

Floating point

From Wikipedia, the free encyclopedia

 *This article is about the method of representing a number. For the album by John McLaughlin, see *Floating Point*.*

In **computing**, **floating point** is the formulaic representation which approximates a **real number** so as to support a **trade-off** between range and **precision**. A number is, in general, represented approximately to a fixed number of **significant digits** (the **significand**) and scaled using an **exponent**; the base for the scaling is normally two, ten, or sixteen. A number that can be represented exactly is of the following form:

$$\text{significand} \times \text{base}^{\text{exponent}}$$

where

$$\text{significand} \in \mathbb{Z}, \text{base} \in \mathbb{N}, \text{exponent} \in \mathbb{Z}.$$

For example:

$$1.2345 = \underbrace{12345}_{\text{significand}} \times \underbrace{10^{-4}}_{\text{base}^{\text{exponent}}}$$

The term *floating point* refers to the fact that a number's **radix point** (*decimal point*, or, more commonly in computers, *binary point*) can "float"; that is, it can be placed anywhere relative to the significant digits of the number. This position is indicated as the exponent component, and thus the floating-point representation can be thought of as a kind of **scientific notation**.

A floating-point system can be used to represent, with a fixed number of digits, numbers of different **orders of magnitude**: e.g. the distance between galaxies or the diameter of an atomic nucleus can be expressed with the same unit of length. The result of this dynamic range is that the numbers that can be represented are not uniformly spaced; the difference between two consecutive representable numbers grows with the chosen scale.^[1]

Over the years, a variety of floating-point representations have been used in computers. However, since the 1990s, the most commonly encountered representation is that defined by the **IEEE 754** Standard.

The speed of floating-point operations, commonly measured in terms of **FLOPS**, is an important characteristic of a computer system, especially for applications that involve intensive mathematical calculations.

Contents

- 1

Overview
- 1.1

Floating-point numbers
- 1.2

Alternatives to floating-point numbers
- 1.3

History
- 2

Range of floating-point numbers
- 3

IEEE 754: floating point in modern computers
- 3.1

Internal representation
- 3.1.1

Piecewise linear approximation to exponential and logarithm
- 3.2

Special values
- 3.2.1

Signed zero
- 3.2.2

Subnormal numbers
- 3.2.3

Infinities
- 3.2.4

NaNs
- 3.2.5

IEEE 754 design rationale
- 4

Representable numbers, conversion and rounding
- 4.1

Rounding modes
- 5

Floating-point arithmetic operations
- 5.1

Addition and subtraction
- 5.2

Multiplication and division
- 6

Dealing with exceptional cases
- 7

Accuracy problems
- 7.1

Incidents
- 7.2

Machine precision and backward error analysis
- 7.3

Minimizing the effect of accuracy problems
- 8

See also
- 9

Notes and references
- 10

Further reading
- 11

External links



An early electromechanical programmable computer, the **Z3**, included floating-point arithmetic (replica on display at **Deutsches Museum** in Munich).

Overview

Floating-point numbers

A number representation (called a **numeral system** in mathematics) specifies some way of encoding a number, usually as a string of digits.

There are several mechanisms by which strings of digits can represent numbers. In common mathematical notation, the digit string can be of any length, and the location of the **radix point** is indicated by placing an explicit **"point" character** (dot or comma) there. If the radix point is not specified, then it is implicitly assumed to lie at the right (at the least significant) end of the string (that is, the number is an **integer**). In **fixed-point** systems, some specific assumption is made about where the radix point is located in the string; for example, the convention could be that the string consists of 8 decimal digits with the decimal point in the middle, so that "00012345" represents the value 1.2345.

In **scientific notation**, the given number is scaled by a **power of 10**, so that it lies within a certain range—typically between 1 and 10, with the radix point appearing immediately after the first digit. The scaling factor, as a power of ten, is then indicated separately at the end of the number. For example, the orbital period of **Jupiter's** moon **Io** is 152 853.5047 seconds, a value that would be represented in standard-form scientific notation as 1.528 535 047 × 10⁵ seconds.

Floating-point representation is similar in concept to scientific notation. Logically, a floating-point number consists of:

- A signed (meaning negative or non-negative) digit string of a given length in a given **base** (or **radix**). This digit string is referred to as the ***significand***,

mantissa, or *coefficient*. The length of the significand determines the *precision* to which numbers can be represented. The radix point position is assumed always to be somewhere within the significand—often just after or just before the most significant digit, or to the right of the rightmost (least significant) digit. This article generally follows the convention that the radix point is set just after the most significant (leftmost) digit.

- A signed integer *exponent* (also referred to as the *characteristic*, or *scale*), which modifies the magnitude of the number.

To derive the value of the floating-point number, the *significand* is multiplied by the *base* raised to the power of the *exponent*, equivalent to shifting the radix point from its implied position by a number of places equal to the value of the exponent—to the right if the exponent is positive or to the left if the exponent is negative.

Using base-10 (the familiar *decimal* notation) as an example, the number 152 853.5047, which has ten decimal digits of precision, is represented as the significand 1528535047 together with 5 as the exponent. To determine the actual value, a decimal point is placed after the first digit of the significand and the result is multiplied by 10⁵ to give 1.528 535 047 × 10⁵, or 152 853.5047. In storing such a number, the base (10) need not be stored, since it will be the same for the entire range of supported numbers, and can thus be inferred.

Symbolically, this final value is:

$$\frac{s}{b^{p-1}} \times b^e$$

where *s* is the significand (ignoring any implied decimal point), *p* is the precision (the number of digits in the significand), *b* is the base (in our example, this is the number *ten*), and *e* is the exponent.

Historically, several number bases have been used for representing floating-point numbers, with base two (*binary*) being the most common, followed by base ten (decimal), and other less common varieties, such as base sixteen (*hexadecimal notation*), and even base three (see *Setun*).

A floating-point number is a *rational number*, because it can be represented as one integer divided by another; for example 1.45 × 10³ is (145/100)*1000 or 145 000/100. The base determines the fractions that can be represented; for instance, 1/5 cannot be represented exactly as a floating-point number using a binary base, but 1/5 can be represented exactly using a decimal base (0.2, or 2 × 10^{−1}). However, 1/3 cannot be represented exactly by either binary (0.010101...) or decimal (0.333...), but in *base 3*, it is trivial (0.1 or 1×3^{−1}). The occasions on which infinite expansions occur depend on the base and its *prime factors*, as described in the article on *Positional Notation*.

The way in which the significand (including its sign) and exponent are stored in a computer is implementation-dependent. The common IEEE formats are described in detail later and elsewhere, but as an example, in the binary single-precision (32-bit) floating-point representation, *p* = 24, and so the significand is a string of 24 *bits*. For instance, the number *π*'s first 33 bits are:

11001001 00001111 11011010 10100010 0

If the leftmost bit is considered the 1st bit, then the 24th bit is zero and the 25th bit is 1; thus, in rounding to 24 bits, let's attribute to the 24th bit the value of the 25th, yielding:

11001001 00001111 11011011

When this is stored using the IEEE 754 encoding, this becomes the significand *s* with *e* = 1 (where *s* is assumed to have a binary point to the right of the first bit) after a left-adjustment (or *normalization*) during which leading or trailing zeros are truncated should there be any, which is unnecessary in this case; as a result of this normalization, the first bit of a non-zero binary significand is always 1, so it need not be stored, saving one bit of storage. In other words, from this representation, *π* is calculated as follows:

$$\begin{aligned} & \left(1 + \sum_{n=1}^{p-1} \text{bit}_n \times 2^{-n} \right) \times 2^e \\ &= \left(1 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-4} + 1 \times 2^{-7} + \cdots + 1 \times 2^{-23} \right) \times 2^1 \\ &= 1.5707964 \times 2 \\ &= 3.1415928 \end{aligned}$$

where *π* is the normalized significand's *n*th bit from the left, where counting starts with 1. Normalization, which is reversed by the addition of the implicit one, can be thought of as a form of compression; it allows a binary significand to be compressed into a field one bit shorter than the maximum precision, at the expense of extra processing.

Alternatives to floating-point numbers [edit]

The floating-point representation is by far the most common way of representing in computers an approximation to real numbers. However, there are alternatives:

- **Fixed-point** representation uses integer hardware operations controlled by a software implementation of a specific convention about the location of the binary or decimal point, for example, 6 bits or digits from the right. The hardware to manipulate these representations is less costly than floating point, and it can be used to perform normal integer operations, too. Binary fixed point is usually used in special-purpose applications on embedded processors that can only do integer arithmetic, but decimal fixed point is common in commercial applications.
- **Binary-coded decimal** (BCD) is an encoding for decimal numbers in which each digit is represented by its own binary sequence. It is possible to implement a floating-point system with BCD encoding.
- **Logarithmic number systems** represent a real number by the logarithm of its absolute value and a sign bit. The value distribution is similar to floating point, but the value-to-representation curve (*i.e.*, the graph of the logarithm function) is smooth (except at 0). Conversely to floating-point arithmetic, in a logarithmic number system multiplication, division and exponentiation are simple to implement, but addition and subtraction are complex. The *level index arithmetic* of Clenshaw, Olver, and Turner is a scheme based on a generalized logarithm representation.
- Where greater precision is desired, floating-point arithmetic can be implemented (typically in software) with variable-length significands (and sometimes exponents) that are sized depending on actual need and depending on how the calculation proceeds. This is called *arbitrary-precision* floating-point arithmetic.
- Some numbers (*e.g.*, 1/3 and 1/10) cannot be represented exactly in binary floating-point, no matter what the precision is. Software packages that perform *rational arithmetic* represent numbers as fractions with integral numerator and denominator, and can therefore represent any rational number exactly. Such packages generally need to use *"bignum"* arithmetic for the individual integers.
- **Computer algebra systems** such as *Mathematica* and *Maxima* can often handle irrational numbers like *π* or $\sqrt{3}$ in a completely "formal" way, without dealing with a specific encoding of the significand. Such a program can evaluate expressions like "*sin 3π*" exactly, because it is programmed to process the underlying mathematics directly, instead of using approximate values for each intermediate calculation.

