Computer representation of numbers and finite-precision arithmetic

1 Computer representation of numbers

Modern processors are (almost) exclusively binary. They contain billions of transistors that are essentially tiny electronic on/off switches which, among other things, are used to store binary digits called *bits* (0 and 1) and to perform arithmetic and logic operations. The main numeral systems in computers are therefore binary numeral systems (i.e., radix 2) where numbers are stored using a binary representation with a finite number of bits. We remark that the term *byte* (a mutation of the word *bite*) is used to refer to a group of 8 bits.

1.1 Integers

Integers can be represented in binary form using the two binary digits 0 and 1. Clearly, n bits make it possible to represent 2^n different numbers. We start by considering nonnegative integers of the form

$$(c_{n-1}c_{n-2}\cdots c_0)_b$$

which are also called *unsigned* integers. The bit c_{n-1} is sometimes referred to as the most significant bit (MSB) whereas c_0 is referred to as least significant bit (LSB).

The obvious choice for representing unsigned integers is to store the n bits $c_0,\ldots,c_{n-1}$ such that the set of possible values is $\{0,1,\ldots,2^n-1\}$. The result of an arithmetic operation involving two such numbers may have as many as $2n$ digits and may be negative, so a natural question is what happens if an arithmetic operations yields a result that is not in the set of possible values. This situation is sometimes referred to as *overflow*. A natural way to deal with this issue is to adopt *modular arithmetic*. Before we delve into modular arithmetic, we start by defining some relevant terminology and notation. Two integers a and b are said to be congruent modulo N (where N is a positive integer) if they differ by a multiple of N, i.e., there exists an integer $k\in\mathbb{Z}$ such that

$$a-b=kN.$$

This may expressed as $a\equiv b \pmod N$. In contrast, a modulo N, denoted $a \bmod N$, is the *modulo operation* which is the remainder of the Euclidean division of a by N (i.e., a is the dividend and N is the divisor). Given $a\in\mathbb{Z}$ and $N\in\mathbb{N}$, $a \bmod N$ is the unique integer r such that $0\le r\le N-1$ and $a\equiv r \pmod N$. Equivalently, we have that $a=qN+r$ where $q=\lfloor a/N\rfloor$ is the integer part of the quotient and $r=a-qN$ is the nonnegative remainder. (We note that in the context of integer division, the term "quotient" is commonly used to refer to the integer part of a/N.)

In modular arithmetic, the result of an arithmetic operation "wraps around" when reaching a given value N (called the modulus). Specifically, modular addition/subtraction/multiplication of two integers $a, b \in \{0, 1, \ldots, N-1\}$ may be expressed as $(a+b) \bmod N$, $(a-b) \bmod N$, and $(ab) \bmod N$, respectively. For example, with $N = 256$, the result of $255+4$ in modular arithmetic is given by $(255+4) \bmod 256 = 3$ (since $259 \equiv 3 \pmod{256}$), and $1-254$ also evaluates to $(1-254) \bmod 256 = 3$ (since $-253 \equiv 3 \pmod{256}$).

In C, arithmetic operations involving unsigned integers are defined modulo $N = 2^n$ where n is the number of bits used to represent the particular unsigned integer type. Thus, the term *overflow* is somewhat a misnomer when dealing with unsigned integers. Indeed, the C99 standard (§6.2.5/9) states the following:

> A computation involving unsigned operands can never overflow, because a result that cannot be represented by the resulting unsigned integer type is reduced modulo the number that is one greater than the largest value that can be represented by the resulting type.

The situation for signed integers is somewhat more complicated due to the fact that existing C standards do not require a specific binary representation of signed integers. As a consequence, signed integer overflow is undefined behavior in C.

