# Computer Arithmetic

**— Perils, Pitfalls, and Practices**

**Nelson H. F. Beebe**

***31 January 2020 [minor updates 06 February 2024]***

This is a 2020 revision of a 2014 draft of a proposed chapter for the GNU C Programming book.

# Short Contents

# Table of Contents

# 1 Numbers and arithmetic

This chapter treats the important issues of numbers and computer arithmetic. While the examples are all in C, many of the issues that are discussed are independent of the programming language, and so can be applied to all languages that are suitable for numeric programming. The chapter includes discussion of several features that are relatively new to C, having been introduced in the ISO C99 Standard, as well as some others that are not yet widely implemented, but should be in the next few years, and should eventually appear in new ISO Standards for C and other languages.

## 1.1 How numbers are represented

Computer hardware usually offers two ways of representing, and working with, numbers: *integer* and *floating point*.

Integers are normally used for counting things, whereas floating-point values are used to represent quantities that may have fractional parts, and may need a wider numeric range than hardware integers can provide. We treat integers first, and then use the rest of this chapter to discuss the more complex floating-point issues.

### 1.1.1 Integer representations

The most familiar number system is integer arithmetic, where, in computers, values are stored as binary (base-2) numbers that occupy a single unit of storage. On modern systems, that is typically an 8-bit `char`, a 16-bit `short int`, a 32-bit `int`, and possibly, a 64-bit `long long int`. Whether a `long int` is a 32-bit or a 64-bit value is system dependent.

In the early days of computing, word sizes varied from 4 to 64 bits, and C has been implemented on at least two platforms with 36-bit words. On hardware manufactured since about 1990, you can be confident that the relevant bit-sizes for integer arithmetic are `8`, `16`, `32`, and `64`. If your compiler conforms to the ISO C Standards of 1999 or 2011, you are guaranteed to have support for 64-bit integers.

The fixed sizes of numeric types necessarily limits their *range*, and the particular encoding of integers decides what that range is.

All modern CPU designs use *two's-complement representation* of signed integers: if they have $n$ bits, their range is from `-2**(n - 1)` to `-1` to `0` to `1` to `+2**(n - 1) - 1`, inclusive, where `**` means *to-the-power*. The leftmost, or high-order, bit is called the *sign bit*.

There is only one zero, and the most negative number lacks a positive companion. As a result, negating that number produces the number back again! We revisit that peculiarity shortly. For example, a two's-complement signed 8-bit integer can represent all decimal numbers from `-128` to `+127`, and `-(-128)` evaluates to `-128`.

When an arithmetic operation produces a value that is too big to represent, the operation is said to *overflow*. For historical reasons having to do with hardware-design choices, integer overflow goes unreported in C programs. What the hardware produces is a longer result whose high-order bits are silently discarded, and the $n$ low-order bits are produced as the result.

In integer arithmetic, if the exact result is representable in $n$ bits, then that is the computed result too, so we often say that *integer arithmetic is exact*, omitting the crucial qualifying phrase *as long as the exact result is representable*.

Two other methods of representing integers need to be mentioned.

In *one's-complement arithmetic*, there are two zeros: one of each sign. As a result, the most negative number does have a positive companion. Machines with one's-complement integers are no longer marketed, but there was as least one C compiler for such a system.

In *sign-magnitude arithmetic*, there are also two zeros. That representation is uncommon in integer arithmetic, but we meet it again in the next section.

In two's-complement representation, negation is done by inverting all of the bits, adding one, propagating any carry, and then discarding any overflow.

To convert a one's-complement integer to its negative, invert all of the bits.

To turn a sign-magnitude integer into its negative, invert just the high-order sign bit.

The encodings are best understood by looking at a table for a tiny integer size, one having just three bits:

| unsigned decimal | signed decimal | bits | sign-mag | one's comp | two's comp |
|---|---|---|---|---|---|
| 0 | 0 | 000 | 000 ( 0) | 111 (-0) | 000 ( 0) |
| 1 | 1 | 001 | 001 ( 1) | 110 (-1) | 111 (-1) |
| 2 | 2 | 010 | 010 ( 2) | 101 (-2) | 110 (-2) |
| 3 | 3 | 011 | 011 ( 3) | 100 (-3) | 101 (-3) |
| 4 | -4 | 100 | 100 (-0) | 011 ( 3) | 100 (-4) |
| 5 | -3 | 101 | 101 (-1) | 010 ( 2) | 011 ( 3) |
| 6 | -2 | 110 | 110 (-2) | 001 ( 1) | 010 ( 2) |
| 7 | -1 | 111 | 111 (-3) | 000 ( 0) | 001 ( 1) |

The parenthesized decimal numbers in the last three column pairs represent the signed values of the indicated binary encodings.

In all three encodings, the interpretations as *unsigned* numbers are identical.

Notice in each of the three systems, the high-order bit is `0` only if the number is non-negative, and the encodings of all three systems are identical for such numbers.

A high-order bit of `1` indicates a negative value in all three systems, but the meanings of the remaining bits differ.

We can now understand the peculiar behavior of negation of the most negative two's-complement integer: start with `100`, invert the bits to get `011`, and add `1`: we get `100`, the value we started with.

The bit pattern of all-bits-one has the value `-1` in two's complement, but `-2**(n - 1) - 1` in sign magnitude, and `-0` in one's complement. Thus, when you mean that particular bit pattern in a C program, represent it as `~0`, which works for all three systems, not as `-1`, which works only in two's complement.

We can also understand overflow behavior in each of the three systems: a sum of two nonnegative values that is too big produces a 1 in the sign bit, so the exact positive result is interpreted as a negative value.

```
sign-magnitude          3 + 1 = 011 + 001 = 100 = (-0)
one's complement        3 + 1 = 011 + 001 = 100 = (-3)
two's complement        3 + 1 = 011 + 001 = 100 = (-4)

sign-magnitude          3 + 2 = 011 + 010 = 101 = (-1)
one's complement        3 + 2 = 011 + 010 = 101 = (-2)
two's complement        3 + 2 = 011 + 010 = 101 = (-3)

sign-magnitude          3 + 3 = 011 + 011 = 110 = (-2)
one's complement        3 + 3 = 011 + 011 = 110 = (-2)
two's complement        3 + 3 = 011 + 011 = 110 = (-3)
=====================================================
```

Although addition, subtraction, multiplication, division, and remainder for integer operands are standardly provided by C and other programming languages, C is unusual in also providing access to bitwise operations, sometimes called *Boolean* or *logical* operations: *AND* (`&`), *OR* (`|`), *eXclusive-OR (XOR)* (`^`), and *NOT* (`!`). The four binary operations, and the single unary operation, can be combined to implement the sixteen possible operations on pairs of input bits. In particular, when those operators are augmented with the *left-shift* (`<<`) and *right-shift* (`>>`) operators, it is possible to access individual bits, and bit fields, in integer words. That ability greatly widens the application area for C code, and has contributed to the widespread use and longevity of C, and its descendant, C++.

### 1.1.2 Floating-point representations

In many practical computations, including most of those in science and engineering, we need to be able to represent numbers with fractional values, and ideally, a larger range than is supplied by hardware integers. The *floating-point representation* satisfies that need, by providing a sign bit, a *significand*, and a power of a fixed base. In the early days of computers, the fixed base was usually either 2 or 10, but some systems used 8, others 16, and one CPU design even used 256, and another used 3. Since the 1970s, base-2 designs have dominated the market, but base-10 formats are returning, and we discuss them later in this chapter.

The wide variation in word size, base, exponent size, and significand size, as well as finer details of how arithmetic operations are really done by the CPU, caused floating-point programmers a good deal of grief and aggravation, imposed significant barriers to software portability to other platforms, and increased the cost of software production and maintenance.

By the late 1970s, it was clear to many numerical analysts that something had to be done, and a committee of volunteers was formed that ultimately produced the *IEEE 754-1985 Standard for Binary Floating-Point Arithmetic*. That standard took several years to develop, and the Intel x86 architecture was the first to implement an early draft, in the 8087 co-processor chip about 1981. Since that time, essentially all new computer designs, and several programming languages, have adopted the IEEE Standard, although some of them, regrettably, fail to implement the full Standard.

Having a standard system for floating-point arithmetic is a great boon to programmers, and the *IEEE 754-2008* revision also requires a separate system for decimal floating-point arithmetic, with a larger significand size that is big enough to satisfy the requirements of

some programming-language standards for representing financial data. At least two CPU architectures provide both binary and decimal floating-point arithmetic, and other vendors are likely to introduce decimal support.