History [edit]

In 1914, *Leonardo Torres y Quevedo* designed an electro-mechanical version of *Charles Babbage's Analytical Engine*, and included floating-point arithmetic.^[2] In 1938, *Konrad Zuse* of Berlin completed the *Z1*, the first binary, programmable mechanical computer;^[3] it uses a 24-bit binary floating-point number representation with a 7-bit signed exponent, a 16-bit significand (including one implicit bit), and a sign bit. The more reliable relay-based *Z3*, completed in 1941, has representations for both positive and negative infinities; in particular, it implements defined operations with infinity, such as $1/\infty = 0$, and it stops on undefined operations, such as $0 \times \infty$.

Zuse also proposed, but did not complete, carefully rounded floating-point arithmetic that includes $\pm\infty$ and NaN representations, anticipating features of the IEEE Standard by four decades.^[4] In contrast, [von Neumann](#) recommended against floating-point numbers for the 1951 [IAS machine](#), arguing that fixed-point arithmetic is preferable.^[5]

The first *commercial* computer with floating-point hardware was Zuse's [Z4](#) computer, designed in 1942–1945. In 1946, Bell Laboratories introduced the Mark V, which implements [decimal floating-point numbers](#).^[6]

The [Pilot ACE](#) has binary floating-point arithmetic, and it became operational in 1950 at [National Physical Laboratory, UK](#). 33 were later sold commercially as the [English Electric DEUCE](#). The arithmetic is actually implemented in software, but with a one megahertz clock rate, the speed of floating-point and fixed-point operations in this machine were initially faster than those of many competing computers.

The mass-produced [IBM 704](#) followed in 1954; it introduced the use of a [biased exponent](#). For many decades after that, floating-point hardware was typically an optional feature, and computers that had it were said to be "scientific computers", or to have "scientific computing" capability. It was not until the launch of the Intel i486 in 1989 that *general-purpose* personal computers had floating-point capability in hardware as a standard feature.

The [UNIVAC 1100/2200 series](#), introduced in 1962, supports two floating-point representations:

- Single precision*: 36 bits, organized as a 1-bit sign, an 8-bit exponent, and a 27-bit significand.
- Double precision*: 72 bits, organized as a 1-bit sign, an 11-bit exponent, and a 60-bit significand.

The [IBM 7094](#), also introduced in 1962, supports single-precision and double-precision representations, but with no relation to the UNIVAC's representations. Indeed, in 1964, IBM introduced proprietary hexadecimal floating-point representations in its [System/360](#) mainframes; these same representations are still available for use in modern [z/Architecture](#) systems. However, in 1998, IBM included IEEE-compatible binary floating-point arithmetic to its mainframes; in 2005, IBM also added IEEE-compatible decimal floating-point arithmetic.

Initially, computers used many different representations for floating-point numbers. The lack of standardization at the mainframe level was an ongoing problem by the early 1970s for those writing and maintaining higher-level source code; these manufacturer floating-point standards differed in the word sizes, the representations, and the rounding behavior and general accuracy of operations. Floating-point compatibility across multiple computing systems was in desperate need of standardization by the early 1980s, leading to the creation of the [IEEE-754](#) standard once the 32-bit (or 64-bit) [word](#) had become commonplace. This standard was significantly based on a proposal from Intel, which was designing the [i8087](#) numerical coprocessor; Motorola, which was designing the [68000](#) around the same time, gave significant input as well.

In 1989, mathematician and computer scientist [William Kahan](#) was honored with the [Turing Award](#) for being the primary architect behind this proposal; he was aided by his student (Jerome Coonen) and a visiting professor (Harold Stone).^[7]

Among the x86 innovations are these:

- A precisely specified floating-point representation at the bit-string level, so that all compliant computers interpret bit patterns the same way. This makes it possible to transfer floating-point numbers from one computer to another (after accounting for [endianness](#)).
- A precisely specified behavior for the arithmetic operations: A result is required to be produced as if infinitely precise arithmetic were used to yield a value that is then rounded according to specific rules. This means that a compliant computer program would always produce the same result when given a particular input, thus mitigating the almost mystical reputation that floating-point computation had developed for its hitherto seemingly non-deterministic behavior.
- The ability of exceptional conditions (overflow, divide by zero, etc.) to propagate through a computation in a benign manner and then be handled by the software in a controlled fashion.

Range of floating-point numbers [edit]

A floating-point number consists of two [fixed-point](#) components, whose range depends exclusively on the number of bits or digits in their representation. Whereas components linearly depend on their range, the floating-point range linearly depends on the significant range and exponentially on the range of exponent component, which attaches outstandingly wider range to the number.

On a typical computer system, a 'double precision' (64-bit) binary floating-point number has a coefficient of 53 bits (one of which is implied), an exponent of 11 bits, and one sign bit. Positive floating-point numbers in this format have an approximate range of 10^{−308} to 10³⁰⁸, because the range of the exponent is [−1022,1023] and 308 is approximately log₁₀(2¹⁰²³). The complete range of the format is from about −10³⁰⁸ through +10³⁰⁸ (see [IEEE 754](#)).

The number of normalized floating-point numbers in a system F (*B*, *P*, *L*, *U*) (where *B* is the base of the system, *P* is the precision of the system to *P* numbers, *L* is the smallest exponent representable in the system, and *U* is the largest exponent used in the system) is: 2(*B* − 1)(*B*^{*P*−1})(*U* − *L* + 1) + 1.

There is a smallest positive normalized floating-point number, Underflow level = UFL = *B*^{*L*} which has a 1 as the leading digit and 0 for the remaining digits of the significand, and the smallest possible value for the exponent.

There is a largest floating-point number, Overflow level = OFL = (1 − *B*^{−*P*})(*B*^{*U*+1}) which has *B* − 1 as the value for each digit of the significand and the largest possible value for the exponent.

In addition there are representable values strictly between −UFL and UFL. Namely, [positive and negative zeros](#), as well as [denormalized numbers](#).

IEEE 754: floating point in modern computers [edit]

Main article: [IEEE floating point](#)

The [IEEE](#) has standardized the computer representation for binary floating-point numbers in [IEEE 754](#) (a.k.a. IEC 60559). This standard is followed by almost all modern machines. IBM mainframes support [IBM's own hexadecimal floating point format](#) and [IEEE 754-2008 decimal floating point](#) in addition to the IEEE 754 binary format. The [Cray T90](#) series had an IEEE version, but the [SV1](#) still uses Cray floating-point format.

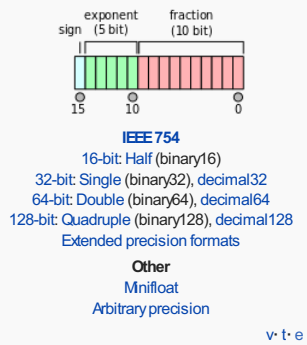
The standard provides for many closely related formats, differing in only a few details. Five of these formats are called *basic formats* and others are termed *extended formats*; three of these are especially widely used in computer hardware and languages:

- Single precision**, usually used to represent the "float" type in the C language family (though this is [not guaranteed](#)). This is a binary format that occupies 32 bits (4 bytes) and its significand has a precision of 24 bits (about 7 decimal digits).
- Double precision**, usually used to represent the "double" type in the C language family (though this is [not guaranteed](#)). This is a binary format that occupies 64 bits (8 bytes) and its significand has a precision of 53 bits (about 16 decimal digits).
- Double extended**, also called "extended precision" format. This is a binary format that occupies at least 79 bits (80 if the hidden/implicit bit rule is not used) and its significand has a precision of at least 64 bits (about 19 decimal digits). A format satisfying the minimal requirements (64-bit precision, 15-bit exponent, thus fitting on 80 bits) is provided by the [x86 architecture](#). In general on such processors, this format can be used with "long double" in the C language family (the [C99](#) and [C11](#) standards "IEC 60559 floating-point



Konrad Zuse, architect of the [Z3](#) computer, which uses a 22-bit binary floating-point representation.

Floating-point precisions



arithmetic extension- Annex F" recommend the 80-bit extended format to be provided as "long double" when available). On other processors, "long double" may be a synonym for "double" if any form of extended precision is not available, or may stand for a larger format, such as quadruple precision.

Increasing the precision of the floating point representation generally reduces the amount of accumulated [round-off error](#) caused by intermediate calculations.^[8]

Less common IEEE formats include:

- [Quadruple precision](#) (binary128). This is a binary format that occupies 128 bits (16 bytes) and its significand has a precision of 113 bits (about 34 decimal digits).
- [Double precision](#) (decimal64) and [quadruple precision](#) (decimal128) decimal floating-point formats. These formats, along with the [single precision](#) (decimal32) format, are intended for performing decimal rounding correctly.
- [Half](#), also called binary16, a 16-bit floating-point value.

