Chapter 2
Floating Point Arithmetic

From 1946–1948 *a great deal of quite detailed coding was done.*
The subroutines for floating-point arithmetic were . . .
produced by Alway and myself in 1947 *. . .*
They were almost certainly the earliest floating-point subroutines.

— J. H. WILKINSON, *Turing's Work at the National Physical Laboratory . . .* (1980)

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.

— MICHAEL L. OVERTON, *Numerical Computing with IEEE Floating Point Arithmetic* (2001)

Arithmetic on Cray computers is interesting because it is driven by a
motivation for the highest possible floating-point performance . . .
Addition on Cray computers does not have a guard digit,
and multiplication is even less accurate than addition . . .
At least Cray computers serve to keep numerical analysts on their toes!

— DAVID GOLDBERG[5], *Computer Arithmetic* (1996)

It is rather conventional to obtain a "realistic" estimate
of the possible overall error due to k *roundoffs,*
when k *is fairly large,*
by replacing k *by* $\sqrt{k}$ *in an expression for* (*or an estimate of*)
the maximum *resultant error.*

— F. B. HILDEBRAND, *Introduction to Numerical Analysis* (1974)

[5] In Hennessy and Patterson [562, 1996, §A.12].

2.1. Floating Point Number System

A floating point number system $F \subset \mathbb{R}$ is a subset of the real numbers whose elements have the form

$$y = \pm m \times \beta^{e-t}. \tag{2.1}$$

The system F is characterized by four integer parameters:

- the *base* β (sometimes called the *radix*),
- the *precision* t, and
- the *exponent range* $e_{\min} \le e \le e_{\max}$.

The *significand*[6] m is an integer satisfying $0 \le m \le \beta^t - 1$. To ensure a unique representation for each nonzero $y \in F$ it is assumed that $m \ge \beta^{t-1}$ if $y \ne 0$, so that the system is *normalized*. The number 0 is a special case in that it does not have a normalized representation. The *range* of the nonzero floating point numbers in F is given by $\beta^{e_{\min}-1} \le |y| \le \beta^{e_{\max}}(1 - \beta^{-t})$. Values of the parameters for some machines of historical interest are given in Table 2.1 (the unit roundoff u is defined on page 38).

Note that an alternative (and more common) way of expressing y is

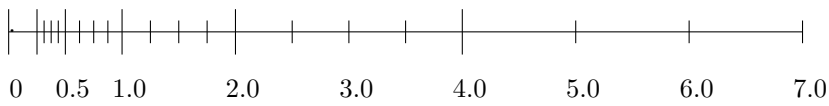
$$y = \pm\beta^e \left(\frac{d_1}{\beta} + \frac{d_2}{\beta^2} + \cdots + \frac{d_t}{\beta^t} \right) = \pm\beta^e \times .d_1 d_2 \ldots d_t, \tag{2.2}$$

where each digit d_i satisfies $0 \le d_i \le \beta - 1$, and $d_1 \ne 0$ for normalized numbers. We prefer the more concise representation (2.1), which we usually find easier to work with. This "nonpositional" representation has pedagogical advantages, being entirely integer based and therefore simpler to grasp. In the representation (2.2), d_1 is called the *most significant digit* and d_t the *least significant digit*.

It is important to realize that the floating point numbers are not equally spaced. If $\beta = 2$, $t = 3$, $e_{\min} = -1$, and $e_{\max} = 3$ then the nonnegative floating point numbers are

0, 0.25, 0.3125, 0.3750, 0.4375, 0.5, 0.625, 0.750, 0.875,
1.0, 1.25, 1.50, 1.75, 2.0, 2.5, 3.0, 3.5, 4.0, 5.0, 6.0, 7.0.

They can be represented pictorially as follows:



[6] The significand is often called the mantissa, but strictly speaking the term mantissa should be used only in conjunction with logarithms.

Table 2.1. *Floating point arithmetic parameters.*

Machine and arithmetic	β	t	$e_{\min}$	$e_{\max}$	u
Cray-1 single	2	48	−8192	8191	4×10^{-15}
Cray-1 double	2	96	−8192	8191	1×10^{-29}
DEC VAX G format, double	2	53	−1023	1023	1×10^{-16}
DEC VAX D format, double	2	56	−127	127	1×10^{-17}
HP 28 and 48G calculators	10	12	−499	499	5×10^{-12}
IBM 3090 single	16	6	−64	63	5×10^{-7}
IBM 3090 double	16	14	−64	63	1×10^{-16}
IBM 3090 extended	16	28	−64	63	2×10^{-33}
IEEE single	2	24	−125	128	6×10^{-8}
IEEE double	2	53	−1021	1024	1×10^{-16}
IEEE extended (typical)	2	64	−16381	16384	5×10^{-20}

Notice that the spacing of the floating point numbers jumps by a factor 2 at each power of 2. The spacing can be characterized in terms of *machine epsilon*, which is the distance ϵ_M from 1.0 to the next larger floating point number. Clearly, $\epsilon_M = \beta^{1-t}$, and this is the spacing of the floating point numbers between 1.0 and β; the spacing of the numbers between 1.0 and $1/\beta$ is $\beta^{-t} = \epsilon_M/\beta$. The spacing at an arbitrary $x \in F$ is estimated by the following lemma.

Lemma 2.1. *The spacing between a normalized floating point number x and an adjacent normalized floating point number is at least $\beta^{-1}\epsilon_M|x|$ and at most $\epsilon_M|x|$.*

Proof. See Problem 2.2. □

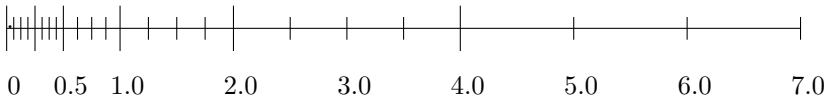
The system F can be extended by including *subnormal numbers* (also known as *denormalized numbers*), which, in the notation of (2.1), are the numbers

$$y = \pm m \times \beta^{e_{\min}-t}, \qquad 0 < m < \beta^{t-1},$$

which have the minimum exponent and are not normalized (equivalently, in (2.2) $e = e_{\min}$ and the most significant digit $d_1 = 0$). The subnormal numbers have fewer digits of precision than the normalized numbers. The smallest positive normalized floating point number is $\lambda = \beta^{e_{\min}-1}$ while the smallest positive subnormal number is $\mu = \beta^{e_{\min}-t} = \lambda\epsilon_M$. The subnormal numbers fill the gap between λ and 0 and are equally spaced, with spacing the same as that of the numbers of F between λ and $\beta\lambda$, namely $\lambda\epsilon_M = \beta^{e_{\min}-t}$. For example, in the system illustrated above with $\beta = 2$, $t = 3$, $e_{\min} = -1$, and $e_{\max} = 3$, we have $\lambda = 2^{-2}$ and $\mu = 2^{-4}$, the subnormal numbers are

0.0625, 0.125, 0.1875,

and the complete number system can be represented as



Let $G \subset \mathbb{R}$ denote all real numbers of the form (2.1) with no restriction on the exponent e. If $x \in \mathbb{R}$ then $fl(x)$ denotes an element of G nearest to x, and the transformation $x \to fl(x)$ is called *rounding*. There are several ways to break ties when x is equidistant from two floating point numbers, including taking $fl(x)$ to be the number of larger magnitude (round away from zero) or the one with an even last digit d_t (round to even); the latter rule enjoys impeccable statistics [160, 1973]. For more on tie-breaking strategies see the Notes and References. Note that rounding is monotonic: $x \geq y$ implies $fl(x) \geq fl(y)$.

Although we have defined fl as a mapping onto G, we are only interested in the cases where it produces a result in F. We say that $fl(x)$ *overflows* if $|fl(x)| > \max\{\,|y| : y \in F\,\}$ and *underflows* if $0 < |fl(x)| < \min\{\,|y| : 0 \neq y \in F\,\}$. When subnormal numbers are included in F, underflow is said to be *gradual*.

The following result shows that every real number x lying in the range of F can be approximated by an element of F with a relative error no larger than $u = \frac{1}{2}\beta^{1-t}$. The quantity u is called the *unit roundoff*. It is the most useful quantity associated with F and is ubiquitous in the world of rounding error analysis.