Before we discuss what is perhaps the most common signed integer representation, the so-called "two's complement" representation, we start by taking a brief look at the so-called "sign and magnitude", "one's complement", and "offset binary" representations. The sign and magnitude representation reserves one bit for storing the sign. The most significant bit, c_{n-1}, represents the sign (by convention, $c_{n-1} = 1$ corresponds to a negative number), and the remaining $n-1$ bits represent the magnitude, i.e., the magnutude is given by $(c_{n-2} \cdots c_0)_b$. The one's complement of an n-bit binary number $x = (c_{n-1} c_{n-2} \cdots c_0)_b$ is defined as the number

$$2^n - 1 - x$$

which corresponds to the bitwise complement of x (zeros become ones and ones become zeros). For example, the one's complement of the 4-bit number $(0010)_b$ is $(1101)_b$. The one's complement representation of signed integers coincides with the sign and magnitude representation for all nonnegative numbers, but negative numbers are represented as the one's complement of their positive counterparts. Both the sign and magnitude and the one's complement representations feature a "signed zero" since zero has two distinct representations. As a consequence, the set of representable integers using n bits is $\mathbb{Z} \cap [-2^{n-1}+1, 2^{n-1}-1]$. The offset binary representation uses an implicit offset that is determined by the minimal negative value. For example, the range from -128 to 127 is mapped to the range 0 to 255 by adding an offset of 128. The offset binary representation is also sometimes referred to as "excess-K" where K is the offset.

The two's complement representation uses n bits to encode integers in the set $\mathbb{Z} \cap [-2^{n-1}, 2^{n-1}-1]$. The two's complement of $x = (c_{n-1} c_{n-2} \cdots c_0)_b$ is defined as the number

$$(2^n - x) \bmod 2^n,$$

and hence $2^n - x \equiv -x \pmod{2^n}$, i.e., the two's complement of x is congruent modulo 2^n to $-x$. It is therefore natural to represent negative numbers as the two's complement of their positive counterparts with the exception of the most negative number, -2^{n-1}, which is represented by the number $x = (10 \cdots 0)_b = 2^{n-1}$. For example, the 4-bit number $(0100)_2$ represents the number 4, its two's complement $2^4 - 4 = 12 = (1100)_2$ represents -4, and $(1000)_2 = 8$ represents the most

negative number -8. It is easy to check that there is no signed zero since the two's complement of 0 is 0. Moreover, the most negative number, -2^{n-1}, is also its own two's complement; we will discuss the implications of this later in this note. The following table shows the unsigned and signed value of the same 8-bit binary representation using the different encodings.

Binary representation	Unsigned	Two's complement	Sign and magnitude	One's complement	Excess-128
00000000	0	0	0	0	-128
00000001	1	1	1	1	-127
00000010	2	2	2	2	-126
⋮	⋮	⋮	⋮	⋮	⋮
01111110	126	126	126	126	-2
01111111	127	127	127	127	-1
10000000	128	-128	-0	-127	0
10000001	129	-127	-1	-126	1
⋮	⋮	⋮	⋮	⋮	⋮
11111110	254	-2	-126	-1	126
11111111	255	-1	-127	-0	127

The fact that the two's complement of a number x is congruent modulo 2^n to $-x$ implies that modular arithmetic operations such as addition, subtraction, and multiplication can be performed using the same arithmetic logical unit (ALU) that is used for unsigned integers. For example, the operation $x-y$ can be performed modulo 2^n by adding the two's complement of y to x, i.e., we have that

$$x - y \equiv x + (2^n - y) \pmod{2^n}.$$

Recall, however, that signed integer overflow is undefined behavior in C. Several compilers have options or extensions that make it possible to either treat signed integer overflow as two's complement wrap-around (i.e., modular arithmetic) or "trap" overflow (e.g., abort the program if overflow occurs). Both GCC and Clang have such options, namely `-fwrapv` which assumes two's complement wrap-around and `-ftrapv` which traps (both signed and unsigned) integer overflow. GCC and Clang also provide a number of (nonstandard) built-in functions for simple arithmetic operations with overflow checking. For example, the function `__builtin_sadd_overflow` implements this for signed integer addition whereas `__builtin_smul_overflow` implements signed integer multiplication. Although these compiler-specific extensions can be quite useful, they can also be detrimental to performance and portability.