We show later how to find the range and precision of the various IEEE 754 formats (see Section 1.16 [Machine epsilon], page 21).

## 1.2 IEEE 754 binary arithmetic

The base IEEE 754-1985 specification defines two main floating-point formats, in sizes of 32 and 64 bits, and an optional format, called *temporary real*, that requires 80 bits. Several vendors subsequently adopted an alternative format that requires 128 bits.

Each of the four formats encodes the floating-point number as a sign bit, followed by an unsigned biased power of 2, followed by the significand. In the 32-, 64-, and 128-bit formats, the significand has an implicit leading 1-bit that is not stored; the binary point lies between that *hidden bit*, and the stored significand. In the 80-bit format, there is no hidden bit, and the binary point lies between the two high-order bits of the stored significand.

All of those formats are therefore *sign-magnitude* representations, and thus, both `+0` and `-0` exist.

In C, the IEEE 754 32-bit format is called `float`, and the 64-bit format, `double`. It is platform and compiler dependent whether the type `long double` is the 80-bit or the 128-bit format. Some compilers support both, with nonstandard types `__float80` and `__float128`, but usually at least one of those is done in software, rather than in hardware.

There are `8`, `11`, `15`, and `15` exponent bits in the four storage formats, respectively, and accounting for the sign bit, and the hidden bit, that leaves `24`, `53`, `64`, and `113` bits for the significands, which correspond roughly to `6`, `15`, `19`, and `34` decimal digits.

When floating-point arithmetic produces a result that requires a larger significand, the exponent is adjusted to bring the significand back to the required base-point alignment and then a suitable rounding rule is applied to reduce the significand to the fixed storage size.

The IEEE 754 standard therefore has important additional requirements:

- There are four possible rounding rules, to be selectable at run time, rather than being fixed at compile time:
  - *round-to-nearest*, with ties to even;
  - *round-up*, to `+Infinity`;
  - *round-down*, to `-Infinity`;
  - *round-to-zero*.

  Thus, under those four rounding rules, a decimal value `-1.2345` that is to be rounded to a four-digit result would become `-1.234`, `-1.234`, `-1.235`, and `-1.234`, respectively.

  The default rounding rule is *always* the *round-to-nearest*, because that has the least bias, and produces the lowest average error. When the true result lies exactly halfway between two representable machine numbers, the result is rounded to the one that ends with an even digit.

  The *round-to-zero* rule was common on many early computer designs, because it is the easiest to implement: it just requires silent truncation of all extra bits.

The two other rules, *round-up* and *round-down*, are essential for implementing *interval arithmetic*, whereby each arithmetic operation produces lower and upper bounds that are guaranteed to enclose the exact result.

- There are *five* basic operations: *add*, *subtract*, *multiply*, *divide*, and *square root*, and they are **required** to produce a result that is equivalent to the exact, possibly infinite-precision, result, then rounded to storage precision according to the current rounding rule.
- There is a signed representation of Infinity, which is produced by operations like `1 / 0`, `Infinity + Infinity`, `Infinity * Infinity`, and `Infinity + finite`, and also for any overflow from a result that is finite, but is just too big to represent.

  The intent is that IEEE 754 floating-point computation is *nonstop*: unlike older arithmetic designs, overflows should never terminate a computation.
- There are two special values, called Not-a-Number (NaN): a quiet NaN (QNaN), and a signaling NaN (SNaN).

  A QNaN is produced by operations for which the value is undefined in real arithmetic, such as `0 / 0`, `sqrt(-1)`, `Infinity - Infinity`, and any basic operation in which an operand is a QNaN.

  The signaling NaN is intended for initializing otherwise-unassigned storage, and the goal is that unlike a QNaN, an SNaN *does* cause an interrupt that can be caught by a software handler, diagnosed, and reported. In practice, little use has been made of signaling NaNs, because the most common CPUs in desktop and portable computers fail to implement the full IEEE 754 Standard, and supply only one kind of NaN, the quiet one. Also, programming-language standards have taken decades to catch up to the IEEE 754 standard, and implementations of those language standards take even longer to be sufficiently widely available that programmers are willing to use them.

  A NaN always has a sign, by simple virtue of its representation as a floating-point number, but the sign is not significant. Some platforms generate negative NaNs, and others, positive NaNs. However, you can create NaNs of either sign in software, such as with `copysign(x, +1.0)` and `copysign(x, -1.0)`, and they will behave identically.

  We discuss Infinity and NaN in more detail in later sections (see Section 1.11 [Handling Infinity], page 14, and see Section 1.12 [Handling NaN], page 15).
- It can happen that a computed floating-point value is too small to represent, such as when two tiny numbers are multiplied. The result is then said to *underflow*, and the traditional behavior before the IEEE 754 Standard was to set it to zero, but possibly to report the underflow in the program output.

  The IEEE 754 Standard is vague about whether rounding happens before underflow and overflow detection, or after, and CPU designers may choose either.

  However, the Standard does something unusual compared to earlier designs, and that is that when the result is smaller than the smallest *normalized* representable value (i.e., one in which the leading significand bit is `1`), the normalization requirement is relaxed, leading zero bits are permitted, and precision is gradually lost until there are no more bits in the significand. That phenomenon is called *gradual underflow*, and it serves important numerical purposes, although it does reduce the precision of the final result. Some floating-point designs allow the programmer to choose at compile time, or even

at run time, whether underflows are gradual, or are flushed abruptly to zero. Numbers that have entered the region of gradual underflow are called *subnormal*.

- *Sticky exception flags* record the occurrence of particular conditions: once set, they remain set until cleared by the programmer. The conditions include *invalid operand*, *division-by-zero*, *inexact result* (i.e., one that required rounding), *underflow*, and *overflow*. Some extended floating-point designs offer several additional exception flags. The C99 functions `feclearexcept()`, `feraiseexcept()`, and `fetestexcept()` provide a standardized interface to those flags.

  One important use of those flags is to do a computation that is normally expected to be exact in floating-point arithmetic, but occasionally might not be, in which case, corrective action is needed. The programmer can clear the *inexact result* flag with a call to `feclearexcept(FE_INEXACT)`, do the computation, and then test the flag with `fetestexcept(FE_INEXACT)`; the result of that call is `0` if all is well and the flag is not set, and otherwise `1` when the flag is set, and more work is needed.

## 1.3 IEEE 754 decimal arithmetic

One of the difficulties that users of computers for numerical work face, whether they realize it or not, is that the computer does not operate in the number base that most people are familiar with. As a result, input decimal fractions must be converted to binary floating-point values for use in computations, and then the final results converted back to decimal for humans. Because the precision is finite and limited, and because algorithms for correct round-trip conversion between number bases were not known until the 1990s, and are still not implemented on most systems and most programming languages, the result is frequent confusion for users, when a simple expression like `3.0*(1.0/3.0)` evaluates to `0.999999` instead of the expected `1.0`. Here is an example from a floating-point calculator that allows rounding-mode control, with the mode set to *round-to-zero*:

```
for (k = 1; k <= 10; ++k)
     (void)printf("%2d\t%10.6f\n", k, k*(1.0/k))
 1        1.000000
 2        1.000000
 3        0.999999
 4        1.000000
 5        0.999999
 6        0.999999
 7        0.999999
 8        1.000000
 9        0.999999
10        0.999999
```

Increasing working precision can sometimes help by reducing intermediate rounding errors, but the reality is that when fractional values are involved, *no amount* of extra precision can suffice for some computations. For example, the nice decimal value `1/10` in C99-style binary representation is `+0x1.999999999999ap-4`; that final digit `a` is the rounding of an infinite string of `9`'s.

Financial computations in particular depend critically on correct arithmetic, and the losses due to *rounding errors* can be large, especially for businesses with large numbers

of small transactions, such as grocery stores and telephone companies. Tax authorities are particularly picky, and demand specific rounding rules, including one that instead of rounding ties to the nearest number, rounds instead in the direction that favors the taxman.

Programming languages used for business applications, notably the venerable Cobol language, have therefore always implemented financial computations in *fixed-point decimal arithmetic* in software, and because of the large monetary amounts that must be processed, successive Cobol standards have increased the minimum number size from 18 to 32 decimal digits, and the most recent one requires a decimal exponent range of at least `[-999, +999]`.

The revised IEEE 754-2008 standard therefore requires decimal floating-point arithmetic, as well as the now-widely used binary formats from 1985. Like the binary formats, the decimal formats also support Infinity, NaN, and signed zero, and the five basic operations are also required to produced correctly rounded representations of infinitely precise exact results.

