -1, giving

$$-\log_{10}(\mathrm{relerr}(x)) > -\log_{10}(\epsilon_{\mathrm{mch}}). \tag{5.14}$$

Although we pointed out in Chapter 4 that defining "significant digits" is problematic, this inequality provides a convenient and well-defined way to do this. We can interpret the left-hand side of (5.14) as *the approximate number of decimal digits* to which $\mathrm{round}(x)$ approximates x, and the right-hand side as a lower bound on this number. Consulting Table **??**, we see that this lower bound is approximately 7 in the case of IEEE single and approximately 16 for IEEE double. Note that since

$$-\log_{10}\left(\frac{1}{2}\epsilon_{\mathrm{mch}}\right) = -\log_{10}(\epsilon_{\mathrm{mch}}) - \log_{10}\left(\frac{1}{2}\right) \approx -\log_{10}(\epsilon_{\mathrm{mch}}) - 0.3,$$

using (5.13) instead of (5.12) increases the lower bound only by about one-third of a decimal digit, which has little significance.

It follows from (5.10) that

$$\mathrm{round}(x) = x(1 + \delta).$$

Combining this with (5.12) and (5.13), we have completed the proof of the following result, which is so important that we state it as the only theorem in this book.

Theorem 5.1 *Let x be any real number in the normalized range of a binary floating point system with precision p. Then*

$$\text{round}(x) = x(1+\delta)$$

for some δ satisfying

$$|\delta| < \epsilon_{\text{mch}},$$

where ϵ_{mch}, machine epsilon, is the gap between 1 *and the next larger floating point number, i.e.,*

$$\epsilon_{\text{mch}} = 2^{-(p-1)}.$$

Furthermore, if the rounding mode in effect is round to nearest,

$$|\delta| < \frac{1}{2}\epsilon_{\text{mch}} = 2^{-p}.$$

Theorem 5.1 shows that, no matter how a number x in the normalized range is stored or displayed, either in a binary format or in a converted decimal format, we may think of its value as *exact within a factor* of $1+\epsilon_{\text{mch}}$, regardless of the rounding mode. This means that IEEE single format numbers are accurate to within a factor of about $1 + 10^{-7}$, which we can interpret to mean that they have *approximately 7 significant decimal digits*. Likewise IEEE double format numbers are accurate to within a factor of about $1 + 10^{-16}$, which we can interpret to mean that they have *approximately 16 significant decimal digits*.

Let us again consider the example from the previous chapter. The number

$$\pi = 3.141592653\ldots$$

is represented in single precision, using *round to nearest* and converting to decimal, as $\text{round}(\pi) \approx 3.1415927$. We have

$$\text{relerr}(\pi) = \frac{|\pi - \text{round}(\pi)|}{|\pi|} \approx 2.783 \times 10^{-8}$$

so

$$-\log 10(\text{relerr}(\pi)) \approx 7.555.$$

Hence we say that $\text{round}(\pi)$ approximates π to about 7 or 8 decimal digits. As another example, the number $x = 1.99999999$ is represented in single precision, using *round to nearest*, by $\text{round}(x) = 2$ (exactly). We have

$$\text{relerr}(x) = \frac{|x - \text{round}(x)|}{|x|} \approx 5 \times 10^{-9}$$

so

$$-\log 10(\text{relerr}(x)) \approx 8.3.$$

Hence we say that $\text{round}(x)$ approximates x to about 8 decimal digits — slightly more than in the previous example.

Exercise 5.10 *Does the result established by Theorem* 5.1 *hold if x is in the subnormal range of the floating point system? Why or why not?*

Chapter 6

Correctly Rounded Floating Point Operations

A key feature of the IEEE standard is that it requires correctly rounded operations, specifically:

- correctly rounded arithmetic operations (add, subtract, multiply, divide)[1]
- correctly rounded remainder and square root operations
- correctly rounded format conversions.

Correctly rounded means rounded to fit the *destination* of the result, using the rounding mode in effect. For example, if the operation is the addition of two floating point numbers that are stored in registers, the destination for the result is normally one of these registers (overwriting one of the operands). On the other hand, the operation might be a store instruction, in which case the destination is a location in memory and a format conversion may be required. Regardless of whether the destination is a register or a memory location, its format could be any of the IEEE formats, depending on the machine being used and the program being executed.

Correctly Rounded Arithmetic

We begin by discussing the arithmetic operations. Very often, the result of an arithmetic operation on two floating point numbers is *not* a floating point number in the destination format. This is most obviously the case for multiplication and division; for example, 1 and 10 are both floating point numbers but we have already seen that 1/10 is not, regardless of the destination format. It is also true of addition and subtraction: for example, 1 and 2^{-24} are IEEE single format numbers, but $1 + 2^{-24}$ is not.

Let x and y be floating point numbers, let $+$, $-$, $\times$, $/$ denote the four standard arithmetic operations, and let $\oplus$, $\ominus$, $\otimes$, $\oslash$ denote the corresponding operations as they are actually implemented on the computer. Thus, $x + y$ may not be a floating point number, but $x \oplus y$ is the floating point number that is the computed approximation of $x + y$. Before the development of the IEEE standard, the results of a floating point operation might be different on two different computers. Occasionally, the results could be quite bizarre. Consider the following questions, where in each case we assume

[1]One more arithmetic operation, *fused multiply-add*, was introduced in the 2008 version of the standard; we discuss this below.

that the destination for the result has the same format as the floating point numbers x and y.

Question 6.1 *If x is a floating point number, is the floating point product $1 \otimes x$ equal to x?*

Question 6.2 *If x is a nonzero (and finite) floating point number, is the floating point quotient $x \oslash x$ equal to 1?*

Question 6.3 *If x is a floating point number, is the floating point product $0.5 \otimes x$ the same as the floating point quotient $x \oslash 2$?*

Question 6.4 *If x and y are floating point numbers, and the floating point difference $x \ominus y$ is zero, does x equal y?*

Normally, the answer to all these questions would be *yes*, but for each of Questions 6.1 through 6.3, there was a widely used computer in the 1960s or 1970s for which the answer was *no* for some input x [Sev98], [Kah00], [PH97, Section 4.12]. These anomalies *cannot* occur with IEEE arithmetic. As for Question 6.4, virtually all systems developed before the standard were such that the answer could be *no* for small enough values of x and y; on some systems, the answer could be *no* even if x and y were both near 1 (see the discussion following equation (6.4)). With IEEE arithmetic, the answer to Question 6.4 is always *yes*; see the next chapter.

When the result of a floating point operation is not a floating point number in the destination format, the IEEE standard requires that the computed result be the rounded value of the exact result, i.e., rounded to fit the destination format, using the rounding mode in effect. It is worth stating this requirement carefully. The rule for correctly rounded arithmetic is as follows: if x and y are floating point numbers, then

$$x \oplus y = \text{round}(x+y),$$

$$x \ominus y = \text{round}(x-y),$$

$$x \otimes y = \text{round}(x \times y),$$

and

$$x \oslash y = \text{round}(x/y),$$

where round is the operation of rounding to the given destination format, using the rounding mode in effect.

Exercise 6.1 *Show that the IEEE rule of correctly rounded arithmetic immediately guarantees that the answers to Questions 6.1 to 6.3 must be yes. Show further that for two out of three of these questions, no rounding is necessary because the exact result is always a floating point number, but that for one of the three, rounding may be necessary in some cases; which question is this and in what cases is rounding needed?*

Using Theorem 5.1, it follows that, as long as $x+y$ is in the normalized range,

$$x \oplus y = (x+y)(1+\delta),$$

where

$$|\delta| < \epsilon_{\text{mch}},$$

machine epsilon for the destination format. This applies to all rounding modes; for *round to nearest*, we have the stronger result

$$|\delta| < \frac{1}{2}\epsilon_{\text{mch}}.$$

For example, if the destination format is IEEE single and the rounding mode is *round to nearest*, floating point addition is accurate to within a factor of $1 + 2^{-24}$, i.e., to approximately seven decimal digits. The same holds for the other operations $\ominus$, $\otimes$, and $\oslash$.

Exercise 6.2 *Suppose that the destination format is IEEE single and the rounding mode is round to nearest. What are* $64 \oplus 2^{20}$, $64 \oplus 2^{-20}$, $32 \oplus 2^{-20}$, $16 \oplus 2^{-20}$, $8 \oplus 2^{-20}$*? Give your answers in binary, not decimal. What are the results if the rounding mode is changed to round up?*

Exercise 6.3 *Recalling how many decimal digits correspond to the* 24*-bit precision of an IEEE single format number, which of the following expressions do you think have the value exactly* 1 *if the destination format is IEEE single and the rounding mode is round to nearest:* $1 \oplus \text{round}(10^{-5})$, $1 \oplus \text{round}(10^{-10})$, $1 \oplus \text{round}(10^{-15})$*?*

Exercise 6.4 *What is the largest floating point number* x *for which* $1 \oplus x$ *is exactly* 1*, assuming the destination format is IEEE single and the rounding mode is round to nearest? What if the destination format is IEEE double?*

It is important to note that the result of a sequence of *two or more* arithmetic operations may *not* be the correctly rounded value of the exact result. For example, consider the computation of $(x + y) - z$, where $x = 1$, $y = 2^{-25}$, and $z = 1$, assuming the destination format for both operations is IEEE single, and with *round to nearest* in effect. The numbers x, y, and z are all IEEE single format floating point numbers, since $x = z = 1.0 \times 2^0$ and $y = 1.0 \times 2^{-25}$. The exact sum of the first two numbers is

$$x + y = (1.0000000000000000000000001)_2.$$

This does not fit the single format, so it is rounded, giving

$$x \oplus y = 1.$$

The final result is therefore

$$(x \oplus y) \ominus z = 1 \ominus 1 = 0.$$

However, the exact result is

$$(x + y) - z = 2^{-25},$$

which *does* fit the single format exactly. Notice that the exact result *would* be obtained if the destination format for the intermediate result $x + y$ is the IEEE double or extended format (see Chapter 8).

Exercise 6.5 *In this example, what is* $x \oplus (y \ominus z)$*, and* $(x \ominus z) \oplus y$*, assuming the destination format for all operations is IEEE single?*

Exercise 6.6 *Using the same example, what is* $(x \oplus y) \ominus z$ *if the rounding mode is round up?*

Exercise 6.7 *Let $x = 1$, $y = 2^{-15}$, and $z = 2^{15}$, stored in the single format. What is $(x \oplus y) \oplus z$, when the destination format for both operations is the single format, using round to nearest? What if the rounding mode is round up?*

Exercise 6.8 *In exact arithmetic, the addition operation is commutative, i.e.,*

$$x + y = y + x$$

for any two numbers x, y, and also associative, i.e.,

$$x + (y + z) = (x + y) + z$$

for any x, y, and z. Is the floating point addition operation $\oplus$ commutative? Is it associative?

The availability of the rounding modes *round down* and *round up* allows a programmer to make any individual computation twice, once with each mode. The two results define an interval that must contain the exact result. *Interval arithmetic* is the name used when sequences of computations are done in this way. See Exercises 10.14, 10.15, 13.5, and 13.11.

Addition and Subtraction

Now we ask the question: How is correctly rounded arithmetic implemented? This is surprisingly complicated. Let us consider the addition of two IEEE single format floating point numbers $x = S \times 2^E$ and $y = T \times 2^F$, assuming the destination format for $x+y$ is also IEEE single. If the two exponents E and F are the same, it is necessary only to add the significands S and T. The final result is $(S + T) \times 2^E$, which then needs further normalization if $S + T$ is greater than or equal to 2, or less than 1. For example, the result of adding $3 = (1.100)_2 \times 2^1$ to $2 = (1.000)_2 \times 2^1$ is

$$\begin{array}{rll} & (\ 1.10000000000000000000000 &)_2 \times 2^1 \\ + & (\ 1.00000000000000000000000 &)_2 \times 2^1 \\ = & (10.10000000000000000000000 &)_2 \times 2^1 \\ \text{Normalize}: & (\ 1.01000000000000000000000 &)_2 \times 2^2. \end{array}$$

However, if the two exponents E and F are different, say with $E > F$, the first step in adding the two numbers is to *align the significands*, shifting T right $E - F$ positions so that the second number is no longer normalized and both numbers have the same exponent E. The significands are then added as before. For example, adding $3 = (1.100)_2 \times 2^1$ to $3/4 = (1.100)_2 \times 2^{-1}$ gives

$$\begin{array}{rll} & (\ 1.10000000000000000000000 &)_2 \times 2^1 \\ + & (\ 0.01100000000000000000000 &)_2 \times 2^1 \\ = & (\ 1.11100000000000000000000 &)_2 \times 2^1. \end{array}$$

In this case, the result does not need further normalization.

Guard Bits

Now consider adding 3 to 3×2^{-23}. We get

$$\begin{array}{rll} & (\ 1.10000000000000000000000 &)_2 \times 2^1 \\ + & (\ 0.00000000000000000000001|1 &)_2 \times 2^1 \\ = & (\ 1.10000000000000000000001|1 &)_2 \times 2^1 \\ \text{Round Down}: & (\ 1.10000000000000000000001 &)_2 \times 2^1 \\ \text{or Round Up}: & (\ 1.10000000000000000000010 &)_2 \times 2^1. \end{array} \tag{6.1}$$

This time, the sum shown in the middle line is not an IEEE single format floating point number, since its significand has 24 bits after the binary point: the 24th is shown beyond the vertical bar. Therefore, the sum must be *correctly rounded*. In the case of rounding to nearest, there is a tie, so the result with its final bit equal to zero is used (round up in this case).

Rounding should not take place before the result is normalized. Consider the example of subtracting the floating point number $1+2^{-22}+2^{-23}$ from 3, or equivalently adding 3 and $-(1+2^{-22}+2^{-23})$. We get

$$\begin{array}{rll} & (\ 1.10000000000000000000000 &)_2 \times 2^1 \\ - & (\ 0.10000000000000000000001|1 &)_2 \times 2^1 \\ = & (\ 0.11111111111111111111110|1 &)_2 \times 2^1 \\ \text{Normalize}: & (\ 1.11111111111111111111101 &)_2 \times 2^0. \end{array} \tag{6.2}$$

Thus, rounding is not needed in this example.

In both examples (6.1) and (6.2), it was necessary to *carry out the operation using an extra bit*, called a *guard bit*, shown after the vertical line following the b_{23} position. Without the guard bit, the correctly rounded result would not have been obtained.

Exercise 6.9 *Work out the details for the examples* $1+2^{-24}$ *and* $1-2^{-24}$. *Make up some more examples where a guard bit is required.*

The following is a particularly interesting example. Consider computing $x-y$ with $x=(1.0)_2\times 2^0$ and $y=(1.1111\ldots1)_2\times 2^{-1}$, where the fraction field for y contains 23 ones after the binary point. (Note that y is only slightly smaller than x; in fact, it is the next floating point number smaller than x.) Aligning the significands, we obtain

$$\begin{array}{rll} & (\ 1.00000000000000000000000| &)_2 \times 2^0 \\ - & (\ 0.11111111111111111111111|1 &)_2 \times 2^0 \\ = & (\ 0.00000000000000000000000|1 &)_2 \times 2^0 \\ \text{Normalize}: & (\ 1.00000000000000000000000|0 &)_2 \times 2^{-24}. \end{array} \tag{6.3}$$

This is an example of *cancellation*, since almost all the bits in the two numbers cancel each other. The result is $(1.0)_2 \times 2^{-24}$, which is a floating point number. As in the previous example, we need a guard bit to get the correct answer; indeed, without it, we would get a completely wrong answer.

The following example shows that more than one guard bit may be necessary. Consider computing $x-y$ where $x=1.0$ and $y=(1.000\ldots01)_2\times 2^{-25}$, where y has 22 zero bits between the binary point and the final 1 bit. Using 25 guard bits and *round to nearest*, we get

$$\begin{array}{rll} & (\ 1.00000000000000000000000| &)_2 \times 2^0 \\ - & (\ 0.00000000000000000000000|0100000000000000000000001 &)_2 \times 2^0 \\ = & (\ 0.11111111111111111111111|1011111111111111111111111 &)_2 \times 2^0 \\ \text{Normalize}: & (\ 1.11111111111111111111111|0111111111111111111111110 &)_2 \times 2^{-1} \\ \text{Round}: & (\ 1.11111111111111111111111 &)_2 \times 2^{-1}. \end{array}$$

This is the correctly rounded value of the exact sum of the numbers, but if we were to use only two guard bits, we would get the result:

$$\begin{array}{rll}
 & (\ \ 1.00000000000000000000000| &)_2 \times 2^0 \\
- & (\ \ 0.00000000000000000000000|01 &)_2 \times 2^0 \\
= & (\ \ 0.11111111111111111111111|11 &)_2 \times 2^0 \\
\text{Normalize :} & (\ \ 1.11111111111111111111111|1 &)_2 \times 2^{-1} \\
\text{Round to Nearest :} & (10.00000000000000000000000 &)_2 \times 2^{-1} \\
\text{Renormalize :} & (\ \ 1.00000000000000000000000 &)_2 \times 2^0.
\end{array}$$

In this scenario, normalizing and rounding results in rounding up (using the tie-breaking rule) instead of down, giving the final result 1.0, which is *not* the correctly rounded value of the exact sum. We get the same wrong answer even if we have 3, 4, or as many as 24 guard bits in this case! Machines that implement correctly rounded arithmetic take such possibilities into account. However, by being a little clever, the need for 25 guard bits can be avoided. Let us repeat the same example, with two guard bits, but with one additional bit "turned on" to indicate that at least one nonzero extra bit was discarded when the bits of the second number, y, were shifted to the right past the second guard bit position. The bit is called *sticky* because once it is turned on, it stays on, regardless of how many bits are discarded. Now, before doing the subtraction, we put the sticky bit in a third guard bit position. For this example, we then get

$$\begin{array}{rll}
 & (\ \ 1.00000000000000000000000| &)_2 \times 2^0 \\
- & (\ \ 0.00000000000000000000000|011 &)_2 \times 2^0 \\
= & (\ \ 0.11111111111111111111111|101 &)_2 \times 2^0 \\
\text{Normalize :} & (\ \ 1.11111111111111111111111|01 &)_2 \times 2^{-1} \\
\text{Round to Nearest :} & (\ \ 1.11111111111111111111111 &)_2 \times 2^{-1},
\end{array}$$

which is the correct answer. In general, it is necessary to use only three extra bits to implement correctly rounded floating point addition and subtraction: two guard bits (often called the guard and round bits) and one sticky bit [Gol95]. For a detailed discussion, see [MB+18, Sec. 7.3].

Exercise 6.10 *Consider the operation $x+y$, where $x = 1.0$ and $y = (1.000\ldots01)_2 \times 2^{-24}$, and y has 22 zero bits between the binary point and the final 1 bit. What is the correctly rounded result, assuming round to nearest is in use? What is computed if only one guard bit is used? What if two guard bits are used? What if two guard bits and a sticky bit are used?*

When the IBM 360 was released in 1965, it did not have any guard bits, and it was only after the strenuous objections of computer scientists that later versions of the machine incorporated one hexadecimal guard digit—still not enough to guarantee correctly rounded arithmetic. Decades later, Cray supercomputers still did not have a guard bit. Let $x = 1$ and let y be the next floating point number smaller than 1, and consider the operation $x - y$, as in example (6.3) above. On one Cray machine, the computed result $x \ominus y$ was wrong by a factor of 2, since a 1 was shifted past the end of the second operand's significand and discarded. This resulted in

$$x \ominus y = 2(x-y) \quad \text{instead of } x \ominus y = (x-y)(1+\delta), \qquad \text{where } |\delta| \leq \epsilon_{\text{mch}}. \tag{6.4}$$

On another Cray machine, the second operand y was rounded before the operation took place. This converted the second operand to the value 1.0 and gave the result $x \ominus y = 0$, so that in this case the answer to Question 6.4 was *no.*

Multiplication and Division

Floating point multiplication and division, unlike addition and subtraction, do not require significands to be aligned. If $x = S \times 2^E$ and $y = T \times 2^F$, then

$$x \times y = (S \times T) \times 2^{E+F},$$

so there are three steps to floating point multiplication: multiply the significands, add the exponents, and normalize and correctly round the result. Suppose x and y are floating point numbers with precision p, including the hidden bit, and that the destination for the result also has precision p. We can think of the significands S and T as p-bit integers scaled by $2^{-(p-1)}$, so we can think of their product as an integer product, with $S \times T$ being a $2p$-bit integer scaled by $2^{-2(p-1)}$. The leading bits of the product will never be discarded as they might be with integer multiplication that overflows (see Chapter 3); instead, the product will be normalized and correctly rounded to p bits, discarding the least significant bits. Of course, the exponent $E + F$ also needs to be taken into account in the final result. Likewise, division requires taking the quotient of the significands and the difference of the exponents. Overall, although floating point multiplication does not involve much more than integer multiplication, it is still a much more complicated operation than floating point addition, and floating point division is even more complicated — primarily because the quotient of two integers is generally not an integer and is not truncated to an integer as would be done with integer division, but needs to be rounded to p bits. For more details on implementing floating point operations, see [MB+18, Ch. 7–8]. Division by zero will be discussed in the next chapter.

In principle it is possible, by using enough space on the chip, to implement the operations so that they are all equally fast. In practice, chip designers build the hardware so that multiplication is approximately as fast as addition, because in many floating point applications addition and multiplication appear together in an inner loop.[2] However, the division operation generally takes significantly longer to execute than addition or multiplication. For an interesting figure showing typical relative speeds of the different arithmetic operations, see [BJ+23, Fig. 1.1]. As explained there, a key development over the past few decades is that although execution times for arithmetic operations has decreased drastically, the time required for passing data to and from various levels of the computer memory hierarchy has decreased much more slowly, by a relative factor of more than 100 since the floating point standard was first published in 1985. This fact has had major impact on computer architecture design and algorithm design.

Exercise 6.11 *Explain why the product of two single format floating point numbers can always be stored exactly as a double format number, with no rounding error. You need to take both the precision and the exponent range of the two formats into consideration. Is this also true of the sum of two single format floating point numbers, the difference, or the quotient?*

Exercise 6.12 *Assume that $x = S \times 2^E$ and $y = T \times 2^F$ are normalized floating point numbers, i.e., $1 \le |S| < 2$, $1 \le |T| < 2$, with (the binary representations of) S and T each having p bits (including the hidden bit). Let U be the exact product of the two significands, i.e., $U = S \times T$.*

1. *What are the possibilities for the number of nonzero bits to the left of the binary point of (the binary representation for) U? What does this tell you about how*

[2]Although this motivation is less important than it was before the fused multiply-add operation became available, it remains significant.

many bits it may be necessary to shift the binary point of U left or right to normalize the result?

2. *What are the possibilities for the number of nonzero bits to the right of the binary point of U? In what cases can U be represented exactly using p bits (including the hidden bit), and in what cases must the result be rounded to fit a p-bit destination?*

The Intel Pentium chip received a lot of bad publicity in 1994 when the fact that it had a floating point hardware bug was exposed. An example of the bug's effects is that, on the original Pentium, the floating point division operation

$$\frac{4195835}{3145727}$$

gave a result with only about 4 correct decimal digits. The error occurred in only a few special cases and could easily have remained undiscovered much longer than it did; it was found by a mathematician doing experiments in number theory. Nonetheless, it created a sensation, mainly because it turned out that Intel knew about the bug but had not released the information. The public outcry against incorrect floating point arithmetic depressed Intel's stock value significantly until the company finally agreed to replace everyone's defective processors, not just those belonging to institutions that Intel thought really needed correct arithmetic! It is hard to imagine a more effective way to persuade the public that floating point accuracy is important than to inform it that only specialists can have it. The event was particularly ironic since no company had done more than Intel to make accurate floating point available to the masses. For details on how the bug arose, see [Ede97]. However, quoting from [MB+18, Sec. 1.1], *[the Pentium bug] has had very positive long-term effects: most arithmetic algorithms used by the manufacturers are now published so that everyone can check them, and everybody understands that a particular effort must be made to build formal proofs of the arithmetic algorithms and their implementation.*

Fused Multiply-Add

Starting in 2008, the floating point standard also requires support for a *fused multiply-add operation* (FMA). Given three floating point numbers a, b and c, the FMA operation computes the correctly rounded value

$$\text{round}(a \times b + c), \tag{6.5}$$

which is usually a more accurate approximation to the exact result than computing

$$\text{round}(\text{round}(a \times b) + c). \tag{6.6}$$

The FMA operation is very useful because many floating point programs include the operation $a \times b + c$ in their inner loops. It also simplifies algorithms for correctly rounded division and square root operations [MB+18, Ch. 4], and can be used to increase accuracy in computations with complex numbers [MB+18, Ch. 11].

The FMA works by adding c to the $2p$-bit product $a \times b$ before it is rounded. Nonetheless, the execution time for an FMA is typically not much more than for a multiplication or addition alone; see [MB+18, Sec. 3.4.2 and 7.5] for details.