We end this section by noting that in C, integer division `a/b` rounds towards zero, and the modulo operation `a%b` yields the *signed* remainder. In fact, the condition `a == (a/b)*b + a%b` is always true, and this implies that the sign of `a%b` is the same as that of `a`.

1.2 Fixed-point numbers

Fixed-point numbers are closely related to integers in that a number with a fixed number of digits before and after the radix point,

$$x = s\,(c_{n-1}c_{n-2}\cdots c_0.d_1d_2\cdots d_l)_b,$$

may be expressed as

$$x = s\,(c_{l+n-1}c_{l+n-2}\cdots c_0)_b \cdot b^{-l},$$

i.e., x is the product of an $l+n$ digit integer and an implicit scale factor b^{-l}. Thus, storing a fixed-point number is essentially no different from storing an integer. Adding and subtracting fixed-point numbers amount to integer arithmetic, but multiplication and division require an additional scaling operation to preserve the radix point. For example, the multiplication of two base-10 fixed-point numbers $x = 3.14$ and $y = 2.99$ (i.e., $b = 10$, $n = 1$, and $l = 2$) may be expressed as

$$xy = (314 \cdot 299) \cdot 10^{-4} = 938.86 \cdot 10^{-2},$$

and hence the result must be rounded to in order to preserve the fixed-point format (we will return to the topic of rounding later in this note). We obtain the result $939 \cdot 10^{-2} = 9.39$ if we round 938.86 towards the nearest integer. Given a real number x, we define $\mathrm{fi}(x)$ as the fixed-point number with a given number of digits that is obtained by rounding x. Thus, if x is rounded towards the nearest representable number, then

$$\mathrm{fi}(x) = x + \delta, \quad |\delta| \leq b^{-l},$$

where δ represents the rounding error. The multiplication of two fixed-point numbers x and y therefore satisfies

$$\mathrm{fi}(xy) = xy + \delta, \qquad |\delta| \leq b^{-l}/2,$$

or equivalently, the absolute error

$$e_{\mathrm{abs}} = |\mathrm{fi}(xy) - xy| = |\delta|$$

is bounded by $b^{-l}/2$. However, the relative error, which may be defined as

$$e_{\mathrm{rel}} = \frac{|\mathrm{fi}(xy) - xy|}{|xy|} = \frac{|\delta|}{|xy|}, \quad xy \neq 0,$$

can be large. For example, if we take $x = y = 1 \cdot b^{-l}$, the fixed-point product $\mathrm{fi}(xy)$ is rounded to zero, and hence the relative error is $e_{\mathrm{rel}} = 1$. We note that arithmetic operations involving two fixed-point numbers can also result in overflow. For example, the product of two three-digit numbers $x = 600 \cdot 10^{-2}$ and $y = 205 \cdot 10^{-2}$ is $xy = 1230 \cdot 10^{-2}$ which clearly requires an additional digit.

Fixed-point numbers are used extensively in various digital signal processing applications, but they are rarely used for general-purpose computing. This is, in part, because of their inability to efficiently and accurately represent both large and small numbers. To illustrate this, consider the decimal numbers $1.43 \cdot 10^4$ and $4.24 \cdot 10^{-8}$ which both have three nonzero digits. Representing these numbers using the same fixed-point format would require (at least) 15 decimal digits. Existing C standards do not provide any fixed-point data types, but some C compilers provide a number of such types through non-standard extensions on certain platforms.

1.3 Floating-point numbers

Floating-point numbers may be viewed as a simple extension of fixed-point numbers in which the radix point is determined by a variable exponent. The floating-point format is therefore more versatile than the fixed-point format.