However, the financial applications of decimal arithmetic introduce some new features:

- There are three decimal formats occupying 32, 64, and 128 bits of storage, and offering exactly 7, 16, and 34 decimal digits of precision. If one size has `n` digits, the next larger size has `2 n + 2` digits. Thus, a future 256-bit format would supply 70 decimal digits, and at least one library already supports the 256-bit binary and decimal formats.
- Decimal arithmetic has an additional rounding mode, called *round-ties-to-away-from-zero*, meaning that a four-digit rounding of `1.2345` is `1.235`, and `-1.2345` becomes `-1.235`. That rounding mode is mandated by financial laws in several countries.
- The decimal significand is an *integer*, instead of a fractional value, and trailing zeros are only removed at user request. That feature allows floating-point arithmetic to emulate the *fixed-point arithmetic* traditionally used in financial computations.

We can easily estimate how many digits are likely to be needed for financial work: seven billion people on Earth, with an average annual income of less than US$10,000, means a world financial base that can be represented in just 15 decimal digits. Even allowing for alternate currencies, future growth, multiyear accounting, and intermediate computations, the 34 digits supplied by the 128-bit format are more than enough for financial purposes.

We return to decimal arithmetic later in this chapter (see Section 1.18 [More on decimal floating-point arithmetic], page 26), after we have covered more about floating-point arithmetic in general.

## 1.4 Exact floating-point arithmetic

Contrary to seemingly widespread belief, repeated in text books and programming classes, floating-point arithmetic is *not* fuzzy. As long as numbers are exactly representable, and intermediate results do not require rounding, then floating-point arithmetic is *exact*. That fact is exploited in many software packages, including some that provide arbitrary-precision arithmetic using floating-point, rather than integer, basic operations.

It is easy to predict how many digits are needed for the results of arithmetic operations:

- addition and subtraction of two `n`-digit values with the *same* exponent requires at most `n + 1` digits, but when the exponents differ, many more digits may be needed;
- multiplication of two `n`-digit values requires exactly `2 n` digits;

- although integer division produces a quotient and a remainder of no more than `n`-digits, floating-point remainder and square root may require an unbounded number of digits, and the quotient can need many more digits than can be stored.

Whenever a result requires more than `n` digits, a rounding rule must be applied to reduce the result to the storage size. We describe later how to detect the use of a rounding rule that makes results inexact (see [fetestexcept], page 6).

## 1.5 Rounding issues

The default IEEE 754 rounding mode minimizes errors, and most normal computations should not suffer any serious accumulation of errors from rounding.

Of course, one can contrive examples where that is not so. Here is one: iterate a square root, then attempt to recover the original value by repeated squaring. This computation was done with the IEEE 754 64-bit format:

```
x = 100.0;
for (n = 10; n <= 100; n += 10)
{
    y = x;
    for (k = 0; k < n; ++k) { y = sqrt(y); }
    for (k = 0; k < n; ++k) { y *= y; }
    (void)printf("n = %3d; x = %.0f\ty = %.6f\n", n, x, y);
}
```

```
n =  10; x = 100        y = 100.000000
n =  20; x = 100        y = 100.000000
n =  30; x = 100        y = 99.999977
n =  40; x = 100        y = 99.981025
n =  50; x = 100        y = 90.017127
n =  60; x = 100        y = 1.000000
n =  70; x = 100        y = 1.000000
n =  80; x = 100        y = 1.000000
n =  90; x = 100        y = 1.000000
n = 100; x = 100        y = 1.000000
```

After 50 iterations, `y` has barely one correct digit, and soon after, there are no correct digits.

## 1.6 Significance loss

A much more serious source of error in floating-point computation is *significance loss* from subtraction of nearly equal values. If the values are close enough, but still unequal, a *single subtraction* can wipe out all correct digits, possibly contaminating all future computations.

Floating-point calculations can sometimes be carefully designed so that significance loss is not possible, such as summing a series where all terms have the same sign. As a comparative example, the series expansions of the trigonometric and hyperbolic sines have terms of identical magnitude, of the general form `x**(2*n + 1) / (2*n + 1)!`. However, those in the trigonometric series alternate in sign, while those in the hyperbolic series are all positive.

Here is the output of two small programs that sum the series, and compare the computed sums with known-to-be-accurate library functions:

```
# sin(x) series summation with alternating signs
# with k terms needed to converge to machine precision

x = 10      k = 51
s(x)   = -0.544_021_110_889_270
sin(x) = -0.544_021_110_889_370

x = 20      k = 81
s(x)   = 0.912_945_250_749_573
sin(x) = 0.912_945_250_727_628

x = 30      k = 109
s(x)   = -0.987_813_746_058_855
sin(x) = -0.988_031_624_092_862

x = 40      k = 137
s(x)   = 0.617_400_430_980_474
sin(x) = 0.745_113_160_479_349

x = 50      k = 159
s(x)   = 57_105.187_673_745_720_532
sin(x) = -0.262_374_853_703_929

# sinh(x) series summation with positive signs
# with k terms needed to converge to machine precision

x = 10      k = 47
t(x)    = 1.101_323_287_470_340e+04
sinh(x) = 1.101_323_287_470_339e+04

x = 20      k = 69
t(x)    = 2.425_825_977_048_951e+08
sinh(x) = 2.425_825_977_048_951e+08

x = 30      k = 87
t(x)    = 5.343_237_290_762_229e+12
sinh(x) = 5.343_237_290_762_231e+12

x = 40      k = 105
t(x)    = 1.176_926_334_185_100e+17
sinh(x) = 1.176_926_334_185_100e+17

x = 50      k = 121
t(x)    = 2.592_352_764_293_534e+21
```

```
sinh(x) = 2.592_352_764_293_536e+21
```

We used an extended C library that supplies underscores in output numbers to enhance readability. It is clear that the `sinh(x)` series with positive terms can be summed to high accuracy, whereas that for `sin(x)` rapidly loses accuracy, so that at `x = 30`, only two correct digits remain, and soon, all digits are wrong, and the answers are complete nonsense.

An important skill in numerical programming is to recognize when significance loss is likely to contaminate a computation, and revise the algorithm so that subtraction losses can be eliminated, or at least reduced. Sometimes, the only practical way to do so is to compute in higher intermediate precision, which is why the extended types like `long double` and `__float128` can be important to have. Sadly, there are numerous platforms where the C implementation simply does not provide access to them, even when the hardware has them.

## 1.7 Fused multiply-add

In 1990, when IBM introduced the POWER architecture, the CPU provided a previously unknown instruction, the *fused multiply-add* (FMA). It computes the value `x * y + z` with an **exact** double-length product, followed by an addition with a *single* rounding. Numerical computation often needs pairs of multiply and add operations, for which the FMA is well-suited.

On the POWER architecture, there are two dedicated registers that hold permanent values of `0.0` and `1.0`, and the normal *multiply* and *add* instructions are just wrappers around the FMA that compute `x * y + 0.0` and `x * 1.0 + z`, respectively.

In the early days, it appeared that the main benefit of the FMA was getting two floating-point operations for the price of one, almost doubling the performance of some algorithms. However, numerical analysts have since shown numerous uses of the FMA for significantly enhancing accuracy. We discuss one of the most important ones in the next section.

A few other architectures have since included the FMA, and most provide variants for the related operations `x * y - z` (FMS), `-x * y + z` (FNMA), and `-x * y - z` (FNMS). The IEEE 754-2008 revision requires implementations to provide the FMA, as a sixth basic operation.

The ISO C99 Standard requires provision of the functions `fmaf(x, y, z)`, `fma(x, y, z)`, and `fmal(x, y, z)` for the usual `float`, `double`, and `long double` data types. Correct implementation of the FMA in software is difficult, and some systems that appear to provide those functions do not satisfy the single-rounding requirement. That situation should change as more programmers use the FMA operation, and more CPUs provide FMA in hardware.

## 1.8 Error recovery

When two numbers are combined by one of the four basic operations, the result often requires rounding to storage precision. For accurate computation, one would like to be able to recover that rounding error. With historical floating-point designs, it was difficult to do so portably, but now that IEEE 754 arithmetic is almost universal, the job is much easier.

For addition with the default *round-to-nearest* rounding mode, we can find a sum and its error with code like this:

```
volatile double err, sum, tmp, x, y;
```

```
if (fabs(x) >= fabs(y))
{
    sum = x + y;
    tmp = sum - x;
    err = y - tmp;
}
else /* fabs(x) < fabs(y) */
{
    sum = x + y;
    tmp = sum - y;
    err = x - tmp;
}
```

Now, `x + y` is *exactly* represented by `sum + err`, and that basic operation, which has come to be called *twosum* in the numerical-analysis literature, is the first key to tracking, and accounting for, rounding error.