Exercise 6.13 *Find single format floating point numbers a, b, and c for which* (6.5) *and* (6.6) *are different, assuming the destination format for each operation is IEEE single.*

Exercise 6.14 *Give some examples of a, b, c for which the FMA returns* NaN.

For more on how computers implement arithmetic operations, see [HP95, PH97, Gol95] and especially [MB+18, Part III]. For a wealth of information on rounding properties of floating point arithmetic at an advanced level, see Goldberg [Gol91] and Kahan [Kah97, Kah96b, Kah00]..

Remainder, Square Root, and Format Conversions

In addition to requiring that the add, subtract, multiply, divide and FMA operations be correctly rounded, the IEEE standard also requires that correctly rounded remainder and square root operations be provided. The remainder operation, x REM y, is valid for finite x and nonzero y and produces $r = x - y \times n$, where n is the integer nearest the exact value x/y. The square root operation is valid for all nonnegative arguments. The standard algorithm for computing square roots is often attributed to Newton, but it was used by Heron of Alexandria 2000 years ago, and seems to have been known to the Babylonians nearly 2000 years before Heron [MB+18, Sec 4.8.1].

Exercise 6.15 *The formula for the length of the hypotenuse of a right-angled triangle is*

$$z = \sqrt{x^2 + y^2},$$

where x and y are the lengths of the legs of the triangle. Suppose this formula is computed using IEEE floating point arithmetic when it happens that all of x, y, and z are integers with $x^2 + y^2 < N_{\max}$ (e.g., 3, 4, and 5 or 5, 12, and 13). Will the floating point result for z necessarily be an integer?

The standard also requires support for number format conversions. These fall into several categories:

- *Conversion between floating point formats.* Conversion from a narrower to a wider precision (e.g., from single to double) must be exact. Conversion from a wider precision to a narrower one requires correct rounding, using the rounding mode in effect.

- *Conversion between floating point and integer formats.* Conversion from a floating point format to an integer format requires rounding to the nearest integer. If the floating point number is already an integer, the conversion should be exact unless this number does not fit the integer format. Conversion from integer format to floating point format may require rounding (see Exercise 3.11).

- *Rounding a floating point number to an integral value.* This is also a required feature, so that rounding to an integral value does not require use of an integer format.

- *Conversion from a binary floating point format to an output decimal string or from an input decimal string to a binary floating point format.* The 2008 version of the standard recommends that, regardless of the length of the input or output decimal string, the result should be correctly rounded using the rounding mode in effect. Thus, if a floating point number is converted to an output format with a specified number of decimal digits, the result should be rounded correctly to that number of digits, and if an input decimal string of any length is converted to a binary floating point format, the result should be correctly rounded to the binary floating point format. Although implementations of IEEE arithmetic are not required to have this strict property, the standard does specify some requirements, too complicated to state here: see [IEE08, Sec. 5.12] and [MB+18, Sec. 3.1.5]. The original version of the standard had less demanding recommendations and requirements, because efficient algorithms for correctly rounded binary to decimal and decimal to binary conversion were not known in 1985. However, efficient conversion algorithms that round correctly in all cases are now known [Gay90]. See [MB+18, Sec. 4.9.2] for more details.

Chapter 7

Exceptions

One of the most difficult things about programming is the need to anticipate exceptional situations. Ideally, a program should handle exceptional data in a manner as consistent as possible with the handling of unexceptional data. For example, a program that reads integers from an input file and echoes them to an output file until the end of the input file is reached should not fail just because the input file is empty. On the other hand, if it is further required to compute the average value of the input data, no reasonable solution is available if the input file is empty. So it is with floating point arithmetic. When a reasonable response to exceptional data is possible, it should be used.

Infinity from Division by Zero

The simplest example of an exception is *division by zero.* Before the IEEE standard was devised, there were two common responses to dividing a positive number by zero. One often used in the 1950s was to generate the largest floating point number as the result. The rationale was that the user would notice the large number in the output and draw the conclusion that something had gone wrong. However, this often led to confusion: for example, the expression $1/0 - 1/0$ would give the result 0, so the user might *not* notice that any error had taken place. Consequently, it was emphasized in the 1960s that division by zero should lead to the interruption or termination of the program, perhaps giving the user an informative message such as "fatal error—division by zero." To avoid this, the burden was on the programmer to make sure that division by zero would never occur.[1]

Suppose, for example, it is desired to compute the total resistance of an electrical circuit with two resistors connected in parallel, with resistances, respectively, R_1 and R_2 ohms, as shown in Figure 7.1. The formula for the total resistance of the circuit is

$$T = \frac{1}{\frac{1}{R_1} + \frac{1}{R_2}}. \tag{7.1}$$

This formula makes intuitive sense: if both resistances R_1 and R_2 are the same value R, then the resistance of the whole circuit is $T = R/2$, since the current divides equally, with equal amounts flowing through each resistor. On the other hand, if R_1 is very much smaller than R_2, the resistance of the whole circuit is somewhat less than R_1, since most of the current flows through the first resistor and avoids the second

[1]The author's high school physics class did not have an electronic computer in 1970, but it had an electromechanical caculator. Students delighted in dividing by zero, which resulted in the machine calculating away forever until it was interrupted.

Figure not available at present, see figure in first edition

Figure 7.1: The Parallel Resistance Circuit

one. What if R_1 is zero? The answer is intuitively clear: since the first resistor offers no resistance to the current, *all* the current flows through that resistor and avoids the second one; therefore, the total resistance in the circuit is zero. The formula for T also makes sense mathematically if we introduce the convention that $1/0 = \infty$ and $1/\infty = 0$. We get

$$T = \frac{1}{\frac{1}{0} + \frac{1}{R_2}} = \frac{1}{\infty + \frac{1}{R_2}} = \frac{1}{\infty} = 0.$$

Why, then, should a programmer writing code for the evaluation of parallel resistance formulas have to worry about treating division by zero as an exceptional situation? In IEEE arithmetic, the programmer is relieved of that burden. The default result obtained by dividing a positive number x by 0 is the floating point number ∞. In the case of the parallel resistance formula, this leads to the correct final result $1/\infty = 0$.

NaN from Invalid Operation

It is true that $a \times 0$ has the value 0 for any *finite* value of a. Similarly, we adopt the convention that $a/0 = \infty$ for any *positive* value of a. Multiplication with ∞ also makes sense: $a \times \infty$ has the value ∞ for any *positive* value of a. But the expressions $0 \times \infty$ and $0/0$ make no mathematical sense. An attempt to compute either of these quantities is called an *invalid operation*, and the IEEE default result of such an operation is NaN (Not a Number). Any subsequent arithmetic computation with an expression that involves a NaN also results in a NaN. When a NaN is discovered in the output of a program, the programmer knows something has gone wrong and can invoke debugging tools to determine what the problem is.

Addition with ∞ makes mathematical sense. In the parallel resistance example, we see that $\infty + \frac{1}{R_2} = \infty$. This is true even if R_2 also happens to be zero, because $\infty + \infty = \infty$. We also have $a - \infty = -\infty$ for any *finite* value of a. But there is no way to make sense of the expression $\infty - \infty$, which therefore yields the result NaN.

These conventions can be justified mathematically by considering addition of limits. Suppose there are two sequences x_k and y_k both diverging to ∞, e.g., $x_k = 2^k$, $y_k = 2k$, for $k = 1, 2, 3, \ldots$, or the other way around. Clearly, the sequence $x_k + y_k$ also diverges to ∞. This justifies the expression $\infty + \infty = \infty$. But it is impossible to make a statement about the limit of $x_k - y_k$, since the result depends on whether one of the sequences diverges faster than the other. Consequently, $\infty - \infty$ is NaN.

Exercise 7.1 *What are the values of the expressions* $\infty/0$, $0/\infty$, *and* ∞/∞*? Justify your answer.*

Exercise 7.2 *For what nonnegative values of* a *is it true that* a/∞ *equals zero?*

Exercise 7.3 *Using the 1950s convention for treatment of division by zero mentioned above, the expression* $(1/0)/10000000$ *results in a number very much smaller than the largest floating point number. What is the result in IEEE arithmetic?*

Exercise 7.4 *The formula* $R_1R_2/(R_1 + R_2)$ *is equivalent to* (7.1) *if* R_1 *and* R_2 *are both nonzero. Does it deliver the correct answer using IEEE arithmetic if* R_1 *or* R_2*, or both, are zero?*

Signed Zeros and Signed Infinities

A question arises: Why should $1/0$ have the value ∞ rather than $-\infty$? This is one motivation for the existence of the floating point number -0, so that the conventions $a/0 = \infty$ and $a/(-0) = -\infty$ may be followed, where a is a positive number. The reverse holds if a is negative. The predicate $0 = -0$ is true,[2] but the predicate $\infty = -\infty$ is false. We are led to the conclusion that it is possible that the predicates $a = b$ and $1/a = 1/b$ have opposite values (the first true, the second false, if $a = 0$, $b = -0$). This phenomenon is a direct consequence of the conventions for handling infinity.

The floating point number -0 is produced by several operations, including the unary operation -0, as well as a/∞ when a is negative, $a \times 0$ when a is negative, and the square root of -0, regardless of the rounding mode, as well as $a - a$ for any finite a when the rounding mode is *round down*. Programming environments typically do not display the sign of zero by default because users rarely need to distinguish between 0 and -0. See Kahan's wonderfully-titled paper "...Much Ado About Nothing's Sign Bit" [Kah87] for another motivation for the use of -0.

Exercise 7.5 *Are there any other cases in which the predicates $a = b$ and $1/a = 1/b$ have opposite values, besides a and b being zeros of opposite sign?*

Exercise 7.6 *What are the values of the expressions $0/(-0)$, $\infty/(-\infty)$, and $-\infty/(-0)$?*

Exercise 7.7 *What is the result for the parallel resistance formula (7.1) if $R_1 = 1$ and $R_2 = -0$?*

More about NaNs

The square root operation provides a good example of the use of NaNs. Before the IEEE standard, an attempt to take the square root of a negative number might result only in the printing of an error message and a positive result being returned. The user might not notice that anything had gone wrong. Alternatively, the program might be terminated with an error message. Under the rules of the IEEE standard, the square root operation is invalid if its argument is negative, the default result is NaN, and the program continues execution. Likewise, the remainder operation a REM b is invalid if a is $\pm\infty$ or b is ± 0, and the default result is NaN.

More generally, NaNs provide a very convenient way for a programmer to handle the possibility of invalid data or other errors in many contexts. Suppose we wish to write a program to compute a function that is not defined for some input values. By setting the output of the function to NaN if the input is invalid or some other error takes place during the computation of the function, the need to return special error messages or codes is avoided. Another good use of NaNs is for initializing variables that are not otherwise assigned initial values when they are declared. Furthermore, the bitstring in the fraction field can, in principle at least, be used to code the origin of the NaN. Consequently, we do not speak of a unique NaN value but of many possible NaN values. For more about NaNs, including the distinction between "quiet" and "signaling" NaNs, see [MB+18, Sec. 3.1.7.1].

When a and b are real numbers, one of three relational conditions holds: $a = b$, $a < b$, or $a > b$. The same is true if a and b are floating point numbers in the conventional sense, even if the values $\pm\infty$ are permitted. However, if either a or b is a NaN none of the three conditions $a = b$, $a < b$, $a > b$ can be said to hold (even

[2]Written $0 == -0$ in most programming languages.

if both a and b are NaNs). Instead, a and b are said to be *unordered*. Consequently, although the predicates $(a \leq b)$ and $(\text{not}(a > b))$ usually have the same value, they have *different* values (the first false, the second true) if either a or b is a NaN.

A simple way to check whether a variable a is a NaN is to check whether $a = a$: if the result is false, a must be a NaN. Checking whether $a = \text{NaN}$ does not work: the result is false regardless of the value of a.

The conversion of a binary format floating point number to an integer or decimal representation that is too big for the format in question is an invalid operation, but it cannot deliver a NaN since there is no floating point destination for the result.

Exercise 7.8 *Extend Exercise* 4.3 *to the case where either x or y may be* ± 0, $\pm\infty$, *or* NaN, *and the result may be "unordered".*

Overflow

In the days before IEEE arithmetic, *overflow* was usually said to occur when the exact result of a floating point operation was finite but with an absolute value that was larger than the largest floating point number, which was a finite number in the absence of ∞. As with division by zero, the usual treatment of overflow was to set the result to (plus or minus) the largest floating point number or to interrupt or terminate the program. In IEEE arithmetic, if an exact result is greater than $N_{\max}$ in magnitude, it is replaced by its correctly rounded value, which might be $\pm\infty$ or $\pm N_{\max}$, depending on the exact value and the rounding mode in effect; see Chapter 5 for details. However, the definition of the overflow exception is a little subtle.

Overflow is said to occur in IEEE arithmetic when the rounded value that is computed is different from what it would be if its exponent E were allowed to be sufficiently large. So, for example, if an exact result is $2N_{\max}$, this is rounded up to ∞ if the rounding mode in effect is *round to nearest* or *round up*, but rounded down to $N_{\max}$ if *round down* is in effect. Nonetheless, overflow is said to occur in all three cases, because if $E_{\max}$ were increased by one, the exact result would be a floating point number. On the other hand, if an exact value x lies strictly between $N_{\max}$ and $N_{\max} + \text{ulp}(N_{\max})/2$ and the rounding mode is *round to nearest*, the result is rounded to $N_{\max}$ *without* overflow occurring, since that would be the correctly rounded result even if $E_{\max}$ were increased. And if an exact result lies strictly between $N_{\max}$ and than $N_{\max} + \text{ulp}(N_{\max})$ when the rounding mode is *round down*, then again the result is rounded to $N_{\max}$ without overflow occurring, since again that would be the correctly rounded result even if $E_{\max}$ were increased.

Gradual Underflow

In the days before IEEE arithmetic, *underflow* was usually said to occur when the exact result of an operation was nonzero but with an absolute value that was smaller than the smallest positive floating point number, which was a normalized number in the absence of subnormals. The usual response to underflow was *flush to zero*: return the result 0. In IEEE arithmetic, if an exact result is smaller in magnitude than $N_{\min}$, the smallest positive normalized number, it is replaced by its correctly rounded value, which might be ± 0, a subnormal number, or $\pm N_{\min}$, depending on its value and the rounding mode in effect; see Chapter 5 for details. This is known as *gradual underflow*, because numbers are not rounded to ± 0 unless their magnitude is less than or equal to $2^{-p}N_{\min}$ (using *round to nearest*). Gradual underflow was the most controversial part of the IEEE standard when it was introduced. Its proponents argued that its use provides many valuable arithmetic rounding properties and significantly adds to the reliability of floating point software (see Coonen [Coo81], Demmel [Dem84], and

Kahan [Kah96b]); its opponents argued that arithmetic with numbers that may be either normalized or subnormal is too complicated to justify inclusion as an operation which will be needed only occasionally. The ensuing debate accounted for much of the delay in the adoption of the IEEE standard in 1985 [Kah10].

Although Intel microprocessors always supported arithmetic involving subnormal numbers in hardware, many other processors provided this support in software instead, which, although allowed by the standard, resulted in much slower computation times. As a consequence, many compilers offered the option of using a *flush to zero* option when underflow occurs, resulting in faster computation but in violation of the requirements of the standard. However, two developments have taken place in more recent years. First, the 2008 version of the standard introduced a new recommendation that language standards provide the opportunity for users to specify an *abrupt underflow* option.[3] When the *round to nearest* rounding mode is in effect, *abrupt underflow* is the same as *flush to zero*, preserving the sign: thus, a result between 0 and $N_{\min}$ would be rounded down to 0 and a result between $-N_{\min}$ and 0 would be rounded up to -0. However, *abrupt underflow* requires rounding up to $N_{\min}$ when the *round up* mode is in effect and the result lies between 0 and $N_{\min}$, and likewise rounding down to $-N_{\min}$ when the *round down* mode is in effect and the result lies between $-N_{\min}$ and 0. The second and much more important development is that many recent floating point processors handle subnormal arithmetic entirely in hardware without any performance penalty [MB+18, Sec. 8.1.2], reducing the appeal of *abrupt underflow.*

The motivation for gradual underflow can be summarized very simply: compare Figure 3.1 with Figure 4.1 to see how the use of subnormal numbers fills in the relatively large gaps between $\pm N_{\min}$ and zero. The immediate consequence is that the worst case absolute rounding error for numbers that underflow to subnormal numbers is the *same* as the worst case absolute rounding error for numbers that round to $N_{\min}$. See a fascinating letter from Donald Knuth endorsing gradual underflow, saying that he was originally skeptical of it "because it appears to be needlessly complicated to gain a few bits at the end of the range." But further thought brought him to the opposite view, that "gradual underflow is a wonderful invention... The thing that I missed was that gradual underflow adds an element of *completeness* to the system that seems impossible to achieve in any other way." See also the long footnote on [Knu98, p. 222].[4]

Consider the following subtraction operation, using the IEEE single format. The second operand is $N_{\min}$ and the first operand is a little bigger:

$$\begin{array}{rl} & (\ 1.01000000000000000000000\ \ )_2 \times 2^{-126} \\ - & (\ 1.00000000000000000000000\ \ )_2 \times 2^{-126} \\ = & (\ 0.01000000000000000000000\ \ )_2 \times 2^{-126} \\ \text{Normalize :} & (\ 1.00000000000000000000000\ \ )_2 \times 2^{-128}. \end{array} \tag{7.2}$$

The last line shows the ideal normalized representation, but this is smaller than $N_{\min}$. Without gradual underflow, we would have to flush the result to zero, so that in this case the answer to Question 6.4 is *no*. With gradual underflow, the answer 2^{-128} can be stored exactly, with the subnormal representation

$$\begin{array}{|c|c|c|}\hline 0 & 00000000 & 01000000000000000000000 \\ \hline \end{array}.$$

This suggests that, when gradual underflow is supported, the answer to Question 6.4

[3]Actually, we are not aware of any current support for *abrupt underflow* that works correctly for all rounding modes in either hardware or software.

[4]The letter is available at https://ieeemilestones.ethw.org/w/images/5/53/Knuth_to_delp_apr1980.pdf

is always *yes.* This is indeed the case; see Exercise 7.12. See also a nice illustration of this property in [MB+18, Fig. 2.1].

The standard specifies two options for how the underflow exception is to be defined. The first is simply that underflow is said to occur if the exact result is nonzero and lies strictly between $\pm N_{\min}$. The second, in analogy with the rule for the overflow exception, is that underflow is said to occur if the rounded value would lie strictly between $\pm N_{\min}$ even if its exponent E were allowed to be sufficiently small (see [MB+18, Sec 2.1.3]). This distinction can be quite confusing, so we give an example using single precision. Let x be the number

$$(1.11111111111111111111111)_2 \times 2^{-127} = (2 - 2^{-23}) \times 2^{-127},$$

where there are 23 ones after the binary point, so x is half-way between $N_{\min} = 2^{-126}$ and the largest subnormal number, which is

$$(0.11111111111111111111111)_2 \times 2^{-126} = (1 - 2^{-23}) \times 2^{-126}.$$

Using *round to nearest* with the tie-breaking rule, x rounds up to 2^{-126}. However, if the normalized exponent range were extended to include $E = -127$, then x would be an exact normalized floating point number, so underflow (using the second definition) *is* said to occur in this case. On the other hand, the number y given by

$$(1.111111111111111111111111)_2 \times 2^{-127} = (2 - 2^{-24}) \times 2^{-127},$$

where now there are 24 ones after the binary point, could not be represented exactly even if the exponent could be reduced to -127, because $2 - 2^{-24}$ requires 24 bits after the binary point to be represented exactly. The number y rounds to 2^{-126} whether or not the normalized exponent range is extended, so in this case underflow *is not* said to occur according to the second definition, although it does occur according to the first definition.

Exercise 7.9 *Using round to nearest, what numbers are rounded down to zero and what numbers are rounded up to the smallest positive subnormal number?*

Exercise 7.10 *Consider the operation*

$$(y \ominus x) \oplus x,$$

where the first part of the operation, $y \ominus x$, underflows. What is the result when gradual underflow *is used? What is the result when* flush to zero *is used? Which gives the exact result? (See [Cod81].)*

Exercise 7.11 *Suppose that x and y are floating point numbers with the property that*

$$\frac{1}{2} \leq \frac{x}{y} \leq 2.$$

Show that the exact difference $x - y$ is also a floating point number, so that, as a consequence, $x \ominus y = x - y$, if gradual underflow *is used. Show that this is not always the case if* flush to zero *is used. (See [Kah96b, Ste74].)*

Exercise 7.12 *Prove that the answer to Question* 6.4 *is always yes in IEEE arithmetic, because of gradual underflow.*

Exercise 7.13 *Is the worst case relative rounding error for numbers that underflow to subnormal numbers the same as the worst case relative rounding error for numbers that round to $N_{\min}$? Why or why not?*

Table 7.1: IEEE Default Results when Exceptions Occur

Exception	Default Result
Invalid Operation	NaN
Division by Zero	$\pm\infty$
Overflow	correctly rounded value: $\pm\infty$ or $\pm N_{\max}$
Underflow	correctly rounded value: ± 0, $\pm N_{\min}$ or subnormal
Inexact	correctly rounded value: could be any floating point number

The Five Exception Types

Altogether, the IEEE standard defines five kinds of exceptions: invalid operation, division by zero, overflow, underflow, and inexact, together with a default result for each of these. All of these have now been described except the last. The inexact exception is, in fact, not exceptional at all because it occurs every time the result of an arithmetic operation is not a floating point number and therefore requires rounding. This occurs in most floating point operations, and, in particular, it occurs when overflow takes place. However, it may or may not occur when underflow takes place, since the correct result might be an exact subnormal number. Table 7.1 summarizes the default results for the five exceptions.

The IEEE standard specifies that when an exception occurs it must be *signaled*; the signal invokes either the default or an alternate exception handler. Default handling means delivering the default result discussed above (see Table 7.1). But the standard also recommends that language standards provide mechanisms for the user to invoke an alternate exception handler.[5] For example, in the case of underflow being signaled, the result could be abrupt underflow instead of the default gradual underflow. Another example, in the case of the divide by zero exception, would be to terminate execution of the program, instead of delivering $\pm\infty$. See [IEE08, Sec. 8] for more details.

The standard also specifies that when an exception is signaled, the *status flag* corresponding to that exception must be set, so that it is possible for a program to determine later that the exception occurred. However, there is a caveat: in the case of the underflow exception, the corresponding status flag is *not* set if the exact result is a subnormal number, so that no rounding is required.

The appearance of a NaN in the output of a program is usually a sign that something has gone wrong. The appearance of ∞ in the output may or may not indicate a programming error, depending on the context. When writing programs where division by zero is a possibility, the programmer should be cautious. Operations with ∞ should not be used unless a careful analysis has ensured that they are appropriate.

The IEEE Philosophy on Exceptions

The IEEE approach to exceptions permits a very efficient and reliable approach to programming in general, which may be summarized as: *Try the easy fast way first; fix it later if an exception occurs.* For example, suppose it is desired to compute

$$\sqrt{x^2 + y^2}. \tag{7.3}$$

[5]An alternate exception handler is also known as a *trap*, and invoking it is also known as *trapping the exception.*

Even if the result is within the normalized range of the floating point system, a direct implementation might result in overflow. The traditional careful implementation would guard against overflow by scaling x and y by $\max(|x|, |y|)$ before squaring. But with IEEE arithmetic, the direct computation may be used. The idea is to first clear the exception status flags, then do the computation, and then check the status flags. In the unlikely event that overflow (or underflow) has occured, the program can take the necessary action. . For a more extensive discussion of how these ideas can be used for many different numerical computations, see [DL94], [Hau96] and [Hig02, Sec. 27.1]. To make such ideas useful, it is essential that the IEEE standard be properly supported by both software and hardware, so that setting and testing the exception status flags is permitted by the programming language in use and does not significantly slow down execution of the program. However, setting and testing status flags is becoming increasingly impractical, because of changes in computer architecture design, as explained in [BJ+23, p. 206].

Hardware support for the standard is discussed in the next chapter, and programming language support is discussed in the following chapter. Although support for the standard is still far from perfect, it has steadily improved over the years. Alternatives to IEEE floating point have been proposed from time to time but, up until now, none have been very influential. For some thoughts on some of the early proposals, see [Dem87] as well as the intriguingly-entitled post [Dem91]. More recent proposals include *unums* and *posits*; for a detailed and balanced appraisal of these, see [Din19]. However, non-IEEE low precision formats, which are nonetheless inspired by the IEEE formats, are becoming common and are discussed in Chapter 15.

Chapter 8

Floating Point Microprocessors

In the early personal computer era, the two leading microprocessor manufacturers incorporated the main ideas of the IEEE standard in their designs. These were Intel (whose chips were used by IBM PCs and clones) and Motorola (whose 68000 series chips were used by the Apple Macintosh II and the early Sun Microsystems workstations). Later microprocessors, such as the Sun Sparc, DEC Alpha, and IBM RS/6000 and Power PC, also followed the standard. Even the IBM 390, the successor to the 360/370 series, offered support for the IEEE standard as an alternative to the long-supported hexadecimal format. We focus here on the x86 microprocessors from Intel and AMD, since these have been and still are by far the most widely used floating point systems in personal computers. Subsequently, we also briefly discuss other processors.