The floating-point representation of real number x with at most p nonzero digits may be expressed as

$$x = s\,(d_0.d_1d_2\ldots d_{p-1})_b \cdot b^E, \tag{1}$$

where the exponent E is an integer that determines the position of the radix point. The number of digits, p, is referred to as the *precision*, and the number $(d_0.d_1d_2\ldots d_{p-1})_b$ is called the *mantissa* or *significant*. The *machine epsilon*, denoted ε, may be defined as the distance from the number 1 to the closest representable number that is larger than one, i.e.,

$$\varepsilon = (0.00\ldots1)_b \cdot b^0 = b^{-(p-1)}.$$

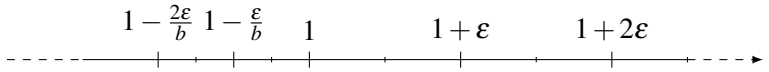
We will refer to half of the machine precision as the *unit round-off*, denoted $u = \varepsilon/2$.

The floating-point representation of a number is not necessarily unique. For example, if $b = 10$ and $p = 5$, then $x = 1.3800 \cdot 10^0$ may also be expressed as $x = 0.1380 \cdot 10^1$ and $x = 0.0138 \cdot 10^2$. The representation of a number is said to be *normal* or in *normalized form* if $d_0 \neq 0$.

A direct consequence of the floating-point format is that the representable numbers are not uniformly spaced. Indeed, given a floating-point number x, the gap between $|x|$ and the next larger representable number depends on the exponent E that is used in the normal representation of x. This gap is referred to as the *unit in the last place* and is given by

$$\text{ulp}(x) = \varepsilon \cdot b^E.$$

For example, if $b = 10$ and $p = 3$, then we have $\varepsilon = 10^{-2}$, and hence $\text{ulp}(1.00 \cdot 10^1) = 10^{-1}$ and $\text{ulp}(5.34 \cdot 10^6) = 10^4$. The uneven spacing of floating-point numbers is illustrated in the figure below.



The range of possible exponents, say, $E_{\min} \leq E \leq E_{\max}$, determines the dynamic range of a floating-point format. The largest representable number is obtained by letting $E = E_{\max}$ and $d_i = b-1$ for $i = 0,\ldots,p-1$, i.e.,

$$N_{\max} = (b-1)\sum_{i=0}^{p-1} b^{E_{\max}-i} = b^{E_{\max}}(b - b^{-(p-1)}).$$

The smallest representable normal number is obtained by letting $E = E_{\min}$, $d_0 = 1$, and $d_i = 0$ for $i = 1,\ldots,p-1$, i.e.,

$$N_{\min} = b^{E_{\min}}.$$

Nonzero numbers that are smaller than $N_{\min}$ must have $d_0 = 0$ and $E = E_{\min}$, and these numbers are referred to as *subnormal* (aka *denormal*) numbers. The smallest subnormal number is obtained by letting $E = E_{\min}$, $d_i = 0$ for $i = 0,\ldots,p-2$, and $d_{p-1} = 1$, i.e., $b^{E_{\min}-(p-1)}$.

As an example of a floating-point number system, we now consider the case where $b = 2$, $p = 3$, and $-1 \leq E \leq 1$. The resulting machine epsilon is $\varepsilon = 2^{-2} = 0.25$, the largest representable number is $N_{\max} = 2^2(1-2^{-3}) = 3.5$, the smallest normal number is $N_{\min} = 2^{-1} = 0.5$, and the smallest subnormal number is $2^{-3} = 0.125$. The set of representable numbers is shown in the figure below (subnormal numbers are shown with red markers).

We note that finite-precision floating-point numbers are rational numbers, and hence a floating-point format essentially provides a rational number approximation to real numbers within some range.

1.3.1 Rounding

Arithmetic operations that involve two p-digit floating-point numbers may yield a result with many more digits. The result must therefore be rounded in order to retain a fixed precision (i.e., a fixed number of digits). This can be done in many ways, e.g., using on of the following methods:

- round to nearest, ties to even,
- round to nearest, ties away from zero,
- round toward zero (aka *truncation*),
- round up (round toward $+\infty$),
- round down (round toward $+\infty$).