For the error in subtraction, just swap the `+` and `-` operators.

We used the `volatile` qualifier in the declaration of the variables, which forces the compiler to store and retrieve them from memory, and prevents optimizations that would rearrange the code and completely destroy the results by reducing `err = y - ((x + y) - x)` to `err = 0`.

More complex code is needed for other rounding modes, and for historical floating-point designs with less-well-specified rounding behavior. Also, the double-length representation does not hold exactly near the underflow and overflow regions. However, for IEEE 754 arithmetic in its default mode, the *twosum* operation is one that deserves to be in math libraries, but so far, has not been included in any ISO Standard for a programming language. It is possible to make a variant of *twosum* that does not require magnitude tests on the operands, but we leave that for further study in the references at the end of this chapter.

For multiplication, the rounding error can be recovered without magnitude tests by using the FMA operation, like this:

```
volatile double err, prod, x, y;
prod = x * y;               /* rounded product */
err  = fma(x, y, -prod);    /* exact product = prod + err */
```

A correct implementation of the FMA is critical: if the library code just computes it as `x * y + z`, with two roundings, then we get a zero value for `err`.

For addition, subtraction, and multiplication, we have *exact* representation of the results with a two-term sum or product. However, division, remainder, and square root potentially require an infinite number of digits, so their errors can only be approximated. Nevertheless, we can get an error term that is close to the true error: it is just that value rounded to machine precision.

For division, we can find `x / y` close to `quo + err` like this:

```
volatile double err, quo, x, y;
quo = x / y;
err = fma(-quo, y, x) / y;
```

For square root, we find `sqrt(x)` nearly equal to `root + err` like this:

```
volatile double err, root, x;
root = sqrt(x);
err = fma(-root, root, x) / (root + root);
```

With the reliable and predictable floating-point design provided by IEEE 754 arithmetic, we now have the tools we need to track errors in the five basic floating-point operations, and we can effectively simulate computing in twice working precision, which is sometimes sufficient to remove almost all traces of arithmetic errors.

## 1.9 Double-rounding problems

It is often true in numerical work that a few more digits of precision, and a somewhat wider exponent range, in intermediate computations can largely mask the effect of cumulative rounding errors, mitigate the possibly disastrous effects of premature underflow and overflow, and simplify algorithms. It was for that reason that the IEEE 754 Standard included an optional *temporary real* format, with 11 more bits in the significand, and 4 more bits in the biased exponent. That format has been provided in hardware on Motorola 68020, Intel x86, and HP/Intel IA-64. It is also available via an x86-like instruction set in the AMD and Intel x86-64 architecture that is now commonly found in desktop computers.

Compilers are free to exploit the longer format, and most do so. That is usually a *good thing*, such as in computing a lengthy sum or product, or in implementing the computation of the hypotenuse of a right-triangle as `sqrt(x*x + y*y)`: the wider exponent range is critical there for avoiding disastrous overflow or underflow.

However, sometimes it is critical to know what the intermediate precision and rounding mode are, such as in tracking errors with the techniques of the preceding section. Some compilers provide options to prevent the use of the 80-bit format in computations with 64-bit `double`, but ensuring that code is always compiled that way may be difficult. The x86 architecture has the ability to force rounding of all operations in the 80-bit registers to the 64-bit storage format, and some systems provide a software interface with the functions `fesetprec()` and `fegetprec()`. Unfortunately, neither of those functions is defined by the ISO Standards for C and C++, and consequently, they are not standardly available on all platforms that use the x86 floating-point design.

When `double` computations are done in the 80-bit format, results necessary involve a *double rounding*: first to the 64-bit significand in intermediate operations in registers, and then to the 53-bit significand when the register contents are stored to memory. Here is an example in decimal arithmetic where such a double rounding results in the wrong answer: round `1_234_999` from seven to five to three digits. The result is `1_240_000`, whereas the correct representation to three significant digits is `1_230_000`.

One way to reduce the use of the 80-bit format is to declare variables as `volatile double`: that way, the compiler is required to store and load intermediates from memory, rather than keeping them in 80-bit registers over long sequences of floating-point instructions. Doing so does not, however, eliminate double rounding. The now-common x86-64 architecture has separate sets of 32-bit and 64-bit floating-point registers, and compiler options may be available to restrict floating-point operations to only those registers, eliminating the possibility of double rounding.

## 1.10 Exact specification of floating-point numbers

One of the frustrations that numerical programmers have suffered with since the dawn of digital computers is the inability to precisely specify numbers in their programs. On the early decimal machines, that was not an issue: you could write a constant `1e-30` and be confident of that exact value being used in floating-point operations. However, when the hardware works in a base other than 10, then human-specified numbers have to be converted to that base, and then converted back again at output time. The two base conversions are rarely exact, and unwanted rounding errors are introduced.

C99 was the first programming language to introduce a solution to that problem: hexadecimal floating-point constants. They are written like this: `+0x1.fffffcp-1`, the number that is the IEEE 754 32-bit value closest to, but below, `1.0`. The significand is represented as a hexadecimal fraction, and the *power of two* is written in decimal following the exponent letter `p` (the traditional exponent letter `e` is not possible, because it is a hexadecimal digit).

The `printf()` and `scanf()` families were extended in C99 to recognize the `%a` format specifier for writing and reading hexadecimal floating-point values, as in this code that reproduces our sample number:

```
(void)printf("%a\n", 1.0 - pow(2.0, -23) );
0x1.fffffcp-1
```

The `strtod()` family was similarly extended to recognize numbers in that new format.

If you want to ensure exact data representation for transfer of floating-point numbers between C/C++ programs on different computers, then hexadecimal constants are an optimum choice. If you instead use the raw I/O functions, `fread()` and `fwrite()`, then you have to deal with byte-order issues: some platforms are *little-endian* designs, whereas others are *big-endian*. In the former, the high-order byte with the sign bit is written last, whereas in the latter, it is written first.

If you just need specific floating-point constants of many digits in your software, you cannot rely on accurate conversion to binary of a long stream of decimal digits. One reasonable way to get accurate constants is to represent them as a sum of nonoverlapping exact high, and approximate low, parts. When that sum is evaluated, the result should be a correctly rounded value for the constant. Even better, you can use the effective higher precision of the sum in calculations that require that constant. For example, to convert from circular degrees to radians, you need to multiply by `pi / 180`. In 64-bit arithmetic, you could use code like this:

```
double degrees, radians;
static const double PI_OVER_180_HI =   157205283378410.0 /
                                      9007199254740992.0;
static const double PI_OVER_180_LO = -2.3991263436588282e-17;

radians = fma(degrees, PI_OVER_180_HI, degrees * PI_OVER_180_LO);
```

Although the numbers in the high part are long, they are each exactly representable in 53 or fewer bits, so their conversion by the compiler from strings to floating-point is exact. The denominator is just `2**53`, a power of the base, so the division is *exact*, and the high part is then too. The low part is *inexact*, but should be close to a correct rounding of the true value. The `fma` operation therefore computes a 106-bit value that is rounded just once to a

53-bit value, so the result should, with high probability, be the nearest machine number to the infinite-precision value of `radians`.

If you wonder where the high-precision values in our constants come from, the answer is Lisp and symbolic-algebra systems, such as `guile` (see `http://www.gnu.org/software/guile/`) and `maxima` (see `http://maxima.sf.net/`) They allow computation in arbitrary-precision arithmetic, and are often essential tools for developing numerical software.

## 1.11 Handling Infinity

As we noted earlier, the IEEE 754 model of computing is *nonstop* operation: exceptional values or conditions are recorded in sticky exception flags, or in results with the special values Infinity and QNaN. In this section, we deal only with the first of those special values; see Section 1.12 [Handling NaN], page 15, for the other.

In principle, you should be able to create a value of signed Infinity in software like this:

```
double x;

x = -1.0 / 0.0;
```

That works with many compilers, but some refuse to accept that code on the flimsy grounds that division by zero is *undefined*. It is not, and IEEE 754 is designed to support exactly that behavior. You should therefore complain to your compiler supplier about its erroneous nonstandard behavior.

C99 supplies the `INFINITY` macro for use as a compile-time constant. Like the expression `-1.0 / 0.0`, it does not set the *overflow* exception flag, because the evaluation happens at compile time, rather than at run time.