Theorem 2.2. *If $x \in \mathbb{R}$ lies in the range of F then*

$$fl(x) = x(1+\delta), \qquad |\delta| < u. \tag{2.3}$$

Proof. We can assume that $x > 0$. Writing the real number x in the form

$$x = \mu \times \beta^{e-t}, \qquad \beta^{t-1} \leq \mu < \beta^t,$$

we see that x lies between the adjacent floating point numbers $y_1 = \lfloor \mu \rfloor \beta^{e-t}$ and $y_2 = \lceil \mu \rceil \beta^{e-t}$. (Strictly, if $\lceil \mu \rceil = \beta^t$ then the floating point representation (2.1) of y_2 is $\lceil \mu \rceil / \beta \cdot \beta^{e-t+1}$.) Thus $fl(x) = y_1$ or y_2 and we have

$$|fl(x) - x| \leq \frac{|y_2 - y_1|}{2} \leq \frac{\beta^{e-t}}{2}.$$

Hence

$$\left| \frac{fl(x) - x}{x} \right| \leq \frac{\frac{1}{2}\beta^{e-t}}{\mu \times \beta^{e-t}} \leq \frac{1}{2}\beta^{1-t} = u.$$

The last inequality is strict unless $\mu = \beta^{t-1}$, in which case $x = fl(x)$, and hence the inequality of the theorem is strict. ∎

Theorem 2.2 says that $fl(x)$ is equal to x multiplied by a factor very close to 1. The representation $1 + \delta$ for the factor is the standard choice, but it is not the only possibility. For example, we could write the factor as e^{α}, with a bound on $|\alpha|$ a little less than u (cf. the rp notation in §3.4).

The following modified version of this theorem can also be useful.

Theorem 2.3. *If $x \in \mathbb{R}$ lies in the range of F then*

$$fl(x) = \frac{x}{1+\delta}, \qquad |\delta| \le u.$$

Proof. See Problem 2.4. □

The widely used IEEE standard arithmetic (described in §2.3) has $\beta = 2$ and supports two precisions. Single precision has $t = 24$, $e_{\min} = -125$, $e_{\max} = 128$, and $u = 2^{-24} \approx 5.96 \times 10^{-8}$. Double precision has $t = 53$, $e_{\min} = -1021$, $e_{\max} = 1024$, and $u = 2^{-53} \approx 1.11 \times 10^{-16}$. IEEE arithmetic uses round to even.

It is easy to see that

$$x = \left(\beta^{t-1} + \frac{1}{2}\right) \times \beta^e \quad \Rightarrow \quad \left|\frac{fl(x) - x}{x}\right| \approx \frac{1}{2}\beta^{1-t},$$
$$x = \left(\beta^{t} - \frac{1}{2}\right) \times \beta^e \quad \Rightarrow \quad \left|\frac{fl(x) - x}{x}\right| \approx \frac{1}{2}\beta^{-t}.$$

Hence, while the relative error in representing x is bounded by $\frac{1}{2}\beta^{1-t}$ (as it must be, by Theorem 2.2), the relative error varies with x by as much as a factor β. This phenomenon is called *wobbling precision* and is one of the reasons why small bases (in particular, $\beta = 2$) are favoured. The effect of wobbling precision is clearly displayed in Figure 2.1, which plots machine numbers x versus the relative distance from x to the next larger machine number, for $1 \le x \le 16$ in IEEE single precision arithmetic. In this plot, the relative distances range from about $2^{-23} \approx 1.19 \times 10^{-7}$ just to the right of a power of 2 to about $2^{-24} \approx 5.96 \times 10^{-8}$ just to the left of a power of 2 (see Lemma 2.1).

The notion of *ulp*, or "unit in last place", is sometimes used when describing the accuracy of a floating point result. One ulp of the normalized floating point number $y = \pm\beta^e \times .d_1 d_2 \ldots d_t$ is $\mathrm{ulp}(y) = \beta^e \times .00\ldots01 = \beta^{e-t}$. If x is any real number we can say that y and x agree to $|y-x|/\mathrm{ulp}(y)$ ulps in y. This measure of accuracy "wobbles" when y changes from a power of β to the next smaller floating point number, since $\mathrm{ulp}(y)$ decreases by a factor β.

In MATLAB the permanent variable `eps` represents the machine epsilon, ϵ_M (not the unit roundoff as is sometimes thought), while `realmax` and `realmin` represent the largest positive and smallest positive normalized floating point number, respectively. On a machine using IEEE standard double precision arithmetic MATLAB returns

```
>> eps
ans =
  2.2204e-016

>> realmax
ans =
  1.7977e+308

>> realmin
ans =
  2.2251e-308
```

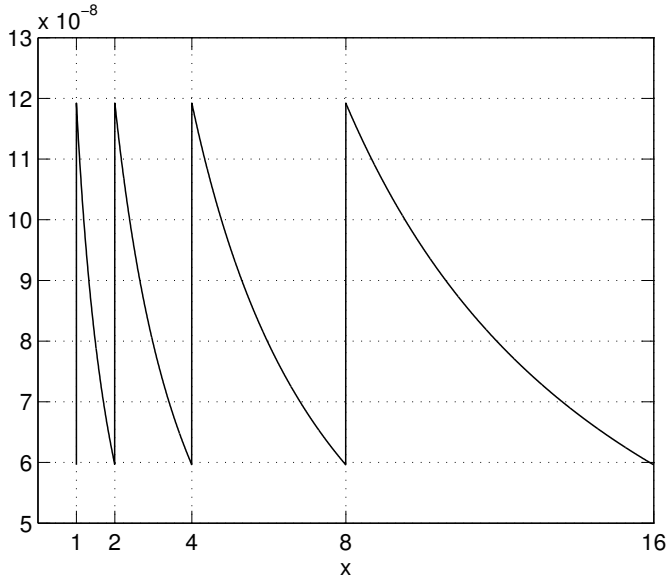


Figure 2.1. *Relative distance from x to the next larger machine number ($\beta = 2$, $t = 24$), displaying wobbling precision.*

2.2. Model of Arithmetic

To carry out rounding error analysis of an algorithm we need to make some assumptions about the accuracy of the basic arithmetic operations. The most common assumptions are embodied in the following model, in which $x, y \in F$:

STANDARD MODEL

$$fl(x \operatorname{op} y) = (x \operatorname{op} y)(1+\delta), \qquad |\delta| \le u, \quad \operatorname{op} = +, -, *, /. \tag{2.4}$$

It is normal to assume that (2.4) holds also for the square root operation.

Note that now we are using $fl(\cdot)$ with an argument that is an arithmetic expression to denote the computed value of that expression. The model says that the computed value of $x \operatorname{op} y$ is "as good as" the rounded exact answer, in the sense that the relative error bound is the same in both cases. However, the model does not require that $\delta = 0$ when $x \operatorname{op} y \in F$—a condition which obviously does hold for the rounded exact answer—so the model does not capture all the features we might require of floating point arithmetic. This model is valid for most computers, and, in particular, holds for IEEE standard arithmetic. Cases in which the model is not valid are described in §2.4.

The following modification of (2.4) can also be used (cf. Theorem 2.3):

$$fl(x \operatorname{op} y) = \frac{x \operatorname{op} y}{1+\delta}, \qquad |\delta| \le u. \tag{2.5}$$

All the error analysis in this book is carried out under the model (2.4), sometimes making use of (2.5). Most results proved using the standard model remain true with the weaker model (2.6) described below, possibly subject to slight increases in the constants. We identify problems for which the choice of model significantly affects the results that can be proved.

2.3. IEEE Arithmetic

IEEE standard 754, published in 1985 [655, 1985], defines a binary floating point arithmetic system. It is the result of several years' work by a working group of a subcommittee of the IEEE Computer Society Computer Standards Committee.

Among the design principles of the standard were that it should encourage experts to develop robust, efficient, and portable numerical programs, enable the handling of arithmetic exceptions, and provide for the development of transcendental functions and very high precision arithmetic.

The standard specifies floating point number formats, the results of the basic floating point operations and comparisons, rounding modes, floating point exceptions and their handling, and conversion between different arithmetic formats. Square root is included as a basic operation. The standard says nothing about exponentiation or transcendental functions such as exp and cos.

Two main floating point formats are defined:

Type	Size	Significand	Exponent	Unit roundoff	Range
Single	32 bits	23+1 bits	8 bits	$2^{-24} \approx 5.96 \times 10^{-8}$	$10^{\pm 38}$
Double	64 bits	52+1 bits	11 bits	$2^{-53} \approx 1.11 \times 10^{-16}$	$10^{\pm 308}$

In both formats one bit is reserved as a sign bit. Since the floating point numbers are normalized, the most significant bit is always 1 and is not stored (except for the denormalized numbers described below). This *hidden bit* accounts for the "+1" in the table.

The standard specifies that all arithmetic operations are to be performed as if they were first calculated to infinite precision and then rounded according to one of four modes. The default rounding mode is to round to the nearest representable number, with rounding to even (zero least significant bit) in the case of a tie. With this default mode, the model (2.4) is obviously satisfied. Note that computing with a single guard bit (see §2.4) will not always give the same answer as computing the exact result and then rounding. But the use of a second guard bit and a third *sticky* bit (the logical OR of all succeeding bits) enables the rounded exact result to be computed. Rounding to plus or minus infinity is also supported by the standard; this facilitates the implementation of interval arithmetic. The fourth supported mode is rounding to zero (truncation, or chopping).

IEEE arithmetic is a closed system: every arithmetic operation produces a result, whether it is mathematically expected or not, and exceptional operations raise a signal. The default results are shown in Table 2.2. The default response to an exception is to set a flag and continue, but it is also possible to take a trap (pass control to a trap handler).