Any integer with absolute value less than 2^{24} can be exactly represented in the single precision format, and any integer with absolute value less than 2^{53} can be exactly represented in the double precision format. Furthermore, a wide range of powers of 2 times such a number can be represented. These properties are sometimes used for purely integer data, to get 53-bit integers on platforms that have double precision floats but only 32-bit integers.

The standard specifies some special values, and their representation: positive [infinity](#) ($+\infty$), negative infinity ($-\infty$), a [negative zero](#) (-0) distinct from ordinary ("positive") zero, and "not a number" values ([NaNs](#)).

Comparison of floating-point numbers, as defined by the IEEE standard, is a bit different from usual integer comparison. Negative and positive zero compare equal, and every NaN compares unequal to every value, including itself. All values except NaN are strictly smaller than $+\infty$ and strictly greater than $-\infty$. Finite floating-point numbers are ordered in the same way as their values (in the set of real numbers).

A project for revising the IEEE 754 standard was started in 2000 (see [IEEE 754 revision](#)); it was completed and approved in June 2008. It includes decimal floating-point formats and a 16-bit floating-point format ("binary16"). binary16 has the same structure and rules as the older formats, with 1 sign bit, 5 exponent bits and 10 trailing significand bits. It is being used in the NVIDIA [Cg](#) graphics language, and in the openEXR standard.^[9]

Internal representation ^[edit]

Floating-point numbers are typically packed into a computer datum as the sign bit, the exponent field, and the significand or mantissa, from left to right. For the IEEE 754 binary formats (basic and extended) which have extant hardware implementations, they are apportioned as follows:

Type	Sign	Exponent	Significand field	Total bits	Exponent bias	Bits precision	Number of decimal digits
Half (IEEE 754-2008)	1	5	10	16	15	11	~3.3
Single	1	8	23	32	127	24	~7.2
Double	1	11	52	64	1023	53	~15.9
x86 extended precision	1	15	64	80	16383	64	~19.2
Quad	1	15	112	128	16383	113	~34.0

While the exponent can be positive or negative, in binary formats it is stored as an unsigned number that has a fixed "bias" added to it. Values of all 0s in this field are reserved for the zeros and [subnormal numbers](#); values of all 1s are reserved for the infinities and NaNs. The exponent range for normalized numbers is [−126, 127] for single precision, [−1022, 1023] for double, or [−16382, 16383] for quad. Normalized numbers exclude subnormal values, zeros, infinities, and NaNs.

In the IEEE binary interchange formats the leading 1 bit of a normalized significand is not actually stored in the computer datum. It is called the "hidden" or "implicit" bit. Because of this, single precision format actually has a significand with 24 bits of precision, double precision format has 53, and quad has 113.

For example, it was shown above that π , rounded to 24 bits of precision, has:

- sign = 0 ; e = 1 ; s = 110010010000111110110111 (including the hidden bit)

The sum of the exponent bias (127) and the exponent (1) is 128, so this is represented in single precision format as

- 0 10000000 100100100001111110110111 (excluding the hidden bit) = 40490FDB^[10] as a [hexadecimal](#) number.

Piecewise linear approximation to exponential and logarithm ^[edit]

See also: [Fast inverse square root § Aliasing to an integer as an approximate logarithm](#)

If one graphs the floating point value of a bit pattern (x-axis is bit pattern, considered as integers, y-axis the value of the floating point number; assume positive), one obtains a piecewise linear approximation of a shifted and scaled exponential function with base 2, 2^y (hence actually $k \cdot 2^{y-l}$). Conversely, given a real number, if one takes the floating point representation and considers it as an integer, one gets a piecewise linear approximation of a shifted and scaled base 2 logarithm, $\log_2(x)$ (hence actually $c \log_2(x + d)$), as shown at right.

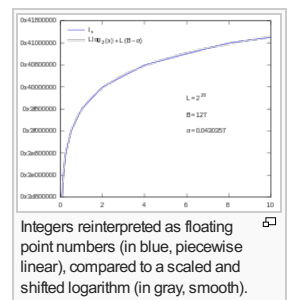
This interpretation is useful for visualizing how the values of floating point numbers vary with the representation, and allow for certain efficient approximations of floating point operations by integer operations and bit shifts. For example, reinterpreting a float as an integer, taking the negative (or rather subtracting from a fixed number, due to bias and implicit 1), then reinterpreting as a float yields the reciprocal. Explicitly, ignoring significand, taking the reciprocal is just taking the additive inverse of the (unbiased) exponent, since the exponent of the reciprocal is the negative of the original exponent. (Hence actually subtracting the exponent from twice the bias, which corresponds to unbiasing, taking negative, and then biasing.) For the significand, near 1 the reciprocal is approximately linear: $1/x \approx 1 - x$ (since the derivative is -1 ; this is the first term of the [Taylor series](#)), and thus for the significand as well, taking the negative (or rather subtracting from a fixed number to handle the implicit 1) is approximately taking the reciprocal.

More significantly, bit shifting allows one to compute the square (shift left by 1) or take the square root (shift right by 1). This leads to [approximate computations of the square root](#); combined with the previous technique for taking the inverse, this allows the [fast inverse square root](#) computation, which was important in graphics processing in the late 1980s and 1990s. This can be exploited in some other applications, such as volume ramping in digital sound processing.^{[[clarification needed](#)]}

Concretely, each time the exponent increments, the value doubles (hence grows exponentially), while each time the significand increments (for a given exponent), the value increases by $2^{(e-b)}$ (hence grows linearly, with slope equal to the actual (unbiased) value of the exponent). This holds even for the last step from a given exponent, where the significand overflows into the exponent: with the implicit 1, the number after 1.11...1 is 2.0 (regardless of the exponent), i.e., an increment of the exponent:

(0...001)0...0 through (0...001)1...1, (0...010)0...0 are equal steps (linear)

Thus as a graph it is linear pieces (as the significand grows for a given exponent) connecting the evenly spaced powers of two (when the significand is 0), with each linear piece having twice the slope of the previous: it is approximately a scaled and shifted exponential 2^x . Each piece takes the same horizontal space, but twice the vertical space of the last. Because the exponent is convex up, the value is always *greater* than or equal to the actual (shifted and scaled) exponential curve through the points with significand 0; by a slightly different shift one can more closely approximate an exponential, sometimes overestimating, sometimes underestimating. Conversely, interpreting a floating point number as an integer gives an approximate shifted and scaled logarithm, with each piece



having half the slope of the last, taking the same vertical space but twice the horizontal space. Since the logarithm is convex down, the approximation is always *less* than the corresponding logarithmic curve; again, a different choice of scale and shift (as at above right) yields a closer approximation.

Special values [edit]

Signed zero [edit]

Main article: [Signed zero](#)

In the IEEE 754 standard, zero is signed, meaning that there exist both a "positive zero" (+0) and a "negative zero" (−0). In most [run-time environments](#), positive zero is usually printed as "0" and the negative zero as "-0". The two values behave as equal in numerical comparisons, but some operations return different results for +0 and −0. For instance, $1/(-0)$ returns negative infinity, while $1/+0$ returns positive infinity (so that the identity $1/(1/\pm\infty) = \pm\infty$ is maintained). Other common [functions with a discontinuity](#) at $x=0$ which might treat +0 and −0 differently include [log](#)(x), [signum](#)(x), and the [principal square root](#) of $y + xi$ for any negative number y . As with any approximation scheme, operations involving "negative zero" can occasionally cause confusion. For example, in IEEE 754, $x = y$ does not imply $1/x = 1/y$, as $0 = -0$ but $1/0 \neq 1/-0$.^[11]

Subnormal numbers [edit]

Main article: [Subnormal numbers](#)

Subnormal values fill the [underflow](#) gap with values where the absolute distance between them are the same as for adjacent values just outside of the underflow gap. This is an improvement over the older practice to just have zero in the underflow gap, and where underflowing results were replaced by zero (flush to zero).

Modern floating-point hardware usually handles subnormal values (as well as normal values), and does not require software emulation for subnormals.

Infinities [edit]

For more details on the concept of infinite, see [Infinity](#).

The infinities of the [extended real number line](#) can be represented in IEEE floating-point datatypes, just like ordinary floating-point values like 1, 1.5, etc. They are not error values in any way, though they are often (but not always, as it depends on the rounding) used as replacement values when there is an [overflow](#). Upon a divide-by-zero exception, a positive or negative infinity is returned as an exact result. An infinity can also be introduced as a numeral (like C's "INFINITY" macro, or "∞" if the programming language allows that syntax).

IEEE 754 requires infinities to be handled in a reasonable way, such as

- $(+\infty) + (+7) = (+\infty)$
- $(+\infty) \times (-2) = (-\infty)$
- $(+\infty) \times 0 = \text{NaN}$ – there is no meaningful thing to do

NaNs [edit]

Main article: [NaN](#)

IEEE 754 specifies a special value called "Not a Number" (NaN) to be returned as the result of certain "invalid" operations, such as $0/0$, $\infty \times 0$, or $\text{sqrt}(-1)$. In general, NaNs will be propagated i.e. most operations involving a NaN will result in a NaN, although functions that would give some defined result for any given floating-point value will do so for NaNs as well, e.g. $\text{NaN}^0 = 1$. There are two kinds of NaNs: the default *quiet* NaNs and, optionally, *signaling* NaNs. A signaling NaN in any arithmetic operation (including numerical comparisons) will cause an "invalid" [exception](#) to be signaled.