A safer way to get an Infinity is via a call to a private function, so that function can hide the messy details. C99 unfortunately does not provide such a function. Here is one possible implementation:

```
double
inf(void)
{
    volatile double x;
    x = 1.0;
    return (1.0 / (x - x));
}
```

Here, the `volatile` qualifier comes to the rescue again, and forces the compiler to actually generate code that computes the difference `x - x` at run-time, rather than optimizing the expression away to zero, and then rejecting the code because of the division by zero. Run-time evaluation means that the *overflow* exception flag is set.

C99 provides a standard function to test for an Infinity: `isinf(x)` returns `1` if the argument is a signed infinity, and `0` if not. You could easily write such a function like this:

```
int
isinf(double x)
{
    volatile double y;
```

```
    y = inf();

    return ( (x == y) || (x == -y) );
}
```

Here is another implementation that avoids a function call, and exploits the fact that the reciprocal of the largest *finite* representable IEEE 754 number in all decimal and binary formats is itself representable, and *nonzero*:

```
int
isinf(double x)
{
    return ( (x != 0.0) && ( (1.0 / x) == 0.0 ) );
}
```

You can also detect an Infinity by extracting the exponent and significand bits to find the particular encoding that identifies such values. However, the code to do so is messy, possibly byte-order dependent, and different for each of the seven binary and decimal floating-point formats. By contrast, only trivial type changes are need to prepare versions of either of our implementations for use with the six other types, and our code is completely portable, and clear.

Notice that Infinities can be compared, and all Infinities of the same sign are *equal*: there is no notion in IEEE 754 arithmetic of different kinds of Infinities, as there are in some areas of mathematics.

Infinities propagate in addition, subtraction, multiplication, and square root, but in division, they disappear, because of the rule that `finite / Infinity` is `0.0`. Thus, an overflow in an intermediate computation that produces an Infinity is likely to be noticed later in the final results. The user can then decide whether that is expected, and acceptable, or whether the code possibly has a bug, or needs to be run in higher precision, or redesigned to avoid the production of the Infinity.

## 1.12 Handling NaN

NaNs are a lovely feature of the IEEE 754 arithmetic system that supply a floating-point representation of values that are not numbers: they represent values from computations that produce undefined results. They are not new with IEEE arithmetic: two supercomputer vendors of the 1960s and 1970s provided them, but called them `Indefinite`. They were also present in machines that Konrad Zuse built in Germany in the late 1930s, but knowledge of those systems was lost for decades, so it is unclear whether Zuse's special values for Infinity and NaN had any influence on their inclusion in the IEEE 754 specification.

NaNs have a distinctive property that makes them unlike any other floating-point value: they are *unequal to everything, including themselves*! Thus, it should be possible in *every* programming language to write a test for a NaN with code similar to this:

```
if (x != x)
    (void)printf("Whoops: x is a NaN\n");
```

Unfortunately, just as with the inline production of Infinity with `1.0 / 0.0`, there are compilers that mishandle the inequality test, and optimize it to zero (false). That behavior too should justify a bitter complaint to the compiler vendor.

In the meantime, it is therefore better to use the C99 function `isnan(x)` to test whether the argument is a NaN. It returns `1` if it is, and `0` otherwise.

A simple implementation could look like this:

```
int
isnan(double x)
{
    return (x != x);
}
```

However, because of the noted compiler issues, it may be necessary to rewrite it like this:

```
int
isnan(double x)
{
    volatile double y;

    y = x;
    return (x != y);
}
```

Just as with Infinity, it is also possible to grovel about in the bits of the exponent and significand to figure out whether a value is a NaN or not, but that too requires separate code for each data type, and for both big-endian and little-endian systems. It is far better to use the closest thing to the original test, `x != x`, because that compiles into just one or two instructions on every architecture.

Up to now, we have avoided getting into the specifics of the floating-point encoding of the various IEEE 754 formats, and we shall continue to do so, except for one important point: Infinity and NaN are encoded in the binary format with the *same exponent*: the maximum representable value, but the stored significand bits are all zero for an Infinity, and at least one of them is nonzero for a NaN. The IEEE 754 Standard does not specify further how quiet and signaling NaNs are encoded, except to forbid the use of the sign bit for that purpose. Some floating-point designs set the high-order *fraction* bit of the significand to `1` to indicate a quiet NaN, and others use that bit to mean a signaling NaN. The remaining bits are set to all zeros, or all ones, depending on the design.

Those design choices mean that there are lots of unused bits in the significand of a NaN, and the questions are: *Do they matter for the floating-point instruction set?*, and, *Are they useful for anything?*

The answer to the first is *no*: apart from the single bit used for the *quiet/signaling* mark, the rest of the bits do not matter. All NaNs of each type behave exactly the same way in arithmetic operations.

The answer to the second is: yes, you can store a *payload* there, and the C99 functions with the prototypes

```
#include <math.h>           /* include for these prototypes */

double      nan  (const char *tagp);
float       nanf (const char *tagp);
long double nanl (const char *tagp);
```

allow you to do so. For example, on one platform, the call `nan("deadbeefcafe")` produces the storage encoding `0x7ffcdead_beefcafe`.

Having control over the NaN payload is useful, because you can mark data differently. For example, otherwise-uninitialized array storage could be filled with NaNs in which the array index, or the low-order 52-bits of the array element's memory address, form the payload. Then, when a NaN later appears in a computation because you referenced an array element without having assigned to it, the payload tells you where it came from.

That error-detection scheme is not completely foolproof, because the behavior of some instruction sets is to return the input NaN as the output NaN, whereas others generate a fresh quiet NaN with an empty payload. Depending on how the code is programmed and compiled, the *invalid operand* flag may or may not be set.

In traditional mathematics, and programming languages before IEEE 754 arithmetic, there are often hidden assumptions that comparisons can be used to classify numbers into *less than zero*, *zero*, and *greater than zero*. Thus, it is common to see code like this:

```
if (x < 0.0)
{
    ... handle negative x ...
}
else if (x == 0.0)
{
    ... handle zero x ...
}
else
{
    ... handle positive x ...
}
```

With IEEE 754 arithmetic, such code is **wrong**! The reason is that comparisons with either, or both, operands that are NaNs evaluate to false, and also, set the *invalid operand* exception flag. Thus, the `else` branch is entered not just when $x$ is positive, but also when $x$ is a NaN.

Sometimes, you want to avoid setting an exception flag in a conditional test. C99 supplies the macros `isgreater()`, `isgreaterequal()`, `isless()`, `islessequal()`, `islessgreater()`, and `isunordered()` for making comparisons without setting the *invalid operand* flag. The `isunordered()` macro returns 1 if either, or both, arguments are NaNs.

One important use of NaNs is marking of *missing data*. For example, in statistics, such data must be omitted from computations. Use of any particular finite value for missing data would eventually collide with real data, whereas such data could never be a NaN, so it is an ideal marker. Functions that deal with collections of data that may have holes can be written to test for, and ignore, NaN values.

It is easy to generate a NaN in computations: evaluating `0.0 / 0.0` is the commonest way, but `Infinity - Infinity`, `Infinity / Infinity`, and `sqrt(-1.0)` also work. Functions that receive out-of-bounds arguments can choose to return a stored NaN value, such as with the C99 `NAN` macro, but that does not set the *invalid operand* exception flag, which is usually undesirable.

Like Infinity, NaNs propagate in computations, but they are even stickier, because they never disappear in division. Thus, once a NaN appears in a chain of numerical operations, it is almost certain to pop out into the final results. The programmer then has to decide whether that is expected, or whether there is a coding or algorithmic error that needs repair.

In general, you should expect any function that gets a NaN argument to return a NaN, and preferably, the precise input NaN, so that you can exploit its payload. However, there are two exceptions in the C99 math-library specification that you need to be aware of, because they violate the *NaNs-always-propagate* rule:

- `pow(x, 0.0)` always returns `1.0`, even if `x` is 0.0, Infinity, or a NaN.
- If just one of the arguments to `fmax(x, y)` or `fmin(x, y)` is a NaN, the other argument is returned. If both arguments are NaNs, a NaN is returned, but there is no requirement about where it comes from: it could be `x`, or `y`, or a fresh quiet NaN.

NaNs are also used for the return values of math-library functions where the result is not representable in real arithmetic, or is mathematically undefined or uncertain, such as `sqrt(-1.0)` and `sin(Infinity)`. However, note that a result that is merely too big to represent should always produce an Infinity, such as with `exp(1000.0)` (too big) and `exp(Infinity)` (truly infinite).