A NaN is a special bit pattern that cannot be generated in the course of unexceptional operations because it has a reserved exponent field. Since the significand

Table 2.2. *IEEE arithmetic exceptions and default results.*

Exception type	Example	Default result
Invalid operation	$0/0$, $0 \times \infty$, $\sqrt{-1}$	NaN (Not a Number)
Overflow		$\pm\infty$
Divide by zero	Finite nonzero/0	$\pm\infty$
Underflow		Subnormal numbers
Inexact	Whenever $fl(x \mathrm{\ op\ } y) \neq x \mathrm{\ op\ } y$	Correctly rounded result

is arbitrary, subject to being nonzero, a NaN can have something about its provenance encoded in it, and this information can be used for retrospective diagnostics. A NaN is generated by operations such as $0/0$, $0 \times \infty$, ∞/∞, $(+\infty) + (-\infty)$, and $\sqrt{-1}$. One creative use of the NaN is to denote uninitialized or missing data. Arithmetic operations involving a NaN return a NaN as the answer. A NaN compares as unordered and unequal with everything including itself (a NaN can be tested with the predicate $x \neq x$ or with the IEEE recommended function `isnan`, if provided).

Zero is represented by a zero exponent field and a zero significand, with the sign bit providing distinct representations for $+0$ and -0. The standard defines comparisons so that $+0 = -0$. Signed zeros provide an elegant way to handle branch cuts in complex arithmetic; for details, see Kahan [694, 1987].

The infinity symbol is represented by a zero significand and the same exponent field as a NaN; the sign bit distinguishes between $\pm\infty$. The infinity symbol obeys the usual mathematical conventions regarding infinity, such as $\infty + \infty = \infty$, $(-1) \times \infty = -\infty$, and $(\text{finite})/\infty = 0$.

The standard requires subnormal numbers to be represented, instead of being flushed to zero as in many systems. This support of gradual underflow makes it easier to write reliable numerical software; see Demmel [308, 1984].

The standard may be implemented in hardware or software. The first hardware implementation was the Intel 8087 floating point coprocessor, which was produced in 1981 and implements an early draft of the standard (the 8087 very nearly conforms to the present standard). This chip, together with its bigger and more recent brothers the Intel 80287, 80387, 80486 and the Pentium series, is used in IBM PC compatibles (the 80486DX and Pentiums are general-purpose chips that incorporate a floating point coprocessor). Virtually all modern processors implement IEEE arithmetic.

The IEEE standard defines minimum requirements for two extended number formats: *single extended* and *double extended*. The double extended format has at least 79 bits, with at least 63 bits in the significand and at least 15 in the exponent; it therefore surpasses the double format in both precision and range, having unit roundoff $u \leq 5.42 \times 10^{-20}$ and range at least $10^{\pm 4932}$. The purpose of the extended precision formats is not to provide for higher precision computation per se, but to enable double precision results to be computed more reliably (by avoiding intermediate overflow and underflow) and more accurately (by reducing the effect of cancellation) than would otherwise be possible. In particular, extended

precision makes it easier to write accurate routines to evaluate the elementary functions, as explained by Hough [640, 1981].

A double extended format of 80 bits is supported by the Intel chips mentioned above and the Motorola 680x0 chips (used on early Apple Macintoshes); these chips, in fact, normally do *all* their floating point arithmetic in 80-bit arithmetic (even for arguments of the single or double format). However, double extended is not supported by Sun SPARCstations or machines that use the PowerPC or DEC Alpha chips. Furthermore, the extended format (when available) is supported little by compilers and packages such as Mathematica and Maple. Kahan [698, 1994] notes that "What you do not use, you are destined to lose", and encourages the development of benchmarks to measure accuracy and related attributes. He also explains that

> For now the 10-byte Extended format is a tolerable compromise between the value of extra-precise arithmetic and the price of implementing it to run fast; very soon two more bytes of precision will become tolerable, and ultimately a 16-byte format ... That kind of gradual evolution towards wider precision was already in view when IEEE Standard 754 for Floating-Point Arithmetic was framed.

A possible side effect of the use of an extended format is the phenomenon of *double rounding*, whereby a result computed "as if to infinite precision" (as specified by the standard) is rounded first to the extended format and then to the destination format. Double rounding (which is allowed by the standard) can give a different result from that obtained by rounding directly to the destination format, and so can lead to subtle differences between the results obtained with different implementations of IEEE arithmetic (see Problem 2.9).

An IEEE Standard 854, which generalizes the binary standard 754, was published in 1987 [656, 1987]. It is a standard for floating point arithmetic that is independent of word length and base (although in fact only bases 2 and 10 are provided in the standard, since the drafting committee "could find no valid technical reason for allowing other radices, and found several reasons for not allowing them" [251, 1988]). Base 10 IEEE 854 arithmetic is implemented in the HP-71B calculator.

2.4. Aberrant Arithmetics

In the past, not all computer floating point arithmetics adhered to the model (2.4). The most common reason for noncompliance with the model was that the arithmetic lacked a guard digit in subtraction. The role of a guard digit is easily explained with a simple example.

Consider a floating point arithmetic system with base $\beta = 2$ and $t = 3$ digits in the significand. Subtracting from 1.0 the next smaller floating number, we have, in binary notation,

$$\begin{array}{ll} 2^1 \times 0.100 - & \\ 2^0 \times 0.111 & \longrightarrow \quad \begin{array}{l} 2^1 \times 0.100 - \\ 2^1 \times 0.0111 \\ \hline 2^1 \times 0.0001 = 2^{-2} \times 0.100 \end{array} \end{array}$$

Notice that to do the subtraction we had to line up the binary points, thereby unnormalizing the second number and using, temporarily, a fourth digit in the significand, known as a *guard digit*. Some old machines did not have a guard digit. Without a guard digit in our example we would compute as follows, assuming the extra digits are simply discarded:

$$\begin{array}{ll} 2^1 \times 0.100 - & \qquad 2^1 \times 0.100 - \\ 2^0 \times 0.111 & \longrightarrow \qquad 2^1 \times 0.011 \quad \text{(last digit dropped)} \\ & \qquad \overline{} \\ & \qquad 2^1 \times 0.001 = 2^{-1} \times 0.100 \end{array}$$

The computed answer is too big by a factor 2 and so has relative error 1! For machines without a guard digit it is not true that

$$fl(x \pm y) = (x \pm y)(1+\delta), \qquad |\delta| \le u,$$

but it is true that

$$fl(x \pm y) = x(1+\alpha) \pm y(1+\beta), \qquad \alpha\beta = 0, \quad |\alpha| + |\beta| \le u.$$

Our model of floating point arithmetic becomes

NO GUARD DIGIT MODEL

$$fl(x \pm y) = x(1+\alpha) \pm y(1+\beta), \qquad |\alpha|, |\beta| \le u, \tag{2.6a}$$

$$fl(x \operatorname{op} y) = (x \operatorname{op} y)(1+\delta), \qquad |\delta| \le u, \quad \operatorname{op} = *, /, \tag{2.6b}$$

where we have stated a weaker condition on α and β that is generally easier to work with.

Notable examples of machines that lacked guard digits are several models of Cray computers (Cray 1, 2, X-MP, Y-MP, and C90). On these computers subtracting any power of 2 from the next smaller floating point number gives an answer that is either a factor of 2 too large (as in the example above—e.g., Cray X-MP or Y-MP) or is zero (Cray 2). In 1992 Cray announced that it would produce systems that use IEEE standard double precision arithmetic by 1995.

The lack of a guard digit is a serious drawback. It causes many algorithms that would otherwise work perfectly to fail some of the time, including Kahan's version of Heron's formula in the next section and compensated summation (see §4.3). Throughout the book we assume a guard digit is used, unless otherwise stated.

Kahan has made these interesting historical comments about guard digits [696, 1990]:

> CRAYs are not the first machines to compute differences blighted by lack of a guard digit. The earliest IBM '360s, from 1964 to 1967, subtracted and multiplied without a hexadecimal guard digit until SHARE, the IBM mainframe user group, discovered why the consequential anomalies were intolerable and so compelled a guard digit to be retrofitted. The earliest Hewlett-Packard financial calculator, the HP-80, had a similar problem. Even now, many a calculator (but not Hewlett-Packard's) lacks a guard digit.

2.5. Exact Subtraction

It is an interesting and sometimes important fact that floating point subtraction is exact if the numbers being subtracted are sufficiently close. The following result about exact subtraction holds for any base β.