The x86/x87 Microprocessors

Intel introduced the 8086 microprocessor in 1978. This chip included a central processing unit (CPU) and an arithmetic-logical unit (ALU) but did not support floating point operations. In 1980, Intel announced the 8087 and 8088 chips, which were used in the first IBM PCs. The 8088 was a modification of the 8086. The 8087 was the floating point coprocessor, providing a floating point unit (FPU) on a separate chip from the 8088. The 8087 was revolutionary in a number of respects. It was unprecedented that so much functionality could be provided by such a small chip. Many of the features of the IEEE standard were first implemented on the 8087. The extended format recommended by the standard was based on the 8087 design.

The immediate successors of the 8087, the 80287 and 80387, were also coprocessors, implemented separately from the main chip (the 80286 and 80386, respectively). However, later microprocessors in the series, namely the 80486 and the many generations of Pentium, included the FPU on the main chip. Collectively, these are known as x86 processors with x87 FPUs. Though each new machine was faster than its predecessor, the architecture of the Pentium x87 FPUs remained essentially the same as that of the 8087.

The x87 floating point instructions operate primarily on data stored in eight 80-bit floating point registers,[1] each of which can accommodate an extended format floating

[1]These registers are organized in a logical stack, as explained in detail in Chapter 8 of the first edition of this book.

point number (see Chapter 4). However, it was expected that programs would usually store variables in memory using the single or double format. The extended precision registers were provided with the idea that a sequence of floating point instructions, operating on data in registers, would produce an accumulated result that is more accurate than if the computations were done using only single or double precision. This more accurate accumulated result would then be rounded to fit the single or double format and stored to memory when the sequence of computations is completed, perhaps giving the correctly rounded value of the exact result, which would be unlikely to happen for a sequence of operations that uses the same precision as the final result.[2] The standard encourages full use of the extended precision registers, but it also requires that the user be able to lower the precision that is used in operations via setting a *precision mode.*[3] This may be set to any of the supported formats; for example, if the precision mode in effect is *double*, the floating point computations must deliver results that are rounded to double precision even when they are are stored in an 80-bit register. A subtle point is that, when the precision mode is *double*, an intermediate floating point value stored in the register that overflows the double format is *not* stored as $\pm\infty$ and does *not* generate an exception, as long as it does not overflow the extended format and the final result stored to memory does not overflow the double format.

The x87 FPUs implement correctly rounded arithmetic, controlled by the four possible rounding modes discussed in Chapter 5. The rounding mode and the precision mode are set in a *control word* stored in a dedicated 16-bit register. There is also a *status word* stored in another 16-bit register. When an exception occurs, the exception status flag is set using a bit in the status word, and a corresponding bit in the control word is examined to see if the default result should be delivered, or if the exception should be trapped with control passed to an alternate exception handler.

AMD (Advanced Micro Devices) also designs and markets x86/x87 microprocessors under a licensing agreement with Intel dating back to the 1980s. According to [War94], the licensing agreement to an outside company was a condition for IBM to agree to use the Intel x86 processors in its early personal computers.

The instruction set for the early x86/x87 processors was called IA32 (Intel Architecture - 32 bit, where the latter refers to the way that the computer memory is organized and addressed). In 2001, Intel released the Itanium microprocessor that used a new instruction set called IA64 (Intel Architecture - 64 bit). However, for a variety of reasons, partly incompatibility with existing x86 machines, this was not successful. In response, AMD introduced its own 64-bit architecture known as AMD64 that is compatible with earlier x86 machines. This architecture, now commonly known as x86-64, was eventually adopted by Intel for its processors, so essentially all recent x86 processors use the x86-64 architecture.

Recent x86 Microprocessors

In 1999, Intel introduced eight SSE 128-bit floating point registers to the x86 architecture, as an alternative to the x87 FPU. Here SSE means Streaming SIMD Extensions, where SIMD stands for Single Instruction, Multiple Data. However, the 128-bit registers do *not* store extended precision or quadruple precision floating point numbers. Instead, each 128-bit register stores four single format (binary32) floating point

[2]However, this “double rounding” process, first to the extended precision register, and later to the single or double format variable in memory, may sometimes give a *less* accurate result than what would be obtained without using extended precision registers. See [MB+18, Sec. 3.2] for details.

[3]Called *rounding precision mode* in the 1985 standard, and replaced by the *preferredWidth* attribute in the 2008 standard.

numbers, so that one instruction can operate on four floating point numbers simultaneously. Later versions, starting with SSE2, support alternatively storing two double format (binary64) numbers in each register. In 2011, Intel introduced AVX (Advanced Vector Extensions) processors, initially with 256-bit registers that can store four double format numbers, eight single format numbers, or 16 half precision numbers (using the binary16 format mentioned in Chapter 4). More recent AVX machines have 512-bit registers which can store eight double format numbers, 16 single format numbers, or 32 half precision numbers. SSE and AVX floating point arithmetic supports all requirements of the IEEE standard, including the FMA instruction mandated as of 2008, as well as setting of the exception flags and alternate exception handling, but they do not support the recommended extended precision format. However, all Intel and AMD x86 processors continue to support x87 FPUs as an alternative, using 80-bit floating point extended precision registers, partly because there is so much legacy code using these. The x87 FPU does not have an FMA instruction, since the x87 architecture predates FMA hardware design and the FMA requirement in the 2008 standard.

As of August 2022, according to the Top 500 list [Top22], 384 of the 500 fastest supercomputers in the world are based on Intel x86 microprocessors and 101 are based on AMD x86 microprocessors. Most if not all of these CPUs have multiple cores, so they can perform many computations in parallel.

Arm Microprocessors

Arm, formerly known as ARM (Advanced RISC Machines, where RISC means Reduced Instruction Set Computer), dominates the market for chips used in embedded systems, including mobile phones. Arm does not manufacture microprocessors. Instead, it designs CPU architectures and licenses those designs to other companies such as Qualcomm, Apple or Samsung who incorporate them into their processors.[4] Not all Arm microprocessors use floating point, but many do, fully complying with the IEEE standard requirements for single and double precison. Apple's Mac computers used Intel x86 chips from 2006 to 2020, but the most recent Macs use M1 and M2 SoCs (Systems on a Chip), which are based on Arm microprocessor designs.

GPUs

A GPU (Graphics Processing Unit) consists of a large set of specialized processors running in parallel. Originally designed for graphics applications, including video games, they are now also used for many other purposes. The early GPUs were not IEEE compliant, but later GPUs using NVIDIA's CUDA (Compute Unified Device Architecture) use IEEE single, double and more recently half precision formats and support most of the requirements of the IEEE standard, including access to the rounding modes, the FMA instruction, and default results when exceptions occur. However, there are no exception status flags, and hence alternative exception handling is not supported.[5] More recently, other manufacturers have also built GPUs that are largely IEEE compliant.

Summary of Hardware Support

The hardware support for the IEEE standard discussed in this chapter is summarized in Table 8.1. It is impressive that most widely used modern microprocessors, including Intel and AMD's SSX and AVX machines, Arm microprocessors and Apple's M1 and

[4]https://www.techspot.com/article/1989-arm-inside/]
[5]https://docs.nvidia.com/cuda/floating-point/index.html.

	Single (binary32)	Double (binary64)	Ext. Prec.	**All Round. Modes**	**FMA**	**Default Excep. Handl.**	**Excep. Status Flags**
x87	yes	yes	yes	yes	no	yes	yes
SSE/AVX	yes	yes	no	yes	yes	yes	yes
Arm/M1/M2	yes	yes	no	yes	yes	yes	yes
CUDA GPU	yes	yes	no	yes	yes	yes	no

Table 8.1: Some of the hardware support for the IEEE standard. Bold face indicates features required by the 2008 and 2019 versions of the standard. Default exception handling requirements are summarized in Table 7.1. Exception status flags are also discussed in Chapter 7.

M2 chips, support all the main requirements of the IEEE standard, namely providing at least one IEEE basic format, the four rounding modes, the default response to exceptions, the FMA instruction, and setting of and access to exception status flags, enabling alternate exception handling. Furthermore, CUDA GPUs support all of these except the last.

However, as noted at the end of the previous chapter, many recently introduced microprocessors are using low precision formats, mostly non-IEEE compliant. We discuss this development in Chapter 15.

Chapter 9

Programming Languages

Programs for the first stored program electronic computers consisted of a list of machine instructions coded in binary. It was a considerable advance when assemblers became available, so that a programmer could use mnemonic codes such as LOAD X, instead of needing to know the binary code for the instruction to load a variable from memory to a register and the physical address of the memory location. The first widely available higher level programming language was developed at IBM in the mid 1950s and called Fortran, for *formula translation.* Programmers could write instructions such as `x = (a + b)*c`, and the compiler would then determine the necessary machine language instructions to be executed. Fortran became extremely popular and is still widely used for scientific computing today. Soon after Fortran was established, the programming language ALGOL was developed by an international group of academic computer scientists. Although ALGOL had many nicer programming constructs than Fortran, it never achieved the latter's success, mainly because Fortran was designed to be highly efficient from its inception, which ALGOL was not.[1] However, many of the innovations of ALGOL, such as the notions of block structure and recursion, influenced later versions of Fortran as well as the design of subsequent programming languages.[2] One of these was C, which was developed by Bell Labs in the 1970s, and emerged as the *lingua franca* of computing in the 1980s. In this chapter, we briefly discuss the status of support for IEEE arithmetic by several programming languages, specifically Fortran, C, MATLAB, Java, Python and Julia, all of which are widely used for floating point computation.

Language Support for IEEE 754

To make the best use of hardware that supports the standard, a programming language should define types that are compatible with the IEEE formats, allow control of the rounding mode, and provide the standard-defined default results for exceptions as the default behavior. Ideally, a programming language should also allow access to and resetting of the exception status flags, and permit alternate exception handling. All of these were provided soon after the publication of IEEE 754 by Apple in SANE (Standard Apple Numerics Environment) [App88], using the Pascal and C languages,

[1]Wilkes wrote in [Wil98], "As an intellectual achievement, Fortran was stillborn by its very success as a practical tool."

[2]The author was fond of a variant called ALGOL W which he used in his student days at UBC and Stanford in the 1970s. The W stood for its designer Niklaus Wirth, who went on to develop Pascal, a language that became widely used for teaching.

but this did not become widely used, and its fine example was not followed by other programming languages and systems for many years. One reason for this is that the 1985 version of the standard did not set requirements or make recommendations for programming language support.

C and Fortran

Although they have very different histories, design philosophies and user bases, C and Fortran have a lot in common. They are both languages that require a compiler to compile a program before it can be executed. Although developed originally by private organizations, for decades the languages have been standardized by national and international committees, and free compilers are widely available. Both languages had important standard revisions with many new features established in 1990, but neither revision supported IEEE arithmetic, although the IEEE standard was published in 1985 and had been in preparation for years. However, thanks to the efforts of the Numerical C Extensions Group (NCEG) and the ANSI Fortran committee X3J3, the situation greatly improved in the late 1990s. The International Standards Organization/International Electrotechnical Commission (ISO/IEC) revision of the C language standard known as C99 was completed in 1999 [ISO99], and an ISO/IEC revision of the Fortran standard known as Fortran 2003 was approved in 2003. Although neither C99 nor Fortran 2003 required every implementation to support IEEE 754, they both provided standardized mechanisms for accessing its features when supported by the hardware.

C99 introduced a macro,[3] `__STDC_IEC_559__`, which is supposed to be predefined by implementations supporting IEEE 754. Any implementation defining this macro must then conform with various requirements. The types *float* and *double* must use the IEEE single and double formats, respectively, and it was recommended that type *long double* fit the IEEE 80-bit extended format requirements.[4] The implementation must provide access to status flags via macro constants such as `FE_DIVBYZERO`, specification of rounding modes via macros such as `FE_DOWNWARD`, and access to special constants via macros such as `INFINITY`. Other macros with function syntax must also be provided, such as `isnan` to determine if a number has a NaN value, `isnormal` to test if a value is in the normalized range, and `fesetround` to set the rounding mode. Details are given in Annex F of [ISO99]. Now, more than 20 years after C99 was published, virtually all C compilers support IEEE 754 and define the `__STDC_IEC_559__` macro accordingly, so there is almost never any need for the programmer to check whether it is predefined.[5] In the next chapter, we discuss how to write simple numerical programs in C, also illustrating the use of some of the other macros mentioned above. For more information on floating point in C, as well as its widely used extension C++, see [Bee17] and [MB+18, Sec. 6.2–6.3]. For information on Fortran 2003, see [MRC18], and for more on floating point in Fortran generally, see [Bee17, Appendix F] and [MB+18, Sec. 6.4].

A significant change in the 2008 version of the IEEE standard was to set requirements and recommendations for programming language standards. Not many requirements were specified, the most important being that programmers should be able to set the rounding modes, but there are many recommendations. These had no effect on

[3] As mentioned earlier, IEC 559 is an international name for IEEE 754.

[4] Since the 80-bit extended format is mostly restricted to the older x87 microprocerssors, in practice, the meaning of the type *long double* varies widely with different compilers, ranging from *binary128* (which is not generally available in hardware) to *double double* (see Chapter 14) to simply *double*.

[5] N. Beebe, private communication, 2022

the C11 and C17 standard revisions, which did not make any changes to floating point support requirements or recommendations, but they will have a significant impact on the next C standard, currently in draft under the name C23 and expected to appear in 2024. See [MB+18, Fig. 6.1] which illustrates "the tangled normalization timeline" of floating-point and C language standards.

C and Fortran have very different rules for how compilers are allowed to evaluate arithmetic expressions: see [MB+18, Sec.6.1.1.3] for details. The availability of the FMA on most machines also raises the question: can the compiler evaluate an expression such as $a \times b + c$ using a single FMA operation, or may it only use the FMA operation when it is explicitly directed to do so by a function call? The answer depends both on the compiler and on the user who, in the case of C, has some control over this [MB+18, Sec. 6.2.3.2]. In most cases, using the FMA whenever possible is desirable, but see [MB+18, Sec. 6.2.3.2] for an example where using an FMA to evaluate the expression $x^2 - y^2$ breaks the symmetry with respect to x and y and hence can give a positive or negative result even when $x = y$.[6]

Many widely used mathematical software libraries are available in both Fortran and C. The most important is LAPACK (Linear Algebra PACKage) [Demmel et al], whose routines can be used to solve many problems arising in linear algebra, such as the solution of systems of linear equations and the computation of eigenvalues and singular values of matrices. LAPACK is written in Fortran, was first released in 1992 and is frequently updated. LAPACK routines call the BLAS (Basic Linear Algebra Subroutines) to efficiently implement the most basic linear algebra computations, such as vector addition, matrix-vector multiply and matrix-matrix multiply; these can be tuned for efficiency on different machines and are available from a variety of sources including Intel and AMD. LAPACK superseded EISPACK and LINPACK, which were historically important Fortran libraries for solving eigenvalue problems and systems of linear equations respectively. There is a C interface called LAPACKE, which largely replaces the earlier CLAPACK. It is also possible to call LAPACK and BLAS routines from many other programming languages, including Java, Python and Julia.

MATLAB

The interactive system MATLAB, standing for Matrix Laboratory, was first developed by Cleve Moler in the late 1970s and early 1980s as a tool for conveniently solving matrix problems, by providing a simple interface to access the relevant software available in EISPACK and LINPACK [Mol18]. The original version was written in Fortran. Moler and others then went on to found The Mathworks and market MATLAB commercially starting in 1984, rewriting the system in C, and eventually incorporating calls to LAPACK and the BLAS. MATLAB is very popular because of its ease of use, its convenient and accurate matrix operations, its extensive software toolboxes, and its graphics capabilities.[7] A particularly important innovation was the introduction of the backslash (\) operator; the assignment $x = A\backslash b$ gives the solution of the linear system of equations $Ax = b$, where A is a matrix and b is a vector. For many years, MATLAB had only one variable type: complex matrix, which was a two-dimensional array of real or complex numbers, with real (and if nonzero, imaginary) parts stored using the IEEE double format, but in more recent years, other data types have been added, including support for the IEEE single format.

MATLAB uses IEEE arithmetic with default exception handling, modified to take

[6]Instead of $x^2 - y^2$, an alternative is $(x + y)(x - y)$, but the computed result can be slightly larger than the computed result of x^2, while this cannot occur for the formula $x^2 - y^2$, with or without the FMA [Jea19, Sec. 5].

[7]The author has been a devoted MATLAB user for decades for all these reasons.

sensible action supporting complex numbers. So, for example, taking the square root of a negative real number x does not generate an exception, but produces the rounded value of $\sqrt{-x}$, using round-to-nearest, times the imaginary unit, denoted `i`. The square root of -0 gives 0, not -0. Unlike in most programming languages, in MATLAB, the user has no access to the FMA operation, although it is used by the BLAS which MATLAB calls. MATLAB does not officially support rounding mode control. Nonetheless, it is possible to change the rounding mode in current versions, although there is no guarantee that this will be supported in the future, because of a variety of technical issues, including concurrency when multithreading. To change the rounding mode, type `feature('setround',r)`, where r is one of `Inf` (for *round up*), `-Inf` (for *round down*), `0` (for *round toward zero*), or `'nearest'` (or `0.5`) (for *round to nearest*, the default). MATLAB does not support access to the exception status flags or alternate exception handling. Many books describe MATLAB; we recommend [**?**].

The plots in Chapters 11 and 12 were produced using MATLAB.

Java

The Java programming language was developed in the 1990s by Sun Microsystems, later taken over by Oracle. The philosophy of Java is that the same program should deliver identical answers on every machine. This is achieved by compiling Java source code to *byte code*, which is then run on a *Java Virtual Machine* designed for the user's hardware.

Java was designed to follow some of the IEEE standard requirements from the beginning. In particular, the Java primitive types *float* and *double* are required to conform to the IEEE single and double floating point formats. Unfortunately, the requirements of the Java language and the IEEE floating point standard have conflicts with each other. One contradiction is that Java insists that the results of arithmetic operations be rounded to nearest, in contrast to the IEEE requirement that four rounding modes be supported. Java supports default exception handling, but it does not allow access to the exception status flags or alternate exception handling. The Java compiler is not allowed to generate an FMA instruction to implement an expression such as $a \times b + c$; it must be explicitly requested by calling an FMA library routine.

A major difficulty has been that although Java programs are required to give identical output on all platforms, the IEEE floating point standard recommends, but does not require, extended precision computation for intermediate results that are later to be rounded to narrower formats. When this is used, the results will likely be different – usually more accurate — than if extended precision is not used. The first version of Java insisted that extended precision be completely disabled, but the consequence was very slow program execution on x86/x87 machines since the floating point hardware could not be used. Consequently, Java 1.2 introduced the *strictfp* (strict floating point) keyword: if this was *not* specified, the x87 precision mode could be used to disable extended precision efficiently, but this nonetheless led to inconsistent behavior of codes on different machines because of the fact that x87 extended precision computations, even with the precision mode set to *double*, have a different overflow threshold than pure double precision computations (see Chapter 8). Starting in 2001, SSE2 became available on x86 machines, supporting both single and double precision floating point operations, so the difficulty was eventually alleviated by specifying that x87 computations should not be used. Effective 2021, with Java 17, the *strictfp* keyword is no longer recognized: all floating point computations must use IEEE single or double precision, not extended precision.

See [MB+18, Sec. 6.5] for an extensive discussion of floating point using Java, and see [Dar98] for early work on the subject.

Python

Python first appeared in the 1990s. It has become extremely popular, and is often chosen as the language for a first course in programming. Like MATLAB, Python uses an interactive interpreter, not a compiler, but its efficiency results from using precompiled libraries. It is *open source*, meaning that it is freely available for possible modification and redistribution under certain licensing conditions, and it is maintained by a large community of developers. It did not become widely used for numerical computing until the introduction of SciPy and NumPy in the early 2000s. The original Python had only one floating point type available, called *float* but using the double format (binary64). However, the NumPy package offers variable types corresponding to IEEE half, single, double and extended floating point formats, uses IEEE default exception handling, and can access the exception status flags and implement alternate exception handling via `numpy.seterr`, but it does not explicitly support setting the rounding modes. NumPy has extensive support for linear algebra and matrix computation, optionally using LAPACK.

Julia

Julia, announced in 2012, might be only the third successful programming language specifically designed for numerical computing, the first two being Fortran and MATLAB. Like Fortran, it is intended to have high performance, but like MATLAB, it is designed to be easy to use, allowing code to be developed interactively before it is compiled for fast execution. Julia supports the IEEE single and double formats as basic types, named *Float32* and *Float64*, and it supports IEEE default exception handling. However, it does not explicitly support setting the rounding modes,[8] does not allow the compiler to generate an FMA instruction without an explicit call to a library routine, and does not support access to exception status flags or alternate exception handling. Like MATLAB, Julia supports computing with complex numbers very conveniently, although with somewhat different syntax. As an example, `sqrt(-1)` delivers NaN, because the square root function is supposed to return a real number when its argument is real, but `sqrt(-1+0im)` delivers `0.0 + 1.0im`, as the square root returns a complex number when its argument is complex; here, `im` denotes the imaginary unit.[9] Julia is built on many modern ideas in computer science and has become very successful extremely rapidly. For more about Julia, see [BE+17] as well as [Jul23].

Summary of Programming Language Support

Support for the IEEE standard by programming languages discussed in this chapter is summarized in Table 9.1. As with hardware support, it is impressive that all languages listed support both IEEE single and double types and support the default results when exceptions occur, including gradual underflow. However, among these languages, full support for the required rounding modes is, at present, limited to C and Fortran, and support for the recommended extended precision computations and alternate error handling using the exception status flags is limited to C, Fortran and Python. It will be interesting to see whether the next revision of the standard reconsiders these requirements and recommendations.

[8]Early versions of Julia did allow setting the rounding modes.

[9]A similar convention is used by NumPy.

	Single (fp32)	Double (fp64)	Ext. Prec.	**All Round. Modes**	**FMA**	**Default Excep. Handl.**	Excep. Status Flags
C	yes	yes	yes	yes	yes	yes	yes
Fortran	yes	yes	yes	yes	yes	yes	yes
MATLAB	yes	yes	no	no	no	yes	no
Java	yes	yes	no	no	yes	yes	no
Python	yes	yes	yes	no	yes	yes	yes
Julia	yes	yes	no	no	yes	yes	no

Table 9.1: Some of the programming language support for the IEEE standard. Bold face indicates features that must be supported according the 2008 and 2019 versions of the IEEE standard. A "yes" in columns 2–4 means that there is a built-in type with the relevant precision, although in the case of extended precision, for C and Fortran, this may not be the case for all compilers [OK?], and in the case of Python, this is via the NumPy library [OK?]. A "yes" in the next column means the language supports specifying any of the four rounding modes. In the next column, a "yes" indicates that the FMA is accessible through a function call, but does not imply that the compiler or interpreter can generate an FMA instruction without explicit direction from the programmer. A "yes" in column 7 means that the language does not, by default, override the default exception handling required by the standard (see Table 7.1). A "yes" in the final column means that the programmer has access to the exception status flags, enabling alternate exception handling; note that although the standard requires that the hardware provide the status flags, it does not require language standards to provide access to them.

Chapter 10

Floating Point in C

In this chapter, we discuss how to get started with floating point computation in C. As explained in the previous chapter, the C99 language standard introduced extensive support for IEEE floating point arithmetic in the C language, and virtually all modern compilers support this, so we frequently refer to C99 below.

Float and Double, Input and Output

In C99, the type *float* refers to the IEEE floating point single format, and when we do computations with variables of type *float*, we say that we are using single precision. Here is an echo program that reads in a floating point number using the standard input routine `scanf` and prints it out again using the standard output routine `printf`:

```
main ()      /*  Program 1: Echo */
{
   float x;
   scanf("%f", &x);
   printf("x = %f", x);
}
```

The second argument to the `scanf` statement is not the *value* of the variable `x` but the *address* of `x`; this is the meaning of the `&` symbol. The address is required because the input routine needs to know where to store the value that is read. On the other hand, the second argument to the `printf` statement is the *value* of the variable `x`. The first argument to both routines is a *control string*. The two standard *format codes* used for specifying floating point numbers in these control strings are `%f` and `%e`. These refer, respectively, to *fixed decimal* and *exponential decimal* formats. Actually, `%f` and `%e` have identical effects when used with the input routine `scanf`, which can process input in either a fixed decimal format (e.g., `0.666`) or an exponential decimal format (e.g., `6.66e-1`, meaning 6.66×10^{-1}). However, different format codes have different effects when used with the output routine `printf`. The `scanf` routine calls a decimal to binary conversion routine to convert the input decimal format to the internal binary floating point representation, and the `printf` routine calls a binary to decimal conversion routine to convert the floating point representation to the output decimal format. The C99 standard recommends that both conversion routines use the rounding mode that is in effect to correctly round the results.