A properly designed I/O library for C99 should always be able to output NaN and Infinity as recognizable text strings, such as `nan` and `inf`, and payloads may be indicated by something like `nan(0xdeadbeef)`. Similarly, those strings should always be acceptable as input values, but be forewarned, on some defective systems, they are not. Once again, such misbehavior needs to be reported to vendors.

## 1.13 Signed zeros

The fact that most historical, and IEEE 754, floating-point representations use the sign-magnitude convention means that, in principle, both positive and negative zeros are possible.

On the once-common VAX architecture, a negative zero is called a *reserved operand*, and its use in arithmetic instructions, and even in load and store instructions, causes an instruction fault that generally terminates the job. On other historical platforms, the hardware would usually ignore the sign of zero, and would never generate a negative zero.

In IEEE 754 arithmetic, however, the sign of zero is significant, and important, because it records the creation of a value that is too small to represent, but came from either the negative axis, or from the positive axis. Such fine distinctions are essential for proper handling of *branch cuts* in complex arithmetic, a system that is also supported by C99, and that we discuss later (see Section 1.17 [Complex arithmetic], page 24).

The key point about signed zeros is that in comparisons, their sign does not matter: `0.0 == -0.0` must *always* evaluate to `1` (true). However, if you write `-0.0` in your code, some compilers misbehave, and convert it to a positive zero. That too is grounds for an annoyed complaint to the compiler vendor. In the meantime, the portable way to deal with that problem is to create a negative zero at run time by small subterfuges, like one of these:

```
volatile double negzero, x;

x = 0.0;
negzero = -x;
```

```
negzero = -1.0 / inf();

negzero = copysign(0.0, -1.0);      /* safest, and recommended, way */

negzero = sin(-x);
```

The last way is, unfortunately, not reliable, for two reasons: (a) some compilers generate code that evaluates the negative zero as a positive zero, so the `sin` function gets, and returns, a positive zero; and (b) some math-library implementations of that function are defective, and do something like this:

```
if (x is sufficiently tiny)
    return (0.0);
```

The correct way to write that code is

```
if (x is sufficiently tiny)
    return (x);
```

In general, any function that has a series expansion for small arguments that begins `x + b * x**2 + c * x**3 + ...` should return `x`, not `0.0`, when the terms after the first are found to be negligible. Even better, instead of just returning `x`, it should return the sum of the first two terms, because that requires arithmetic operations that obey the current rounding rule, which might not be the default one.

Input and output functions should always preserve the sign of zero, but alas, that is not always the case. When it matters, and the code needs to be portable, then you may need to develop your own I/O support, or coding and file-storage practices, to work around those limitations.

Signed zeros, Infinity, and NaN prevent some optimizations by programmers and compilers that might otherwise have seemed obvious:

- `x + 0` and `x - 0` are not the same as `x` when `x` is zero, because the result depends on the rounding rule.
- `x * 0.0` is not the same as `0.0` when `x` is Infinity, a NaN, or negative zero.
- `x / x` is not the same as `1.0` when `x` is Infinity, a NaN, or zero.
- `(x - y)` is not the same as `-(y - x)` because when the operands are finite and equal, one evaluates to `+0` and the other to `-0`.
- `x - x` is not the same as `0.0` when $x$ is Infinity or a NaN.
- `x == x` and `x != x` cannot be simplified to `1` and `0` when $x$ is a NaN.
- `x < y` and `isless(x, y)` are not equivalent, because the first sets a sticky exception flag when an operand is a NaN, whereas the second does not affect that flag. The same holds for the other `isxxx()` functions that are companions to relational operators.

Even though IEEE 754 arithmetic has been available for more than three decades, such mistaken optimizations are still sometimes encountered, and harm floating-point software portability.

## 1.14 Scaling by powers of the base

We have discussed rounding errors several times in this chapter, but it is important to remember that when results require no more bits than the exponent and significand bits can represent, those results are *exact*.

One particularly useful exact operation is scaling by a power of the base. While one, in principle, could do that with code like this:

```
y = x * pow(2.0, (double)k);        /* undesirable scaling: avoid! */
```

that is not advisable, because it relies on the quality of the math-library power function, and that happens to be one of the most difficult functions in the C math library to make accurate. What is likely to happen on many systems is that the returned value from `pow()` will be close to a power of two, but slightly different, so the subsequent multiplication introduces rounding error.

The correct, and fastest, way to do the scaling is either via the traditional C library function, or with its C99 equivalent:

```
y = ldexp(x, k);            /* traditional pre-C99 style */
y = scalbn(x, k);           /* C99 style */
```

With either function, $y$ is set to `x * 2**k`.

## 1.15 Rounding control

IEEE 754 requires that rounding control be possible at run time, and the C99 revisions include a standard way to access that control. Prototypes for the needed functions are supplied in a new header file, like this:

```
#include <fenv.h>           /* include for these prototypes */

int fegetround(void);
int fesetround(int round);
```

The header file also provides a set of constants that can be used to select the rounding rule: `FE_DOWNWARD`, `FE_TONEAREST`, `FE_TOWARDZERO`, and `FE_UPWARD`. One of those should be returned by the call `fegetround()`. However, it returns a negative value if the requested action fails, and user code should be prepared to deal with that possibility, even though it should never happen on a platform with IEEE 754 arithmetic.

On some architectures, changing the current rounding rule is a relatively expensive operation, so it is desirable to minimize the number of changes. For interval arithmetic, we seem to need three changes for each operation, but we really only need two, because we can write code like this example for interval addition of two reals:

```
interval_double v;
volatile double x, y;
int rule;

rule = fegetround();

if (fesetround(FE_UPWARD) == 0)
{
```

```
        v.hi = x + y;
        v.lo = -(-x - y);
    }
    else
        fatal("ERROR: failed to change rounding rule");

    if (fesetround(rule) != 0)
        fatal("ERROR: failed to restore rounding rule");
```

The now-familiar `volatile` qualifier is essential to prevent an optimizing compiler from producing the same value for both bounds. The code also needs to be compiled with an option that forbids the compiler from using higher precision for the basic operations, because that would introduce double rounding that could spoil the bounds guarantee of interval arithmetic.

At the time of writing this, decimal rounding is not yet supported in GCC on those systems where the compiler can be built with support for decimal floating-point arithmetic.

## 1.16 Machine epsilon

In any floating-point system, three attributes are particularly important to know: *base*, *precision* (how many digits in the significand?), and *range* (how small or big can a number be?). The allocation of bits between exponent and significand decides the answers to those questions.

With historical designs, it was often the case that the exponent size was *fixed* for all significand precisions, with a range that might be approximately similar to `[1e-38, 1e+38]`. However, more than a half-century of numerical analysis has shown that practical applications generally require the range to increase when the significand precision increases. The IEEE 754 design supplies that need, providing, as we noted earlier, `8`, `11`, `15`, and `15` bits for the exponent field in the four formats.

A measure of the precision is the answer to the question: what is the smallest number that can be added to `1.0` such that the sum differs from `1.0`? That number is called the *machine epsilon*. For the four IEEE 754 binary formats, it is `0x1p-23`, `0x1p-52`, `0x1p-63`, and `0x1p-112`, respectively. Notice that each is an exact power of the base (two). For the three decimal formats, it is `1e-6`, `1e-15`, and `1e-33`. The machine epsilon is always an exactly representable number that is equal to the base raised to the negative of one less than the number of significand digits. For the IEEE 754 formats with `n`-digit significands, that means `2**(-(n - 1))` or `10**(-(n - 1))`.

Given that most C code now can rely on having IEEE 754 arithmetic, and a C99 compiler, one could define the needed machine-epsilon constants like this:

```
static const float  epsf       = 0x1p-23;  /* about 1.192e-07 */
static const double eps        = 0x1p-52;  /* about 2.220e-16 */
static const long double epsl  = 0x1p-63;  /* about 1.084e-19 */
static const __float128 eps128 = 0x1p-112; /* about 1.926e-34 */
```

Instead of the hexadecimal constants, we could also have used the Standard C macros, `FLT_EPSILON`, `DBL_EPSILON`, and `LDBL_EPSILON`. There are no such macros for the nonstandard types `__float80` and `__float128`.