The last three methods a so-called *directed rounding* methods, and these may introduce a systematic bias in calculations. The first method, round-to-nearest with the ties-to-even tie-breaking rule, avoids such bias. This method is also known as *banker's rounding* and *scientific rounding*. As an example of how the tie-breaking rule works, suppose we have a floating-point format with $b = 10$ and $p = 2$. The number $x = 1.05$ is then exactly in the middle of two representable numbers, namely 1.0 and 1.1. Using the ties to even tie-breaking rule, x is rounded to 1.0 since this is the option for which the last digit is even. In contrast, the *away from zero* tie-breaking rule, which is typically the one being taught in elementary school, would result in x being rounded to 1.1. The different rounding methods are illustrated with some examples in the table below.

x	Nearest, even	Nearest, away from 0	Truncation	Up	Down
1.05	1.0	1.1	1.0	1.1	1.0
1.15	1.2	1.2	1.1	1.2	1.1
-1.05	-1.0	-1.1	-1.0	-1.0	-1.1
-1.15	-1.2	-1.2	-1.1	-1.1	-1.2

Given a real number x, we define $\mathrm{fl}(x)$ as the floating-point number with given precision that is obtained by rounding x. This operation satisfies

$$\mathrm{fl}(x) = x + \xi, \tag{2}$$

where ξ represents the rounding error. This error depends on the magnitude of x, the precision p, and the rounding method. If x is normal and it rounded to the nearest representable number, then the rounding error ξ satisfies $|\xi| \leq \frac{1}{2}\mathrm{ulp}(x)$, and directed rounding implies the bound $|\xi| \leq \mathrm{ulp}(x)$. It follows from the definition of $\mathrm{ulp}(x)$ that $\mathrm{ulp}(x) \leq \varepsilon|x|$, and as a consequence,

we have that $|\xi| \leq u|x|$ or $|\xi| \leq 2u|x|$, depending on the method of rounding. These bounds allow us to characterize the rounding operation as

$$\text{fl}(x) = x(1+\delta), \quad |\delta| \leq \alpha u, \tag{3}$$

where $x\delta$ corresponds to the rounding error ξ in (2) and $\alpha \in \{1,2\}$, depending on the rounding method ($\alpha = 1$ correponds to round to nearest whereas $\alpha = 2$ corresponds to directed rounding). We will see that this characterization is useful for analysis purposes.

1.3.2 Arithmetic operations and rounding errors

Let x and y be two finite-precision floating-point numbers, and let $\odot$ denote an arithmetic operator (say, addition, subtraction, multiplication, or division). We will assume that $\text{fl}(x \odot y)$ is the nearest representable floating-point number (with some tie-breaking rule) and that it has a normal representation (i.e., $\text{fl}(x \odot y)$ is not a subnormal number). The rounded result satisfies

$$\text{fl}(x \odot y) = (x \odot y)(1+\delta), \quad |\delta| \leq u,$$

and this implies that the absolute error incurred by rounding satisfies the bound

$$e_{\text{abs}} = |\text{fl}(x \odot y) - x \odot y| = |\delta(x \odot y)| \leq u|(x \odot y)|,$$

while the relative error satisfies the bound

$$e_{\text{rel}} = \frac{|\text{fl}(x \odot y) - x \odot y|}{|x \odot y|} = |\delta| \leq u, \quad x \odot y \neq 0.$$

In other words, the relative error is bounded by u.

1.3.3 Cancellation

We now turn to the case where one of the operands cannot be represented as a finite-precision floating-point number. Specifically, we will consider the case where x must be rounded and where $\odot$ corresponds to subtraction. In other words, we will store and work with $\hat{x} = \text{fl}(x) = x(1+\delta_1)$ instead of x. We will see that this additional rounding operation can lead to a relative error that is significantly larger than u when subtracting two nearly equal numbers, and as a result, the number of significant digits is reduced considerably. This phenomenon is known as *catastrophic cancellation*, and we now demonstrate this with an example. We will use a floating-point format with $b = 10$ and $p = 5$ and the two nearly equal numbers $x = 4/9 = 0.\overline{44}$ and $y = 4.4442 \cdot 10^{-1}$. The number x is a non-terminating decimal, and the nearest floating-point number is $\hat{x} = \text{fl}(x) = 4.4444 \cdot 10^{-1}$. The floating-point operation $\text{fl}(\text{fl}(x) - y)$ is therefore equal to