The representation of NaNs specified by the standard has some unspecified bits that could be used to encode the type or source of error; but there is no standard for that encoding. In theory, signaling NaNs could be used by a [runtime system](#) to flag uninitialized variables, or extend the floating-point numbers with other special values without slowing down the computations with ordinary values, although such extensions are not common.

IEEE 754 design rationale [edit]

It is a common misconception that the more esoteric features of the IEEE 754 standard discussed here, such as extended formats, NaN, infinities, subnormals etc., are only of interest to [numerical analysts](#), or for advanced numerical applications; in fact the opposite is true: these features are designed to give safe robust defaults for numerically unsophisticated programmers, in addition to supporting sophisticated numerical libraries by experts. The key designer of IEEE 754, [William Kahan](#) notes that it is incorrect to "... [deem] features of IEEE Standard 754 for Binary Floating- Point Arithmetic that ...[are] not appreciated to be features usable by none but numerical experts. The facts are quite the opposite. In 1977 those features were designed into the Intel 8087 to serve the widest possible market... Error-analysis tells us how to design floating-point arithmetic, like IEEE Standard 754, moderately tolerant of well-meaning ignorance among programmers".^[12]

- The special values such as infinity and NaN ensure that the floating-point arithmetic is algebraically completed, such that every floating-point operation produces a well-defined result and will not -by default- throw a machine interrupt or trap. Moreover, the choices of special values returned in exceptional cases were designed to give the correct answer in many cases, e.g. continued fractions such as $R(z) := 7 - 3/(z - 2 - 1/(z - 7 + 10/(z - 2 - 2/(z - 3))))$ will give the correct answer in all inputs under IEEE-754 arithmetic as the potential divide by zero in e.g. $R(3)=4.6$ is correctly handled as +infinity and so can be safely ignored.^[13] As noted by Kahan, the unhandled trap consecutive to a floating-point to 16-bit integer conversion overflow that caused the [loss of an Ariane 5](#) rocket would not have happened under the default IEEE 754 floating-point policy.^[12]
- Subnormal numbers ensure that for *finite* floating-point numbers x and y , $x - y = 0$ if and only if $x = y$, as expected, but which did not hold under earlier floating-point representations.^[14]
- On the design rationale of the x87 [80-bit format](#), Kahan notes: "This Extended format is designed to be used, with negligible loss of speed, for all but the simplest arithmetic with float and double operands. For example, it should be used for scratch variables in loops that implement recurrences like polynomial evaluation, scalar products, partial and continued fractions. It often averts premature Over/Underflow or severe local cancellation that can spoil simple algorithms.^[15] Computing intermediate results in an extended format with high precision and extended exponent has precedents in the historical practice of scientific [calculation](#) and in the design of scientific calculators e.g. Hewlett- Packard's financial calculators performed arithmetic and financial functions to three more significant decimals than they stored or displayed.^[15] The implementation of extended precision enabled standard elementary function libraries to be readily developed that normally gave double precision results within one [unit in the last place](#) (ULP) at high speed.
- Correct rounding of values to the nearest representable value avoids systematic biases in calculations and slows the growth of errors. Rounding ties to even removes the statistical bias that can occur in adding similar figures.
- Directed rounding was intended as an aid with checking error bounds, for instance in interval arithmetic. It is also used in the implementation of some functions.
- The mathematical basis of the operations enabled high precision multiword arithmetic subroutines to be built relatively easily.
- The single and double precision formats were designed to be easy to sort without using floating-point hardware.



William Kahan. A primary architect of the Intel [80x87](#) floating-point coprocessor and [IEEE 754](#) floating-point standard.

Representable numbers, conversion and rounding [edit]

By their nature, all numbers expressed in floating-point format are [rational numbers](#) with a terminating expansion in the relevant base (for example, a terminating decimal expansion in base-10, or a terminating binary expansion in base-2). Irrational numbers, such as [π](#) or $\sqrt{2}$, or non-terminating rational numbers, must be approximated. The number of digits (or bits) of precision also limits the set of rational numbers that can be represented exactly. For example, the number

123456789 cannot be exactly represented if only eight decimal digits of precision are available.

When a number is represented in some format (such as a character string) which is not a native floating-point representation supported in a computer implementation, then it will require a conversion before it can be used in that implementation. If the number can be represented exactly in the floating-point format then the conversion is exact. If there is not an exact representation then the conversion requires a choice of which floating-point number to use to represent the original value. The representation chosen will have a different value to the original, and the value thus adjusted is called the *rounded value*.

Whether or not a rational number has a terminating expansion depends on the base. For example, in base-10 the number 1/2 has a terminating expansion (0.5) while the number 1/3 does not (0.333...). In base-2 only rationals with denominators that are powers of 2 (such as 1/2 or 3/16) are terminating. Any rational with a denominator that has a prime factor other than 2 will have an infinite binary expansion. This means that numbers which appear to be short and exact when written in decimal format may need to be approximated when converted to binary floating-point. For example, the decimal number 0.1 is not representable in binary floating-point of any finite precision; the exact binary representation would have a "1100" sequence continuing endlessly:

$$e = -4; s = 1100110011001100110011001100110011\dots,$$

where, as previously, *s* is the significand and *e* is the exponent.

When rounded to 24 bits this becomes

$$e = -4; s = 110011001100110011001101,$$

which is actually 0.100000001490116119384765625 in decimal.

As a further example, the real number π , represented in binary as an infinite sequence of bits is

$$1.100100100001111110110101000100010000101101000110000100011010011\dots$$

but is

$$1.10010010000111111011011$$

when approximated by [rounding](#) to a precision of 24 bits.

In binary single-precision floating-point, this is represented as $s = 1.10010010000111111011011$ with $e = 1$. This has a decimal value of

$$3.1415927410125732421875,$$

whereas a more accurate approximation of the true value of π is

$$3.14159265358979323846264338327950\dots$$

The result of rounding differs from the true value by about 0.03 parts per million, and matches the decimal representation of π in the first 7 digits. The difference is the [discretization error](#) and is limited by the [machine epsilon](#).

The arithmetical difference between two consecutive representable floating-point numbers which have the same exponent is called a [unit in the last place](#) (ULP). For example, if there is no representable number lying between the representable numbers 1.45a70c22_{hex} and 1.45a70c24_{hex}, the ULP is 2×16^{-8} , or 2^{-31} . For numbers with a base-2 exponent part of 0, i.e. numbers with an absolute value higher than or equal to 1 but lower than 2, an ULP is exactly 2^{-23} or about 10^{-7} in single precision, and exactly 2^{-53} or about 10^{-16} in double precision. The mandated behavior of IEEE-compliant hardware is that the result be within one-half of a ULP.

Rounding modes [\[edit\]](#)

Rounding is used when the exact result of a floating-point operation (or a conversion to floating-point format) would need more digits than there are digits in the significand. IEEE 754 requires *correct rounding*: that is, the rounded result is as if infinitely precise arithmetic was used to compute the value and then rounded (although in implementation only three extra bits are needed to ensure this). There are several different [rounding schemes](#) (or *rounding modes*). Historically, [truncation](#) was the typical approach. Since the introduction of IEEE 754, the default method (*round to nearest, ties to even*, sometimes called Banker's Rounding) is more commonly used. This method rounds the ideal (infinitely precise) result of an arithmetic operation to the nearest representable value, and gives that representation as the result.^[16] In the case of a tie, the value that would make the significand end in an even digit is chosen. The IEEE 754 standard requires the same rounding to be applied to all fundamental algebraic operations, including square root and conversions, when there is a numeric (non-NaN) result. It means that the results of IEEE 754 operations are completely determined in all bits of the result, except for the representation of NaNs. ("Library" functions such as cosine and log are not mandated.)

Alternative rounding options are also available. IEEE 754 specifies the following rounding modes:

- round to nearest, where ties round to the nearest even digit in the required position (the default and by far the most common mode)
- round to nearest, where ties round away from zero (optional for binary floating-point and commonly used in decimal)
- round up (toward $+\infty$; negative results thus round toward zero)
- round down (toward $-\infty$; negative results thus round away from zero)
- round toward zero (truncation; it is similar to the common behavior of float-to-integer conversions, which convert -3.9 to -3 and 3.9 to 3)

Alternative modes are useful when the amount of error being introduced must be bounded. Applications that require a bounded error are multi-precision floating-point, and [interval arithmetic](#). The alternative rounding modes are also useful in diagnosing numerical instability: if the results of a subroutine vary substantially between rounding to $+$ and $-$ infinity then it is likely numerically unstable and affected by round-off error.^[17]

Floating-point arithmetic operations [\[edit\]](#)

For ease of presentation and understanding, decimal [radix](#) with 7 digit precision will be used in the examples, as in the IEEE 754 *decimal32* format. The fundamental principles are the same in any [radix](#) or precision, except that normalization is optional (it does not affect the numerical value of the result). Here, *s* denotes the significand and *e* denotes the exponent.