It is useful to be able to compute the machine epsilons at run time, and we can easily generalize the operation by replacing the constant `1.0` with a user-supplied value:

```
double
macheps(double x)
{   /* return machine epsilon for x such that x + macheps(x) > x */
    static const double base = 2.0;
    volatile double eps;

    if (isnan(x))
        eps = x;
    else
    {
        volatile double v;

        v = x;
        eps = (v == 0.0) ? 1.0 : v;

        while ((v + eps / base) != v)
            eps /= base;                    /* always EXACT! */
    }

    return (eps);
}
```

If we call that function with arguments from `0` to `10`, as well as Infinity and NaN, and print the returned values in hexadecimal, we get output like this:

```
macheps(  0) = 0x1.0000000000000p-1074
macheps(  1) = 0x1.0000000000000p-52
macheps(  2) = 0x1.0000000000000p-51
macheps(  3) = 0x1.8000000000000p-52
macheps(  4) = 0x1.0000000000000p-50
macheps(  5) = 0x1.4000000000000p-51
macheps(  6) = 0x1.8000000000000p-51
macheps(  7) = 0x1.c000000000000p-51
macheps(  8) = 0x1.0000000000000p-49
macheps(  9) = 0x1.2000000000000p-50
macheps( 10) = 0x1.4000000000000p-50
macheps(Inf) = infinity
macheps(NaN) = nan
```

Notice that our code in `macheps()` had to make a special test for a NaN to prevent an infinite loop.

We introduced the `volatile` variable *v* in the `else` block to force the loop test to be evaluated in storage precision, in order to foil compilers that use normally higher intermediate precision in expressions. That is a common problem in x86 and x86-64 platforms.

Our code made another test for a zero argument to avoid getting a zero return. The returned value in that case is the smallest representable floating-point number, here the subnormal value `2**(-1074)`, which is about `4.941e-324`.

No special test is needed for an Infinity, because the `eps`-reduction loop then terminates at the first iteration.

The code in `macheps()` illustrates some general rules for numerical programming with IEEE 754 arithmetic: first supply any special treatment needed for NaN, Infinity, and signed-zero arguments, and then deal with the general case for finite nonzero arguments. Well-written numerical code should therefore almost always have an outer `if` statement with one or more blocks to handle the special and general cases. Too often, programmers neglect to test for those special values, with the result that their code fails disastrously when presented with one of those special arguments, producing nonsensical results, or going into an infinite loop. Such coding errors can lie undetected for a long time. An important old rule of programming is that functions should always sanity-check their arguments, to avoid garbage-in/garbage-out misbehavior.

C99 introduces some related functions that can also be used to determine machine epsilons at run time:

```
#include <math.h>           /* include for these prototypes */

double      nextafter  (double x, double y);
float       nextafterf (float x, float y);
long double nextafterl (long double x, long double y);
```

They return the machine number nearest $x$ in the direction of $y$. For example, `nextafter(1.0, 2.0)` produces the same result as `1.0 + macheps(1.0)` and `1.0 + DBL_EPSILON`.

It is important to know that the machine epsilon is not symmetric about all numbers. At the boundaries where normalization changes the exponent, the epsilon below $x$ is smaller than that just above $x$ by a factor `1 / base`. For example, `macheps(1.0)` returns `+0x1p-52`, whereas `macheps(-1.0)` returns `+0x1p-53`. Some authors distinguish those cases by calling them the *positive* and *negative*, or *big* and *small*, machine epsilons. With the C99 functions, you could get them like this:

```
eps_neg = 1.0 - nextafter(1.0, -1.0);
eps_pos = nextafter(1.0, +2.0) - 1.0;
```

If the first argument is not an obvious constant, write the assignments like this:

```
eps_neg = x - nextafter(x, -inf());
eps_pos = nextafter(x, +inf()) - x;
```

If $x$ is Infinity, the function behaves according to this statement from the C99 Standard: *The* `nextafter` *functions return* $y$ *if* $x$ *equals* $y$. Our two assignments then produce `+0x1.fffffffffffffp+1023` (about 1.798e+308) for *eps_neg* and Infinity for *eps_pos*. Thus, the call `nextafter(INFINITY, -INFINITY)` can be used to find the largest representable finite number, and with the call `nextafter(0.0, 1.0)`, the smallest representable number (here, `0x1p-1074` (about 4.491e-324), a number that we saw before as the output from `macheps(0.0)`).

You can use those functions to find the number ranges for the seven IEEE 754 binary and decimal formats:

```
================================================================================================
base     size   min subnormal     min normal    max normal
================================================================================================
```

```
binary   32-bit   0x1p-149        0x1p-126      0x1.ffff_fep+127
         approx   1.401e-45       1.175e-38     3.403e+38

binary   64-bit   0x1p-1074       0x1p-1022     0x1.ffff_ffff_ffff_fp+1023
         approx   4.941e-324      2.225e-308    1.798e+308

binary   80-bit   0x1p-16445      0x1p-16382    0x1.ffff_ffff_ffff_fffep+16383
         approx   3.645e-4951     3.362e-4932   1.190e+4932

binary  128-bit   0x1p-16494      0x1p-16382    0x1.ffff_ffff_ffff_ffff_ffff_ffff_ffffp+16383
         approx   6.475e-4966     3.362e-4932   1.190e+4932

decimal  32-bit   1e-101           1e-95        9.999_999e+96

decimal  64-bit   1e-398           1e-383       9.999_999_999_999_999e+384

decimal 128-bit   1e-6176          1e-6143      9.999_999_999_999_999_999_999_999_999_999e+6144
================================================================================================
```

Notice that the decimal-format exponent ranges are significantly wider than their binary-format companions, and that the exponent ranges increase substantially as the significand size grows. As a rough rule of thumb, the range increases by a factor of 4 to 16 when the significand doubles in size.

## 1.17 Complex arithmetic

The Fortran language has had complex floating-point types since the 1960s, but C did not supply that capability until C99 became available. We do not describe here the numerous library functions that are available for complex types, but in general, most functions in the C math library for real arguments have counterparts for complex arguments that begin with the letter `c`: `cexp()`, `clog()`, `cpow()`, `csin()`, and so on.

Complex numbers in C99 are declared with the `complex` type modifier on a binary floating-point type (complex decimal types are not yet supported), and the imaginary unit from mathematics is available as the macro `I`. Unlike Fortran, which writes complex numbers as parenthesized comma-separated lists of real and imaginary parts, in C99, you write them as sums:

```
#include <complex.h>       /* needed for complex type definitions */

double x, y;
complex double z;

z = x + I*y;
z = x + y*I;               /* alternate style with same meaning */
```

If you prefer to use a constructor function or macro, do so like this:

```
#define CMPLX(x, y) ((x) + I * (y))

z = CMPLX(x, y);    /* constructor (like Fortran cmplx(x,y)) */
```

Even though you write complex numbers as a sum of real and imaginary parts, neither an addition nor a multiplication is needed to evaluate them: the compiler recognizes those forms, and simply stores the two parts in appropriate locations.

You can recover the real and imaginary parts with type-generic C99 macros like this:

```
x = creal(z);
y = cimag(z);
```

Just as in Fortran, complex numbers in C are stored as a two-element object containing adjacent real and imaginary parts. For cross-language communication, you can treat them like a two-element array. In C, they are actually implemented as a two-element `struct`.

What is important to discuss here are some issues that are unlikely to be obvious to programmers without extensive experience in both numerical computing, and in complex arithmetic in mathematics.

The first important point is that, unlike real arithmetic, in complex arithmetic, the danger of significance loss is *pervasive*, and affects *every one* of the basic operations, and *almost all* of the math-library functions. To understand why, recall the rules for complex multiplication and division:

```
   a = u + I*v              /* first operand */
   b = x + I*y              /* second operand */

prod = a * b
     = (u + I*v) * (x + I*y)
     = (u * x - v * y) + I*(v * x + u * y)

quo  = a / b
     = (u + I*v) / (x + I*y)
     = [(u + I*v) * (x - I*y)] / [(x + I*y) * (x - I*y)]
     = [(u * x + v * y) + I*(v * x - u * y)] / (x**2 + y**2)
```

There are four critical observations about those formulas:

- the multiplications on the right-hand side introduce the possibility of disastrous premature underflow or overflow;
- the products must be accurate to twice working precision;
- there is *always* one subtraction on the right-hand sides that is subject to catastrophic significance loss; and
- complex multiplication has up to *six* rounding errors, and complex division has *ten* rounding errors.

Because complex arithmetic is new in C99, one might expect library weaknesses in any or all of those areas, and numerical experiments across different platforms confirm that fear. The situation is even worse for the math-library functions for complex arithmetic, because most of them have many more places where accuracy can be compromised.