$$\text{fl}(\hat{x} - y) = 0.0002 \cdot 10^{-1}.$$

To normalize this number, we move the decimal point four positions to the right and adjust the exponent accordingly, i.e., the normalized result is

$$\text{fl}(\hat{x} - y) = 2.0000 \cdot 10^{-5}.$$

In comparison, the exact result is

$$x - y = \frac{22}{9} \cdot 10^{-5} = 2.\overline{44} \cdot 10^{-5},$$

which confirms that $\text{fl}(\hat{x}-y)$ only has a single significant digit. Moreover, the relative error is

$$e_{\text{rel}} = \frac{|\text{fl}(\hat{x}-y)-(x-y)|}{|x-y|} = \frac{2}{11} = 0.\overline{18},$$

which is significantly larger than $u = \varepsilon/2 = 0.0005$. Note also that in this example, we have that $\text{fl}(\text{fl}(x)-y) = \hat{x}-y$, i.e., it is only the rounding of x to the nearest representable floating-point number that incurs a rounding error.

Loss of significance does not always result in catastrophic cancellation. Indeed, we have seen that when x and y are both floating-point numbers, then $\text{fl}(x \odot y) = (x \odot y)(1+\delta)$ for some $|\delta| \leq u$, and hence the relative error is bounded by u. For example, if $x = 1.0002$ and $y = 1.0000$, then $\text{fl}(x-y) = 2.0000 \cdot 10^{-4}$ has a single significant digit, but the relative error is zero. This is referred to as *benign cancellation*.

We now more formally analyze the loss of significant digits when subtracting two normal floating-point numbers x and y with precision p and in base b. Without loss of generality, we may assume that $x > y > 0$, and we will express the numbers as $x = m_x \cdot b^{E_x}$ and $y = m_y \cdot b^{E_y}$ where the mantissas are normalized, i.e., $m_x \in [1,b)$ and $m_y \in [1,b)$. In exact arithmetic we have that

$$\begin{aligned} x-y &= \left(m_x - m_y \cdot b^{E_y-E_x}\right) \cdot b^{E_x} \\ &= m_x(1-y/x) \cdot b^{E_x}, \end{aligned}$$

or equivalently,

$$x-y = m_x(1-y/x) \cdot b^{\gamma} \cdot b^{E_x-\gamma} = m \cdot b^{E},$$

with mantissa $m = m_x(1-y/x) \cdot b^{\gamma}$ and exponent $E = E_x - \gamma$. Normalizing the mantissa m therefore amounts to choosing $\gamma \in \mathbb{Z}$ such that $m \in [1,b)$. Equivalently, we must move the radix point by γ places to the right and adjust the exponent accordingly to normalize the mantissa, thereby loosing γ significant digits. This leads us to the result that if

$$b^{-\beta} \leq 1 - \frac{y}{x} \leq b^{-\alpha}, \quad \alpha, \beta \in \mathbb{N}_0,$$

then at most β and at least α significant digits will be lost in the subtraction $x-y$ (i.e., $\alpha \leq \gamma \leq \beta$). This result is also known as the *loss of precision theorem*. In the previous example where $\hat{x} = 4.4444 \cdot 10^{-1}$ and $y = 4.4442 \cdot 10^{-1}$ with $b = 10$ and $p = 5$, we have that

$$1 - \frac{\hat{x}}{y} \approx 0.000045,$$

and hence $10^{-5} \leq 1-\hat{x}/y \leq 10^{-4}$. Consequently, at least 4 and at most 5 significant digits will be lost in the subtraction $\hat{x}-y$.