Addition and subtraction [\[edit\]](#)

A simple method to add floating-point numbers is to first represent them with the same exponent. In the example below, the second number is shifted right by three digits, and we then proceed with the usual addition method:

$$\begin{aligned} 123456.7 &= 1.234567 \times 10^5 \\ 101.7654 &= 1.017654 \times 10^2 = 0.001017654 \times 10^5 \end{aligned}$$

$$\begin{aligned} \text{Hence:} \\ 123456.7 + 101.7654 &= (1.234567 \times 10^5) + (1.017654 \times 10^2) \\ &= (1.234567 \times 10^5) + (0.001017654 \times 10^5) \\ &= (1.234567 + 0.001017654) \times 10^5 \\ &= 1.235584654 \times 10^5 \end{aligned}$$

In detail:

```
e=5;  s=1.234567    (123456.7)
+ e=2; s=1.017654    (101.7654)
```

```
e=5;  s=1.234567
+ e=5; s=0.001017654  (after shifting)
-----
e=5;  s=1.235584654  (true sum: 123558.4654)
```

This is the true result, the exact sum of the operands. It will be rounded to seven digits and then normalized if necessary. The final result is

```
e=5;  s=1.235585    (final sum: 123558.5)
```

Note that the low three digits of the second operand (654) are essentially lost. This is [round-off error](#). In extreme cases, the sum of two non-zero numbers may be equal to one of them:

```
e=5;  s=1.234567
+ e=-3; s=9.876543
```

```
e=5;  s=1.234567
+ e=5; s=0.0000009876543 (after shifting)
-----
e=5;  s=1.23456709876543 (true sum)
e=5;  s=1.234567          (after rounding/normalization)
```

Note that in the above conceptual examples it would appear that a large number of extra digits would need to be provided by the adder to ensure correct rounding: in fact for binary addition or subtraction using careful implementation techniques only two extra *guard* bits and one extra *sticky* bit need to be carried beyond the precision of the operands.^[11]

Another problem of loss of significance occurs when two close numbers are subtracted. In the following example $e = 5$; $s = 1.234571$ and $e = 5$; $s = 1.234567$ are representations of the rationals 123457.1467 and 123456.659.

```
e=5;  s=1.234571
- e=5; s=1.234567
-----
e=5;  s=0.000004
e=-1; s=4.000000 (after rounding/normalization)
```

The best representation of this difference is $e = -1$; $s = 4.877000$, which differs more than 20% from $e = -1$; $s = 4.000000$. In extreme cases, all significant digits of precision can be lost (although gradual underflow ensures that the result will not be zero unless the two operands were equal). This [cancellation](#) illustrates the danger in assuming that all of the digits of a computed result are meaningful. Dealing with the consequences of these errors is a topic in [numerical analysis](#); see also [Accuracy problems](#).

Multiplication and division [\[edit\]](#)

To multiply, the significands are multiplied while the exponents are added, and the result is rounded and normalized.

```
e=3;  s=4.734612
x e=5; s=5.417242
-----
e=8;  s=25.648538980104 (true product)
e=8;  s=25.64854         (after rounding)
e=9;  s=2.564854         (after normalization)
```

Similarly, division is accomplished by subtracting the divisor's exponent from the dividend's exponent, and dividing the dividend's significand by the divisor's significand.

There are no cancellation or absorption problems with multiplication or division, though small errors may accumulate as operations are performed in succession.^[11] In practice, the way these operations are carried out in digital logic can be quite complex (see [Booth's multiplication algorithm](#) and [Division algorithm](#)).^[18] For a fast, simple method, see the [Horner method](#).

Dealing with exceptional cases [\[edit\]](#)

Floating-point computation in a computer can run into three kinds of problems:

- An operation can be mathematically undefined, such as ∞/∞ , or division by zero.
- An operation can be legal in principle, but not supported by the specific format, for example, calculating the square root of -1 or the inverse sine of 2 (both of which result in [complex numbers](#)).
- An operation can be legal in principle, but the result can be impossible to represent in the specified format, because the exponent is too large or too small to encode in the exponent field. Such an event is called an [overflow](#) (exponent too large), [underflow](#) (exponent too small) or [denormalization](#) (precision loss).

Prior to the IEEE standard, such conditions usually caused the program to terminate, or triggered some kind of [trap](#) that the programmer might be able to catch. How this worked was system-dependent, meaning that floating-point programs were not [portable](#). (Note that the term "exception" as used in IEEE-754 is a general term meaning an exceptional condition, which is not necessarily an error, and is a different usage to that typically defined in programming languages such as a C++ or Java, in which an ["exception"](#) is an alternative flow of control, closer to what is termed a "trap" in IEEE-754 terminology).

Here, the required default method of handling exceptions according to IEEE 754 is discussed (the IEEE-754 optional trapping and other "alternate exception handling" modes are not discussed). Arithmetic exceptions are (by default) required to be recorded in "sticky" status flag bits. That they are "sticky" means that they are not reset by the next (arithmetic) operation, but stay set until explicitly reset. The use of "sticky" flags thus allows for testing of exceptional conditions to be delayed until after a full floating-point expression or subroutine: without them exceptional conditions that could not be otherwise ignored would require explicit testing immediately after every floating-point operation. By default, an operation always returns a result according to specification without interrupting computation. For instance, $1/0$ returns $+\infty$, while also setting the divide-by-zero flag bit (this default of ∞ is designed so as to often return a finite result when used in subsequent operations and so be safely ignored).

The original IEEE 754 standard, however, failed to recommend operations to handle such sets of arithmetic exception flag bits. So while these were implemented

in hardware, initially programming language implementations typically did not provide a means to access them (apart from assembler). Over time some programming language standards (e.g., [C99/C11](#) and Fortran) have been updated to specify methods to access and change status flag bits. The 2008 version of the IEEE 754 standard now specifies a few operations for accessing and handling the arithmetic flag bits. The programming model is based on a single thread of execution and use of them by multiple threads has to be handled by a [means](#) outside of the standard (e.g. [C11](#) specifies that the flags have [thread-local storage](#)).

IEEE 754 specifies five arithmetic exceptions that are to be recorded in the status flags ("sticky bits"):

- **inexact**, set if the rounded (and returned) value is different from the mathematically exact result of the operation.
- **underflow**, set if the rounded value is tiny (as specified in IEEE 754) *and* inexact (or maybe limited to if it has denormalization loss, as per the 1984 version of IEEE 754), returning a subnormal value including the zeros.
- **overflow**, set if the absolute value of the rounded value is too large to be represented. An infinity or maximal finite value is returned, depending on which rounding is used.
- **divide-by-zero**, set if the result is infinite given finite operands, returning an infinity, either $+\infty$ or $-\infty$.
- **invalid**, set if a real-valued result cannot be returned e.g. $\sqrt{-1}$ or $0/0$, returning a quiet NaN.

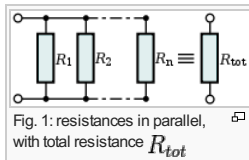


Fig. 1: resistances in parallel, with total resistance R_{tot}

The default return value for each of the exceptions is designed to give the correct result in the majority of cases such that the exceptions can be ignored in the majority of codes. *inexact* returns a correctly rounded result, and *underflow* returns a denormalized small value and so can almost always be ignored.^[19] *divide-by-zero* returns infinity exactly, which will typically then divide a finite number and so give zero, or else will give an *invalid* exception subsequently if not, and so can also typically be ignored. For example, the effective resistance of n resistors in parallel (see fig. 1) is given by $R_{tot} = 1/(1/R_1 + 1/R_2 + \dots + 1/R_n)$. If a short-circuit develops with R_1 set to 0, $1/R_1$ will return $+\infty$ which will give a final R_{tot} of 0, as expected^[20] (see the continued fraction example of [IEEE 754 design rationale](#) for

another example).

Overflow and *invalid* exceptions can typically not be ignored, but do not necessarily represent errors: for example, a [root-finding](#) routine, as part of its normal operation, may evaluate a passed-in function at values outside of its domain, returning NaN and an *invalid* exception flag to be ignored until finding a useful start point.^[21]

Accuracy problems [\[edit\]](#)

The fact that floating-point numbers cannot precisely represent all real numbers, and that floating-point operations cannot precisely represent true arithmetic operations, leads to many surprising situations. This is related to the finite [precision](#) with which computers generally represent numbers.

For example, the non-representability of 0.1 and 0.01 (in binary) means that the result of attempting to square 0.1 is neither 0.01 nor the representable number closest to it. In 24-bit (single precision) representation, 0.1 (decimal) was given previously as $e = -4$; $s = 110011001100110011001101$, which is

0.100000001490116119384765625 exactly.

Squaring this number gives

0.010000000298023226097399174250313080847263336181640625 exactly.

Squaring it with single-precision floating-point hardware (with rounding) gives

0.010000000707805156707763671875 exactly.

But the representable number closest to 0.01 is

0.009999999776482582092285156250 exactly.

Also, the non-representability of π (and $\pi/2$) means that an attempted computation of $\tan(\pi/2)$ will not yield a result of infinity, nor will it even overflow. It is simply not possible for standard floating-point hardware to attempt to compute $\tan(\pi/2)$, because $\pi/2$ cannot be represented exactly. This computation in C:

```
/* Enough digits to be sure we get the correct approximation. */
double pi = 3.1415926535897932384626433832795;
double z = tan(pi/2.0);
```

will give a result of 16331239353195370.0. In single precision (using the `tanf` function), the result will be -22877332.0 .