Theorem 2.4 (Ferguson). *If x and y are floating point numbers for which $e(x-y) \le \min(e(x), e(y))$, where $e(x)$ denotes the exponent of x in its normalized floating point representation, then $fl(x-y) = x-y$ (assuming $x-y$ does not underflow).*

Proof. From the condition of the theorem the exponents of x and y differ by at most 1. If the exponents are the same then $fl(x-y)$ is computed exactly, so suppose the exponents differ by 1, which can happen only when x and y have the same sign. Scale and interchange x and y if necessary so that $\beta^{-1} \le y < 1 \le x < \beta$, where β is the base. Now x is represented in base β as $x_1.x_2 \dots x_t$ and the exact difference $z = x - y$ is of the form

$$\begin{array}{l} x_1.x_2 \dots x_t - \\ \quad 0.y_1 \dots y_{t-1} y_t \\ \hline \\ z_1.z_2 \dots z_t z_{t+1} \end{array}$$

But $e(x-y) \le e(y)$ and $y < 1$, so $z_1 = 0$. The algorithm for computing z forms $z_1.z_2 \dots z_{t+1}$ and then rounds to t digits; since z has at most t significant digits this rounding introduces no error, and thus z is computed exactly. ∎

The next result is a corollary of the previous one but is more well known. It is worth stating as a separate theorem because the conditions of the theorem are so elegant and easy to check (being independent of the base), and because this weaker theorem is sufficient for many purposes.

Theorem 2.5 (Sterbenz). *If x and y are floating point numbers with $y/2 \le x \le 2y$ then $fl(x-y) = x-y$ (assuming $x-y$ does not underflow).* ∎

With gradual underflow, the condition "assuming $x-y$ does not underflow" can be removed from Theorems 2.4 and 2.5 (see Problem 2.19).

Theorem 2.5 is vital in proving that certain special algorithms work. A good example involves Heron's formula for the area A of a triangle with sides of length a, b, and c:

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

This formula is inaccurate for needle-shaped triangles: if $a \approx b+c$ then $s \approx a$ and the term $s-a$ suffers severe cancellation. A way around this difficulty, devised by Kahan, is to rename a, b, and c so that $a \ge b \ge c$ and then evaluate

$$A = \frac{1}{4}\sqrt{\big(a+(b+c)\big)\big(c-(a-b)\big)\big(c+(a-b)\big)\big(a+(b-c)\big)}. \tag{2.7}$$

The parentheses are essential! Kahan has shown that this formula gives the area with a relative error bounded by a modest multiple of the unit roundoff [496, 1991, Thm. 3], [692, 1983], [701, 1997], [702, 2001] (see Problem 2.23). If a guard digit is not used in subtraction, however, the computed result can be very inaccurate.

2.6. Fused Multiply-Add Operation

Some computers have a *fused multiply-add* (FMA) operation that enables a floating point multiplication followed by an addition or subtraction, $x * y + z$ or $x * y - z$, to be performed as though it were a single floating point operation. An FMA operation therefore commits just one rounding error:

$$fl(x \pm y * z) = (x \pm y * z)(1 + \delta), \qquad |\delta| \leq u.$$

The Intel IA-64 architecture, as first implemented on the Intel Itanium chip, has an FMA instruction [273, 1999], as did the IBM RISC System/6000 and IBM Power PC before it. The Itanium's FMA instruction enables a multiply and an addition to be performed in the same number of cycles as one multiplication or addition, so it is advantageous for speed as well as accuracy.

An FMA is a welcome capability in that it enables the number of rounding errors in many algorithms to be approximately halved. Indeed, by using FMAs an inner product $x^T y$ between two n-vectors can be computed with just n rounding errors instead of the usual $2n - 1$, and any algorithm whose dominant operation is an inner product benefits accordingly.

Opportunities to exploit an FMA are frequent. Consider, for example, Newton's method for solving $f(x) = a - 1/x = 0$. The method is

$$\begin{aligned} x_{k+1} &= x_k - \frac{f(x_k)}{f'(x_k)} = x_k - \frac{a - 1/x_k}{x_k^{-2}} \\ &= x_k + (1 - x_k a)x_k, \end{aligned}$$

and its quadratic convergence can be seen from $1 - x_{k+1}a = (1 - x_k a)^2$. This method was used on early computers to implement reciprocation in terms of multiplication and thence division as $a/b = a * (1/b)$ [551, 1946], [1222, 1997], and this technique is still in use [714, 1997]. Since the computation of x_{k+1} from x_k can be expressed as two multiply-adds, the method is attractive when an FMA is available; an FMA also has the advantage of enabling a correctly rounded quotient to be achieved more easily [562, 1996, §A.7]. The floating point divide algorithms for the IA-64 architecture use this Newton-based approach [273, 1999].

An FMA makes it possible to compute an exact representation of the product of two floating point numbers x and y: by computing $\widehat{a} = fl(xy)$ and $\widehat{b} = fl(xy - \widehat{a})$ with two FMAs, $\widehat{a} + \widehat{b} \equiv xy$ (see Problem 2.26). Furthermore, clever use of an FMA enables highly accurate computation of, for example, the determinant of a 2×2 matrix (see Problem 2.27).

However, the presence of an FMA introduces some tricky programming language issues [700, 1996]. If a programmer writes `a*d + c*b` how is this expression to be compiled for a machine with an FMA? There are three possibilities—two using the FMA and one without it—and they can all yield different answers. An example of where this question is particularly important is the product of complex numbers

$$(x + iy)^*(x + iy) = x^2 + y^2 + i(xy - yx).$$

The product is obviously real, and the right-hand side evaluates as real in IEEE arithmetic, but if an FMA is employed in evaluating $xy - yx$ then a nonreal result will generally be obtained.

In the course of solving the quadratic $ax^2 - 2bx + c = 0$ for x, the expression $\sqrt{b^2 - ac}$ must be computed. Can the discriminant under the square root evaluate to a negative number when $b^2 \geq ac$? In correctly rounded arithmetic the answer is no: the monotonicity of rounding implies $fl(b^2) - fl(ac) \geq 0$ and the floating point subtraction of these incurs a small relative error and hence produces a nonnegative result. However, evaluation as $fl(fl(b^2) - ac)$ using an FMA can produce a negative result (for example, if $b^2 = ac$ and $fl(b^2) < b^2$).

In conclusion, as Kahan [700, 1996] puts it, "[FMAs] should not be used indiscriminately". Unfortunately, compiler writers, in their efforts to obtain maximum performance, may not give programmers the capability to inhibit FMAs in those subexpressions where their use can have undesirable effects.

2.7. Choice of Base and Distribution of Numbers

What base β is best for a floating point number system? Most modern computers use base 2. Most hand-held calculators use base 10, since it makes the calculator easier for the user to understand (how would you explain to a naive user that 0.1 is not exactly representable on a base 2 calculator?). IBM mainframes traditionally have used base 16. Even base 3 has been tried—in an experimental machine called SETUN, built at Moscow State University in the late 1950s [1208, 1960].

Several criteria can be used to guide the choice of base. One is the impact of wobbling precision: as we saw at the end of §2.1, the spread of representation errors is smallest for small bases. Another possibility is to measure the worst-case representation error or the mean square representation error. The latter quantity depends on the assumed distribution of the numbers that are represented. Brent [160, 1973] shows that for the logarithmic distribution the worst-case error and the mean square error are both minimal for (normalized) base 2, provided that the most significant bit is not stored explicitly.

The logarithmic distribution is defined by the property that the proportion of base β numbers with leading significant digit n is

$$\log_\beta(n+1) - \log_\beta n = \log_\beta\Big(1 + \frac{1}{n}\Big).$$

It appears that in practice real numbers *are* logarithmically distributed. In 1938, Benford [102, 1938] noticed, as had Newcomb [889, 1881] before him, that the early pages of logarithm tables showed greater signs of wear than the later ones. (He was studying dirty books!) This prompted him to carry out a survey of 20,229 "real-life" numbers, whose decimal representations he found matched the logarithmic distribution closely.

The observed logarithmic distribution of leading significant digits has not been fully explained. Some proposed explanations are based on the assumption that the actual distribution is scale invariant, but this assumption is equivalent to the observation to be explained [1170, 1984]. Barlow [67, 1981], [68, 1981], [70, 1988] and Turner [1169, 1982], [1170, 1984] give useful insight by showing that if uniformly distributed numbers are multiplied together, then the resulting distribution converges to the logarithmic one; see also Boyle [157, 1994]. Furthermore, it is an interesting result that the leading significant digits of the numbers q^k,

$k = 0, 1, 2, \ldots$, are logarithmically distributed if q is positive and is not a rational power of 10; when $q = 2$ and the digit is 7 this is Gelfand's problem [939, 1981, pp. 50–51].

The nature of the logarithmic distribution is striking. For decimal numbers, the digits 1 to 9 are not equally likely to be a leading significant digit. The probabilities are as follows:

1	2	3	4	5	6	7	8	9
0.301	0.176	0.125	0.097	0.079	0.067	0.058	0.051	0.046

As an example, here is the leading significant digit distribution for the elements of the inverse of one random 100×100 matrix from the normal $N(0, 1)$ distribution:

1	2	3	4	5	6	7	8	9
0.334	0.163	0.100	0.087	0.077	0.070	0.063	0.056	0.051

For an entertaining survey of work on the distribution of leading significant digits see Raimi [968, 1976] (and also the popular article [967, 1969]).