We end this section by deriving an upper bound for the relative error associated with the floating-point computation $\text{fl}(\text{fl}(x)-y)$. Expressing $\text{fl}(x)$ as $\hat{x} = x(1+\delta_1)$ for some $|\delta_1| \leq u$, we may expand $\text{fl}(\text{fl}(x)-y)$ as

$$\begin{aligned} \text{fl}(\hat{x}-y) &= (x(1+\delta_1)-y)(1+\delta_2) \\ &= (x-y) + (\delta_1+\delta_2+\delta_1\delta_2)x - \delta_2 y, \end{aligned}$$

where $|\delta_1| \leq u$ and $|\delta_2| \leq u$. Now, assuming that $x \neq y$, the relative error is given by

$$\begin{aligned} e_{\text{rel}} &= \frac{|\text{fl}(\hat{x}-y)-(x-y)|}{|x-y|} \\ &= \frac{|(\delta_1+\delta_2+\delta_1\delta_2)x-\delta_2 y|}{|x-y|}. \end{aligned}$$

It follows from the bounds on δ_1 and δ_2 that the numerator satisfies

$$\begin{aligned} |(\delta_1+\delta_2+\delta_1\delta_2)x-\delta_2 y| &\leq |(2u+u^2)x|+|uy| \\ &\leq (|x|+|y|)(2u+u^2), \end{aligned}$$

and hence

$$e_{\text{rel}} \leq \frac{|x|+|y|}{|x-y|}(2+u)u. \tag{4}$$

The factor $(2+u)u$ only depends on the machine precision whereas the factor $(|x|+|y|)/|x-y|$ only depends on the problem data x and y. Thus, for a given precision, the bound on the relative error is determined by the problem data. In particular, the bound is large when $|x-y|$ is small compared to $|x|+|y|$. It is therefore natural to ask how conservative or pessimistic is this bound? To answer this question, we first observe that the factor $(|x|+|y|)/|x-y|$ is equal to 1 if x and y have opposite signs. This corresponds to the addition of two numbers with the same sign (i.e., $x+(-y)$), and the resulting bound on the relative error reduces to $2u+u^2$ which is approximately equal to $2u$ when u is small. On the other hand, suppose x and y are nearly equal and have the same sign. For example, if we let $x = 1+u$ and $y = 1$, then $\text{fl}(x) = 1$ and $\text{fl}(\text{fl}(x)-1) = 0$, resulting in the relative error $e_{\text{rel}} = 1$. To compare with the bound (4), we first note that

$$\frac{|x|+|y|}{|x-y|} = \frac{2+u}{u} = \frac{4}{\varepsilon}+1 = 4b^{p-1}+1,$$

and hence we obtain the bound $e_{\text{rel}} \leq (4b^{p-1}+1)(2+u)u \approx 4$. We end this section by noting that the magnitude of this bound is comparable to the actual relative error which suggests that the bound is useful.

1.3.4 A standard for floating-point arithmetic

In 1980, Intel introduced the so-called 8087 floating-point unit (FPU) to accompany their 8086 line of CPUs. The purpose of the FPU was to speed up floating-point computations, as the 8086 line of CPUs did not support floating-point arithmetic in hardware. The 8087 supported three different floating-point formats (represented using 32 bits, 64 bits, and 80 bits, respectively) and could perform around 50,000 floating-point operations (FLOPS) per second. The design phase of the 8087 FPU spawned the development a floating-point standard, the so-called *IEEE Standard for Floating-Point Arithmetic* (aka *IEEE 754*), which was adopted in 1985 and most recently revised in 2019. The motivation for the standard was to improve portability and robustness, thereby making it easier to develop mathematical software. To quote William Kahan, one of the main contributors to the standard, "error-analysis tells us how to design floating-point arithmetic, like IEEE Standard 754, moderately tolerant of well-meaning ignorance among programmers." [1]

The IEEE 754 standard defines a number of different floating-point formats of the form (1). Each of these formats is defined by three parameters: the base b, the precision p, the maximum

exponent $E_{\max}$. The minimum exponent is given by $E_{\min} = 1 - E_{\max}$ for all of these formats. Some of the formats and their parameters are shown in the table below.

Format	b	p	$E_{\max}$
binary16 (half)	2	11	15
binary32 (single)	2	24	127
binary64 (double)	2	53	1023
binary128 (quadruple)	2	113	16383
decimal32	10	7	96
decimal64	10	16	384
decimal128	10	34	6144

In addition to normal and subnormal floating-point numbers of the form (1), these formats also define a way to represent the following:

- signed zero (-0 and $+0$)
- signed infinity ($-\infty$ and ∞)
- "not a number" or NaN.