By the same token, an attempted computation of $\sin(\pi)$ will not yield zero. The result will be (approximately) 0.1225×10^{-15} in double precision, or -0.8742×10^{-7} in single precision.^[22]

While floating-point addition and multiplication are both [commutative](#) ($a + b = b + a$ and $a \times b = b \times a$), they are not necessarily [associative](#). That is, $(a + b) + c$ is not necessarily equal to $a + (b + c)$. Using 7-digit significant decimal arithmetic:

$a = 1234.567$, $b = 45.67834$, $c = 0.0004$

```
(a + b) + c:
1234.567   (a)
+  45.67834 (b)
-----
1280.24534 rounds to 1280.245
```

```
1280.245   (a + b)
+  0.0004   (c)
-----
1280.2454 rounds to 1280.245 <--- (a + b) + c
```

```
a + (b + c):
45.67834 (b)
+  0.0004 (c)
-----
45.67874
```

```
1234.567   (a)
+  45.67874 (b + c)
-----
1280.24574 rounds to 1280.246 <--- a + (b + c)
```


They are also not necessarily [distributive](#). That is, $(a + b) \times c$ may not be the same as $a \times c + b \times c$:

```
1234.567 × 3.333333 = 4115.223
1.234567 × 3.333333 = 4.115223
4115.223 + 4.115223 = 4119.338
but
1234.567 + 1.234567 = 1235.802
1235.802 × 3.333333 = 4119.340
```

In addition to loss of significance, inability to represent numbers such as π and 0.1 exactly, and other slight inaccuracies, the following phenomena may occur:

- **Cancellation**: subtraction of nearly equal operands may cause extreme loss of accuracy.^[23] When we subtract two almost equal numbers we set the most significant digits to zero, leaving ourselves with just the insignificant, and most erroneous, digits. For example, when determining a [derivative](#) of a function the following formula is used:

$$Q(h) = \frac{f(a+h) - f(a)}{h}.$$

Intuitively one would want an h very close to zero, however when using floating-point operations, the smallest number won't give the best approximation of a derivative. As h grows smaller the difference between $f(a+h)$ and $f(a)$ grows smaller, cancelling out the most significant and least erroneous digits and making the most erroneous digits more important. As a result the smallest number of h possible will give a more erroneous approximation of a derivative than a somewhat larger number. This is perhaps the most common and serious accuracy problem.

- Conversions to integer are not intuitive: converting (63.0/9.0) to integer yields 7, but converting (0.63/0.09) may yield 6. This is because conversions generally truncate rather than round. [Floor and ceiling functions](#) may produce answers which are off by one from the intuitively expected value.
- Limited exponent range: results might overflow yielding infinity, or underflow yielding a [subnormal number](#) or zero. In these cases precision will be lost.
- Testing for [safe division](#) is problematic: Checking that the divisor is not zero does not guarantee that a division will not overflow.
- Testing for equality is problematic. Two computational sequences that are mathematically equal may well produce different floating-point values.

Incidents [\[edit\]](#)

- On 25 February 1991, a [loss of significance](#) in a [MIM-104 Patriot](#) missile battery prevented it intercepting an incoming [Scud](#) missile in [Dhahran](#), [Saudi Arabia](#), contributing to the death of 28 soldiers from the U.S. Army's [14th Quartermaster Detachment](#).^[24] See also: [Failure at Dhahran](#)

Machine precision and backward error analysis [\[edit\]](#)

Machine precision is a quantity that characterizes the accuracy of a floating-point system, and is used in [backward error analysis](#) of floating-point algorithms. It is also known as unit roundoff or [machine epsilon](#). Usually denoted E_{mach} , its value depends on the particular rounding being used.

With rounding to zero,

$$E_{\text{mach}} = B^{1-P},$$

whereas rounding to nearest,

$$E_{\text{mach}} = \frac{1}{2}B^{1-P}.$$

This is important since it bounds the [relative error](#) in representing any non-zero real number x within the normalized range of a floating-point system:

$$\left| \frac{fl(x) - x}{x} \right| \leq E_{\text{mach}}.$$

Backward error analysis, the theory of which was developed and popularized by [James H. Wilkinson](#), can be used to establish that an algorithm implementing a numerical function is numerically stable.^[25] The basic approach is to show that although the calculated result, due to roundoff errors, will not be exactly correct, it is the exact solution to a nearby problem with slightly perturbed input data. If the perturbation required is small, on the order of the uncertainty in the input data, then the results are in some sense as accurate as the data "deserves". The algorithm is then defined as [backward stable](#). Stability is a measure of the sensitivity to rounding errors of a given numerical procedure; by contrast, the [condition number](#) of a function for a given problem indicates the inherent sensitivity of the function to small perturbations in its input and is independent of the implementation used to solve the problem.^[26]

As a trivial example, consider a simple expression giving the inner product of (length two) vectors x and y , then

$$\begin{aligned} fl(x \cdot y) &= fl(fl(x_1 * y_1) + fl(x_2 * y_2)) \text{ where } fl() \text{ indicates correctly rounded floating-point arithmetic} \\ &= fl((x_1 * y_1)(1 + \delta_1) + (x_2 * y_2)(1 + \delta_2)) \text{ where } \delta_n \leq E_{\text{mach}}, \text{ from above} \\ &= ((x_1 * y_1)(1 + \delta_1) + (x_2 * y_2)(1 + \delta_2))(1 + \delta_3) \\ &= (x_1 * y_1)(1 + \delta_1)(1 + \delta_3) + (x_2 * y_2)(1 + \delta_2)(1 + \delta_3) \end{aligned}$$

and so

$$\begin{aligned} fl(x \cdot y) &= \hat{x} \cdot \hat{y} \text{ where} \\ \hat{x}_1 &= x_1(1 + \delta_1); \hat{x}_2 = x_2(1 + \delta_2); \\ \hat{y}_1 &= y_1(1 + \delta_3); \hat{y}_2 = y_2(1 + \delta_3) \\ \text{where } \delta_n &\leq E_{\text{mach}}, \text{ by definition} \end{aligned}$$

which is the sum of two slightly perturbed (on the order of E_{mach}) input data, and so is backward stable. For more realistic examples in [numerical linear algebra](#) see Higham 2002 and other references below).

Minimizing the effect of accuracy problems [\[edit\]](#)

Although, as noted previously, individual arithmetic operations of IEEE 754 are guaranteed accurate to within half a ULP, more complicated formulae can suffer from larger errors due to round-off. The loss of accuracy can be substantial if a problem or its data are [ill-conditioned](#), meaning that the correct result is hypersensitive to tiny perturbations in its data. However, even functions that are well-conditioned can suffer from large loss of accuracy if an algorithm [numerically unstable](#) for that data is used: apparently equivalent formulations of expressions in a programming language can differ markedly in their numerical stability. One approach to remove the risk of such loss of accuracy is the design and analysis of numerically stable algorithms, which is an aim of the branch of mathematics known as [numerical analysis](#). Another approach that can protect against the risk of numerical instabilities is the computation of intermediate (scratch) values in an algorithm at a higher precision than the final result requires,^[27] which can remove, or reduce by orders of magnitude,^[28] such risk: [IEEE 754 quadruple precision](#) and [extended precision](#) are designed for this purpose when computing at double precision.^{[29][30]}

For example, the following algorithm is a direct implementation to compute the function $A(x) = (x-1)/(\exp(x-1) - 1)$ which is well-conditioned at 1.0,^[31] however it can be shown to be numerically unstable and lose up to half the significant digits carried by the arithmetic when computed near 1.0.^[12]

```
double A(double X)
```

```

{
    double Y, Z;  // [1]
    Y = X - 1.0;
    Z = exp(Y);
    if (Z != 1.0) Z = Y / (Z - 1.0); // [2]
    return (Z);
}

```

If, however, intermediate computations are all performed in extended precision (e.g. by setting line [1] to [C99 long double](#)), then up to full precision in the final double result can be maintained.^[32] Alternatively, a numerical analysis of the algorithm reveals that if the following non-obvious change to line [2] is made:

```

if (Z != 1.0) Z = log(Z) / (Z - 1.0);

```

then the algorithm becomes numerically stable and can compute to full double precision.

To maintain the properties of such carefully constructed numerically stable programs, careful handling by the [compiler](#) is required. Certain "optimizations" that compilers might make (for example, reordering operations) can work against the goals of well-behaved software. There is some controversy about the failings of compilers and language designs in this area: [C99](#) is an example of a language where such optimizations are carefully specified so as to maintain numerical precision. See the external references at the bottom of this article.

A detailed treatment of the techniques for writing high-quality floating-point software is beyond the scope of this article, and the reader is referred to,^{[33][34]} and the other references at the bottom of this article. Kahan suggests several rules of thumb that can substantially decrease by orders of magnitude^[34] the risk of numerical anomalies, in addition to, or in lieu of, a more careful numerical analysis. These include: as noted above, computing all expressions and intermediate results in the highest precision supported in hardware (a common rule of thumb is to carry twice the precision of the desired result i.e. compute in double precision for a final single precision result, or in double extended or quad precision for up to double precision results^[35]); and rounding input data and results to only the precision required and supported by the input data (carrying excess precision in the final result beyond that required and supported by the input data can be misleading, increases storage cost and decreases speed, and the excess bits can affect convergence of numerical procedures.^[36] notably, the first form of the iterative example given below converges correctly when using this rule of thumb). Brief descriptions of several additional issues and techniques follow.