2.8. Statistical Distribution of Rounding Errors

Most rounding error analyses, including all the ones in this book, are designed to produce worst-case bounds for the error. The analyses ignore the signs of rounding errors and are often the result of many applications of the triangle inequality and the submultiplicative inequality. Consequently, although the bounds may well give much insight into a method, they tend to be pessimistic if regarded as error estimates.

Statistical statements about the effect of rounding on a numerical process can be obtained from statistical analysis coupled with probabilistic models of the rounding errors. For example, a well-known rule of thumb is that a more realistic error estimate for a numerical method is obtained by replacing the dimension-dependent constants in a rounding error bound by their square root; thus if the bound is $f(n)u$, the rule of thumb says that the error is typically of order $\sqrt{f(n)}u$ (see, for example, Wilkinson [1232, 1963, pp. 26, 102]). This rule of thumb can be supported by assuming that the rounding errors are independent random variables and applying the central limit theorem. Statistical analysis of rounding errors goes back to one of the first papers on rounding error analysis, by Goldstine and von Neumann [501, 1951].

As we noted in §1.17, rounding errors are *not* random. See Problem 2.10 for an example of how two rounding errors cancel in one particular class of computations. Forsythe [424, 1959] points out that rounding errors do not necessarily behave like independent random variables and proposes a random form of rounding (intended for computer testing) to which statistical analysis is applicable.

Henrici [564, 1962], [565, 1963], [566, 1964] assumes models for rounding errors and then derives the probability distribution of the overall error, mainly in the context of difference methods for differential equations. Hull and Swenson [650, 1966] give an insightful discussion of probabilistic models, pointing out that "There is no claim that ordinary rounding and chopping are random processes, or that successive errors are independent. The question to be decided is whether or

not these particular probabilistic models of the processes will adequately describe what actually happens" (see also the ensuing note by Henrici [567, 1966]). Kahan [699, 1996] notes that "The fact that rounding errors are neither random nor uncorrelated will not in itself preclude the possibility of modelling them usefully by uncorrelated random variables. What will jeopardize the utility of such models is their failure to mimic important properties that actual rounding errors possess." In the last sentence Kahan is referring to results such as Theorem 2.5.

Several authors have investigated the distribution of rounding errors under the assumption that the significands of the operands are from a logarithmic distribution and for different modes of rounding; see Barlow and Bareiss [72, 1985] and the references therein.

Other work concerned with statistical modelling of rounding errors includes that of Tienari [1137, 1970] and Linnainmaa [789, 1975]; see also the discussion of the CESTAC and PRECISE systems in §26.5.

2.9. Alternative Number Systems

The floating point format is not the only means for representing numbers in finite precision arithmetic. Various alternatives have been proposed, though none has achieved widespread use.

A particularly elegant system is the "level index arithmetic" of Clenshaw, Olver, and Turner, in which a number $x > 1$ is represented by $\xi = l + f$, where $f \in [0, 1]$ and

$$x = e^{e^{\cdot^{\cdot^{\cdot^{e^f}}}}}, \quad \text{or} \quad f = \ln(\ln(\ldots(\ln x)\ldots)),$$

where the exponentiation or logarithm is performed l times (l is the "level"). If $0 < x < 1$, then x is represented by the reciprocal of the representation for $1/x$. An obvious feature of the level index system is that it can represent much larger and smaller numbers than the floating point system, for similar word lengths. A price to be paid is that as the size of the number increases the precision with which it is represented shrinks. If l is sufficiently large then adjacent level index numbers are so far apart that even their exponents in base 10 differ. For very readable introductions to level index arithmetic see Clenshaw and Olver [240, 1984] and Turner [1171, 1991], and for more details see Clenshaw, Olver, and Turner [241, 1989]. Level index arithmetic is somewhat controversial in that there is disagreement about its advantages and disadvantages with respect to floating point arithmetic; see Demmel [310, 1987]. A number system involving levels has also been proposed by Matsui and Iri [826, 1981]; in their system, the number of bits allocated to the significand and exponent is allowed to vary (within a fixed word size).

Other number systems include those of Swartzlander and Alexopolous [1115, 1975], Matula and Kornerup [831, 1985], and Hamada [539, 1987]. For summaries of alternatives to floating point arithmetic see the section "Alternatives to Floating-Point—Some Candidates", in [241, 1989], and Knuth [744, 1998, Chap. 4].

2.10. Elementary Functions

The term elementary functions refers to the base 2, e and 10 logarithms and exponentials, and the trigonometric and hyperbolic functions and their inverses. These functions are much harder to implement than the elementary operations $+, -, *, /$ and square root and, as noted earlier, the IEEE arithmetic standard does not have anything to say about elementary functions.

In the past, elementary function libraries were of variable quality, but with increased demand for high-quality floating point arithmetic, the ability now to build on IEEE standard elementary operations, and continuing research into elementary function evaluation, the quality of algorithms and libraries is improving.

Ideally, we would like the computed value of an elementary function at a machine number to be the correctly rounded result. The tablemaker's dilemma (see §1.2) casts doubt on whether this goal is achievable. However, Lindemann's result that the exponential of a nonzero algebraic number cannot be algebraic implies that the exponential of a nonzero *machine* number cannot be a machine number or half-way between two machine numbers. Therefore for a given machine number x and precision t there is a finite m so that computing m digits of e^x is sufficient to determine e^x correctly rounded to t digits. Similar arguments apply to the other elementary functions, and so the tablemaker's dilemma does not occur in the context of floating point arithmetic [778, 1998]. In extensive computations, Lefèvre, Muller, and Tisserand [778, 1998], [777, 2001] have determined the maximum value of m over all double precision IEEE arithmetic numbers ($t = 53$) for each of the elementary functions. They find that m is usually close to $2t = 106$. This knowledge of the worst-case m makes the goal of correctly rounded results achievable.

Other desirable properties of computed elementary functions are preservation of monotonicity (e.g., $x < y \;\Rightarrow\; e^x < e^y$), preservation of symmetries (e.g., $\sin(x) = -\sin(-x)$), and preservation of mathematical relations and identities (e.g., $\sin(x) \in [-1, 1]$ and $\cos^2(x)+\sin^2(x) = 1$). Unfortunately, these requirements can conflict. Lefèvre, Muller, and Tisserand [778, 1998] note that in IEEE single precision arithmetic the machine number closest to $\arctan(2^{30})$ is slightly greater than $\pi/2$, so a correctly rounded approximation to $\arctan(2^{30})$ necessarily violates the requirement that $\arctan(x) \in [-\pi/2, \pi/2]$.

Describing methods for evaluating elementary functions is outside the scope of this book. Details can be found in the excellent book by Muller [876, 1997], which is the best overall reference for this topic. Algorithms for evaluating elementary functions in IEEE arithmetic are developed by Tang [1121, 1989], [1123, 1990], [1125, 1992], Gal and Bachelis [451, 1991], and Ziv [1285, 1991]. Tang [1124, 1991] gives a very readable description of table lookup algorithms for evaluating elementary functions, which are used in a number of current chips and libraries.

Algorithms for evaluating complex elementary functions that exploit exception handling and assume the availability of algorithms for the real elementary functions are presented by Hull, Fairgrieve, and Tang [649, 1994]. For details of how elementary functions are evaluated on many of today's pocket calculators see Schelin [1016, 1983], as well as Muller [876, 1997].

Table 2.3. *Test arithmetics.*

Hardware	Software	$\lvert 3 \times (4/3 - 1) - 1 \rvert$ [a]
Casio fx-140 ($\approx$ 1979)		1×10^{-9}
Casio fx-992VB ($\approx$ 1990)		1×10^{-13}
HP 48G (1993)		1×10^{-11}
Sharp EL-5020 (1994)		0.0 [b]
Pentium III	MATLAB 6.1 (2001)	$2.2\ldots \times 10^{-16}$
486DX	WATFOR-77[c] V3.0 (1988)	$2.2\ldots \times 10^{-16}$
486DX	FTN 90[d] (1993)	$2.2\ldots \times 10^{-16}$
486DX	MS-DOS QBasic 1.1	$1.1\ldots \times 10^{-19}$ [e]

[a] Integers in the test expression are typed as real constants 3.0, etc., for the Fortran tests.
[b] 1×10^{-9} if 4/3 is stored and recalled from memory.
[c] WATCOM Systems Inc.
[d] Salford Software/Numerical Algorithms Group, Version 1.2.
[e] $2.2\ldots \times 10^{-16}$ if 4/3 is stored and recalled from a variable.

2.11. Accuracy Tests

How can you test the accuracy of the floating point arithmetic on a computer or pocket calculator? There is no easy way, though a few software packages are available to help with the tasks in specific programming languages (see §27.6). There are, however, a few quick and easy tests that *may* reveal weaknesses. The following list is far from comprehensive and good performance on these tests does not imply that an arithmetic is correct. Results from the tests are given in Tables 2.4–2.5 for the selected floating point arithmetics described in Table 2.3, with incorrect digits in boldface and underlined. Double precision was used for the compiled languages. The last column of Table 2.3 gives an estimate of the unit roundoff (see Problem 2.14). The estimate produced by QBasic indicates that the compiler used extended precision in evaluating the estimate.