Assuming Program 1 has been saved in a file and compiled, let us consider the output when it is executed. Table 10.1 shows the output of Program 1 for various output formats in the `printf` statement, when the input is 0.66666666666666666666 (the

Table 10.1: Output of Program 1 for Various Output Format Codes

Format code	Output
`%f`	`0.666667`
`%e`	`6.666667e-01`
`%8.3f`	`0.667`
`%8.3e`	`6.667e-01`
`%21.15f`	`0.666666686534882`
`%21.15e`	`6.666666865348816e-01`

value of 2/3 to 20 digits). The first line shows that the `%f` format code generates output in a fixed decimal output, while the second line shows that `%e` generates exponential notation. Neither of them echoes the input to the accuracy given originally, since it is not possible to store the value of 2/3 to 20 accurate digits using single precision. Instead, the input value is correctly rounded to 24 bits of precision in its significand, which, as we discussed in Chapter 5, corresponds to approximately 7 significant decimal digits. Consequently, the format codes `%f` and `%e` print, by default, 6 digits after the decimal point, but `%e` shows a little more accuracy than `%f` since the digit before the decimal point is nonzero. In both cases, the decimal output is rounded, using the default *round to nearest* mode; this explains the final digit 7. The next two lines of Table 10.1 show how to print the number to *less* precision if so desired. The 8 refers to the total field width, and the 3 to the number of digits after the decimal point. The last two lines show an attempt to print the number to *more* precision, but we see that about half the digits have no significance. The output is the result of converting the single precision binary representation of 2/3 to more than the number of decimal digits that can be accurately represented by the single format. . The output would be exactly the same if, instead of reading the value 0.66666666666666666666 for `x`, we set the value of `x` equal to the quotient 2.0/3.0. It is important to realize that regardless of the decimal output format, the floating point variables are *always* stored using the IEEE binary formats described in Chapter 4.

Using the `%f` format code is not a good idea unless it is known that the numbers are neither too small nor too large. For example, if Program 1 is run on the input `1.0e-10`, the output using `%f` is 0.000000, since it is not possible to print the desired value more accurately using only 6 digits after the decimal point without using exponential notation. Using `%e` we get the desired result `1.000000e-10`. A useful alternative is `%g`, which chooses either a fixed or an exponential display, whichever is shorter.

We can represent the value of 2/3 more accurately by declaring `x` to have type *double* instead of *float*. Type *double* uses the IEEE floating point double format,[1] and when we do computations with variables of type *double*, we say that we are using double precision. But if we simply change *float x* to *double x* in Program 1, using the same input, we get a completely wrong answer such as `-6.392091e-236` (using the `%e` output format code). The problem here is that the `%f` in the input format code instructs the `scanf` routine to store the input value in the IEEE single format at the address given by `&x`; `scanf` does not know the type of `x`, only its address. When `x` is printed, its value is converted to decimal assuming it is stored in the double format. Therefore, any `scanf` statement that reads a double variable must use the control format code `%lf` or `%le` (for long float or long exponential) instead of `%f` or `%e`, so that

[1]The name *long float* was used in the past but is obsolete in this context.

the result is stored in the IEEE double format. Do not attempt to use `%d` to refer to the double format; `%d` actually means integer format.

The `printf` control string does *not* need to use `%lf` or `%le` (as opposed to `%f` or `%e`) when printing the values of *double* variables. This is because `printf` *always* expects *double* variables, and so *float* variables are always automatically converted to *double* values before being passed to the output routine. Therefore, since `printf` always receives *double* arguments, it treats the control strings `%e` and `%le` exactly the same; likewise `%f` and `%lf`, `%g` and `%lg`. However, the default, if no field width and precision are specified, is to output the number to approximately the precision of the single format, so we need to specify higher precision output to see more digits. In summary, then, if we change *float* to *double* in the variable declaration, change `%f` to `%lf` in the `scanf` statement, and change `%e` to `%21.15e` in the `printf` statement, we obtain an output that has 15 significant digits after the decimal point.

Choice of the wrong output format code can have serious implications! A good example is the story of a German state election in 1992. According to the rules of this election, a minimum of 5% of the vote was required for a party to be allocated any seats in the parliament. The evening of the election, results were announced declaring that the Green party had received exactly 5% of the vote. After midnight it was discovered that the Greens actually had only 4.97% of the vote. The program that printed out the percentages showed only one place after the decimal, and had rounded the count up to 5% [WW92].

Exercise 10.1 *Supposing that the German election program was written in C, what output format code would have led to the incorrect conclusion that the Greens had exactly 5% of the vote? What would have been a better output format code?*

Exercise 10.2 *(D. Gay) The wrong answer obtained by Program* 1 *if float is changed to double without any change to the* `scanf` *format depends on whether the machine uses Big Endian or Little Endian addressing (see the end of Chapter* 4*). Why is this? If you have machines of both kinds available, experiment to see what results you obtain on each.*

Two Loop Programs

Let us now consider a program with a "while loop":

```
main()   /* Program 2: First Loop Program */
{
   int n = 0;
   float x = 1;

   /* Repeatedly divide x by 2 until it underflows to 0 */

   while (x > 0) {
      x = x/2;
      n++;
      printf("(2 raised to the power -%d) = %e \n", n, x);
   }
}
```

Program 2 initializes an IEEE single format floating point variable `x` to 1, and then repeatedly divides it by 2. We could, for clarity, replace 2 in the statement `x = x/2` by `2.0`, but this is not necessary because `x` has type *float*. (If `x` had type *int*, the integer division operator would be used instead, giving the result 0, for example, if `x` had the value 1.) The termination test is `x > 0`: the loop continues to execute as long as `x` is positive, but terminates when `x` is zero. The statement `n++` increments the integer `n` by 1 during every pass through the loop, keeping track of how many times it is executed. (This statement is shorthand for `n = n + 1`.) The `printf` statement displays the values of `n` and `x`, which by construction satisfy the equation

$$\mathtt{x} = 2^{-\mathtt{n}}$$

as long as the arithmetic is done exactly. The `\n` is the *newline* character, needed so that the output is displayed one line at a time. If the arithmetic were to be done exactly, Program 2 would run forever, but in floating point, the value of the variable `x` will underflow to zero eventually. Here is the output with IEEE arithmetic:

```
(2 raised to the power -1) = 5.000000e-01
(2 raised to the power -2) = 2.500000e-01
(2 raised to the power -3) = 1.250000e-01
(2 raised to the power -4) = 6.250000e-02
(2 raised to the power -5) = 3.125000e-02

   .... 140 lines omitted ....

(2 raised to the power -146) = 1.121039e-44
(2 raised to the power -147) = 5.605194e-45
(2 raised to the power -148) = 2.802597e-45
(2 raised to the power -149) = 1.401298e-45
(2 raised to the power -150) = 0.000000e+00
```

Here is the explanation. The variable `x` is reduced from its initial value of 1 to 1/2, 1/4, 1/8, etc. After the first 126 times through the loop, `x` has been reduced to the smallest IEEE single normalized number, $N_{\min} = 2^{-126}$. If there were no subnormal numbers, `x` would underflow to zero after one more step through the loop. Instead, gradual underflow (see Chapter 7) requires that the next step reduce `x` to the subnormal number 2^{-127}, which has the representation

$$\begin{array}{|c|c|c|}\hline 0 & 00000000 & 10000000000000000000000 \\ \hline \end{array}.$$

The next step reduces `x` to 2^{-128}, with the representation

$$\begin{array}{|c|c|c|}\hline 0 & 00000000 & 01000000000000000000000 \\ \hline \end{array}.$$

This continues 21 more times, until `x` reaches 2^{-149}, with the representation

$$\begin{array}{|c|c|c|}\hline 0 & 00000000 & 00000000000000000000001 \\ \hline \end{array}.$$

After one more step, we would like `x` to have the value 2^{-150}, but this is not representable in the IEEE single format. It could be rounded up to 2^{-149} or down to 0. In either case, the absolute rounding error is 2^{-150}. The default rounding mode, round to nearest, chooses the one with the 0 final bit, namely, the number 0.

Now let us consider a different loop program:

```
main()   /* Program 3: Second Loop Program */
{
   int n = 0;
   float x = 1, y = 2;

   /* Repeatedly divide x by 2 until y = (1 + x) rounds to 1 */

   while (y > 1) {
      x = x/2;
      y = 1 + x;
      n++;
      printf("1 added to (2 raised to the power -%d) = %e \n", n, y);
   }
}
```

In Program 3, the *float* variable `x` is again initialized to 1 and repeatedly divided by 2, but this time the termination test is different: the loop continues as long as `y` is greater than 1, where `y` is set to the value of $1 + \texttt{x}$, but terminates if the value of `y` is exactly 1 (or smaller). What is the output using IEEE arithmetic?

Here is the answer:

```
1 added to (2 raised to the power -1) = 1.500000e+00
1 added to (2 raised to the power -2) = 1.250000e+00
1 added to (2 raised to the power -3) = 1.125000e+00
1 added to (2 raised to the power -4) = 1.062500e+00
1 added to (2 raised to the power -5) = 1.031250e+00

             .... 10 lines omitted ....

1 added to (2 raised to the power -16) = 1.000015e+00
1 added to (2 raised to the power -17) = 1.000008e+00
1 added to (2 raised to the power -18) = 1.000004e+00
1 added to (2 raised to the power -19) = 1.000002e+00
1 added to (2 raised to the power -20) = 1.000001e+00
1 added to (2 raised to the power -21) = 1.000000e+00
1 added to (2 raised to the power -22) = 1.000000e+00
1 added to (2 raised to the power -23) = 1.000000e+00
1 added to (2 raised to the power -24) = 1.000000e+00
```

Program 3 terminates much sooner than Program 2 does. Recall that $1 + 2^{-23}$ has the exact representation

0	01111111	00000000000000000000001

.

However, even though the number 2^{-24} can be represented exactly using the IEEE single format, the number $1 + 2^{-24}$ cannot. It could be rounded up to $1 + 2^{-23}$ or down to 1. As earlier, both choices are equally close, this time with an absolute rounding error of 2^{-24}. Again, the default rounding mode, round to nearest, chooses the answer with the zero final bit, namely, the number 1. Consequently, the loop terminates.

At first sight, it seems from the output that the loop should have terminated earlier, when `x` reached the value 2^{-21}. However, this is because the output format code `%e` used in the `printf` statement does not display enough decimal digits. Six digits after the decimal point are not quite enough. We can insist on seeing seven by changing the `printf` statement to

```
    printf("1 added to (2 raised to the power -%d) = %.7e \n", n, y);
```

in which case the last few lines of output become

```
1 added to (2 raised to the power -20) = 1.0000010e+00
1 added to (2 raised to the power -21) = 1.0000005e+00
1 added to (2 raised to the power -22) = 1.0000002e+00
1 added to (2 raised to the power -23) = 1.0000001e+00
1 added to (2 raised to the power -24) = 1.0000000e+00
```

We could have coded Program 3 without using the variable `y`, replacing it by `1 + x` in both the while loop termination test and the `printf` statement. However, when we do this, we are no longer sure that the value of `1 + x` will be rounded to the IEEE single format. Indeed, on an x87 machine the value of `1 + x` would then likely be held in an extended 80-bit register. If so, the program would run through the loop more times before it terminates. Another possibility is that a compiler might simplify `1 + x > 1` to `x > 0`, resulting in the same output as Program 2.

Exercise 10.3 *Write a C program to store the value of* $1/10$ *in a float variable and then repeatedly divide the number by* 2 *until it is subnormal. Continue dividing by* 2 *until about half the precision of the number is lost. Then reverse the process, multiplying by* 2 *the same number of times you divided by* 2*. Display the final result. How many significant digits does it have? Explain why this happened.*

Exercise 10.4 *What would happen if Programs* 2 *and* 3 *were executed using the rounding mode round up? Make a prediction and then do the experiment.*

Exercise 10.5 *Write a C program to find the smallest positive integer* x *such that the floating point expression*

$$(1 \oslash x) \otimes x$$

is not 1*, using single precision. Make sure that the variable* `x` *has type float, and assign the value of the expression* $1 \oslash x$ *to a float variable before doing the multiplication operation, to prevent the use of extended precision or an optimization by the compiler from defeating your experiment. Repeat with double precision.*

Exercise 10.6 *Write a C program to find the smallest positive integer* x *such that*

$$1 \oslash (1 \oslash x)$$

is not x*, using single precision. Repeat with double precision. (See the comments in the previous exercise.)*

Table 10.2: Parallel Resistance Results

R_1	R_2	Total resistance
1	1	5.000000e-01
1	10	9.090909e-01
1	1000	9.990010e-01
1	1.0e5	9.999900e-01
1	1.0e10	1.000000e+00
1	0.1	9.090909e-02
1	1.0e-5	9.999900e-06
1	1.0e-10	1.000000e-10
1	0	0.000000e+00

Infinity and Division by Zero

Now let us turn our attention to exceptions. Program 4 implements the parallel resistance formula (7.1):

```
main() /* Program 4: Parallel Resistance Formula */
{
   float r1, r2, total;

   printf("Enter the two resistances \n");
   scanf("%f %f", &r1, &r2);  /* input the resistances of the two
                                 resistors connected in parallel */
   printf("r1 = %e    r2 = %e \n", r1, r2);
   total = 1 / (1/r1 + 1/r2); /* formula for total resistance */
   printf("Total resistance is %e \n",total);
}
```

Table 10.2 summarizes the output of Program 4 for various input data. In the first five lines, R_1 is held fixed equal to 1 and R_2 is increased from 1 to a large number. The larger R_2 is, the more the current tends to flow through the first resistor, and so the closer the total resistance is to 1. With R_1 fixed equal to 1, and R_2 sufficiently large, i.e., $1/R_2$ sufficiently small, the floating point sum of $1/R_2$ and $1/R_1$ is precisely 1 even though the exact result would be strictly greater than 1 for all finite nonnegative values of R_2. Thus, the final result (the inverse of the sum) is precisely 1, even though the mathematical result is strictly less than 1. This is, of course, because of the limited precision of an IEEE single format floating point number.

The last four lines of the table show what happens when R_1 is fixed equal to 1 and R_2 is decreased below 1. The smaller R_2 is, the more the current tends to flow through the second resistor.

The last line of the table shows that the output zero is obtained when $R_2 = 0$. As explained in Chapter 7, this is a sensible mathematical result, and will be obtained using IEEE default exception handling, since $1/0$ evaluates to ∞, $1 + \infty$ evaluates to ∞, and $1/\infty$ evaluates to 0. Furthermore, as explained in Chapter 9, this result will be obtained using any compiler following the C99 standard, which is essentially every modern compiler.

Exercise 10.7 *If $R_1 = 1$, for approximately what range of values for R_2 (what powers*

Table 10.3: Some Standard C Math Library Functions

fabs	absolute value: `fabs(x)` returns $\lvert x\rvert$
sqrt	square root: `sqrt(x)` returns $\sqrt{x}$
exp	exponential (base e): `exp(x)` returns e^x
log	logarithm (base e): `log(x)` returns $\log_e(x)$
log10	logarithm (base 10): `log10(x)` returns $\log_{10}(x)$
sin	sine (argument given in radians)
cos	cosine (argument given in radians)
atan	arctangent (result in radians between $-\pi/2$ and $\pi/2$)
pow	power (two arguments: `pow(x,y)` returns x^y)

of 10*) does Program* 4 *give a result exactly equal to* 1*? Try to work out the answer before you run the program.*

Exercise 10.8 *How does the answer to Exercise* 10.7 *change if* $R_1 = 1000$*: approximately what range of values for* R_2 *give a result exactly equal to* 1000*? Explain your reasoning.*

Exercise 10.9 *How does the answer to Exercise* 10.7 *change if the rounding mode is changed to round up?*

The Math Library

Very often, numerical programs need to evaluate standard mathematical functions. The *exponential* function is defined by

$$\exp(x) = e^x = \lim_{n\to\infty}\left(1+\frac{x}{n}\right)^n. \tag{10.1}$$

It maps the extended real line (including $\pm\infty$) to the nonnegative extended real numbers, with the convention that $\exp(-\infty) = 0$ and $\exp(\infty) = \infty$. Its inverse is the *logarithm* function (base e), denoted $\log(x)$, satisfying

$$\exp(x) = y \quad \text{if and only if} \quad x = \log(y).$$

The function $\log(y)$ is defined for $y \geq 0$ with the conventions that $\log(0) = -\infty$ and $\log(\infty) = \infty$. These functions, along with many others such as the trigonometric functions sine, cosine, etc., are provided by all C compilers as part of an associated *math library.* Some of them are listed in Table 10.3. They all expect *double* arguments and return *double* values, but they can be called with *float* arguments and their values assigned to *float* variables; the conversions are done automatically. The C99 standard also calls for the support of variants such as `expf` and `expl`, which expect and return *float* and *long double* types. The C99 standard also calls for support for other mathematical functions not traditionally provided, including `fma`, the fused multiply-add operation (with a single rounding error) described in Chapter 6. For a complete list of math library functions reqired by C99 and the subset required by its predecessor, the C89 standard, see [Bee17, Table 3.4].

Neither the IEEE standard nor the C99 standard prescribes the accuracy of the math library functions (with the exception of the square root function). However, the 2008 version of the IEEE standard does make extensive recommendations for many functions to provide correctly rounded results for all rounding modes, including all the functions listed in Table 10.3; for more, see [IEE08, Table 9.1]. One of the motivations for the provision of extended precision by the IEEE standard was to allow fast accurate computation of the library functions [Hou81], but as already noted, extended precision hardware is largely limited to the older x87 microprocessors. There is a fundamental difficulty in computing correctly rounded values of the exponential, logarithmic, and trigonometric functions, called the Tablemaker's Dilemma: one might have to compute an arbitrarily large number of correct digits before one would know whether to round the result up or down. Nonetheless, clever algorithms can achieve correctly rounded results by using sufficiently high precision when necessary [Mul97], [MB+18, Ch. 10], [Bee17, Ch.10–11]. This is an active topic of current research: very fast implementations are becoming available, such as in the ongoing CORE-MATH project.[2] However, at present, these algorithms are not routinely used by the many different C math libraries provided by compilers and microprocessors, which are generally *not* guaranteed to produce correctly results [InnZim23].

It is important to note that when we say a function f returns correctly rounded results, this means it returns the correctly rounded value of $f(x)$, where the input x is a floating point number. It does not mean that it will return the correctly rounded value of, say, $f(y)$, where y is *not* a floating point number. The reason is that then f will have to be evaluated at $\text{round}(y)$, the correctly rounded value of y, producing the correctly rounded value of $f(\text{round}(y))$, which may not be nearly the same as $f(y)$. This phenomenon is discussed in Chapter 12.

C99 and the 2008 IEEE standard call for the math library functions to return infinite or NaN values when appropriate. For example, log returns $-\infty$ when its argument is ± 0 and NaN when its argument is negative. The divide-by-zero exception is signaled when the argument is ± 0, because $-\infty$ is considered to be the exact result; the invalid operation exception is signaled when the argument is negative. A complete set of special function values prescribed by the 2008 standard appears in [IEE08, Table 9.1].

When a C program contains calls to the math library, it should have the following line at the start of the program:

```
#include <math.h>
```

The "include file" `math.h` tells the C compiler about the return types and other calling-sequence details of the functions provided in the math library. Also, when the program is compiled, it is necessary to *link* the math library; the syntax for this is system-dependent. If an error message says that the math functions cannot be found, the math library is not properly linked.

The *mathcw* library designed and implemented by Beebe [Bee17] is an extensive, portable math library with many nice features. The book that documents it has a wealth of information about floating point computing in C as well as in many other languages including Fortran, C++, C# and Java. The name *mathcw* is inspired by the pioneering work of Cody and Waite [CW80].[3]

Exercise 10.10 *Check to see what your C math library returns for* $\log(\pm 0)$, $\log(\pm 1)$, *and* $\log(\pm\infty)$. *Are these results what you expected?*

Exercise 10.11 *The domain of a function is the set of values on which it is defined, and its range is the set of all possible values that it returns.*

[2]https://core-math.gitlabpages.inria.fr.

[3]Beebe's dedication in [Bee17] is to the three Williams: Cody, Kahan and Waite. He wrote: "They taught us that floating point arithmetic is interesting, intricate, and worth doing right, and showed us how to do it better.

1. *What are the ranges of the* sin *and* cos *functions?*
2. *What values should* $\sin(\infty)$ *and* $\cos(\infty)$ *return? Run a program to see if your C math library does what you expect.*
3. *The arcsine and arccosine functions,* asin *and* acos*, are, respectively, the inverses of the* sin *and* cos *functions on a restricted domain. By experimenting, determine these restricted domains, i.e., the ranges of the* asin *and* acos *functions.*
4. *What values should* $\text{asin}(x)$ *and* $\text{acos}(x)$ *return if x is outside the range of the* sin *and* cos *functions? Run a program to test your answer.*

Exercise 10.12 *What is* $\text{pow}(x, 0)$ *when x is nonzero? What is* $\text{pow}(0, x)$ *when x is nonzero? What is* $\text{pow}(0, 0)$*? What is the mathematical justification for these conventions? (See [Gol91], as well as [IEE19, p. 63] for more special values prescribed for x^y.)*

Exercise 10.13 *It's unlikely that HAL, quoted on p.* vi*, was IEEE compliant, since the movie 2001: A Space Odyssey predated the development of the standard by nearly two decades [Cla99]. What precision does HAL appear to be using? Does he report correctly rounded results for the square root and log functions before Dave starts dismantling his circuits? In order to answer this with certainty, you may need to use the long double (extended precision) versions of the math library functions, namely, sqrtl, expl, and log10l.*

Exercise 10.14 *As mentioned in Chapter* 6*, interval arithmetic means floating point computing where, for each variable, lower and upper bounds on the exact value of the variable are maintained. This can be implemented using the round down and round up modes, assuming these are supported by your C compiler. Write a program to read a sequence of positive numbers and add them together in a sum. Include an option to set the rounding mode to any of round down, round up, and round to nearest. The round down mode should give you a lower bound on the exact result, the round up mode should give you an upper bound, and round to nearest should give you an intermediate result. Use type float but print the results to double precision so you can see the rounding effects clearly. Avoid using very simple input values (such as integers) that may not need rounding at all. Describe the results that you obtain. To how many digits do the three answers agree? The answer will depend on your data.*

Exercise 10.15 *Assuming the rounding modes are supported by your C compiler, use the ideas of interval arithmetic (see previous exercise) to compute upper and lower bounds on the quantity*

$$\frac{a+b}{c+d},$$

where a, b, c, d are input values. Use float variables but print the results to double precision so you can see the effect of rounding. Think about how to do this carefully. Do you have to change the rounding mode dynamically, i.e., during the computation? Be sure to try a variety of input data to fully test the program. Avoid using very simple input values (such as integers) that may not need rounding at all.

Exercise 10.16 *Avoiding overflow in a product (J. Demmel).*

1. *Write a C program to read a sequence of positive numbers and compute the product. Assume that the input numbers do not overflow the IEEE single format. The program should have the following properties:*

- *The variables in the program should have type either float or int. Double or extended precision variables are not permitted.*
- *The program should print the product of the numbers in the following non-standard format: a floating point number F (in standard decimal exponential format), followed by the string*

  `times 10 to the power`,

  followed by an integer K.
- *The result should not overflow, i.e., the result should not be ∞, even if the final value, or an intermediate value generated along the way, is bigger than $N_{\max}$, the biggest IEEE single format floating point number.*
- *The program should be reasonably efficient, doing no unnecessary computation (except for comparisons) when none of the intermediate or final values are greater than $N_{\max}$. In this case, the integer K displayed should be zero.*

The way to accomplish these goals is as follows. Suppose the input consists of two numbers, both `1.0e30`*, so that the product is too big to fit in the IEEE single floating point format. With default exception handling, the result is ∞, and by observing this result the program can divide one of the numbers by a power of 10 and try again. By choosing the power of* 10 *correctly (using a loop), the product can be made small enough not to overflow the IEEE single format. In this way, a final product is computed that requires final scaling by a certain power of* 10*: this is the integer that should be output, and you can assume this is not bigger than the biggest integer that can be stored.*

An important part of the assignment is to choose a good test set to properly check the program.

Note: When you multiply two numbers together and compare the result to ∞, you might not get the answer you expect unless you first store the product of the numbers in a float variable (since the registers may use the double or extended format).

2. *Extend the program so that the result does not underflow to zero regardless of how small the intermediate and final values are.*

3. *Should you also avoid underflow to subnormal intermediate and final values? Why or why not? How can this be done?*

Chapter 11

Cancellation

Consider the two numbers

$$x = 3.141592653589793$$

and

$$y = 3.141592653585682.$$

The first number, x, is a 16-digit approximation to π, while the second number agrees with π to only 12 digits. Their difference is

$$z = x - y = 0.000000000004111 = 4.111 \times 10^{-12}. \qquad (11.1)$$

This difference is in the normalized range of the IEEE single format. However, if we compute the difference $z = x - y$ in a C program, using the single format to store the variables x and y before doing the subtraction, and display the result to single precision, we find that the result displayed for z is

```
0.000000e+00.                                     (11.2)
```

The reason is simple enough. The input numbers x and y are first converted from decimal to the single binary format; they are not exact floating point numbers, so the decimal to binary conversion requires some rounding. Because they agree to 12 digits, both x and y round to exactly the same single format number. Thus, all bits in their binary representation cancel when the subtraction is done; we say that we have a complete loss of significance in the computation $z = x - y$.