As decimal fractions can often not be exactly represented in binary floating-point, such arithmetic is at its best when it is simply being used to measure real-world quantities over a wide range of scales (such as the orbital period of a moon around Saturn or the mass of a [proton](#)), and at its worst when it is expected to model the interactions of quantities expressed as decimal strings that are expected to be exact.^{[37][38]} An example of the latter case is financial calculations. For this reason, financial software tends not to use a binary floating-point number representation.^[39] The "decimal" data type of the [C#](#) and [Python](#) programming languages, and the decimal formats of the [IEEE 754-2008](#) standard, are designed to avoid the problems of binary floating-point representations when applied to human-entered exact decimal values, and make the arithmetic always behave as expected when numbers are printed in decimal.

Expectations from mathematics may not be realized in the field of floating-point computation. For example, it is known that $(x + y)(x - y) = x^2 - y^2$, and that $\sin^2 \theta + \cos^2 \theta = 1$, however these facts cannot be relied on when the quantities involved are the result of floating-point computation.

The use of the equality test (`if (x==y) ...`) requires care when dealing with floating-point numbers. Even simple expressions like `0.6/0.2-3==0` will, on most computers, fail to be true^[40] (in IEEE 754 double precision, for example, `0.6/0.2-3` is approximately equal to -4.44089209850063e-16). Consequently, such tests are sometimes replaced with "fuzzy" comparisons (`if (abs(x-y) < epsilon) ...`), where [epsilon](#) is sufficiently small and tailored to the application, such as 1.0E-13). The wisdom of doing this varies greatly, and can require numerical analysis to bound epsilon.^[41] Values derived from the primary data representation and their comparisons should be performed in a wider, extended, precision to minimize the risk of such inconsistencies due to round-off errors.^[34] It is often better to organize the code in such a way that such tests are unnecessary. For example, in [computational geometry](#), exact tests of whether a point lies off or on a line or plane defined by other points can be performed using adaptive precision or exact arithmetic methods.^[42]

Small errors in floating-point arithmetic can grow when mathematical algorithms perform operations an enormous number of times. A few examples are [matrix inversion](#), [eigenvector](#) computation, and differential equation solving. These algorithms must be very carefully designed, using numerical approaches such as [iterative refinement](#), if they are to work well.^[43]

Summation of a vector of floating-point values is a basic algorithm in [scientific computing](#), and so an awareness of when loss of significance can occur is essential. For example, if one is adding a very large number of numbers, the individual addends are very small compared with the sum. This can lead to loss of significance. A typical addition would then be something like

```

3253.671
+ 3.141276
-----
3256.812

```

The low 3 digits of the addends are effectively lost. Suppose, for example, that one needs to add many numbers, all approximately equal to 3. After 1000 of them have been added, the running sum is about 3000; the lost digits are not regained. The [Kahan summation algorithm](#) may be used to reduce the errors.^[44]

Round-off error can affect the convergence and accuracy of iterative numerical procedures. As an example, [Archimedes](#) approximated π by calculating the perimeters of polygons inscribing and circumscribing a circle, starting with hexagons, and successively doubling the number of sides. As noted above, computations may be rearranged in a way that is mathematically equivalent but less prone to error ([numerical analysis](#)). Two forms of the recurrence formula for the circumscribed polygon are:

$$t_0 = \frac{1}{\sqrt{3}}$$

$$\text{first form : } t_{i+1} = \frac{\sqrt{t_i^2 + 1} - 1}{t_i} \quad \text{second form : } t_{i+1} = \frac{t_i}{\sqrt{t_i^2 + 1} + 1}$$

$$\pi \sim 6 \times 2^i \times t_i, \quad \text{converging as } i \rightarrow \infty$$

Here is a computation using IEEE "double" (a significand with 53 bits of precision) arithmetic:

i	$6 \times 2^i \times t_i$, first form	$6 \times 2^i \times t_i$, second form
---	--	---

0	3.4641016151377543863	3.4641016151377543863
1	3.2153903091734710173	3.2153903091734723496
2	3.1596599420974940120	3.1596599420975006733
3	3.1460862151314012979	3.1460862151314352708
4	3.1427145996453136334	3.1427145996453689225
5	3.1418730499801259536	3.1418730499798241950
6	3.1416627470548084133	3.1416627470568494473

7	3.1416101765997805905	3.1416101766046906629
8	3.1415970343230776862	3.1415970343215275928
9	3.1415937488171150615	3.1415937487713536668
10	3.1415929278733740748	3.1415929273850979885
11	3.1415927256228504127	3.1415927220386148377
12	3.1415926717412858693	3.1415926707019992125
13	3.1415926189011456060	3.1415926578678454728
14	3.1415926717412858693	3.1415926546593073709
15	3.1415919358822321783	3.1415926538571730119
16	3.1415926717412858693	3.1415926536566394222
17	3.1415810075796233302	3.1415926536065061913
18	3.1415926717412858693	3.1415926535939728836
19	3.1414061547378810956	3.1415926535908393901
20	3.1405434924008406305	3.1415926535900560168
21	3.1400068646912273617	3.1415926535898608396
22	3.1349453756585929919	3.1415926535898122118
23	3.1400068646912273617	3.1415926535897995552
24	3.2245152435345525443	3.1415926535897968907
25		3.1415926535897962246
26		3.1415926535897962246
27		3.1415926535897962246
28		3.1415926535897962246

The true value is 3.14159265358979323846264338327...

While the two forms of the recurrence formula are clearly mathematically equivalent,^[49] the first subtracts 1 from a number extremely close to 1, leading to an increasingly problematic loss of **significant digits**. As the recurrence is applied repeatedly, the accuracy improves at first, but then it deteriorates. It never gets better than about 8 digits, even though 53-bit arithmetic should be capable of about 16 digits of precision. When the second form of the recurrence is used, the value converges to 15 digits of precision.

See also [edit]

- C99 for code examples demonstrating access and use of IEEE 754 features.
- Computable number
- Coprocessor
- Decimal floating point
- Double precision
- Experimental mathematics—utilizes high precision floating-point computations
- Fixed-point arithmetic
- FLOPS
- Gal's accurate tables
- GNU Multi-Precision Library
- Half precision
- IEEE 754 — Standard for Binary Floating-Point Arithmetic
- IBM Floating Point Architecture
- Kahan summation algorithm
- Microsoft Binary Format (MBF)
- Minifloat
- Q (number format) for constant resolution
- Quad precision
- Significant digits




Notes and references [edit]






- ↑ W.Smith, Steven (1997). "Chapter 28, Fixed versus Floating Point". *The Scientist and Engineer's Guide to Digital Signal Processing*. California Technical Pub. p. 514. ISBN 0966017633. Retrieved December 31, 2012.
- ↑ B. Randell (1982). *From analytical engine to electronic digital computer: the contributions of Ludgate, Torres, and Bush*. *IEEE Annals of the History of Computing*, 04(4). pp. 327–341.
- ↑ "Konrad Zuse's Legacy: The Architecture of the Z1 and Z3" (PDF). *IEEE Annals of the History of Computing* **19** (2): 5–15. 1997. doi:10.1109/85.586067.
- ↑ William Kahan (15 July 1997). "The Baleful Effect of Computer Languages and Benchmarks upon Applied Mathematics, Physics and Chemistry" (PDF).
- ↑ "The Baleful Effect of Computer Languages and Benchmarks upon Applied Mathematics, Physics and Chemistry. John von Neumann Lecture" (PDF). 16 July 1997. p. 3.
- ↑ Randell, Brian, ed. (1982) [1973]. *The Origins of Digital Computers: Selected Papers* (3rd ed.). Berlin; New York: Springer-Verlag. p. 244. ISBN 3-540-11319-3.
- ↑ Severance, Charles (20 February 1998). "An Interview with the Old Man of Floating-Point" .
- ↑ "W. Kahan. "On the Cost of Floating-Point Computation Without Extra-Precise Arithmetic" " (PDF). 20 November 2004. Retrieved 19 February 2012.
- ↑ "openEXR" . openEXR. Retrieved 25 April 2012.
- ↑ http://babbar.cs.qc.edu/IEEE-754/32bit.html
- ↑ ^ ^ ^ Goldberg, David (1991). "What Every Computer Scientist Should Know About Floating-Point Arithmetic" (PDF). *ACM Computing Surveys* **23**: 5–48. doi:10.1145/103162.103163. Retrieved 13 August 2015.
- ↑ ^ ^ ^ William Kahan (1 March 1998). "How JAVA's Floating-Point Hurts Everyone Everywhere" (PDF).
- ↑ William Kahan (12 February 1981). "Why do we need a floating-point arithmetic standard?" (PDF).
- ↑ Charles Severance (20 February 1998). "An Interview with the Old Man of Floating-Point" .
- ↑ ^ ^ ^ William Kahan (1 October 1997). "The Baleful Effect of Computer Benchmarks upon Applied Mathematics, Physics and Chemistry" (PDF).
- ↑ Computer hardware doesn't necessarily compute the exact value; it simply has to produce the equivalent rounded result as though it had computed the infinitely precise result.
- ↑ William Kahan (11 January 2006). "How Futile are Mindless Assessments of Roundoff in Floating-Point Computation ?" (PDF).
- ↑ The enormous complexity of modern division algorithms once led to a famous error. An early version of the Intel Pentium chip was shipped with a division instruction that, on rare occasions, gave slightly incorrect results. Many computers had been shipped before the error was discovered. Until the defective computers were replaced, patched versions of compilers were developed that could avoid the failing cases. See *Pentium FDIV bug*.
- ↑ William Kahan (1 October 1997). "Lecture Notes on the Status of IEEE Standard 754 for Binary Floating-Point Arithmetic" (PDF).
- ↑ "Intel® 64 and IA-32 Architectures Software Developers' Manuals. Volume 1. section D.3.2.1" .
- ↑ William Kahan (1 October 1997). "Lecture Notes on the Status of IEEE Standard 754 for Binary Floating-Point Arithmetic (page 9)" (PDF).
- ↑ But an attempted computation of cos(π) yields −1 exactly. Since the derivative is nearly zero near π, the effect of the inaccuracy in the argument is far smaller than the spacing of the floating-point numbers around −1, and the rounded result is exact.
- ↑ Richard Harris (October 2010). "You're Going To Have To Think!" . *Overload* (99): 5–10. ISSN 1354-3172. Retrieved 24 September 2011. "Far more worrying is cancellation error which can yield catastrophic loss of precision."
- ↑ "Patriot missile defense. Software problem led to system failure at Dhahran, Saudi Arabia: GAO report IMTEC 02-26" . U.S. Government Accounting Office