1. (Cody [249, 1982]) Evaluate $\sin(22) = -8.8513\ 0929\ 0403\ 8759\ 2169 \times 10^{-3}$ (shown correct to 21 digits). This is a difficult test for the range reduction used in the sine evaluation (which brings the argument within the range $[-\pi/2, \pi/2]$, and which necessarily uses an approximate value of π), since 22 is close to an integer multiple of π.

2. (Cody [249, 1982]) Evaluate $2.5^{125} = 5.5271\ 4787\ 5260\ 4445\ 6025 \times 10^{49}$ (shown correct to 21 digits). One way to evaluate $z = x^y$ is as $z = \exp(y \log x)$. But to obtain z correct to within a few ulps it is not sufficient to compute exp and log correct to within a few ulps; in other words, the composition of two functions evaluated to high accuracy is not necessarily obtained to the same accuracy. To examine this particular case, write

$$w := y \log x, \qquad z = \exp(w).$$

   If $w \rightarrow w + \Delta w$ then $z \rightarrow z + \Delta z$, where $z + \Delta z = \exp(w + \Delta w) =$

Table 2.4. *Sine test.*

Machine	sin(22)
Exact	$-8.8513\ 0929\ 0403\ 8759 \times 10^{-3}$
Casio fx-140	$-8.8513\ \underline{\mathbf{62}} \times 10^{-3}$
Casio fx-992VB	$-8.8513\ 0929\ 0\underline{\mathbf{96}} \times 10^{-3}$
HP 48G	$-8.8513\ 0929\ 040 \times 10^{-3}$
Sharp EL-5020	$-8.8513\ 09\underline{\mathbf{15\ 4}} \times 10^{-3}$
MATLAB 6.1	$-8.8513\ 0929\ 0403\ 876 \times 10^{-3}$
WATFOR-77	$-8.8513\ 0929\ 0403\ 8\underline{\mathbf{80}} \times 10^{-3}$
FTN 90	$-8.8513\ 0929\ 0403\ 876 \times 10^{-3}$
QBasic	$-8.8513\ 0929\ 0403\ 876 \times 10^{-3}$

Table 2.5. *Exponentation test. No entry for last column means same value as previous column.*

Machine	2.5^{125}	$\exp(125\log(2.5))$
Exact	$5.5271\ 4787\ 5260\ 4446 \times 10^{49}$	$5.5271\ 4787\ 5260\ 4446 \times 10^{49}$
Casio fx-140	$5.5271\ 47\underline{\mathbf{7}} \times 10^{49}$	$5.5271\ 4\underline{\mathbf{63}} \times 10^{49}$
Casio fx-992VB	$5.5271\ 4787\ 526 \times 10^{49}$	
HP 48G	$5.5271\ 4787\ 526 \times 10^{49}$	$5.5271\ 4787\ \underline{\mathbf{377}} \times 10^{49}$
Sharp EL-5020	$5.5271\ 4787\ \underline{\mathbf{3}} \times 10^{49}$	$5.5271\ 47\underline{\mathbf{96\ 2}} \times 10^{49}$
MATLAB 6.1	$5.5271\ 4787\ 5260\ 44\underline{\mathbf{4}} \times 10^{49}$	$5.5271\ 4787\ 5260\ 4\underline{\mathbf{59}} \times 10^{49}$
WATFOR-77	$5.5271\ 4787\ 5260\ 4\underline{\mathbf{50}} \times 10^{49}$	$5.5271\ 4787\ 5260\ 4\underline{\mathbf{60}} \times 10^{49}$
FTN 90	$5.5271\ 4787\ 5260\ 445 \times 10^{49}$	$5.5271\ 4787\ 5260\ 4\underline{\mathbf{59}} \times 10^{49}$
QBasic	$5.5271\ 4787\ 5260\ 44\underline{\mathbf{4}} \times 10^{49}$	

$\exp(w)\exp(\Delta w) \approx \exp(w)(1+\Delta w)$, so $\Delta z/z \approx \Delta w$. In other words, the relative error of z depends on the absolute error of w and hence on the size of w. To obtain z correct to within a few ulps it is necessary to use extra precision in calculating the logarithm and exponential [256, 1980, Chap. 7].

3. (Karpinski [715, 1985]) A simple test for the presence of a guard digit on a pocket calculator is to evaluate the expressions

$$9/27 * 3 - 1, \qquad 9/27 * 3 - 0.5 - 0.5,$$

which are given in a form that can be typed directly into most four-function calculators. If the results are equal then a guard digit is present. Otherwise there is probably no guard digit (we cannot be completely sure from this simple test). To test for a guard digit on a computer it is best to run one of the diagnostic codes described in §27.5.

2.12. Notes and References

The classic reference on floating point arithmetic, and on all aspects of rounding error analysis, is Wilkinson's *Rounding Errors in Algebraic Processes* (READ) [1232,

1963]. Wilkinson was uniquely qualified to write such a book, for not only was he the leading expert in rounding error analysis, but he was one of the architects and builders of the Automatic Computing Engine (ACE) at the National Physical Laboratory [1226, 1954]. The Pilot (prototype) ACE first operated in May 1950, and an engineered version was later sold commercially as the DEUCE Computer by the English Electric Company. Wilkinson and his colleagues were probably the first to write subroutines for floating point arithmetic, and this enabled them to accumulate practical experience of floating point arithmetic much earlier than anyone else [395, 1976], [1243, 1980].

In REAP, Wilkinson gives equal consideration to fixed point and floating point arithmetic. In fixed point arithmetic, all numbers are constrained to lie in a range such as $[-1, 1]$, as if the exponent were frozen in the floating point representation (2.1). Preliminary analysis and the introduction of scale factors during the computation are needed to keep numbers in the permitted range. We consider only floating point arithmetic in this book. REAP, together with Wilkinson's second book, *The Algebraic Eigenvalue Problem* (AEP) [1233, 1965], has been immensely influential in the areas of floating point arithmetic and rounding error analysis.

Wilkinson's books were preceded by the paper "Error Analysis of Floating-Point Computation" [1228, 1960], in which he presents the model (2.4) for floating point arithmetic and applies the model to several algorithms for the eigenvalue problem. This paper has hardly dated and is still well worth reading.

Another classic book devoted entirely to floating point arithmetic is Sterbenz's *Floating-Point Computation* [1062, 1974]. It contains a thorough treatment of low-level details of floating point arithmetic, with particular reference to IBM 360 and IBM 7090 machines. It also contains a good chapter on rounding error analysis and an interesting collection of exercises. R. W. Hamming has said of this book, "Nobody should ever have to know that much about floating-point arithmetic. But I'm afraid sometimes you might" [942, 1988]. Although Sterbenz's book is now dated in some respects, it remains a useful reference.

A third important reference on floating point arithmetic is Knuth's *Seminumerical Algorithms* [744, 1998, Sec. 4.2], from his *Art of Computer Programming* series. Knuth's lucid presentation includes historical comments and challenging exercises (with solutions).

The first analysis of floating point arithmetic was given by Samelson and Bauer [1009, 1953]. Later in the same decade Carr [205, 1959] gave a detailed discussion of error bounds for the basic arithmetic operations.

An up-to-date and very readable reference on floating point arithmetic is the survey paper by Goldberg [496, 1991], which includes a detailed discussion of IEEE arithmetic. A less mathematical, more hardware-oriented discussion can be found in the appendix "Computer Arithmetic" written by Goldberg that appears in the book on computer architecture by Hennessy and Patterson [562, 1996].

A fascinating historical perspective on the development of computer floating point arithmetics, including background to the development of the IEEE standard, can be found in the textbook by Patterson and Hennessy [929, 1998, §4.12].

The idea of representing floating point numbers in the form (2.1) is found, for example, in the work of Forsythe [428, 1969], Matula [830, 1970], and Dekker [302, 1971].

An alternative definition of $fl(x)$ is the nearest $y \in G$ satisfying $|y| \leq |x|$. This operation is called *chopping*, and does not satisfy our definition of rounding. Chopped arithmetic is used in the IBM/370 floating point system.

The difference between chopping and rounding (to nearest) is highlighted by a discrepancy in the index of the Vancouver Stock Exchange in the early 1980s [963, 1983]. The exchange established an index in January 1982, with the initial value of 1000. By November 1983 the index had been hitting lows in the 520s, despite the exchange apparently performing well. The index was recorded to three decimal places and it was discovered that the computer program calculating the index was chopping instead of rounding to produce the final value. Since the index was recalculated thousands of times a day, each time with a nonpositive final error, the bias introduced by chopping became significant. Upon recalculation with rounding the index almost doubled!