If we use the double format to store x and y and their difference z, and if we display the result to double precision, we find that z has the value

```
4.110933815582030e-12.                            (11.3)
```

This agrees with the exact answer (11.1) to about four digits, but what is the meaning of the other digits? The answer is that the result displayed is the correctly rounded difference of the double format representations of x and y. Although we might prefer to see (11.1), this will not happen on a binary machine, as it would on a decimal calculator with enough digits. It is important to realize that in this case, we may ignore all but the first four or five digits of (11.3). The rest may be viewed as garbage, in the sense that they do not reflect the original data x and y. We say that we have a partial loss of significance in the computation $z = x - y$.

Regardless of whether the loss of significance is complete or partial, the phenomenon is called *cancellation*. It occurs when one number is subtracted from another number that is nearly equal to it. Equivalently, it occurs if two numbers with opposite sign but nearly equal magnitude are added together.

Approximating a Derivative by a Difference Quotient

An excellent illustration of cancellation is provided by the example of computing an approximation to a derivative. Let f be a differentiable function of a single real variable. . Suppose that we do not have a formula for f, but only a program that evaluates $f(x)$ for any given value x. How would we estimate the value of $f'(x)$, the derivative of f at x?

By definition, $f'(x)$ is the slope of the line tangent to the graph of f at $(x, f(x))$, i.e., the limit of the difference quotient

$$\frac{f(x+h)-f(x)}{h} \tag{11.4}$$

as h converges to zero. This difference quotient is the slope of the line passing through the points $(x+h, f(x+h))$ and $(x, f(x))$. A natural idea, then, is to evaluate (11.4) for some "small" value of h—but how small? Setting h to zero will give us 0/0, i.e., NaN. Program 5 tries values of h ranging from 10^{-1} down to 10^{-20}, assuming that $x = 1$ and f is the sine function. Since we know that the derivative of $\sin(x)$ is $\cos(x)$, we can evaluate this at $x = 1$ and compare the result to the difference quotient. The absolute value of the difference between the two is called the *error*. The program uses type *double*.

```
#include <math.h>
main()  /* Program 5: Approximate a Derivative by a
                      Difference Quotient*/
{
   int n = 1;
   double x = 1.0, h = 1.0, deriv = cos(x), diffquo, error;

   printf(" deriv =%13.6e \n", deriv);
   printf("   h      diffquo  abs(deriv - diffquo) \n");

   /* Let h range from 10^{-1} down to 10^{-20} */

   while(n <= 20) {
      h = h/10;                        /* h = 10^(-n) */
      diffquo = (sin(x+h)-sin(x))/h;  /* difference quotient */
      error = fabs(deriv - diffquo);
      printf("%5.1e %13.6e %13.6e \n", h, diffquo, error);
      n++;
   }
}
```

Here is the output:

```
 deriv = 5.403023e-01
   h      diffquo  abs(deriv - diffquo)
1.0e-01  4.973638e-01  4.293855e-02
1.0e-02  5.360860e-01  4.216325e-03
1.0e-03  5.398815e-01  4.208255e-04
1.0e-04  5.402602e-01  4.207445e-05
1.0e-05  5.402981e-01  4.207362e-06
1.0e-06  5.403019e-01  4.207468e-07
```

```
1.0e-07  5.403023e-01  4.182769e-08
1.0e-08  5.403023e-01  2.969885e-09
1.0e-09  5.403024e-01  5.254127e-08
1.0e-10  5.403022e-01  5.848104e-08
1.0e-11  5.403011e-01  1.168704e-06
1.0e-12  5.403455e-01  4.324022e-05
1.0e-13  5.395684e-01  7.339159e-04
1.0e-14  5.440093e-01  3.706976e-03
1.0e-15  5.551115e-01  1.480921e-02
1.0e-16  0.000000e+00  5.403023e-01
1.0e-17  0.000000e+00  5.403023e-01
1.0e-18  0.000000e+00  5.403023e-01
1.0e-19  0.000000e+00  5.403023e-01
1.0e-20  0.000000e+00  5.403023e-01
```

The error (the absolute value of `deriv - diffquo`) is plotted as a function of h in Figure 11.1, using a log–log scale. The results are quite interesting. Reading the graph from right (large h) to left (small h), we see that the approximation gets better, i.e., the error gets smaller, as h gets smaller—as we might expect—but only up to a certain point. When h gets *too* small, the approximation starts to get *worse!* Why?

After a little thought, the reason is clear. If $x = 1$ and h is smaller than half of machine epsilon (about 10^{-16} in the double format), then $x+h$, i.e., $1+h$, is rounded to 1, and so naturally the formula that is being displayed gives the result zero, since the values $\sin(x+h)$ and $\sin(x)$ completely cancel. In other words, the final answer has *no significant digits*. When h is a *little* bigger than machine epsilon, the values do not completely cancel but they still *partially cancel*. For example, suppose that the first 10 digits of $\sin(x+h)$ and $\sin(x)$ are the same. Then, even though $\sin(x+h)$ and $\sin(x)$ both have about 16 significant digits, the difference has only about 6 significant digits. Since the difference is stored as a normalized double format number, it appears at first to have 16 significant digits, but only the first 6 are meaningful. We may summarize the situation by saying that although using very small values of h reduces the *discretization error* inherent in approximating the derivative $f'(x)$ by the difference quotient (11.4), this is dominated by the much larger *cancellation error* inherent in computing (11.4). On the other hand, for large h, the cancellation error is small, but it is dominated by the large discretization error. For the function $f(x) = \sin(x)$ at $x = 1$, the best choice of h is about 10^{-8}, approximately *the square root of machine epsilon.*

A closer look at the output shows that, for the larger values of h, the error drops by approximately a factor of 10 every time h is reduced by a factor of 10—until the cancellation error starts to take over. There is a reason for this. To explain it, we assume that f is twice differentiable, as is the case for the sine function. Then there exists z between x and $x+h$ such that

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(z), \tag{11.5}$$

where $f''(z)$ is the second derivative of f at z. Formula (11.5) is called a truncated

Figure not available at present, see figure in first edition

Figure 11.1: Error (Absolute Value of Derivative Minus Difference Quotient) as a Function of h (Log–Log Scale)

Taylor series. Therefore,

$$\frac{f(x+h)-f(x)}{h} - f'(x) = \frac{h}{2}f''(z). \tag{11.6}$$

This quantity is the difference between what we are computing, the difference quotient, and what we want, the exact derivative. Its absolute value is the *discretization error*. Equation (11.6) shows that if h is reduced by a factor of 10, the discretization error also decreases by a factor of about 10 (not exactly, since the point z between x and $x+h$ changes when h changes). Thus, we say the discretization error is $O(h)$. This explains the factors of 10 observed in the table (and the corresponding straight line of data on the right side of Figure 11.1).

The lesson to be learned here is that *cancellation*, which occurs when subtraction of nearly equal values takes place, should be avoided when possible. Using the formula for the derivative of a function is much more accurate than approximating it by difference quotients.

The Central Difference Quotient

As long as f is smooth enough, we can construct a more accurate approximation to the derivative of f at x by computing the slope of the line passing through $(x+h, f(x+h))$ and $(x-h, f(x-h))$, i.e.,

$$\frac{f(x+h)-f(x-h)}{2h}.$$

This is called the *central difference quotient.* Assume that the third derivative f''' exists. For small enough h (but large enough that cancellation is not a problem), the central difference quotient gives a more accurate approximation to the derivative $f'(x)$ than the difference quotient (11.4). Here is the explanation. Truncated Taylor series give us

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{6}f'''(z_1) \tag{11.7}$$

for some z_1 between x and $x+h$ and

$$f(x-h) = f(x) - hf'(x) + \frac{h^2}{2}f''(x) - \frac{h^3}{6}f'''(z_2) \tag{11.8}$$

for some z_2 between x and $x-h$. Subtracting (11.8) from (11.7) and dividing through by $2h$, we get

$$\frac{f(x+h)-f(x-h)}{2h} - f'(x) = \frac{h^2}{12}(f'''(z_1) + f'''(z_2)).$$

This gives the discretization error for the central difference quotient. Thus, the discretization error for the central difference quotient is $O(h^2)$ instead of $O(h)$.

Exercise 11.1 *Change Program* 5 *to use centered differences, and observe that when* h *is reduced by a factor of* 10*, the discretization error is reduced by a factor of about* 100*, confirming the* $O(h^2)$ *behavior. But, as before, when* h *becomes too small, the cancellation error dominates and the results become useless. Approximately what* h *gives the best results?*

Exercise 11.2 *Using a spreadsheet program such as Excel, implement the finite difference formula in a spreadsheet. Can you tell from the results what precision the spreadsheet program is using? If necessary, use the menu options to change the output format used.*

It's important to note that the problem with cancellation has nothing to do with the subtraction operation somehow being inaccurate. Like the other arithmetic operations, it always produces correctly rounded results in IEEE arithmetic, and in fact the subtraction of two nearly equal numbers gives the exact result, thanks to the availability of subnormal numbers. The issue is that if x and y are nearly equal, many or even most of the significant digits in the numbers x and y are lost when the difference $x - y$ is computed. For more on cancellation, see a nice discussion in [Hig02, Sec. 1.7].

Chapter 12

Conditioning of Problems

Suppose we wish to solve some problem using numerical computing. Roughly speaking, the conditioning of the problem measures how accurately one can expect to be able to solve it using a given floating point precision, independently of the algorithm used. We confine our discussion to the problem of evaluating a real function of a real variable,

$$y = f(x),$$

assuming that f is differentiable and that x and $f(x)$ are in the normalized range of the floating point precision. Define

$$\widehat{x} = \text{round}(x).$$

Evaluating the function f on the computer using floating point arithmetic, the *best* we can hope for is to compute the value

$$\widehat{y} = f(\widehat{x}).$$

In fact, even this is an unreasonable hope because we will not be able to evaluate f exactly, but for simplicity, let us suppose for now that we can. Now, we know from (5.12) in Chapter 5 that the relative rounding error satisfies the bound

$$\frac{|\widehat{x} - x|}{|x|} < \epsilon_{\text{mch}},$$

where ϵ_{mch} is machine epsilon (with an additional factor of 1/2 if the rounding mode is *round to nearest*). It follows that

$$-\log_{10}\left(\frac{|\widehat{x} - x|}{|x|}\right) > -\log_{10}(\epsilon_{\text{mch}}). \tag{12.1}$$

As noted in the discussion following the equivalent inequality (5.14), the left-hand side of (12.1) approximates the number of decimal digits to which $\widehat{x}$ agrees with x—at least about seven digits in the case of IEEE single precision. The question we now ask is: To how many digits can we expect $\widehat{y}$ to agree with y? To find out, we must look at the quantity

$$-\log_{10}\left(\frac{|\widehat{y} - y|}{|y|}\right).$$

We have

$$\frac{\widehat{y} - y}{y} = \frac{f(\widehat{x}) - f(x)}{\widehat{x} - x} \times \frac{x}{f(x)} \times \frac{\widehat{x} - x}{x}. \tag{12.2}$$

The first factor,

$$\frac{f(\widehat{x}) - f(x)}{\widehat{x} - x}, \tag{12.3}$$

approximates $f'(x)$, the derivative of f at x. Therefore,

$$\frac{|\widehat{y} - y|}{|y|} \approx \kappa_f(x) \times \frac{|\widehat{x} - x|}{|x|}, \tag{12.4}$$

where

$$\kappa_f(x) = \frac{|x| \times |f'(x)|}{|f(x)|}. \tag{12.5}$$

The quantity $\kappa_f(x)$ is called the *condition number of f at x*. It measures *approximately how much the relative rounding error in x is magnified by evaluation of f at x*. (For a more rigorous derivation that eliminates the approximation symbol $\approx$, see Exercise 12.8.)

Now we are in a position to answer the question: To how many digits can we expect $\widehat{y}$ to agree with y? The left-hand side of (12.4) is a relative measure of how well $\widehat{y}$ approximates y. The second factor on the right-hand side of (12.4) is a relative measure of how well $\widehat{x}$ approximates x. Taking logarithms (base 10) on both sides, we get

$$-\log_{10}\left(\frac{|\widehat{y} - y|}{|y|}\right) \approx -\log_{10}\left(\frac{|\widehat{x} - x|}{|x|}\right) - \log_{10}\left(\kappa_f(x)\right). \tag{12.6}$$

Here the left-hand side is approximately the number of digits to which $\widehat{y}$ agrees with y, and the first term on the right-hand side is approximately the number of digits to which $\widehat{x}$ agrees with x, which we know from (12.1) and Table **??** is at least about seven when IEEE single precision is in use. Consequently, we conclude with a rule of thumb.[1]

Rule of Thumb 12.1 *To estimate the number of digits to which $\widehat{y} = f(\widehat{x})$ agrees with $y = f(x)$, subtract*

$$\log_{10}\left(\kappa_f(x)\right)$$

*from the approximate number of digits to which $\widehat{x} = \mathrm{round}(x)$ agrees with x, i.e., 7 when using IEEE single precision or 16 when using IEEE double precision (see Table **??**). Here $\kappa_f(x)$ is the condition number of f at x, defined in (12.5), and we assume that f is differentiable and that x and $f(x)$ are in the normalized range of the floating point system.*

Since evaluating the condition number of f at x requires first evaluating f at x as well as the derivative $f'(x)$, the condition number does *not* help us solve our original problem, the evaluation of f. On the contrary, evaluating the condition number is harder than evaluating the function. However, the condition number *does* give us insight into difficulties that may arise when we evaluate f at certain values of x.

Exercise 12.1 *Determine the condition numbers of the functions*

$$g(x) = \frac{x}{10}$$

and

$$h(x) = x - 10,$$

and discuss what values of x, if any, give large condition numbers $\kappa_g(x)$ or $\kappa_h(x)$.

Table 12.1: Sample Condition Numbers

f	x	$f(x)$	$f'(x)$	$\kappa_f(x)$	$\log_{10}(\kappa_f(x))$
exp	1	e	e	1	0
exp	0	1	1	0	$-\infty$
exp	-1	$1/e$	$1/e$	1	0
log	e	1	$1/e$	1	0
log	1	0	1	∞	∞
log	$1/e$	-1	e	1	0
sin	π	0	-1	∞	∞
sin	$\pi/2$	1	0	0	$-\infty$
sin	0	0	1	NaN	NaN

Figure not available at present, see figure in first edition

Figure 12.1: Exponential, Logarithmic, and Sine Functions

Table 12.1 tabulates the function, derivative, and condition number of three functions, the exponential, logarithmic, and sine functions, each at three different values of x. These values are all exact. The three functions and the relevant points $(x, f(x))$ are shown in Figure 12.1. The derivatives of $\exp(x)$, $\log(x)$, and $\sin(x)$ are

$$\exp(x), \quad \frac{1}{x}, \quad \cos(x),$$

respectively, so the condition numbers are

$$\kappa_{\exp}(x) = |x|, \quad \kappa_{\log}(x) = \frac{1}{|\log(x)|}, \quad \kappa_{\sin}(x) = \frac{|x|}{|\tan(x)|}. \tag{12.7}$$

Altogether, Table 12.1 gives nine examples. Four of these examples have condition numbers equal to 1. In these examples, we expect that if $\widehat{x}$ approximates x to about seven significant digits, then $\widehat{y} = f(\widehat{x})$ should approximate $y = f(x)$ to about seven digits. In two examples, we see a condition number equal to 0. The problem of evaluating $f(x)$ in these examples is said to be *infinitely well conditioned*, and we expect that $\widehat{y}$ should approximate y to *many more* digits than $\widehat{x}$ approximates x. On the other hand, two other examples have condition numbers equal to ∞. The problem of evaluating $f(x)$ in these cases is said to be *infinitely ill conditioned*, and we expect that $\widehat{y}$ should approximate y to *many fewer* digits than $\widehat{x}$ approximates x. Finally, there is one case in which the condition number is not defined because both x and $f(x)$ are zero. This is indicated by the NaN in the table. However, see Exercise 12.3.

Exercise 12.2 *Construct a table like Table* 12.1 *to display the condition number of the functions* log10*,* cos*, and* asin*, using points where they can be computed exactly, if possible.*

Exercise 12.3 *The condition number of the sine function is not defined at $x = 0$, but the limit of the condition number is defined as $x \to 0$. What is this limit? Does* (12.4) *still hold if we define this limit to be the condition number?*

[1]A rule of thumb is any method of measuring that is practical though not precise [Web96].

Checking the Rule of Thumb

We used Program 6 to evaluate the math library functions exp, log, and sin near the points x shown in Table 12.1. The values e, $1/e$, π, and $\pi/2$ were input to *double* accuracy and stored in the *double* variable `xD`. The values 1, 0, and -1 would have been stored exactly, so instead we input numbers *near* these values. For each input value, Program 6 computes the relevant function value in two ways. The output is summarized in Table 12.2. The fourth column displays the *double* results computed by the function when its argument is the *double* variable `xD`. The third column displays the *double* results computed by the function when its argument is *rounded to single precision before being passed to the function.* In both cases, the function evaluation uses double precision and the value is displayed to 16 digits. The only difference is the precision of the *argument passed to the function.*

```
#include <math.h>
main ()     /*  Program 6: Function Evaluation */
{
   int funcode;                        /* function code */
   float xS;                           /* IEEE single */
   double xD, fxS, fxD, relerr, cond;  /* IEEE double */

   printf("enter 1 for exp, 2 for log, 3 for sin ");
   scanf("%d", &funcode);
   printf("enter an input value \n");
   scanf("%lf", &xD);       /* read using double format */
   xS = xD;                 /* force single format */
   switch (funcode) {
      case 1: fxS = exp(xS); fxD = exp(xD);
                             cond = fabs(xD); break;
      case 2: fxS = log(xS); fxD = log(xD);
                             cond = fabs(1/log(xD)); break;
      case 3: fxS = sin(xS); fxD = sin(xD);
                             cond = fabs(xD/tan(xD)); break;
   }
   printf("funcode %d\n", funcode);
   printf("xS = %22.15e   f(xS) = %22.15e \n", xS, fxS);
   printf("xD = %22.15e   f(xD) = %22.15e \n", xD, fxD);
   /* relative error */
   relerr = fabs((fxD - fxS)/fxD);
   /* approximate number of digits they are in agreement */
   printf("relative error   = %e   ", relerr);
   printf("approx digits agreeing = %2.0f\n", -log10(relerr));
   /* log base 10 of condition number */
   printf("condition number = %e   ", cond);
   printf("log10 condition number = %2.0f\n", log10(cond));
}
```

We may view the numbers in the third column of Table 12.2 as reasonably accurate evaluations of $\widehat{y}$, the value of the function f at its rounded-to-single argument $\widehat{x}$. We may view the numbers in the fourth column as reasonably accurate evaluations of the exact value $y = f(x)$—at least, the rounded-to-double argument `xD` is much closer to the input value x than the rounded-to-single argument `xS`. The column headed Agree

Table 12.2: Actual Function Evaluations: Exp, Log, and Sin

f	x	$f(x)$, x rounded to single	$f(x)$, x rounded to double	Agree	Loss
exp	`1.000001`	`2.718284420815846e+00`	`2.718284546742233e+00`	7	0
exp	`0.000001`	`1.000001000000498e+00`	`1.000001000000500e+00`	15	-6
exp	`-1.000001`	`3.678790903344350e-01`	`3.678790732921851e-01`	7	0
log	e (double)	`9.999999696321400e-01`	`1.000000000000000e+00`	8	0
log	`1.001`	`9.995470164405893e-04`	`9.995003330834232e-04`	4	3
log	`1.000001`	`9.536738616591883e-07`	`9.999994999180668e-07`	1	6
log	$1/e$ (double)	`-9.999999751283870e-01`	`-1.000000000000000e+00`	8	0
sin	π (double)	`-8.742278000372475e-08`	`1.224646799147353e-16`	-9	16
sin	$\pi/2$ (double)	`9.999999999999990e-01`	`1.000000000000000e+00`	15	-15
sin	`0.000001`	`9.999999974750761e-07`	`9.999999999998333e-07`	9	0

in Table 12.1 estimates the number of digits to which these two computations agree, using the formula on the left-hand side of (12.6), rounded to the nearest integer. The column headed Loss shows the log (base 10) of the condition number of the function at `xD`, using (12.7) and rounding to the nearest integer. According to Rule of Thumb 12.1, the number shown in the column headed Agree should be approximately 7 minus the number shown in the column headed Loss. This is exactly the case for several of the input values, e.g., the log function at $x = 1.001$ (where the condition number is about 1000) and $x = 1.000001$ (where it is about 10^6). These are two of the ill-conditioned examples. In the extremely well-conditioned cases, the exp function at $x = 10^{-6}$ (condition number 10^{-6}) and the sine function very near $\pi/2$ (condition number 10^{-15}), we have, as expected, more agreement in the function values than there is in their arguments; however, this is limited to about 15 digits, about the most we can expect using double precision computations. In the case of the sine function near π, which is the worst conditioned example of all (condition number 10^{16}), the displayed agreement of -9 digits is a consequence of the division by `fxD` in the computation of the log (base 10) of the relative error; if we divided by `fxS` instead, we would be informed that we have 0 digits of agreement.[2]

Exercise 12.4 *As mentioned in Chapter* 4*, deciding to how many digits two numbers agree is problematic. Devise a rule that you think is reasonable and test it on the numbers in Table* 12.2*. How do your answers compare with the log (base* 10*) of the relative error reported in the column headed Agree?*

Exercise 12.5 *Modify Program* 6 *to evaluate the functions whose condition numbers were displayed in Exercise* 12.2*, and display the results in a table like Table* 12.2*. Don't forget to use the correct formulas for the condition numbers defining the output for the Loss column. Do the results support Rule of Thumb* 12.1*?*

Exercise 12.6 *Determine the condition number of the parallel resistance formula with variable R_1 and fixed $R_2 = 1$, i.e., the condition number of*

$$f(x) = \frac{1}{\frac{1}{x} + 1}.$$

[2]The reason the value of the sine function at π reported in columns 3 and 4 is not exactly zero is that the input to `sin` is not exactly π, but the single or double precision approximation to π, respectively. There is another function `sinPi` which, given x, returns the correctly rounded value of $\sin(\pi \times x)$ using a different method, so that if $x = 1$, `sinPi` returns exactly 0 [Bee17, Sec 11.7]. However, even though `sinPi` returns the exact answer at $x = 1$, the condition number of `sinPi` is nonetheless ∞ at $x = 1$.

Exercise 12.7 *Suppose that g is a function that is twice differentiable. Determine the condition number of the derivative*

$$f(x) = g'(x)$$

and the condition number of the difference quotient

$$f_h(x) = \frac{g(x+h) - g(x)}{h}$$

for fixed $h > 0$. Does the latter converge to the former as h converges to zero?

Exercise 12.8 *Replace* (12.4) *by a more precise statement that does not use the $\approx$ symbol, using the truncated Taylor series* (11.5) *together with the assumption that f is twice differentiable.*

The notion of conditioning extends far beyond simple function evaluation to more complicated and challenging problems. In other settings, the definition of condition number is more subtle than (12.5). For example, suppose the problem to be solved is to compute $y = Ax$, where A is a matrix and x is a vector, or to solve a system of linear equations $Ay = x$ for y, where A is a square nonsingular matrix and x is a vector. If we take A to be fixed, we need to know how relative errors in the solution vector y depend on relative errors in the data vector x, where we must introduce the notion of a *norm* to quantify the magnitude of a vector. The condition number is then defined to measure the *worst case* of such dependence over all data vectors x with fixed norm; see [Dem97], [Hig02], or [TB97]. In the case of simple function evaluation discussed in this chapter, where x is a scalar, not a vector, this crucial worst case aspect of the condition number is not present.

Chapter 13

Stability of Algorithms

An algorithm is a well-defined computational method to solve a given class of problems. In computer science, the study of algorithms is traditionally concerned with efficiency; it is understood that an algorithm is supposed to get the correct answer, though proving that this will happen is not necessarily easy. However, numerical algorithms, which solve problems using floating point arithmetic, almost never find the exact solution to a problem. Instead, the goal is "approximately correct" answers. These are by no means guaranteed. Although each individual floating point operation is correctly rounded, a poor choice of algorithm may introduce unnecessarily large rounding errors.

We saw in the previous chapter that the conditioning of a problem measures how accurately one can expect to be able to solve it using a given floating point precision, independently of the algorithm used. The stability of an algorithm measures how good a job the algorithm does at solving problems to the achievable accuracy defined by their conditioning. For whatever problem one might want to solve, some algorithms are better than others. Those algorithms that get *unnecessarily inaccurate* answers are called *unstable*.