Despite the presence of two zeros, the standard requires that comparisons ignore the sign of zero (i.e., $+0 = -0$). The two infinities provide a way to represent that a number is outside the range of representable numbers, e.g., if the result of an operation overflows. Similarly, NaN provides a way to represent something undefined and a mechanism for handling exceptions. For example, the result of an invalid operation yields a NaN result. Some examples are $0 \cdot \infty$, $\infty - \infty$, $0/0$, ∞/∞, and the square root of a negative number.

The numeric suffix in each format name (e.g., "64" in "binary64") designates the number of bits that is used to encode the floating-point format. For example, binary64 uses 64 bits: 1 bit for the sign, 52 bits for the mantissa, and 11 bits for the exponent. Notice that the number of bits for the mantissa is one less than the precision. The reason is that a normal floating-point number in base 2 always has $d_0 = 1$, so there is no need to store this bit explicitly. Notice also that the 11 bits for the exponent allow 2048 different values, but the range from $E_{\min}$ to $E_{\max}$ uses only 2046 of these. The two remaining values are used to represent zero, subnormals, infinity, and NaN.

The IEEE 754 standard requires four different rounding modes. It also requires that basic arithmetic operations (addition, subtraction, multiplication, division), the square root operation, and a so-called *fused multiply add* (FMA) operation are correctly rounded, as determined by the applicable rounding mode. It is not a requirement that other functions (e.g., trigonometric functions and the exponential and logarithm functions) are correctly rounded, but it is recommended.

We end this section by mentioning that the standard also defines some optional *extended* formats associated with the basic format. These formats can be used to avoid overflow and reduce round-off errors in intermediate results. The following table shows the minumum precision and exponent range for the extended formats.

Extended formats	$p \geq$	$E_{\max} \geq$
binary32 ext.	32	1023
binary64 ext.	64	16383

Extended formats	$p \geq$	$E_{\max} \geq$
binary128 ext.	128	65535
decimal64 ext.	22	6144
decimal128 ext.	40	24576

The most widely used extended format is perhaps the 80-bit format (aka *double extended*) that debuted with the Intel 8087 FPU. This format coincides with the minimum requirements for the binary64 extended format defined in the IEEE 754 standard.

We end this section by noting that in addition to the floating-point formats included in the IEEE 754 standard, there are a number of other formats. Perhaps the most notable example is the "bfloat16" format (also known as "brain floating-point"), which was developed in the 2010s by Google Brain for AI processors/accelerators such as Google's tensor processing units (TPUs). The format is similar to the "binary16" format but has a wider dynamic range (larger $E_{\max}$) and at the expense of precision (smaller p), as shown in the following table.

Format	b	p	$E_{\max}$
bfloat16	2	8	127

Interested readers can find additional information about floating-point formats and arithmetic in [2]–[6].

References

[1] W. Kahan, "Lecture notes on the status of IEEE standard 754 for binary floating-point arithmetic." 1997.

[2] D. Goldberg, "What every computer scientist should know about floating-point arithmetic," *ACM Computing Surveys (CSUR)*, vol. 23, no. 1, pp. 5–48, Mar. 1991, doi: 10.1145/103162.103163.

[3] C. Severance, "IEEE 754: An interview with William Kahan," *Computer*, vol. 31, no. 3, pp. 114–115, Mar. 1998, doi: 10.1109/mc.1998.660194.

[4] M. Overton, *Numerical computing with IEEE floating point arithmetic*. Philadelphia, PA: Society for Industrial; Applied Mathematics, 2001.

[5] D. G. Hough, "The IEEE standard 754: One for the history books," *Computer*, vol. 52, no. 12, pp. 109–112, Dec. 2019, doi: 10.1109/mc.2019.2926614.

[6] "IEEE standard for floating-point arithmetic," *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pp. 1–84, 2019.