When there is a tie in rounding, two possible strategies are to round to the number with an even last digit and to round to the one with an odd last digit. Both are stable forms of rounding in the sense that

$$fl\big(\big(((x+y)-y)+y\big)-y\big) = fl\big((x+y)-y\big),$$

as shown by Reiser and Knuth [982, 1975], [744, 1998, Sec. 4.2.2, Thm. D]. For other rules, such as round away from zero, repeated subtraction and addition of the same number can yield an increasing sequence, a phenomenon known as *drift*. For bases 2 and 10 rounding to even is preferred to rounding to odd. After rounding to even a subsequent rounding to one less place does not involve a tie. Thus we have the rounding sequence 2.445, 2.44, 2.4 with round to even, but 2.445, 2.45, 2.5 with round to odd. For base 2, round to even causes computations to produce integers more often [706, 1979] as a consequence of producing a zero least significant bit. Rounding to even in the case of ties seems to have first been suggested by Scarborough in the first edition (1930) of [1014, 1950]. Hunt [651, 1997] notes that in the presentation of meteorological temperature measurements round to odd is used as a tie-breaking strategy, and he comments that this may be to avoid the temperatures 0.5°C and 32.5°F being rounding down and the incorrect impression being given that it is freezing.

Predictions based on the growth in the size of mathematical models solved as the memory and speed of computers increase suggest that floating point arithmetic with unit roundoff $u \approx 10^{-32}$ will be needed for some applications on future supercomputers [57, 1989].

The model (2.4) does not fully describe any floating point arithmetic. It is merely a tool for error analysis—one that has been remarkably successful in view of our current understanding of the numerical behaviour of algorithms. There have been various attempts to devise formal models of floating point arithmetic, by specifying sets of axioms in terms of which error analysis can be performed. Some attempts are discussed in §27.7.4. No model yet proposed has been truly successful. Priest [955, 1992] conjectures that the task of "encapsulating all that we wish to know about floating point arithmetic in a single set of axioms" is impossible, and he gives some motivation for this conjecture.

Under the model (2.4), floating point arithmetic is not associative with respect to any of the four basic operations: $(a \boxed{\text{op}} b) \boxed{\text{op}} c \neq a \boxed{\text{op}} (b \boxed{\text{op}} c)$, $\text{op} = +, -, *, /$,

where $a \boxed{\text{op}} b := fl(a \text{ op } b)$. Nevertheless, floating point arithmetic enjoys some algebraic structure, and it is possible to carry out error analysis in the "$\boxed{\text{op}}$ algebra". Fortunately, it was recognized by Wilkinson and others in the 1950s that this laboured approach is unnecessarily complicated, and that it is much better to work with the exact equations satisfied by the computed quantities. As Parlett [925, 1990] notes, though, "There have appeared a number of ponderous tomes that do manage to abstract the computer's numbers into a formal structure and burden us with more jargon."

A draft proposal of IEEE Standard 754 is defined and described in [657, 1981]. That article, together with others in the same issue of the journal *Computer*, provides a very readable description of IEEE arithmetic. In particular, an excellent discussion of gradual underflow is given by Coonen [270, 1981]. A draft proposal of IEEE Standard 854 is presented, with discussion, in [253, 1984].

W. M. Kahan of the University of California at Berkeley received the 1989 ACM Turing Award for his contributions to computer architecture and numerical analysis, and in particular for his work on IEEE floating point arithmetic standards 754 and 854. For interesting comments by Kahan on the development of IEEE arithmetic and other floating point matters, see [1029, 1998], [1251, 1997],

An interesting examination of the implications of the IEEE standard for high-level languages such as Fortran is given by Fateman [405, 1982]. Topics discussed include trap handling and how to exploit NaNs. For an overview of hardware implementations of IEEE arithmetic, and software support for it, see Cody [251, 1988].

Producing a fast and correct implementation of IEEE arithmetic is a difficult task. Correctness is especially important for a microprocessor (as opposed to a software) implementation, because of the logistical difficulties of correcting errors when they are found. In late 1994, much publicity was generated by the discovery of a flaw in the floating point divide instruction of Intel's Pentium chip. Because of some missing entries in a lookup table on the chip, the FPDIV instruction could give as few as four correct significant decimal digits for double precision floating point arguments with certain special bit patterns [1036, 1994], [380, 1997]. The flaw had been discovered by Intel in the summer of 1994 during ongoing testing of the Pentium processor, but it had not been publically announced. In October 1994, a mathematician doing research into prime numbers independently discovered the flaw and reported it to the user community. Largely because of the way in which Intel responded to the discovery of the flaw, the story was reported in national newspapers (e.g., the *New York Times* [816, 1994]) and generated voluminous discussion on Internet newsgroups (notably `comp.sys.intel`). Intel corrected the bug in 1994 and, several weeks after the bug was first reported, offered to replace faulty chips. For a very readable account of the Pentium FPDIV bug story, see Moler [866, 1995]. To emphasize that bugs in implementations of floating point arithmetic are far from rare, we mention that the Calculator application in Microsoft Windows 3.1 evaluates $fl(2.01 - 2.00) = 0.0$.

Computer chip designs can be tested in two main ways: by software simulations and by applying formal verification techniques. Formal verification aims to prove mathematically that the chip design is correct, and this approach has been in use by chip manufacturers for some time [491, 1995]. Some relevant references are

Barrett [79, 1989] for the Inmos T800 (or Shepherd and Wilson [1037, 1989] for a more informal overview), and Moore, Lynch, and Kaufmann [871, 1998] and Russinoff [1003, 1998] for AMD processors.

For low-level details of computer arithmetic several textbooks are available. We mention only the recent and very thorough book by Parhami [921, 2000].

The floating point operation op (op $= +, -, *,$ or $/$) is *monotonic* if $fl(a \operatorname{op} b) \leq fl(c \operatorname{op} d)$ whenever a, b, c, and d are floating point numbers for which $a \operatorname{op} b \leq c \operatorname{op} d$ and neither $fl(a \operatorname{op} b)$ nor $fl(c \operatorname{op} d)$ overflows. IEEE arithmetic is monotonic, as is any correctly rounded arithmetic. Monotonic arithmetic is important in the bisection algorithm for finding the eigenvalues of a symmetric tridiagonal matrix; see Demmel, Dhillon, and Ren [320, 1995], who give rigorous correctness proofs of some bisection implementations in floating point arithmetic. Ferguson and Brightman [410, 1991] derive conditions that ensure that an approximation to a monotonic function preserves the monotonicity on a set of floating point numbers.

On computers of the 1950s, (fixed point) multiplication was slower than (fixed point) addition by up to an order of magnitude [773, 1980, Apps. 2, 3]. For most modern computers it is a rule of thumb that a floating point addition and multiplication take about the same amount of time, while a floating point division is 2–10 times slower, and a square root operation (in hardware) is 1–2 times slower than a division.

During the design of the IBM 7030, Sweeney [1116, 1965] collected statistics on the floating point additions carried out by selected application programs on an IBM 704. He found that 11% of all instructions traced were floating point additions. Details were recorded of the shifting needed to align floating point numbers prior to addition, and the results were used in the design of the shifter on the IBM 7030.

The word *bit*, meaning binary digit, first appeared in print in a 1948 paper of Claude E. Shannon, but the term was apparently coined by John W. Tukey [1160, 1984]. The word *byte*, meaning a group of (usually eight) bits, did not appear in print until 1959 [173, 1981].

The earliest reference we know for Theorem 2.5 is Sterbenz [1062, 1974, Thm. 4.3.1]. Theorem 2.4 is due to Ferguson [409, 1995], who proves a more general version of the theorem that allows for trailing zero digits in x and y. A variation in which the condition is $0 \leq y \leq x \leq y + \beta^e$, where $e = \min\{\, j : \beta^j \geq y \,\}$, is stated by Ziv [1285, 1991] and can be proved in a similar way.

For more on the choice of base, see Cody [255, 1973] and Kuki and Cody [754, 1973]. Buchholz's paper *Fingers or Fists*? [172, 1959] on binary versus decimal representation of numbers on a computer deserves mention for its clever title, though the content is only of historical interest.

The model (2.4) ignores the possibility of underflow and overflow. To take underflow into account the model must be modified to

$$fl(x \operatorname{op} y) = (x \operatorname{op} y)(1 + \delta) + \eta, \qquad \operatorname{op} = +, -, *, /. \tag{2.8}$$

As before, $|\delta| \leq u$. If underflow is gradual, as in IEEE arithmetic, then $|\eta| \leq \frac{1}{2}\beta^{e_{\min}-t} = \lambda u$, which is half the spacing between the subnormal numbers ($\lambda = \beta^{e_{\min}-1}$ is the smallest positive normalized floating point number); if underflows

are flushed to zero then $|\eta| \leq \lambda$. Only one of δ and η is nonzero: δ if no underflow occurs, otherwise η. With gradual underflow the *absolute error* of an underflowed result is no greater than the smallest (bound for the) absolute error that arises from an operation $fl(x \text{ op } y)$ in which the arguments and result are normalized. Moreover, with gradual underflow we can take $\eta = 0$ for op $= +, -$ (see Problem 2.19). For more details, and a thorough discussion of how error analysis of standard algorithms is affected by using the model (2.8), see the perceptive paper by Demmel [308, 1984]. Another relevant reference is Neumaier [884, 1985].