We continue to confine our attention to the problem of evaluating a real function of a real variable,

$$y = f(x),$$

assuming that f is differentiable and that x and $f(x)$ are in the normalized range of the floating point precision. As earlier, define

$$\widehat{x} = \text{round}(x).$$

We commented in the previous chapter that, using floating point arithmetic, the *best* we can hope for is to compute the value

$$\widehat{y} = f(\widehat{x}), \tag{13.1}$$

and we showed that

$$\frac{|\widehat{y} - y|}{|y|} \approx \kappa_f(x) \frac{|\widehat{x} - x|}{|x|},$$

where $\kappa_f(x)$ is the condition number of f at x. However, it is generally too much to expect an algorithm to find $\widehat{y}$ satisfying (13.1), and in fact this is often impossible since $f(\widehat{x})$ may not be a floating point number. So, we say that an algorithm to compute $f(x)$ is *stable* if instead of $\widehat{y}$ it returns $\widetilde{y}$ satisfying

$$\frac{|\widetilde{y} - y|}{|y|} \approx \kappa_f(x) \frac{|\widehat{x} - x|}{|x|}, \tag{13.2}$$

where, as earlier, we deliberately avoid a specific definition for the symbol $\approx$, meaning approximately equal.[1] For rigorous definitions of stability, see [TB97, Section III] and [Hig02, Chapter 1]. If an algorithm to compute $f(x)$ delivers $\widetilde{y}$ for which the left-hand side of (13.2) is much greater than the right-hand side, we say the algorithm is *unstable*.

Compound Interest

We illustrate these concepts by considering algorithms for computing compound interest. Suppose we invest a_0 dollars in a bank that pays 5% interest per year, compounded quarterly. This means that at the end of the first quarter of the year, the value of our investment is

$$a_1 = a_0 \times (1 + (.05)/4)$$

dollars, i.e., the original amount plus one quarter of 5% of the original amount. At the end of the second quarter, the bank pays interest not only on the original amount a_0, but also on the interest earned in the first quarter; thus, the value of the investment at the end of the second quarter is

$$a_2 = a_1 \times (1 + (.05)/4) = a_0 \times (1 + (.05)/4)^2$$

dollars. At the end of the third quarter the bank pays interest on this amount, so that the investment is now worth

$$a_3 = a_2 \times (1 + (.05)/4) = a_0 \times (1 + (.05)/4)^3,$$

and at the end of the whole year the bank pays the last installment of interest on the amount a_3, so that the investment is finally worth

$$a_4 = a_3 \times (1 + (.05)/4) = a_0 \times (1 + (.05)/4)^4.$$

In general, if a_0 dollars are invested at an interest rate x with compounding n times per year, at the end of the year the final value is

$$a_0 \times C_n(x)$$

dollars, where

$$C_n(x) = \left(1 + \frac{x}{n}\right)^n.$$

This is the *compound interest formula*. It is well known that, for fixed x, the compound interest formula $C_n(x)$ has a limiting value as $n \to \infty$, namely, $\exp(x)$, as already displayed in (10.1). Consequently, excessively high compounding frequencies are pointless.

Nonetheless, it is interesting to evaluate $C_n(x)$ for various choices of n. Before considering algorithms to do this, let us investigate the condition number of $C_n(x)$. From the chain rule, the derivative is

$$C_n'(x) = n\left(1 + \frac{x}{n}\right)^{n-1} \frac{1}{n} = \frac{C_n(x)}{1 + \frac{x}{n}}.$$

Thus, for n sufficiently large compared to $|x|$, $C_n(x)$ is close to being its own derivative, which is not surprising, since the derivative of the limiting function $\exp(x)$ is itself. Therefore, the condition number of C_n is

$$\kappa_{C_n}(x) = \frac{|x|}{|C_n(x)|} \times \frac{|C_n(x)|}{|1 + \frac{x}{n}|} = \frac{|x|}{|1 + \frac{x}{n}|},$$

[1]A more demanding definition than (13.2) known as *backward stability* requires that $\widetilde{y} = f(\widetilde{x})$ for some $\widetilde{x}$ not necessarily equal to x or $\widehat{x}$, but close to x, in a relative sense, just as $\widehat{x}$ is.

which converges to $|x|$, the condition number of $\exp(x)$ (see (12.7)), as $n \to \infty$. Consequently, the compound interest formula is a well-conditioned function even for very large n, as long as $|x|$ is not large.

We first state the simplest, though not the most efficient, algorithm for computing $C_n(x)$.

Algorithm 13.1

1. *Compute* $z = 1 + \frac{x}{n}$, *and set* $w = 1$.
2. *Repeatedly (n times) perform the multiplication* $w \leftarrow w \times z$, *and return* w.

Since n may be large, the following more efficient algorithm makes use of the C library function `pow`.

Algorithm 13.2

1. *Compute* $z = 1 + \frac{x}{n}$.
2. *Return* $\mathrm{pow}(z, n)$.

A third algorithm makes use of the properties of the exponential and logarithmic functions. Writing

$$C_n(x) = z^n$$

and taking logarithms (base e) of both sides, we obtain

$$\log C_n(x) = n \times \log(z).$$

Therefore,

$$C_n(x) = \exp(n \times \log(z)).$$

Algorithm 13.3

1. *Compute* $z = 1 + \frac{x}{n}$.
2. *Compute* $v = \log(z)$ *and return* $\exp(n \times v)$.

Program 7 implements all three algorithms in C using single precision. The output for various n is summarized in Table 13.1. In all cases the input for x is 0.05, i.e., an interest rate of 5%.

```
#include <math.h>
main ()     /*  Program 7: Compound Interest */
{
   int n,i;
   float x,z,w,v;

   printf("enter input values for x (float) and n (integer) \n");
   scanf("%f  %d", &x, &n );
   z = 1 + x/n;
   w = 1;
   for (i=0; i<n; i++) {
      w = w*z;
   }
   v = log(z);
   printf("Alg 1: %e \n", w);
   printf("Alg 2: %e \n", pow(z,n));
   printf("Alg 3: %e \n", exp(n*v));
}
```

Table 13.1: Compound Interest at 5%, Single Precision

n	Algorithm 13.1	Algorithm 13.2	Algorithm 13.3
4	1.050946	1.050946	1.050946
365	1.051262	1.051262	1.051262
1000	1.051215	1.051216	1.051216
10,000	1.051331	1.051342	1.051342
100,000	1.047684	1.048839	1.048839
1,000,000	1.000000	1.000000	1.000000

The results are alarming! They look reasonable only for $n = 4$ (compounding quarterly) and $n = 365$ (compounding daily). For $n = 1000$, all three algorithms give results that are *less* than the result for $n = 365$. This is certainly not correct; we know that compounding more often may not give much more interest, but it certainly should not give less! We get our first clue as to what is happening when we come to the last line in the table. When $n = 1,000,000$, the computation in step 1 of all three algorithms,

$$z = 1 + \frac{.05}{1000000},$$

rounds exactly to 1 using single precision. Thus, the crucial interest rate information is completely lost, and all three algorithms return a result exactly equal to 1, as if the interest rate had been zero. Likewise, when $n = 10,000$ or $n = 100,000$, *some* but not *all* of the interest rate information is being lost in the computation of z.

On the other hand, it is clear that z is being computed correctly to about seven digits—there is no cancellation here! So why does the loss of the subsequent digits matter?

The heart of the matter is that *all three algorithms are unstable.* The rounding error in step 1 of each algorithm has a dramatically bad effect because the condition number of the function being computed in step 2 is *much worse* than the condition number of C_n. In fact, step 2 of all three algorithms computes the same function,

$$P_n(z) = z^n.$$

The derivative of $P_n(z)$ is nz^{n-1}, so the condition number is

$$\kappa_{P_n}(z) = \frac{|z \times nz^{n-1}|}{|z^n|} = n,$$

which, unlike the condition number of $C_n(x)$, grows without bound as $n \to \infty$. For example, when $n = 100,000$, the log (base 10) of the condition number is 5, and so, according to Rule of Thumb 12.1, although z computed in step 1 has seven significant digits, the result computed in step 2 of Algorithms 13.1 and 13.2 has only about two accurate digits. Thus, ill conditioning has been introduced, even though it was not present in the original function to be computed. Consequently, the algorithms are unstable.

Algorithm 13.3 does not avoid the instability with its use of exp and log. We already observed in the previous chapter that $\log(z)$ has a large condition number near $z = 1$, so although z is accurate to about seven digits, $v = \log(z)$ is accurate to only about two digits when $x = .05$ and $n = 100,000$ (see Table 12.2).

Table 13.2: Compound Interest at 5%

n	Algorithm 13.1 (double)	Algorithm 13.2 (double)	Algorithm 13.3 (double)	Algorithm 13.4 (single)
4	1.050945	1.050945	1.050945	1.050945
365	1.051267	1.051267	1.051267	1.051267
1000	1.051270	1.051270	1.051270	1.051270
10,000	1.051271	1.051271	1.051271	1.051271
100,000	1.051271	1.051271	1.051271	1.051271
1,000,000	1.051271	1.051271	1.051271	1.051271

The easiest way to get more accurate answers is to change Program 7 so that all computations are done in double precision. All we need to do is change `float` to `double`, and change `%f` to `%lf` in the `scanf` statement. The results, shown in Table 13.2, are correct to single precision accuracy because we are doing the computations in double precision. Of course, the algorithms are still not stable. If n is sufficiently large, inaccurate answers will again appear.

Exercise 13.1 *For what n does the double precision version of Program* 7 *give poor answers? Display the results in a table like Table* 13.1.

Surprisingly, there is no obvious stable algorithm to compute the compound interest formula using only the library functions pow, exp, and log. See [Gol91] for a simple stable algorithm that uses only exp and log, but one that is clever and far from obvious; a related discussion is given in [Hig02, Section 1.14.1]. However, there is a Taylor series expansion of the logarithm function near one that provides exactly what we need:

$$\log(1+s) = s - \frac{s^2}{2} + \frac{s^3}{3} - \frac{s^4}{4} + \ldots. \tag{13.3}$$

C99 provides a math library function `log1p` that computes this:

$$\text{log1p}(s) = \log(1+s).$$

This function is well-conditioned at and near $s = 0$ (see Exercise 13.2). This eliminates the need for the addition in step 1 of Algorithm 13.3 and gives us the following stable algorithm.

Algorithm 13.4

```
1. Compute $u = \frac{x}{n}$.
2. Compute $v = \text{log1p}(u)$, and return $\exp(n \times v)$.
```

This is implemented in Program 8, and the results are shown in Table 13.2. The stable algorithm gives accurate results using single precision; the unstable algorithms give such accurate results only when double precision is used.

```
#include <math.h>
main ()     /*  Program 8: Stable Algorithm for Compound Interest */
{
   int n;
   float x,u,v;

   printf("enter input values for x (float) and n (integer) \n");
   scanf("%f  %d", &x, &n );
   u = x/n;
   v = log1p(u);
   printf("Alg 4: %e \n", exp(n*v));
}
```

Exercise 13.2 *What is condition number of the function* log1p(s) *as a function of* s*? What is the limit of the condition number as* $s \to 0$*? See Exercise* 12.3.

The function `log1p`, and a related function `expm1` (see Exercise 13.4), are among the many specified in [IEE08, Table 9.1] as functions that should be implemented to provide correctly rounded results. For more details about using these functions to compute compound interest and a related quantity, the present value of an annuity, see [Bee17, Ch. 10.6].

Instability via Cancellation

In fact, the phenomenon of cancellation described in Chapter 11 can be completely explained by conditioning. The condition number of the function

$$f(x) = x - 1$$

is

$$\kappa_f(x) = \frac{|x|}{|x-1|},$$

which is arbitrarily large for x close to 1. Consequently, an algorithm that introduces cancellation unnecessarily is introducing ill conditioning unnecessarily and is unstable.

In Chapter 11, we discussed the idea of approximating a derivative $g'(x)$ by a difference quotient. A working of Exercise 12.7 shows that the difference quotient has the same condition number in the limit as $h \to 0$ as the derivative itself, suggesting that approximating $g'(x)$ by the difference quotient might be a stable algorithm. Unfortunately, the first step in evaluating the difference quotient, computing $g(x+h) - g(x)$, has a large condition number for small h, and hence computing the difference quotient without the use of intermediate higher precision is unstable. Better algorithms exist to approximate the derivative, e.g., using the central difference quotient with larger h or still more accurate difference quotients with still larger h. Of course, using the formula for the derivative is preferable if it is known. Furthermore, automatic differentiation of functions is a technique that has been known for decades and has now become widely used, especially in machine learning.

Exercise 13.3 *Why is the formula*

$$\frac{x^2 - 1}{x - 1}$$

an unstable way to compute $f(x) = x + 1$*? For what values of* x *is it unstable?*

Exercise 13.4 *Consider the function $f(x) = \exp(x) - 1$.*

1. *What is the condition number of $f(x)$? What is the limit of the condition number as $x \to 0$? See Exercise* 12.3.
2. *Write a C program to compute $f(x)$ using the exp function. Is the algorithm stable? If not, what are the values of x that cause trouble?*
3. *Write a C program to compute $f(x)$ directly by calling the math library function* `expm1`*, intended exactly for this purpose. Does it give more accurate results?*

Exercise 13.5 *This is an extension to Exercise* 10.14*. Instead of using only positive input data, run your interval sum program to add up numbers with both positive and negative values. Choose some of your input values so that the numbers cancel out and the result is zero or close to zero. To how many significant digits do your three answers (upper bound, lower bound, and intermediate) agree? Is it as many as before? Is the difficulty that the problem of adding data with alternating signs is ill conditioned, or that the addition algorithm is unstable? Does it help to add up the positive and negative terms separately? Be sure to try a variety of input data to fully test the program.*

Computing the Exponential Function without a Math Library

For our second example illustrating stability and instability, let us attempt to compute the exponential function $\exp(x)$ directly, without any calls to library functions. From (12.7), we know that exp is a well-conditioned function as long as $|x|$ is not too large. We use the well-known Taylor series

$$\exp(x) = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots .$$

This allows us to approximately compute the limiting sum by means of a simple loop, noting that successive terms are related by

$$\frac{x^n}{n!} = \left(\frac{x^{n-1}}{(n-1)!}\right) \times \frac{x}{n}.$$

Thus, each term is easily computed from the previous term by multiplying by x and dividing by n. How should we terminate the loop? The simplest way would be to continue until the new term in the sum underflows to zero, as in Program 2 (Chapter 10). A better solution is to use the idea in Program 3 (Chapter 10): the loop may be terminated when *the new term is small enough that adding it to the previous terms does not change the sum.* Program 9 implements this idea using single precision.

```
#include <math.h>
main()  /* Program 9: Compute exp(x) from its Taylor series */
{
   int n;
   float x, term, oldsum, newsum;

   printf("Enter x \n");
   scanf("%e", &x);
   n = 0;
   oldsum = 0.0;
   newsum = 1.0;
   term = 1.0;
   /* terminates when the new sum is no different from the old sum */
   while (newsum !=oldsum){
      oldsum = newsum;
      n++;
      term = (term*x)/n;    /* term has the value (x^n)/(n!) */
      newsum = newsum + term;  /* approximates exp(x) */
      printf("n = %3d   term = %13.6e   newsum = %13.6e \n",
             n,term,newsum);
   }
   printf("From summing the series,     exp(x)=%e \n", newsum);
   printf("Using the standard function, exp(x)=%e \n", exp(x));
}
```

Here is the output of Program 9 for $x = .05$:

```
n =   1   term =  5.000000e-02   newsum =  1.050000e+00
n =   2   term =  1.250000e-03   newsum =  1.051250e+00
n =   3   term =  2.083333e-05   newsum =  1.051271e+00
n =   4   term =  2.604167e-07   newsum =  1.051271e+00
n =   5   term =  2.604167e-09   newsum =  1.051271e+00
From summing the series,     exp(x)=1.051271e+00
Using the standard function, exp(x)=1.051271e+00
```

We see that the value of `term` printed at each step grows rapidly smaller, and the loop terminates when `term` is so small that two successive values of `newsum` are identical. Actually, it looks from the output that this occurs a few lines before the loop terminates, but that is just the decimal conversion of `newsum` to seven digits; the binary floating point values are different until the last line. The final value of `newsum`, 1.051271, agrees with the value computed by the library function `exp` to all digits shown.

Table 13.3: Computing the Exponential Function

x	Computed by summing series	Computed by call to `exp(x)`
10	2.202647e+04	2.202647e+04
1	2.718282e+00	2.718282e+00
.05	1.051271e+00	1.051271e+00
-1	3.678794e-01	3.678794e-01
-5	6.738423e-03	6.737947e-03
-10	-6.256183e-05	4.539993e-05

Now let us run Program 9 again for a larger value of x, say 10.0:

```
n =   1   term =  1.000000e+01   newsum =  1.100000e+01
n =   2   term =  5.000000e+01   newsum =  6.100000e+01
n =   3   term =  1.666667e+02   newsum =  2.276667e+02

      ..........4 lines omitted..........

n =   8   term =  2.480159e+03   newsum =  7.330842e+03
n =   9   term =  2.755732e+03   newsum =  1.008657e+04
n =  10   term =  2.755732e+03   newsum =  1.284231e+04
n =  11   term =  2.505211e+03   newsum =  1.534752e+04
n =  12   term =  2.087676e+03   newsum =  1.743519e+04
n =  13   term =  1.605905e+03   newsum =  1.904110e+04
n =  14   term =  1.147075e+03   newsum =  2.018817e+04
n =  15   term =  7.647164e+02   newsum =  2.095289e+04

      .........13 lines omitted..........

n =  29   term =  1.130996e-02   newsum =  2.202646e+04
n =  30   term =  3.769987e-03   newsum =  2.202647e+04
n =  31   term =  1.216125e-03   newsum =  2.202647e+04
n =  32   term =  3.800390e-04   newsum =  2.202647e+04
From summing the series,     exp(x)=2.202647e+04
Using the standard function, exp(x)=2.202647e+04
```

We find that `term` grows larger than its initial value before it starts to get smaller, but it does eventually grow smaller when $n > 10$, and the loop eventually terminates as before, with an answer 22026.47 that again agrees with the library function `exp` to single precision.

Exercise 13.6 *Termination of the loop takes place when the decimal exponents of* `newsum` *and* `term` *differ by about* 7 *or* 8. *Why is this?*

Table 13.3 shows the output of Program 9 for various values of x. The first column shows the value computed by summing the series, and the second column shows the result returned by the library function `exp`. The results agree to single precision for $x = 10$, $x = 1$, $x = 0.05$, and $x = -1$. However, the final line of Table 13.3 shows that the value computed by the loop for $x = -10$ is completely wrong! And the previous line for $x = -5$ is correct to only about four digits.

Let's look at the details for $x = -10$.

```
n =   1   term = -1.000000e+01   newsum = -9.000000e+00
n =   2   term =  5.000000e+01   newsum =  4.100000e+01

      ..........6 lines omitted..........

n =   9   term = -2.755732e+03   newsum = -1.413145e+03
n =  10   term =  2.755732e+03   newsum =  1.342587e+03

      .........34 lines omitted..........

n =  45   term = -8.359650e-12   newsum = -6.256183e-05
n =  46   term =  1.817315e-12   newsum = -6.256183e-05
From summing the series,     exp(x)=-6.256183e-05
Using the standard function, exp(x)=4.539993e-05
```

We see that the values of `term` are the same as for $x = 10$ except that they alternate in sign, which makes sense, since when n is odd,

$$\frac{(-x)^n}{n!} = \frac{-(x^n)}{n!}.$$

Therefore, since the terms alternate in sign, they cancel with each other, and eventually the value of `newsum` starts to get smaller. The final result is a small number; this is to be expected, since $\exp(x) < 1$ for $x < 0$. Also, the loop takes longer to terminate than it did for $x = 10$, since the decimal exponents of `term` and `newsum` must differ by about 7 or 8 and `newsum` is smaller than it was before. Looking at the final value for `newsum`, however, we see that the answer is completely wrong, since it is not possible for $\exp(x)$ to be negative for any value of x. What is happening?

To find out, we examine the line-by-line output of Program 9 more carefully. We see that for $x = -10$, the size (i.e., absolute value) of `term` increases to 2.75×10^3 (for $n = 10$) before it starts decreasing to zero. We know that `term` is accurate to at most about seven digits, since it is an IEEE single format number. Consequently, its largest value, about 2.75×10^3, must have an absolute rounding error that is at least about 10^{-4}. The same error must be present in `newsum`, since it is obtained by adding values of `term` together. As more terms are added to `newsum`, this error is *not* reduced, even though the value of `newsum` continues to get smaller as the terms cancel each other out. In fact, the final value of `newsum` is smaller, in absolute value, than the error and consequently has *no significant digits.* The source of the difficulty is the size of the intermediate results together with the alternating sign of the terms, which cancel each other out, leading to a small final result even though the individual terms are quite large. For $x > 0$, there is no difficulty, since all the terms are positive and no cancellation takes place. But for $x < 0$, the results are meaningless for $|x|$ sufficiently large.

Now let's run Program 9 for $x = -5$:

```
n =   1   term = -5.000000e+00   newsum = -4.000000e+00
n =   2   term =  1.250000e+01   newsum =  8.500000e+00
n =   3   term = -2.083333e+01   newsum = -1.233333e+01
n =   4   term =  2.604167e+01   newsum =  1.370833e+01
n =   5   term = -2.604167e+01   newsum = -1.233333e+01
n =   6   term =  2.170139e+01   newsum =  9.368057e+00

      .........20 lines omitted..........

n =  27   term = -6.842382e-10   newsum =  6.738423e-03
n =  28   term =  1.221854e-10   newsum =  6.738423e-03
From summing the series,     exp(x)=6.738423e-03
Using the standard function, exp(x)=6.737947e-03
```

The final result agrees with the library function `exp` to a few digits, but not to full precision. When $x = -5$, the difficulty is not as severe, since the size of `term` grows only to 2.6×10^1 before it starts decreasing to 0. This value of `term`, which is again accurate to at most about seven digits, has an absolute rounding error at least about 10^{-6}. The final answer, which is computed to be 6.738423×10^{-3}, must have an error of size at least about 10^{-6} and is therefore accurate to only about three digits.

Since the problem of computing $\exp(x)$ is well conditioned when $|x|$ is not large, the inevitable conclusion is that Program 9 implements an algorithm that is unstable for $x < 0$.

How can we change Program 9 so that it is stable? The answer is simple: if x is negative, sum the series for the positive value $-x$ and compute the reciprocal of the final amount, using the fact that

$$\exp(x) = \frac{1}{\exp(-x)}. \tag{13.4}$$

Thus we add to the end of the code

```
   printf("One over the sum=%e \n", 1/newsum);
   printf("Call to exp(-x) =%e \n", exp(-x));
```

and run it for $x = 10$ instead of $x = -10$. The final two lines of output are

```
One over the sum=4.539992e-05
Call to exp(-x) =4.539993e-05
```

It may seem amazing that this simple trick could work so well, but the reason it works is that *no cancellation takes place.* Dividing 1 by a large number with about six or seven significant digits results in a small number that also has about six or seven significant digits. (See Exercise 13.7.)

This confirms that the difficulty with Program 9 was not inherent in the problem that it solves, which is not ill conditioned. We chose an unstable algorithm that introduced cancellation, and therefore ill conditioning, unnecessarily.

Although Program 9 works well when cancellation is avoided, it cannot be concluded that summing a series until the sum is unchanged will always give such good answers. See Exercises **??**–13.13.

The library function `exp` uses a more clever but more complicated method, which is both highly accurate and very fast. The purpose of this discussion has been to show that a good answer can be obtained with a simple program, but also that a completely wrong answer can be obtained if precautions are not taken. For much more detailed—yet very readable—discussions of algorithms for computing the functions in the math library, see [Mul97], [Bee17] and [MB+18].

Exercise 13.7 *Determine the condition number of the function*

$$f(x) = \frac{1}{x}.$$

Are there any finite, nonzero numbers x for which f has a large condition number?

Exercise 13.8 *Suppose Program* 9 *is changed to use double precision. For what range of values of x (approximately) does it give no significant digits, and why? Modify Program* 9 *further using the formula* (13.4) *to correct the difficulty.*

Exercise 13.9 *Suppose Program* 9 *is changed to add the positive and negative terms in two separate sums, and take their difference at the end. Does this improve its stability? Why or why not?*

Exercise 13.10 *If $|x|$ is large enough, overflow occurs in Program* 9*. If the standard response to overflow is used, what results are generated* (a) *if $x > 0$,* (b) *if $x < 0$, and* (c) *if $x < 0$ but the program is modified using the formula* (13.4)*? Explain why these results are obtained.*

Exercise 13.11 *Modify Program* 9 *using the interval arithmetic idea. Compute lower and upper bounds on the sum, using round down and round up. You may have to change the termination condition in order to avoid an infinite loop; why? How well do the lower and upper bounds agree when $x = 10$? How well do they agree when $x = -10$? Can you also get lower and upper bounds for the result computed by the stable version of the algorithm incorporating* (13.4)*? (This requires thought; see Exercise* 10.15.*) Do you conclude that the rounding modes are useful for testing the stability of an algorithm?*

A working of Exercises 10.14, 10.15, 13.5, and 13.11 demonstrates both the power and the limitations of interval arithmetic. The power is that guaranteed lower and upper bounds on the desired solution are computed, but the limitation is that these bounds may not be close together, especially if the problem being solved is ill conditioned. It should be clear from a working of the exercises that carrying out a complicated calculation with interval arithmetic would be very clumsy with only the rounding modes as tools. However, software packages are available that carry out interval computations automatically. INTLAB [Rum] is an efficient and powerful MATLAB toolbox for interval arithmetic, which depends on the ability to set the rounding modes in MATLAB as described in Chapter 9. For more on interval arithmetic see [Rum10] and [Tuc11]. An IEEE standard for interval arithmetic was published in 2015 as IEEE 1788-2015.