24. ^a Faulty missile test, software problem led to system failure at Dhahran, Saudi Arabia, GAO report IMTC-92-20 (S. US Government Accounting Office).
25. ^a James H. Wilkinson; Anthony Ralston(ed); Edwin D. Reilly(ed); David Hemmendinger(ed) (8 September 2003). *"Error Analysis" in Encyclopedia of Computer Science*. pp. 669-674 ^a. Wiley. ISBN 978-0-470-86412-8. Retrieved 14 May 2013.
26. ^a Bo Einarsson (2005). *Accuracy and reliability in scientific computing* ^a. SIAM. pp. 50–. ISBN 978-0-89871-815-7. Retrieved 14 May 2013.
27. ^a Suelly Oliveira; David E. Stewart (7 September 2006). *Writing Scientific Software: A Guide to Good Style* ^a. Cambridge University Press. pp. 10–. ISBN 978-1-139-45862-7.
28. ^a Kahan estimates that the incidence of excessively inaccurate results near singularities is reduced by a factor of approx. 1/2000 using the 11 extra bits of precision of double extended- William Kahan (15 July 2005). *"Floating-Point Arithmetic Besieged by "Business Decisions". Keynote Address, IEEE-Sponsored ARITH 17 Symposium on Computer Arithmetic*, p. 18" ^a (PDF). Retrieved 23 May 2013.
29. ^a William Kahan (3 August 2011). *"Desperately Needed Remedies for the Undebuggability of Large Floating-Point Computations in Science and Engineering. IFIP/SIAM/NIST Working Conference on Uncertainty Quantification in Scientific Computing. Boulder CO."* ^a (PDF). p. 33.
30. ^a Kahan notes: "Except in extremely uncommon situations, extra-precise arithmetic generally attenuates risks due to roundoff at far less cost than the price of a competent error-analyst."
31. ^a note: the Taylor expansion of this function demonstrates that it is well-conditioned near 1: $A(x) = 1 - (x-1)/2 + (x-1)^2/12 - (x-1)^4/720 + (x-1)^6/30240 - (x-1)^8/1209600 + \dots$ for $|x-1| < \pi$
32. ^a if long double is IEEE quad precision then full double precision is retained; if long double is IEEE double extended precision then additional, but not full, precision is retained
33. ^a Higham, Nicholas (2002). *"Designing stable algorithms" in Accuracy and Stability of Numerical Algorithms (2 ed)*. SIAM. pp. 27–28.
34. ^a ^b ^c William Kahan. *"Four Rules of Thumb for Best Use of Modern Floating-point Hardware" in Marketing versus Mathematics* ^a (PDF). p. 47.
35. ^a William Kahan (12 February 1981). *"Why do we need a floating-point arithmetic standard? (page 26)"* ^a (PDF).
36. ^a William Kahan (transcribed by David Bindel) (4 June 2001). *"Lecture notes of System Support for Scientific Computation"* ^a (PDF).
37. ^a William Kahan (27 August 2000). *"Marketing versus Mathematics (p 15)"* ^a (PDF).
38. ^a William Kahan (5 July 2005). *"Floating-Point Arithmetic Besieged by "Business Decisions": Keynote Address for the IEEE-Sponsored ARITH 17 Symposium on Computer Arithmetic"* ^a (PDF). p. 6.
39. ^a *"General Decimal Arithmetic"* ^a. Speleotrove.com. Retrieved 25 April 2012.
40. ^a Tom Christiansen, Nathan Torkington, and others (2006). *"perfaq4 / Why is int() broken?"* ^a. perldoc.perl.org. Retrieved 11 January 2011.
41. ^a Higham, Nicholas (2002). *"Subtleties of floating point arithmetic" in Accuracy and Stability of Numerical Algorithms (2 ed)*. SIAM. p. 493.
42. ^a Jonathan Richard Shewchuk (1997). *"Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates, Discrete & Computational Geometry 18:305-363"* ^a.
43. ^a William Kahan and Melody Y. Ivory (3 July 1997). *"Roundoff Degrades an Idealized Cantilever"* ^a (PDF).
44. ^a Higham, Nicholas (2002). *Summation in "Subtleties of floating point arithmetic" in Accuracy and Stability of Numerical Algorithms (2 ed)*. SIAM. pp. 110–123.
45. ^a The equivalence of the two forms can be verified algebraically by noting that the denominator of the fraction in the second form is the conjugate of the numerator of the first. By multiplying the top and bottom of the first expression by this conjugate, one obtains the second expression.

Further reading [\[edit\]](#)

- *What Every Computer Scientist Should Know About Floating-Point Arithmetic* , by David Goldberg, published in the March, 1991 issue of Computing Surveys. Edited reprint from Sun's *Numerical Computation Guide*, which contains an addendum *Differences Among IEEE 754 Implementations* by Doug Priest.
- Nicholas Higham. *Accuracy and Stability of Numerical Algorithms*, Second Edition. SIAM, 2002. [ISBN 0-89871-355-2](#).
- Gene F. Golub and Charles F. van Loan. *Matrix Computations*, Third Edition. Johns Hopkins University Press, 1986. [ISBN 0-8018-5413-X](#)
- Donald Knuth. *The Art of Computer Programming*, Volume 2: *Seminumerical Algorithms*, Third Edition. Addison-Wesley, 1997. [ISBN 0-201-89684-2](#). Section 4.2: Floating Point Arithmetic, pp. 214–264.
- Press et al. *Numerical Recipes in C++*. *The Art of Scientific Computing*, [ISBN 0-521-75033-4](#).
- James H. Wilkinson. *Rounding errors in algebraic processes*. 1963. -- Classic influential treatises on floating point arithmetic.
- James H. Wilkinson. *The Algebraic Eigenvalue Problem*, Clarendon Press, 1965.
- P.H. Sterbenz. *Floating point computation*. 1974. -- Another classic book on floating point and [error analysis](#).
- Jean-Michel Muller, Nicolas Brisebarre, Florent de Dinechin, Claude-Pierre Jeannerod, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé, and Serge Torres. *Handbook of Floating-Point Arithmetic*. 2010. [ISBN 978-0-8176-4705-6](#).

External links [\[edit\]](#)

- Kahan, William and Darcy, Joseph (2001). [How Java's floating-point hurts everyone everywhere](#) . Retrieved 5 September 2003.
- [Survey of Floating-Point Formats](#)  This page gives a very brief summary of floating-point formats that have been used over the years.
- *The pitfalls of verifying floating-point computations* , by David Monniaux, also printed in *ACM Transactions on programming languages and systems (TOPLAS)*, May 2008: a compendium of non-intuitive behaviors of floating point on popular architectures, with implications for program verification and testing
- <http://www.opencores.org>  The OpenCores website contains open source floating-point IP cores for the implementation of floating-point operators in FPGA or ASIC devices. The project, `double_fpu`, contains verilog source code of a double precision floating-point unit. The project, `fpuvhdl`, contains vhdl source code of a single precision floating-point unit.
- [http://msdn.microsoft.com/en-us/library/aa289157\(v=vs.71\).aspx](http://msdn.microsoft.com/en-us/library/aa289157(v=vs.71).aspx)  "Microsoft Visual C++ Floating-Point Optimization", by Eric Fleegal, MSDN, 2004

v · t · e	Data types	[hide]
Uninterpreted	Bit · Byte · Trit · Tryle · Word	
Numeric	Bignum · Complex · Decimal · Fixed point · Floating point (Double precision · Extended precision · Half precision · Minifloat · Octuple precision · Quadruple precision · Single precision) · Integer (signedness) · Interval · Rational	
Text	Character · String (null-terminated)	
Pointer	Address (physical · virtual) · Reference	
Composite	Algebraic data type (generalized) · Array · Associative array · Class · Dependent · Equality · Inductive · List · Object (metaobject) · Option type · Product · Record · Set · Union (tagged)	
Other	Boolean · Bottom type · Collection · Enumerated type · Exception · Function type · Opaque data type · Recursive data type · Semaphore · Stream · Top type · Type class · Unit type · Void	
Related topics	Abstract data type · Data structure · Generic · Kind (metaclass) · Parametric polymorphism · Primitive data type · Protocol (interface) · Subtyping · Type constructor · Type conversion · Type system	

Categories: [Floating point types](#) | [Computer arithmetic](#)