Hauser [553, 1996] gives a thorough and very readable treatment of exception handling, covering underflow, overflow, indeterminate or undefined cases, and their support in software.

An important problem not considered in this chapter is the conversion of numbers between decimal and binary representations. These conversions are needed whenever numbers are read into a computer or printed out. They tend to be taken for granted, but if not done carefully they can lead to puzzling behaviour, such as a number read in as 0.1 being printed out as $0.099\ldots9$. To be precise, the problems of interest are (a) convert a number represented in decimal notation to the best binary floating point representation of a given precision, and (b) given a binary floating point number, print a correctly rounded decimal representation, either to a given number of significant digits or to the smallest number of significant digits that allows the number to be re-read without loss of accuracy. Algorithms for solving these problems are given by Clinger [247, 1990] and Steele and White [1060, 1990]; Gay [470, 1990] gives some improvements to the algorithms and C code implementing them. Precise requirements for binary–decimal conversion are specified in the IEEE arithmetic standard. A program for testing the correctness of binary–decimal conversion routines is described by Paxson [930, 1991]. Early references on base conversion are Goldberg [497, 1967] and Matula [829, 1968], [830, 1970]. It is interesting to note that, in Fortran or C, where the output format for a "print" statement can be precisely specified, most compilers will, for an (in)appropriate choice of format, print a decimal string that contains many more significant digits than are determined by the floating point number whose value is being represented.

Other authors who have analysed various aspects of floating (and fixed) point arithmetic include Diamond [339, 1978], Urabe [1174, 1968], and Feldstein, Goodman, and co-authors [510, 1975], [407, 1982], [511, 1985], [408, 1986]. For a survey of computer arithmetic up until 1976 that includes a number of references not given here, see Garner [460, 1976].

Problems

The exercise had warmed my blood,
and I was beginning to enjoy myself amazingly.

— JOHN BUCHAN, *The Thirty-Nine Steps* (1915)

2.1. How many normalized numbers and how many subnormal numbers are there in the system F defined in (2.1) with $e_{\min} \leq e \leq e_{\max}$? What are the figures for IEEE single and double precision (base 2)?

2.2. Prove Lemma 2.1.

2.3. In IEEE arithmetic how many double precision numbers are there between any two adjacent nonzero single precision numbers?

2.4. Prove Theorem 2.3.

2.5. Show that

$$0.1 = \sum_{i=1}^{\infty} (2^{-4i} + 2^{-4i-1})$$

and deduce that 0.1 has the base 2 representation $0.000\overline{1100}$ (repeating last 4 bits). Let $\widehat{x} = fl(0.1)$ be the rounded version of 0.1 obtained in binary IEEE single precision arithmetic ($u = 2^{-24}$). Show that $(x - \widehat{x})/x = -\frac{1}{4}u$.

2.6. What is the largest integer p such that all integers in the interval $[-p, p]$ are exactly representable in IEEE double precision arithmetic? What is the corresponding p for IEEE single precision arithmetic?

2.7. Which of the following statements is true in IEEE arithmetic, assuming that a and b are normalized floating point numbers and that no exception occurs in the stated operations?

1. $fl(a \operatorname{op} b) = fl(b \operatorname{op} a)$, $\quad \operatorname{op} = +, *$.
2. $fl(b - a) = -fl(a - b)$.
3. $fl(a + a) = fl(2 * a)$.
4. $fl(0.5 * a) = fl(a/2)$.
5. $fl((a + b) + c) = fl(a + (b + c))$.
6. $a \le fl((a + b)/2) \le b$, given that $a \le b$.

2.8. Show that the inequalities $a \le fl((a + b)/2) \le b$, where a and b are floating point numbers with $a \le b$, can be violated in base 10 arithmetic. Show that $a \le fl(a + (b - a)/2) \le b$ in base β arithmetic, for any β.

2.9. What is the result of the computation $\sqrt{1 - 2^{-53}}$ in IEEE double precision arithmetic, with and without double rounding from an extended format with a 64-bit significand?

2.10. A theorem of Kahan [496, 1991, Thm. 7] says that if $\beta = 2$ and the arithmetic rounds as specified in the IEEE standard, then for integers m and n with $|m| < 2^{t-1}$ and $n = 2^i + 2^j$ (some i, j), $fl((m/n) * n) = m$. Thus, for example, $fl((1/3) * 3) = 1$ (even though $fl(1/3) \ne 1/3$). The sequence of allowable n begins $1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 16, 17, 18, 20$, so Kahan's theorem covers many common cases. Test the theorem on your computer.

2.11. Investigate the leading significant digit distribution for numbers obtained as follows.

1. k^n, $n = 0\colon 1000$ for $k = 2$ and 3.
2. $n!$, $n = 1\colon 1000$.

3. The eigenvalues of a random symmetric matrix.
4. Physical constants from published tables.
5. From the front page of the *London Times* or the *New York Times*.

(Note that in writing a program for the first case you can form the powers of 2 or 3 in order, following each multiplication by a division by 10, as necessary, to keep the result in the range $[1, 10]$. Similarly for the second case.)

2.12. (Edelman [379, 1994]) Let x be a floating point number in IEEE double precision arithmetic satisfying $1 \le x < 2$. Show that $fl(x * (1/x))$ is either 1 or $1 - \epsilon/2$, where $\epsilon = 2^{-52}$ (the machine epsilon).

2.13. (Edelman [379, 1994]) Consider IEEE double precision arithmetic. Find the smallest positive integer j such that $fl(x * (1/x)) \neq 1$, where $x = 1 + j\epsilon$, with $\epsilon = 2^{-52}$ (the machine epsilon).

2.14. Kahan has stated that "an (over-)estimate of u can be obtained for almost any machine by computing $|3 \times (4/3 - 1) - 1|$ using rounded floating-point for every operation". Test this estimate against u on any machines available to you.

2.15. What is 0^0 in IEEE arithmetic?

2.16. Evaluate these expressions in any IEEE arithmetic environment available to you. Are the values returned what you would expect? (None of the results is specified by the IEEE standard.)

1. 1^∞.
2. 2^∞.
3. $\exp(\infty)$, $\exp(-\infty)$.
4. sign(NaN), sign(−NaN).
5. NaN^0.
6. ∞^0.
7. 1^{NaN}.
8. $\log(\infty)$, $\log(-\infty)$, $\log(0)$.

2.17. Let $x_{\max}$ denote the largest representable number in IEEE single or double precision arithmetic. In what circumstances does $2x_{\max}$ not overflow?

2.18. Can Theorem 2.4 be strengthened to say that $fl(x-y)$ is computed exactly whenever the exponents of $x \ge 0$ and $y \ge 0$ differ by at most 1?

2.19. (Hauser [553, 1996]) Show that with gradual underflow, if x and y are floating point numbers and $fl(x \pm y)$ underflows then $fl(x \pm y) = x \pm y$.

2.20. Two requirements that we might ask of a routine for computing $\sqrt{x}$ in floating point arithmetic are that the identities $\sqrt{x^2} = |x|$ and $(\sqrt{x})^2 = |x|$ be satisfied. Which, if either, of these is a reasonable requirement?

2.21. Are there any floating point values of x and y (excepting values both 0, or so huge or tiny as to cause overflow or underflow) for which the computed value of $x/\sqrt{x^2 + y^2}$ exceeds 1?

2.22. (Kahan) A natural way to compute the maximum of two numbers x and y is with the code

```
% max(x, y)
if x > y then
   max = x
else
   max = y
end
```

Does this code always produce the expected answer in IEEE arithmetic?

2.23. Prove that Kahan's formula (2.7) computes the area of a triangle accurately. (Hint: you will need one invocation of Theorem 2.5.)

2.24. (Kahan) Describe the result of the computation $y = (x+x)-x$ on a binary machine with a guard digit and one without a guard digit.

2.25. (Kahan) Let $f(x) = \big(((x-0.5)+x)-0.5\big)+x$. Show that if f is evaluated as shown in single or double precision binary IEEE arithmetic then $f(x) \neq 0$ for all floating point numbers x.

2.26. Show that if x and y are floating point numbers and $\widehat{a} = fl(xy)$ and $\widehat{b} = fl(xy - \widehat{a})$ are computed using fused multiply-add operations, then $\widehat{a} + \widehat{b} \equiv xy$.

2.27. (Kahan) Show that with the use of a fused multiply-add operation the algorithm

$$
\begin{aligned}
w &= bc \\
e &= w - bc \\
x &= (ad - w) + e
\end{aligned}
$$

computes $x = \det(\left[\begin{smallmatrix} a & b \\ c & d \end{smallmatrix}\right])$ with high relative accuracy.

2.28. Suppose we have an iterative algorithm for computing $z = x/y$. Derive a convergence test that terminates the iteration (only) when full accuracy has been achieved. Assume the use of IEEE arithmetic with gradual underflow (use (2.8)).