Exercise 13.12 *Write a stable modification of Program* 9 *that computes the function* $\exp(x) - 1$ *without any calls to the math library. Compare your answer with the result computed by* `expm1` *(see Exercise* 13.4*).*

Exercise 13.13 *It is well known that the series*

$$1 + \frac{1}{4} + \frac{1}{9} + \frac{1}{16} + \cdots$$

converges to $\pi^2/6$. Write a C program to sum this series, terminating if the sum is unchanged by the addition of a term. How good are the results using single precision? Using double precision? Use an integer `n` *to control the loop but assign this to a float variable before you square it, to avoid integer overflow.*

Chapter 14

Higher Precision Computations

Although double precision (binary64) is sufficient for most problems, it may not be adequate if a problem is extremely ill-conditioned, or many digits of accuracy are required. Such problems arise in many different scientific applications, with black hole physics [Kha13] being one example. A completely different kind of application arises in computational geometry, where, for example, it may be necessary to know whether or not a given point is inside a circle determined by three other points and high precision computations may be needed to determine this with any certainty [She97, BF+23]. However, as noted in Chapter 4, the quadruple precision (binary128) interchange format introduced in the 2008 version of the standard is not supported in hardware by most current microprocessors, let alone other wider floating point formats.

Arbitrary precision systems, that is, systems with arbitrarily high precision specified by the user, have been available for many decades, but since no computer provides hardware support for this, such systems must be implemented in software. For a detailed discussion of algorithms for both arbitrary precision integer arithmetic and arbitrary precision floating point arithmetic, including correct rounding and other features inspired by the IEEE standard, see the book by Brent and Zimmerman [BZ11]. As explained there, MPFR[1] is a C library which extends the main ideas of the IEEE standard to arbitrary precision arithmetic, providing correctly rounded arithmetic and standard responses to exceptions such as $1/0$ and $0/0$ resulting in ∞ and NaN respectively. For more details see [FH+07] and [MB+18, Sec. 14.4.4]. Other packages, such as the well known Maple and Mathematica computer algebra packages, also provide arbitrary precision calculations but without any correct rounding guarantees. We also mention briefly here that Advanpix[2] is an efficient and easy-to-use arbitrary precision package for MATLAB. Although efficiency is a primary concern in these state-of-the-art arbitrary precision packages, computations are necessarily substantially slower than using standard floating point arithmetic.

Double-Double and Quad-Double

An efficient alternative to arbitrary precision uses standard floating point hardware operations to implement floating point computation with higher than standard accuracy. This development has been driven partly by the widespread acceptance of the

[1]http://www.mpfr.org
[2]https://www.advanpix.com/

IEEE standard, ensuring correctly rounded results, and partly by the highly optimized performance of floating point hardware operations on most microprocessors. In such systems, a number is represented by an "unevaluated sum" of a specified number of fixed precision floating point numbers. The most common example is a sum of two double precision numbers, often called *double-double*.[3] For example, suppose we have a number x which is known to 111-bit accuracy:

$$x = (b_0.b_1 \ldots b_{52}b_{53} \ldots b_{105}b_{106} \ldots b_{110})_2 \times 2^E,$$

with $b_0 = 1$ so x is normalized. This number can be approximated by the double-double (h, t), where, if the bit b_{53} is 1 *and* the bit b_{106} is 0, then h, the "head" (or leading precision) component is given by the normalized double format

$$h = (b_0.b_1 \ldots b_{52})_2 \times 2^E,$$

and t, the "tail" (or trailing precision) component is given by the normalized double format

$$t = (b_{53}.b_{54} \ldots b_{105})_2 \times 2^{E-53}.$$

Then $h + t$ is precisely x truncated to 106 bits, or equivalently, since the first omitted bit is zero, rounded to the nearest 106-bit number. If b_{53} is 0, the tail t would need to be normalized, shifting more bits of x into t until its first bit is 1. If the first omitted bit is 1, rounding to nearest may require rounding up, redefining t and possibly also h. We may think of the precision of a double-double as $p = 106$, although it is potentially higher. For example, the number $1 + 2^{-1000}$ can be exactly represented to 1001 bit accuracy by the double-double (h, t) with $h = 1$ and $t = 2^{-1000}$, but if this is then added to another double-double, it's likely (or at least possible) that the result will have only 106 bit accuracy. Such precision is sometimes called "wobbling" [MB+18, p. 515], and it is somewhat reminiscent of the unpredictable precision of the hexadecimal system used by the IBM 360 and its successors (see Chapter 8).

Arithmetic with double-doubles and extensions such as quad-doubles, each of which is the unevaluated sum of four double format numbers, is somewhat complicated, but some clever and efficient algorithms have been devised. These depend on sub-calculations such as TwoSum, which returns not only the sum of two ordinary double format numbers x and y, but also the rounding error

$$x + y - (x \oplus y) = x + y - \text{round}(x + y).$$

As long as round-to-nearest (with any tie-breaking rule) and gradual underflow are in effect, and $x \oplus y$ is finite and nonzero, it turns out that this quantity is an exactly representable double format floating point number; see [RD18], where C code to implement TwoSum is given. Another sub-calculation that is needed for implementing multiplication of double-double numbers is TwoProduct, which returns not only the product of two ordinary double format numbers x and y, but also the rounding error

$$x \times y - (x \otimes y) = x \times y - \text{round}(x \times y).$$

In most cases, this is also a double format floating point number, but in rare cases it may need rounding. It's interesting to note that TwoProduct is trivial to implement using the fused-multiply add instruction (see Chapter 6), via

$$z = x \otimes y; \quad \text{error} = \text{FMA}(x, y, -z).$$

[3]The name may bring Shakespeare's Macbeth, Act IV, Scene 1 to mind.

The correctness of this immediately follows from the fact that the FMA instruction delivers the correctly rounded value of $x \times y - z$. To see why TwoProduct is needed to correctly round the product of two double-doubles (h_a, t_a) and (h_b, t_b), simply observe that the product of the 53-bit precision heads, $h_a \times h_b$, has 106 bits of precision, of which the last 53 have the same significance as the first 53 bits of each of the products $h_a \times t_b$ and $h_b \times t_a$, so it is important not to neglect these when rounding $h_a \times h_b$ to 53 bits.

The demand for TwoSum and TwoProduct led to the only significant technical addition to the IEEE standard in 2019, with the recommendation that language standards provide the operations *augmentedAddition* and *augmentedMultiplication* implementing TwoSum and TwoProduct respectively. In connection with these operations, the 2019 standard also recommended an alternative tie-breaking rule for rounding called *roundTiesToZero* (see Chapter 4). Although this rule is not needed for implementation of double-double arithmetic, it simplifies the implementation of the new recommended operations, and, in addition, it enables reproducible arithmetic when summations of many numbers are implemented using parallel computing where the order of the summations is not specified. For more details, including the long history of extending the precision of standard floating point operations, see [RD18], [MB+18, Chap. 14], [BJ+23, Sec. 5], and, for information on high-precision software packages, [Bai].

The Hundred Digit Challenge

In 2002, Nick Trefethen announced the SIAM Hundred Digit Challenge, specifying 10 fiendishly difficult numerical problems and challenging his readers to solve each to 10-digit accuracy. The competition turned out to be far more popular than Trefethen expected, and 20 correct solutions were submitted. For a detailed account of the problems and their solutions from some of the winners, including an interview with Trefethen, see [BL+04]. As Higham writes in [Hig17]: *Although high precision arithmetic could be used in the solutions as part of a brute force attack, it turned out to be generally not necessary. This example serves as a reminder that mathematical ingenuity in the choice of algorithm can enable a great deal to be done in double precision arithmetic, so one should always think carefully before resorting to higher precision arithmetic, with its attendant increase in cost.*

Chapter 15

Lower Precision Computations

Up until recently, most floating point computing used the single and double formats (*binary32* and *binary64*) introduced in the original IEEE 754 standard published in 1985. The 2008 revision of the standard introduced a shorter (or narrower) 16-bit format called *binary16*, also known as *half precision*, but this was called an interchange format; the intention was not that it be used for computation. However, the rapid rise of machine learning in the past decade, specifically deep learning using neural networks, involves massive amounts of numerical computing, and this has motivated the design of microprocessors using short floating point formats for computation, saving both time and energy at the cost of lower precision results.

16-bit Floating Point Formats

Many recent floating point microprocessors support 16-bit floating point numbers. Intel, AMD, Arm and NVIDIA all support hardware computation using the IEEE format *binary16*, now often abbreviated *fp16*, which uses one bit for the sign, 5 bits for the exponent field, and 10 bits for the fractional part of the significand.[1] This means the precision is $p = 11$, including the hidden bit, so the machine epsilon is $2^{-10} \approx 10^{-3}$, corresponding to about 3 decimal digits of accuracy (see (5.14)), but the normalized range of the numbers is very limited, from $N_{\min} = 2^{-14}$ to $N_{\max} \approx 2^{16}$, so overflow and underflow are much more frequent occurrences using *binary16* than they are using *binary32*. Computations with *binary16* numbers became widespread in recent years because of their use in machine learning [ND+18], but the limitations on the range were among the challenges.

For this reason, in 2016 Google announced a new 16-bit format called *bfloat16* (Brain Float16, sometimes abbreviated *bf16*) which, although it is not compliant with IEEE 754, is inspired by it [Dea20]. This format has the same exponent range as the IEEE single format (binary32), which means the bit width of the exponent field is the same as in the single format, namely, 8 bits. However, this leaves only 7 bits remaining for the fractional part of the significand, so the precision is $p = 8$, including the hidden bit, and the machine epsilon is $2^{-7} \approx 10^{-2}$. Hence, *bfloat16* floating point numbers have only about 2 decimal digits of accuracy. Some research indicates that neural network computation is much more sensitive to the range of the exponent than

[1]Arm also offers a modified *binary16* format that eliminates Infs and NaNs, a major violation of the IEEE standard requirements.

Format	*bfloat16* (bf16)	*binary16* (fp16)
p	8	11
$\epsilon_{\text{mch}} = 2^{-(p-1)}$	$\approx 7.8 \times 10^{-3}$	$\approx 9.8 \times 10^{-4}$
$w = 16 - p$	8	5
$E_{\max} = 2^{w-1} - 1$	127	15
$E_{\min} = -2^{w-1} + 2$	-126	-14
$bias = E_{\max}$	127	15
$N_{\max} = 2^{E_{\max}}\left(2 - 2^{-(p-1)}\right)$	$\approx 3.4 \times 10^{38}$	$\approx 6.6 \times 10^{4}$
$N_{\min} = 2^{E_{\min}}$	$\approx 1.2 \times 10^{-38}$	$\approx 6.1 \times 10^{-5}$
$S_{\min} = 2^{E_{\min}-(p-1)}$	$\approx 9.2 \times 10^{-41} *$	$\approx 6.0 \times 10^{-8}$

Table 15.1: Parameters for the two16-bit floating point formats that are widely used at present: precision p, machine epsilon ϵ_{mch}, exponent bit width w, maximum exponent, minimum exponent, exponent bias, maximum normalized number, minimum positive normalized number, minimum positive subnormal number. The asterisk (*) indicates that implementations of *bfloat16* often do not support subnormals. Note that the formulas for the various parameters are all consistent with those shown in Table 4.3.

the width of the significand [WK19]. An advantage of the *bfloat16* format is that converting it to the IEEE single format is trivial (just append 16 zeros to the significand), and rounding IEEE single to *bfloat16* is also a simple operation, with no overflow or underflow in almost all cases. Many microprocessors now support *bfloat16* format in addition to *binary16.* Early implementations do not support subnormal numbers and therefore flush underflowed values to zero. However, in principle, subnormal arithmetic could be supported. Parameters for *bfloat16* and *binary16* are summarized in Table 15.1. The term *half precision* does not apply to *bfloat16* since the precision of this format, 8, is much less than half of 24, the precision of the single format.

Exercise 15.1 *Explain why the product of two binary16 format floating point numbers can always be stored exactly as a single format (binary32) number, with no rounding error. You need to take both the precision and the exponent range of the two formats into consideration. Is this also true of the product of two bfloat16 numbers? (See Exercise 6.11.)*

8-bit Floating Point Formats

Because of the widespread successful use of 16-bit floating point formats in machine learning, there is now a lot of interest in 8-bit floating point formats, sometimes called *quarter precision*, although their precision varies and is generally less than a quarter of 24, the precision of the single format. A new IEEE working group, known as P3109, was established in 2021 to standardize these formats and has recently released an interim report on its work.[2] The report recommends that several formats be defined, namely *binary8p3*, *binary8p4* and *binary8p5*, with precisions 3, 4 and 5 respectively, and hence exponent bit widths of 5, 4 and 3 respectively. Its recommendations, while

[2]See https://sagroups.ieee.org/P3109wgpublic/. Although the official scope of the committee is *Arithmetic Formats for Machine Learning*, almost all its discussions so far have focused on 8-bit formats, as *bfloat16* has become a *de facto* 16-bit standard format in addition to *binary16*. The author is a member of this working group. The interim report is available from https://github.com/P3109/Public/tree/main/Shared%20Reports.

Format	*binary8p3*	*binary8p4*	*binary8p5*
p	3	4	5
$\epsilon_{\text{mch}} = 2^{-(p-1)}$	0.25	0.125	.0625
$w = 8 - p$	5	4	3
$E_{\max} = 2^{w-1} - 1$	15	7	3
$E_{\min} = -2^{w-1} + 1$	-15	-7	-3
$bias = 2^{w-1}$	16	8	4
$N_{\max} = 2^{E_{\max}}\left(2 - 2^{-(p-2)}\right)$	49152	224	15
$N_{\min} = 2^{E_{\min}}$	$\approx 3.1 \times 10^{-5}$	$\approx 7.8 \times 10^{-3}$	$\approx 1.3 \times 10^{-1}$
$S_{\min} = 2^{E_{\min}-(p-1)}$	$\approx 7.6 \times 10^{-6}$	$\approx 9.8 \times 10^{-4}$	$\approx 7.8 \times 10^{-3}$

Table 15.2: Parameters for three 8-bit floating point formats recommended by the P3109 interim report: precision p, exponent bit width w, maximum exponent, minimum exponent, exponent bias, maximum normalized number, minimum positive normalized number, minimum positive subnormal number. Notice the different formulas for $E_{\min}$, *bias* and $N_{\max}$ compared to those given in Table 4.3 for the three basic formats in the 2019 IEEE standard and in Table 15.1 for the commonly used 16-bit formats.

Value	Bit String
$-\infty$	11111111
NaN	10000000
0	00000000
∞	01111111

Table 15.3: Four special values for the recommended 8-bit formats. The bit string representations are the same regardless of the precision p.

based on the IEEE 754 formats, introduce some significant departures from them. The primary change is that because there are only 256 distinct 8-bit strings, it is proposed to reduce the number of representations for the number zero from two to one, with a positive sign, and also to reduce the number of NaN formats to just one. This is done by using the usual representation for -0, namely the bitstring $100\ldots0$ (sign bit one, exponent and significand bits all zero), to denote NaN. Representations for subnormal numbers in the usual way are included, and representations for $\pm\infty$ are also endorsed, but using a modified representation, with an exponent bit string $a_1 \ldots a_w = 11\ldots1$ as usual, but a significand bit string $b_1 \ldots b_{p-1} = 11\ldots1$ instead of $10\ldots0$. The elimination of -0 and the reduction of the number of NaNs to just one frees up $2^{p-1} - 1$ representations for positive numbers, namely with $a_1a_2\ldots a_w = 11\ldots1$ and $b_1b_2\ldots b_{p-1}$ free to be anything except all ones. It also frees up the same number of representations for negative numbers. These patterns are used to represent the largest finite numbers in magnitude. Noting that there is a lack of symmetry in the IEEE 754 formats in the sense that $E_{\max} = 2^{w-1} - 1$ but $E_{\min} = -2^{w-1} + 2$, it is recommended that for the 8-bit format representation, $E_{\max}$ remains equal to $2^{w-1} - 1$ but $E_{\min}$ is changed to to $-2^{w-1} + 1$. This is accomplished by setting the exponent bias to 2^{w-1}, so that the largest *biased* exponent is $2^{w-1} - 1 + 2^{w-1} = 2^w - 1$ (the largest integer that can be represented with w bits), instead of the IEEE 754 value $2^w - 2$, while the smallest *biased* exponent is $-2^{w-1} + 1 + 2^{w-1} = 1$, as it is for IEEE 754 formats. This discussion is summarized in Table 15.2.

It follows that the recommended 8-bit format floating point representations have just four special values, namely 0, $-\infty$, ∞ and NaN. Their bitstring representations are given in Table 15.3. A nice property is that these bitstrings are exactly the same for all precisions p. All 252 other bitstrings represent distinct, finite, nonzero normalized or subnormal numbers (a different set for each choice of p).

These formats can be extended more generally to *binary8pp*, where in principle the precision p might be any integer from 1 to 7, should there be a demand for this. It is instructive to consider two extreme cases. In the case $p = 1$, so $w = 7$, there would be no fractional significand bits, so all nonzero finite numbers would be powers of two, the largest finite number would need to have a smaller biased exponent than the exponent field in the representation for ∞ (as with the usual 754 formats), and the formulas for $E_{\max}$, $E_{\min}$ and *bias* would become 63, -62 and 63 respectively. Hence, the numbers represented would be $\pm 2^{-62}, \pm 2^{-61}, \ldots, \pm 2^{63}$, as well as $0, \pm\infty$ and NaN. In the case $p = 7$, so $w = 1$, the formulas given in Table 15.2 give $E_{\max} = E_{\min} = 0$, so all normalized and subnormal numbers would have the exponent 0. Hence, the format would become essentially a scaled sign-and-magnitude integer format, with subnormal values $\pm 1/64, \ldots, \pm 63/64$ and normalized values $\pm 64/64, \ldots, \pm 127/64$, as well as $0, \pm\infty$ and NaN. The cases $p = 0$ and $p = 8$ are excluded. Setting $p = 0$ would mean no significand bits, not even a hidden bit. Setting $p = 8$ would mean no exponent bits, so all numbers would need to be subnormal in order to represent zero, which somewhat suprisingly results in the same set of possible values as $p = 7$.

The choice of names for 8-bit format floating point has led to the suggestion that *binary16* and *bfloat16* formats be renamed *binary16p11* and *binary16p8*, respectively, as well as the possibility of other choices for p in the future.

Another major change from IEEE arithmetic being recommended by the P3109 working group is an option for *saturation arithmetic*, likely the default, where, when overflow occurs, the result is set to $\pm N_{\max}$ instead of $\pm\infty$. The argument in favor of this is that because the 8-bit formats have such a limited exponent range, overflow will be a frequent occurrence and that, at least in the context of machine learning, computations may give more useful results using the saturation option. Gradual underflow using subnormal numbers is recommended as usual. However, overflow and underflow can sometimes be avoided by carefully scaling the input data to the various computations required for machine learning [PZ+23]. In a way, this is a return to von Neumann's way of thinking in the 1940s (see Chapter 3): bits are too precious to use for floating point exponents when this can be avoided by careful programming.

Exercise 15.2 *Which of the formats binary8p3, binary8p4 and binary8p5 have the property that the product $x \times y$, where x and y are numbers in this format, can be stored in the binary16 format without rounding? What about bfloat16? Remember to take account of both the precision and the exponent range of the format. (See Exercises 6.11 and 15.1.)*

Mixed Precision

Mixed precision computations use two or more precisions together. As noted in [Hig17], like so many ideas, this goes back to Wilkinson in the early days of floating point, but in the two decades following the publication of the IEEE standard in 1985, when computing was dominated by x87 microprocessors, there was little incentive to use single precision in preference to double, or a mix of both, since all computations used the same 80-bit extended precision registers. However, this began to change with Intel's introduction of SSE and SSE2, which could carry out single precision computations twice as fast as double, motivating the use of single precision, or mixed single and double, in preference to double precision only. More recently, because of the massive computational demands of machine learning and the resulting use of lower precision formats, mixed precision computations have moved to center stage.

Although deep learning methods for machine learning require a lot of computation, many of the operations needed are simple and can be performed in parallel on multiple data. A fundamental building block is the computation of inner products (or dot products, or scalar products) of vectors. Let us write two vectors, or one-dimensional arrays of numbers, as

$$a = [a_1 \; a_2 \; \dots \; a_n] \quad \text{and} \quad b = [b_1 \; b_2 \; \dots \; b_n],$$

where n is an integer and $a_i, b_i, i = 1, \dots, n$, are floating point numbers. Then the inner product of a and b is defined as

$$a \cdot b = a_1 \times b_1 + a_2 \times b_2 + \cdots + a_n \times b_n.$$

We know from Chapter 6 that if each a_i and b_i, $i = 1, \dots, n$, has precision p, that is, has a significand with p bits including the hidden bit, then the exact product $a_i \times b_i$ has a significand with at most $2p$ nonzero bits, as the significands of a_i and b_i are essentially scaled integers with p bits. Mixed precision inner products treat each product $a_i \times b_i$ as a precision p product, but, instead of rounding the result to p bits as a floating point multiplication would usually do, all $2p$ bits of the product are saved in a format with a higher precision q, where $q \geq 2p$. Then all the products are accumulated (added together) using precision q, giving a precision q result for the inner product $a \cdot b$.

Among other places, inner products are used in a matrix multiplication operation. Let us write two matrices, or two-dimensional arrays of numbers, as

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} b_{11} & b_{12} & \dots & b_{1n} \\ b_{21} & b_{22} & \dots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \dots & b_{nn} \end{bmatrix}.$$

Then the matrix product $A \cdot B$ is defined as the $n \times n$ matrix whose (i, j) entry is the inner product of the ith row of A and the jth column of B.

Matrix multiplication is so fundamental in deep learning algorithms that many specialized microprocessors have recently been designed and built to carry it out efficiently using mixed precision. These new processors are sometimes known as tensor cores (made by NVIDIA) or tensor processing units (TPUs, made by Google); a tensor is a generalization of a matrix to a k-dimensional array of numbers with $k > 2$. NVIDIA's tensor cores use *binary16* and *binary32* formats ($p = 11$ and $q = 24$ in the notation used in the previous paragraph), while Google's TPUs use *bfloat16* and *binary32* formats ($p = 8$ and $q = 24$). The basic matrix-multiply-plus-addition mixed precision operation which can be carried out by these machines,

$$D = A \cdot B + C,$$

where A and B are matrices with precision p, C is a matrix with higher precision $q \geq 2p$, and the resulting matrix D also has precision q, is called a block FMA in [BH+22]. Details of rounding and other properties of these machines are not publicly available, but an experimental investigation of the properties of NVIDIA's tensor cores is given in [FH+21]. Tensor cores and TPUs are much more efficient, in terms of both time and power consumption, than they would be if they were built entirely using single precision. More recently, machine learning algorithms are using mixed precision computations where the lower precision uses an 8-bit format (typically with p between 3 and 5, as in Table 15.2), and the higher precision uses either *binary16* ($q = 11$) or *bfloat16* ($q = 8$) (although $p = 5$ and $q = 8$ would not satisfy the inequality $q \geq 2p$). See [MS+22], [NJ+22] for more details.

Speaking of the FMA, it is worth noting here that a correctly rounded mixed-precision FMA (for floating point numbers, not matrices) was introduced in 2011 in [BD+11]. Suppose that floating point numbers a and b have precision p, and recall that the fused multiply-add operation first computes the product $a \times b$, whose significand has length $2p$, and then adds c to this before rounding to the destination format. If this also has precision p, then many of the bits of $a \times b$ are lost in this process. The mixed-precision FMA instead retains all bits of $a \times b$ in a format with precision q, with $q \geq 2p$. This also allows the third operand c to have the same higher precision q, and then the addition of $a \times b$ and c can be done using precision q. See [MB+18, Sec. 7.8.2–7.8.4] for details.

Stochastic Rounding

The idea of stochastic rounding is that instead of rounding deterministically as dictated by the rounding mode in effect, the choice of whether to round a non-exact result up or down is made randomly. This is an old idea that has long been out of favor, as is evident by the fact that it is not mentioned by any version of the IEEE standard. However, with the advent of much lower precision floating point formats, it is becoming apparent that stochastic rounding may play an important role, and some microprocessors are now providing it as a hardware operation. Suppose that the exact result of an operation is x which is not a floating point number, so we must decide whether to round down to x_- or up to x_+. One variant of stochastic rounding chooses each with 50% probability, but a variant with better properties sets

$$\text{round}(x) = \begin{cases} x_- \text{ with probability } (x_+ - x)/(x_+ - x_-), \\ x_+ \text{ with probability } (x - x_-)/(x_+ - x_-). \end{cases}$$

One of the motivations for stochastic rounding is as follows. Consider an infinite sum (or series) such as

$$1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots$$

It is well known that this sum does not converge to a finite value, but that it diverges to ∞. However, if we write a program to compute it using floating point arithmetic, using *round to nearest*, then for sufficiently large n the term $1/n$ will be so small that adding it to the previous terms does not change the result, and the same will be true as n continues to increase, so we will always obtain convergence to a finite value. We say that the computation stagnates. On the other hand, using stochastic rounding, even adding very small terms to the previous terms must round up eventually.

Another motivation for stochastic rounding is illustrated by again considering the computation of the vector inner product $a \cdot b$. The standard rounding error analysis using *round to nearest*, due to Wilkinson, states that the worst-case rounding error at the end of the computation is, roughly speaking, of the order of $n\epsilon_{\text{mch}}$, where n is the length of the vectors. When we use double precision, with $\epsilon_{\text{mch}} \approx 10^{-16}$, then $n\epsilon_{\text{mch}}$ is much less than one for n as large as 10^{10} or more. However, with the rapid recent improvements in technology, it is now common for n to be much larger than in previous years, and if at the same time, we use lower precision, say *binary16* with $\epsilon_{\text{mch}} \approx 10^{-3}$, we typically find that $n\epsilon_{\text{mch}}$ is much bigger than one making the standard rounding error analysis useless. One way to compensate for this is to make a probabilistic analysis, which, even continuing to use *round to nearest*, can be expected to reduce the factor of n to $\sqrt{n}$. However, this is difficult to make rigorous. Using stochastic rounding, however, such a result can be shown to be true with a relatively straightforward analysis. See [CF+22] for more details.

Outlook for Low Precision Computing

Low precision floating point computing has become widespread very rapidly, as a consequence of the dramatic growth of machine learning. The use of 8-bit floating point, not even seriously contemplated 10 years ago, is already widespread, and it is not clear whether the P3109 working group will have the same kind of standardizing effect that p754 did nearly 40 years ago. Even 4-bit floating point is being used by some researchers [SW+20]. The stated rationale for introducing short floating point formats has been that they will be used primarily for applications in machine learning, where the argument is made that high or even modest accuracy in not generally needed [Dea20], but now that these machines are starting to become generally available at low cost, there is little doubt that they will be used in other applications too. The impact of this dramatic change in floating point hardware remains to be seen.

Chapter 16

Conclusion

Here is a summary of some of the most important ideas in this book.

• Floating point representation of numbers is ubiquitous in numerical computing, since fixed point numbers have very restricted ranges. Floating point uses exponential notation, storing a sign, an exponent, and a significand in each floating point word.

• The range of possible values for IEEE single format floating point numbers is from tiny (approximately 10^{-38}) to huge (approximately 10^{38}). In addition, there are the corresponding range of negative numbers, the subnormal numbers, and the special numbers ± 0 and $\pm\infty$. NaN is used for the result of invalid operations. Double format numbers have a much greater finite range.

• Floating point numbers are inherently accurate only to a certain number of bits or digits. The *precision* p is the number of bits in the significand, including the *hidden bit*, which is not stored. In the case of the IEEE single format (*binary32*), numbers have precision $p = 24$, corresponding to approximately 7 significant decimal digits, and in the case of the double format (binary64), precision $p = 53$, corresponding to approximately 16 significant decimal digits. Theorem 5.1 says that, when precision p is in use, the rounded value of a number x satisfies

$$\text{round}(x) = x(1+\delta), \quad \text{where} \quad |\delta| < 2^{-(p-1)},$$

with $|\delta| < 2^{-p}$ when the rounding mode is *round to nearest* (the default choice). The quantity $|\delta|$ is called the relative rounding error and its size depends only on the precision p of the floating point system and not on the size of x. The absolute rounding error $|x - \text{round}(x)|$ does depend on the size of x, since the gap between floating point numbers is larger for larger numbers. These results apply to normalized numbers. Subnormal numbers are less accurate. Because of its greater precision, the double format is generally preferred for most scientific computing applications.

• One floating point arithmetic operation is required, under the rules of the IEEE standard, to give the exact result rounded correctly using the relevant rounding mode and precision. Such a result is by definition accurate to 24 bits (about 7 digits) when the destination format is IEEE single, and to 53 bits (about 16 digits) when the destination format is IEEE double, unless the number is subnormal. Exceptional cases may yield a result equal to $\pm\infty$ or NaN.

• A sequence of floating point operations generally does not give correctly rounded exact results. Furthermore, one cannot expect the results to be accurate to 7 significant digits (or 16 digits when the double format is in use). Accuracy of computations is limited by the condition number of the problem being solved. Rule of Thumb 12.1 says that the number of significant digits in the computed results can be expected to be, at best, about 7 minus the base 10 logarithm of the condition number using the single format (or 16 minus the base 10 logarithm of the condition number if the double format is in use).

• A stable algorithm is one that solves a problem to approximately the accuracy predicted by Rule of Thumb 12.1. A poor choice of algorithm may give much worse results; in this case the algorithm is said to be unstable. This may happen because of cancellation or, more generally, because of intermediate steps that introduce ill conditioning.

For diving deeper into many technical aspects of floating point technology, see the recent survey paper [BJ+23] and the comprehensive treatise [MB+18].

Numerical Algorithms and Numerical Analysis

In this book, we have not even begun to discuss algorithms for the solution of nontrivial numerical problems, nor the analysis that underlies them. Many fine books, old and new, contain a wealth of information on these most classical of scientific subjects. Numerical algorithms and analysis form such a fundamental part of computer science and applied mathematics that Knuth, in the preface to his celebrated multivolume series *The Art of Computer Programming*, commented that his subject might be called "nonnumerical analysis" [Knu68]. He felt this was too negative, so he suggested instead "analysis of algorithms," a name that stuck.

We cannot list more than a tiny fraction of the many books on numerical algorithms and analysis, but we mention a few favorite books. Two good general undergraduate texts are [AG11] and [SM03], respectively oriented primarily towards computer science and mathematics students. For linear algebra, see [Dem97, Hig02, TB97]. For differential equations, see [TBD18]. For optimization, see [BV04] and [NW06].

Numerical analysis is, according to Trefethen [Tre97, p. 323], the study of algorithms for the problems of continuous mathematics—*not* just the study of rounding errors. We completely agree. As Trefethen says, "The central mission of numerical analysis is to compute quantities that are typically uncomputable, from an analytical point of view, and do it with lightning speed." Floating point computing is the workhorse that makes this possible.

Reliability Is Paramount

There is one thing that is even more important than lightning speed, and that is reliability. This applies to all kinds of computing and is an issue that receives an enormous amount of attention; a common refrain is: bugs are everywhere.

As Kahan says, speed should not be confused with throughput [Kah00]. Fast programs that break down occasionally and therefore require a lot of user interaction may be less useful than highly reliable, slower programs. Floating point hardware operations have become both very fast and, thanks in large part to the IEEE standard, very reliable. Although the computer industry has, by and large, been hugely supportive of the IEEE standard, it is not clear how long this will be the case. One concern is the recent proliferation of nonstandard half and even quarter-precision microprocessors, largely inspired by the massive computing requirements of machine learning.

We conclude by noting that catastrophic system failures because of difficulties arising from careless floating point programming *sometimes happen in the real world.* Perhaps the most dramatic example is the error that triggered the destruction of Ariane 5, the European Space Agency's billion-dollar rocket, in June 1996. Thirty-seven seconds after liftoff, a program tried to convert the rocket's horizontal velocity from a double format to a short integer format. The number in question was easily within the normalized range of the double floating point format, but was too big for the 16-bit short integer format. When the invalid operation occurred, the program, instead of taking some appropriate action, shut down the entire guidance system and the rocket self-destructed [Inq96].

In the modern world, many critical matters are dependent on complex computer programs, from air traffic control systems to heart machines. Many of these codes depend, in one way or another, on floating point computing.

Bibliography

[App88] *Apple Numerics Manual.* Addison-Wesley, Reading, MA, Menlo Park, CA, second edition, 1988.

[AG11] U. M. Ascher and C. Greif. *A First Course in Numerical Methods.* SIAM, Philadelphia, 2011.

[Bai] D.H. Bailey. High-precision software directory. https://www.davidhbailey.com/dhbsoftware/.

[BD+11] N. Brunie, F. de Dinechin and B. de Dinechin. A mixed-precision fused multiply and add. 2011 Conference Record of the Forty Fifth Asilomar Conference on Signals, Systems and Computers (ASILOMAR), Pacific Grove, CA, USA, 2011, pp. 165-169. doi: 10.1109/ACSSC.2011.6189977.

[BF+23] T. Bartels, V. Fisikopoulos and M. Weiser. Fast floating-point filters for robust predicates. *BIT Numerical Mathematics* (2023) 63, 31 pp. https://doi.org/10.1007/s10543-023-00975-x

[Bee17] N. Beebe. *The Mathematical-Function Computation Handbook*, Springer, 2017.

[BE+17] J. Bezanson, A. Edelman, S. Karpinski, V.B. Shah. Julia: A Fresh Approach to Numerical Computing, *SIAM Review* 59, pp. 65–98, 2017.

[Bha22] A. Bhattacharya. *The Man from the Future: The Visionary Life of John von Neumann*, Norton, 2022.

[BH+22] P. Blanchard, N.J. Higham, F. Lopez, T. Mary, S.Pranesh. Mixed precision block fused multiply-add: Error analysis and application to GPU tensor cores. *SIAM Journal on Scientific Computing* 42 (3), pp. C124–C141, 2020.

[BH+02] D. H. Bailey, Y. Hida, X.S. Li and B. Thompson. ARPREC: An arbitrary precision computation package, 2002. http://crd.lbl.gov/˜dhbailey/dhbpapers/arprec.pdf

[BJ+23] S. Boldo, C.-P. Jeannerod, G. Melquiond and J.-M. Muller. Floating-point arithmetic. *Acta Numerica* (2023), pp. 203–290.

[BV04] S. Boyd and L. Vandenberghe. *Convex Optimization.* Cambridge University Press, 2004.

[BZ11] R.P Brent and P. Zimmerman. *Modern Computer Arithmetic*, Cambridge University Press, 2011.

[Cha79] A. B. Chace. *The Rhind Mathematical Papyrus.* National Council of Teachers of Mathematics, Reston, VA, 1979. The cited quote is from Volume 1, pp. 48–49.

[Cla99] Arthur C. Clarke. *2001: A Space Odyssey.* New American Library, Penguin Putnam, New York, 1999. Based on a screenplay by Stanley Kubrick and Arthur C. Clarke, 1968.

[CW80] W. J. Cody, Jr. and W. Waite, *Software Manual for the Elementary Functions.* Prentice-Hall, 1980.

[Cod81] W. J. Cody. Analysis of proposals for the floating-point standard. *Computer*, 14(3):63–69, 1981.

[Coo81] J. T. Coonen. Underflow and the denormalized numbers. *Computer*, 14(3):75–87, 1981.

[CF+22] M. Croci, M. Fasi, N. J. Higham, T. Mary and M. Mikaitis. Stochastic rounding: implementation, error analysis and applications. *Roy. Soc. Open Sci.* 9: 211631. https://doi.org/10.1098/rsos.211631

[Dar98] J.D. Darcy. *Borneo: adding IEEE 754 support to Java*, M.S. thesis, University of California, 1998. http://www.jddarcy.org/Borneo/borneo.pdf

[Dea20] J. Dean. The deep learning revolution and its implications for computer architecture and chip design. In 2020 IEEE International Solid-State Circuits Conference (ISSCC), IEEE, pp. 8–14. arXiv:1911.05289

[Dem84] J. W. Demmel. Underflow and the reliability of numerical software. *SIAM J. Sci. Stat. Comput.*, 5:887–919, 1984.

[Dem87] J. W. Demmel. *On error analysis in arithmetic with varying relative precision.* In 8th Symposium on Computer Arithmetic (ARITH-8), 1987.

[Dem91] J. W. Demmel. On the odor of IEEE arithmetic. *NA Digest*, 91(39) Sept. 29, 1991. http://www.netlib.org/na-digest-html/91/v91n39.html#15

[Dem97] J. W. Demmel. *Applied Numerical Linear Algebra.* SIAM, Philadelphia, 1997.

[Din19] F. de Dinechin, L. Forget, J.-M. Muller and Y. Uguen. Posits: the good, the bad and the ugly. Conference for Next Generation Arithmetic 2019, DOI:10.1145/3316279.3316285.

[Dys12] G. Dyson. *Turing's Cathedral: The Origins of the Digital Universe*, Pantheon, 2012.

[DL94] J. W. Demmel and X. Li. Faster numerical algorithms via exception handling. *IEEE Trans. Comput.*, 43:983–992, 1994.

[Ede97] A. Edelman. The mathematics of the Pentium division bug. *SIAM Review*, 39:54–67, 1997.

[Ede94] A. Edelman. When is $x * (1/x) \neq 1$?, 1994. http://www-math.mit.edu/~edelman

[BL+04] F. Bornemann, D. Laurie, S. Wagon and J. Waldvogel. *The SIAM 100-Digit Challenge: A Study in High-Accuracy Numerical Computing*. With a foreword by D.H. Bailey. SIAM, 2004. https://epubs-siam-org/doi/book/10.1137/1.9780898717969

[FH+21] M. Fasi, N.J. Higham, M. Mikaitis, S. Pranesh S. 2021. Numerical behavior of NVIDIA tensor cores. *Peer J. Comput. Sci.* 7:e330 DOI 10.7717/peerj-cs.330

[FH+07] L. Fousse, G. Hanrot, V. Lefèvre, P. Pelissier and P. Zimmermann. MPFR: A multiple-precision binary floating-point library with correct rounding *ACM Transactions on Mathematical Software*, 33 (2), Article 13, 2007. https://doi.org/10.1145/1236463.1236468

[Gay90] D. M. Gay. Correctly rounded binary-decimal and decimal-binary conversions. Technical report, 1990, AT&T Bell Labs Numerical Analysis Manuscript 90-10. http://www.ampl.com/ampl/REFS/rounding.ps.gz

[Gol91] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computer Surveys*, 23:5–48, 1991.

[Gol95] D. Goldberg. *Computer Arithmetic*. Kaufmann, San Mateo, CA, second edition, 1995. Appendix in [HP95].

[Hau96] J. Hauser. Handling floating-point exceptions in numeric programs. *ACM Transactions on Programming Languages and Systems*, Vol. 18, No. 2, March 1996, pp. 139–174.

[HP95] J. L. Hennessy and D. L. Patterson. *Computer Architecture: A Quantitative Approach*. Kaufmann, San Mateo, CA, second edition, 1995. (Need to update to later edition.)

[HH00] D. J. Higham and N. J. Higham. *MATLAB Guide*. SIAM, Philadelphia, third edition, 2017.

[Hig02] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM, Philadelphia, second edition, 2002.

[Hig17] N.J. Higham. A multiprecision world. *SIAM News*, 2017. https://sinews.siam.org/Details-Page/a-multiprecision-world

[HM22] N.J. Higham and T. Mary. Mixed precision algorithms in numerical linear algebra. *Acta Numerica* 31: 347–414, 2022.

[Hou81] D. Hough. Applications of the proposed IEEE 754 standard for floating-point arithmetic. *Computer*, 14(3):70–74, 1981.

[Hou19] David G. Hough. The IEEE Standard 754: One for the History Books. *ACM Signum Newsletter*, 14:13-21, 2019. https://ieeemilestones.ethw.org/w/images/f/f2/Hough_one_for_the_history_books.pdf

[IEE85] IEEE 754-1985 - IEEE Standard for Binary Floating-Point Arithmetic, 1985. Reprinted in *SIGPLAN Notices* 22(2):9–25, 1987. https://ieeexplore.ieee.org/document/30711

[IEE87] IEEE 854-1987 - IEEE Standard for Radix-Independent Floating-Point Arithmetic, 1987. https://ieeexplore.ieee.org/document/27840

[IEE08] IEEE 754-2008 - IEEE Standard for Floating-Point Arithmetic, 2008. https://ieeexplore.ieee.org/document/4610935

[IEE19] IEEE 754-2019 - IEEE Standard for Floating-Point Arithmetic, 2019. https://ieeexplore.ieee.org/document/8766229

[IEE23] Milestones: IEEE Standard 754 for Floating Point Arithmetic. (J. Coonen, proposer) https://ieeemilestones.ethw.org/Milestones:IEEE_Standard_754_for_Floating_Point_Arithmetic

[InnZim23] V. Innocente and P. Zimmermann. Accuracy of mathematical functions in single, double, extended double and quadruple precision. https://inria.hal.science/hal-03141101.

[Inq96] Inquiry board traces Ariane 5 failure to overflow error. *SIAM News*, 29(8), Oct. 1996, pp. 1, 12, 13. http://www.siam.org/siamnews/general/ariane.htm

[ISO99] ISO/IEC 9899:1999 Standard for the C programming language (C99), 1999. http://www.iso.ch/. January 1999 draft available at http://anubis.dkuug.dk/JTC1/SC22/WG14/www/docs/n869/

[Jav] Java Numerics. http://math.nist.gov/javanumerics/

[Jea19] C.-P. Jeannerod. The relative accuracy of (x+y)*(x-y). *J. Comput. Appl. Math.* 369. https://doi.org/10.1016/j.cam.2019.112613

[Jul23] https://julialang.org/.

[Kah87] W. Kahan. Branch cuts for complex elementary functions or much ado about nothing's sign bit. *The State of the Art in Numerical Analysis*, pp. 165–188. Clarendon Press (1987).

[Kah96a] W. Kahan. The baleful effect of computer benchmarks upon applied mathematics, physics and chemistry, 1996. http://www.cs.berkeley.edu/~wkahan/ieee754status/baleful.ps

[Kah96b] W. Kahan. Lecture notes on the status of IEEE standard 754 for binary floating-point arithmetic, 1996. http://www.cs.berkeley.edu/~wkahan/ieee754status/ieee754.ps

[Kah97] W. Kahan. The John von Neumann lecture at the SIAM 45th annual meeting, 1997. http://www.cs.berkeley.edu/~wkahan/SIAMjvnl.ps

[Kah98] W. Kahan and J.D. Darcy. How Java's floating-point hurts everyone everywhere, 1998. http://www.cs.berkeley.edu/~wkahan/JAVAhurt.pdf

[Kah00] W. Kahan. Ruminations on the design of floating-point arithmetic, 2000. http://www.cs.nyu.edu/cs/faculty/overton/book/docs/KahanTalk.pdf

[Kah10] W. Kahan. Pete's unsung contribution to IEEE standard 754 for binary floating-point: A talk at a conference to celebrate G.W. "Pete" Stewart's 70th Birthday, 2010. https://people.eecs.berkeley.edu/ wkahan/19July10.pdf

[Kha13] G. Khanna. High-precision numerical simulations on a CUDA GPU: Kerr black hole tails. *J. Sci. Comput.* 56, 366–380, 2013. https://doi.org/10.1007/s10915-012-9679-3

[Knu68] D. E. Knuth. *The Art of Computer Programming, Volume* 1: *Fundamental Algorithms*. Addison-Wesley, Reading, MA, 1968.

[Knu98] D. E. Knuth. *The Art of Computer Programming, Volume* 2: *Seminumerical Algorithms*. Addison-Wesley, Reading, MA, third edition, 1998.

[MS+22] P. Micikevicius, D. Stosic et al. FP8 formats for deep learning. arXiv:2209.05433, 2022.

[MRC18] M. Metcalf, J. Reid and M. Cohen. *Modern Fortran Explained, Incorporating Fortran 2018.* Oxford University Press, Oxford, 2018.

[MHR80] N. Metropolis, J. Howlett, and G.-C. Rota, editors. *A History of Computing in the Twentieth Century*. Academic Press, New York, 1980.

[Mol18] C.B. Moler. *A brief history of MATLAB*. https://www.mathworks.com/company/newsletters/articles/a-brief-history-of-matlab.html).

[Mul97] J.-M. Muller. *Elementary Functions: Algorithms and Implementation*. Birkhaüser, Boston, Basel, Berlin, second edition, 2005.

[MB+18] J.-M. Muller, N. Brunie, F. de Dinechin, C.-P. Jeannerod, M. Joldes, V. Lefévre, G. Melquiond, N. Revol, S. Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser (Springer group), second edition, 2018.

[ND+18] S. Narang, G. Diamos, E. Elsen, P. Micikevicius, J. Alben, D. Garcia, B.Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, H.Wu. Mixed precision training. arXiv:1710.03740v3

[NJ+22] B. Noune, P. Jones, D. Justus, D. Masters, and C. Luschi. 8-bit numerical formats for deep neural networks. arXiv:2206.02915, 2022.

[NW06] J. Nocedal and S. J. Wright. *Numerical Optimization*. Springer, New York, second edition, 2006.

[PZ+23] S.P. Perez, Y. Zhang, J. Briggs, C. Blake, J. Levy-Kramer, P. Balanca, C. Luschi, S. Barlow, A. Fitzgibbon. Training and inference of large language models using 8-bit floating point. arXiv:2309.17224, 2023.

[PH97] D. L. Patterson and J. L. Hennessy. *Computer Organization and Design: the Hardware/Software Interface*. Kaufmann, San Mateo, CA, second edition, 1997. (Need to update edition and merge with earlier entry.)

[RD18] J. Riedy and J. Demmel. Augmented arithmetic operations proposed for IEEE-754 2018. 25th IEEE Symposium on Computer Arithmetic (ARITH 2018). https://ieeexplore.ieee.org/document/8464813

[Rob95] E. S. Roberts. *The Art and Science of C.* Addison-Wesley, Reading, MA, Menlo Park, CA, 1995.

[Rum] S. M. Rump. INTLAB: Interval Laboratory, a MATLAB toolbox for interval arithmetic. http://www.ti3.tu-harburg.de/rump/intlab/

[Rum10] S. Rump. Verification Methods: Rigorous results using floating-point arithmetic, *Acta Numerica*, pp. 287-449, Cambridge University Press, 2010.

[SW+20] X. Sun, N. Wang et al. Ultra-low precision 4-bit training of deep neural networks. NeurIPS 2020.

[Sev98] C. Severance. An interview with the old man of floating-point: Reminiscences elicited from William Kahan, 1998. http://www.cs.berkeley.edu/~wkahan/ieee754status/754story.html A condensed version appeared in *Computer*, 31:114–115, 1998.

[She97] J. R. Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete Comput. Geom.*, 18(3):305–363, 1997.

[Ste74] P. Sterbenz, *Floating Point Computation*, Prentice-Hall, Englewood Cliffs, NJ, 1974.

[SM03] E. Süli and D. Mayers. *An Introduction to Numerical Analysis.* Cambridge University Press (2003).

[TBD18] L. N. Trefethen, A. Birkisson and T. A. Driscoll. *Exploring ODEs.* SIAM, Philadelphia, 2018.

[TB97] L. N. Trefethen and D. Bau, III. *Numerical Linear Algebra.* SIAM, Philadelphia, 1997.

[Top22] Top 500 list, 2022. https://en.wikipedia.org/wiki/TOP500

[Tre97] L. N. Trefethen. The definition of numerical analysis. In [TB97], pp. 321–327.

[Tuc11] W. Tucker. *Validated Numerics: A Short Introduction to Rigorous Computations.* Princeton University Press, 2011.

[WK19] S. Wang and P. Kanwar. BFloat16: The secret to high performance on cloud TPUs, 2019. https: //cloud.google.com/blog/products/aimachine-learning/bfloat16-the-secretto-high-performance-on-cloud-tpus/.

[War94] F. Warshofsky. *The Patent Wars: The Battle to Own the World's Technology.* Wiley, 1994. Reviewed in the Harvard Journal of Law & Technology Volume 9, Number 1 Winter 1996, http://jolt.law.harvard.edu/articles/pdf/v09/09HarvJLTech219.pdf.

[Web96] *Webster's New World College Dictionary.* Macmillan, New York, 1996.

[Wil63] J. H. Wilkinson. *Rounding Errors in Algebraic Processes.* Prentice-Hall, 1963.

[Wil23] J. H. Wilkinson. *Rounding Errors in Algebraic Processes.* Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 2023. With a foreword by Nicholas J. Higham.

[Wil71] J. H. Wilkinson. *Modern Error Analysis.* SIAM Review 13, pp. 548–568, 1971.

[Wil85] M. R. Williams. *A History of Computing Technology.* Prentice-Hall, Englewood Cliffs, NJ, 1985.

[Wil98] M. V. Wilkes. A revisionist account of early language development. *Computer*, 31:22–26, 1998.

[WW92] D. Weber-Wulff. Rounding error changes parliament makeup. *The Risks Digest*, 13(37), 1992. http://catless.ncl.ac.uk/Risks/13.37.html#subj4

[Zus93] K. Zuse. *The Computer—My Life.* Springer-Verlag, Berlin, New York, 1993.