Another point that needs careful study is the fact that many functions in complex arithmetic have *branch cuts*. You can view a function with a complex argument, `f(z)`, as `f(x + I*y)`, and thus, it defines a relation between a point `(x, y)` on the complex plane with an elevation value on a surface. A branch cut looks like a tear in that surface, so approaching the cut from one side produces a particular value, and from the other side, a quite different value. Great care is needed to handle branch cuts properly, and even small numerical errors can push a result from one side to the other, radically changing the

returned value. As we reported earlier, correct handling of the sign of zero is critically important for computing near branch cuts.

The best advice that we can give to programmers who need complex arithmetic in C99 is to always use the *highest precision available*, and then to carefully check the results of test calculations to gauge the likely accuracy of the computed results. It is easy to supply test values of real and imaginary parts where all five basic operations in complex arithmetic, and almost all of the complex math functions, lose *all* significance, and fail to produce even a single correct digit.

Even though complex arithmetic makes some programming tasks easier, it may be numerically preferable to rework the algorithm so that it can be carried out in real arithmetic. That is commonly possible in matrix algebra.

## 1.18 More on decimal floating-point arithmetic

In the proposed extensions to the C language for decimal floating-point arithmetic, decimal constants are suffixed `df`, `dd`, and `dl` for the 32-bit, 64-bit, and 128-bit storage formats. Those suffixes may be in either lettercase.

Decimal library functions use the name of the corresponding `double` function for binary arithmetic, but with suffixes `df`, `d`, and `dl`. Notice the suffix differences between constants and functions in the 64-bit format, and that the decimal functions always have lowercase names.

The proposed extensions introduce new types called `_Decimal32`, `_Decimal64`, and `_Decimal128`. However, until they are standardized, it seems better to avoid those names in code by hiding them with suitable `typedef` statements. In the code samples below, we just attach the prefix `decimal_` to the binary type names.

We stated earlier (see [decimal-significand], page 7) that the significand in decimal floating-point arithmetic is an integer, rather than fractional, value. Decimal instructions do not normally alter the exponent by normalizing nonzero significands to remove trailing zeros. That design feature is intentional: it allows emulation of the fixed-point arithmetic that has commonly been used for financial computations.

One consequence of the lack of normalization is that there are multiple representations of any number that does not use all of the significand digits. Thus, in the 32-bit format, the values `1.DF`, `1.0DF`, `1.00DF`, ..., `1.000_000DF`, all have different bit patterns in storage, even though they compare equal. Thus, programmers need to be careful about trailing zero digits, because they appear in the results, and affect scaling. For example, `1.0DF * 1.0DF` evaluates to `1.00DF`, which is stored as `100 * 10**(-2)`.

In general, when you look at a decimal expression with fractional digits, you should mentally rewrite it in integer form with suitable powers of ten. Thus, a multiplication like `1.23 * 4.56` really means `123 * 10**(-2) * 456 * 10**(-2)`, which evaluates to `56088 * 10**(-4)`, and would be output as `5.6088`.

Another consequence of the decimal significand choice is that initializing decimal floating-point values to a pattern of all-bits-zero does not produce the expected value `0.`: instead, the result is the subnormal values `0.e-101`, `0.e-398`, and `0.e-6176` in the three storage formats.

The exponent in decimal arithmetic is called the *quantum*, and financial computations require that the quantum always be preserved. If it is not, then rounding may have happened, and additional scaling is required.

The function `samequantumd(x,y)` for 64-bit decimal arithmetic returns `1` if the arguments have the same exponent, and `0` otherwise.

The function `quantized(x,y)` returns a value of $x$ that has been adjusted to have the same quantum as $y$; that adjustment could require rounding of the significand. For example, `quantized(5.dd, 1.00dd)` returns the value `5.00dd`, which is stored as `500 * 10**(-2)`. As another example, a sales-tax computation might be carried out like this:

```
decimal_long_double amount, rate, total;

amount = 0.70DL;
rate   = 1.05DL;
total  = quantizedl(amount * rate, 1.00DL);
```

Without the call to `quantizedl`, the result would have been `0.7350`, instead of the correctly rounded `0.74`. That particular example was chosen because it illustrates yet another difference between decimal and binary arithmetic: in the latter, the factors each require an infinite number of bits, and their product, when converted to decimal, looks like `0.734_999_999...`. Thus, rounding the product in binary format to two decimal places always gets `0.73`, which is the *wrong* answer for tax laws.

In financial computations in decimal floating-point arithmetic, the `quantized` function family is expected to receive wide use whenever multiplication or division change the desired quantum of the result.

The function call `normalized(x)` returns a value that is numerically equal to $x$, but with trailing zeros trimmed. Here are some examples of its operation:

```
=====================================================
function call                          result
=====================================================
normalized(+0.00100DD)                 +0.001DD
normalized(+1.00DD)                    +1.DD
normalized(+1.E2DD)                    +1E+2DD
normalized(+100.DD)                    +1E+2DD
normalized(+100.00DD)                  +1E+2DD
normalized(+NaN(0x1234))               +NaN(0x1234)
normalized(-NaN(0x1234))               -NaN(0x1234)
normalized(+Infinity)                  +Infinity
normalized(-Infinity)                  -Infinity
=====================================================
```

The NaN examples show that payloads are preserved.

Because the `printf` and `scanf` families were designed long before IEEE 754 decimal arithmetic, their format items do not support distinguishing between numbers with identical values, but different quanta, and yet, that distinction is likely to be needed in output.

The solution adopted by one early library for decimal arithmetic is to provide a family of number-to-string conversion functions that preserve quantization. Here is a code fragment and its output that shows how they work.

```
decimal_float x;

x = 123.000DF;
(void)printf("%%He:    x = %He\n", x);
(void)printf("%%Hf:    x = %Hf\n", x);
(void)printf("%%Hg:    x = %Hg\n", x);
(void)printf("ntosdf(x) = %s\n", ntosdf(x));

%He:    x = 1.230000e+02
%Hf:    x = 123.000000
%Hg:    x = 123
ntosdf(x) = +123.000
```

The format modifier letter `H` indicates a 32-bit decimal value, and the modifiers `DD` and `DL` correspond to the two other formats.

## 1.19 Exact round-trip base conversion

Because most programmers, and programming languages, have to deal with at least two floating-point number bases, 10 for humans, and 2 (usually) for computers, base conversion occurs somewhere in almost every numerical program. Often it is hidden inside the compiler's processing of the source code, or in the input/output routines in the run-time library. You might expect then, that because the problem has been around for more than half a century, it must be well understood, and handled correctly. If so, you are almost certainly wrong!

In separate papers submitted about the same time in 1967–1968, Bennett Goldberg and David Matula (pronounced Ma-tóo-la) first showed that the base-conversion problem is more subtle than most people had believed (and many still believe). They answer this important question: *How many decimal digits do I need to print for binary floating-point numbers in the computer so that I can later read those numbers back in, and recover the original binary values exactly?* The naive formula that most people come up with using powers and logarithms is off by one or two digits, and that error has unfortunately been enshrined in default digit counts for floating-point output in several programming languages, including Fortran and C.

We do not go into the details just yet, but we report here a table of values relevant for IEEE 754 binary and decimal arithmetic:

```
================================================
   Digit counts for input/output base conversion
================================================
binary in       24      53      64      113     237
decimal out      9      17      21       36      73

decimal in       7      16       34      70
binary out      25      55      114     234
================================================
```

The table numbers can be computed from these two functions:

```
int
```

```
goldberg(int ndec)
{   /* return output bits needed for ndec-digits input */
    return ((int)ceil((double)ndec / log10(2.0) + 1.0));
}

int
matula(int nbits)
{   /* return output decimal digits needed for nbits-bits input */
    return ((int)ceil((double)nbits / log2(10.0) + 1.0));
}
```

One significant observation from those numbers is that we cannot achieve correct round-trip conversion between the decimal and binary formats in the same storage size! For example, we need 25 bits to represent a 7-digit value from the 32-bit decimal format, but the binary format only has 24 available. Similar observations hold for each of the other conversion pairs.

The general input/output base-conversion problem is astonishingly complicated, and solutions were not generally known until the publication of two papers in 1990 that are listed later near the end of this chapter. For the 128-bit formats, the worst case needs more than 11,500 decimal digits of precision to guarantee correct rounding in a binary-to-decimal conversion!

## 1.20 Further reading

The subject of floating-point arithmetic is much more complex than many programmers seem to think, and few books on programming languages spend much time in that area. In this chapter, we have tried to expose the reader to some of the key ideas, and to warn of easily overlooked pitfalls that can soon lead to nonsensical results. There are a few good references that we recommend for further reading, and for finding other important material about computer arithmetic. Entries are ordered by the family name of the first author, and then by year.

- Paul H. Abbott and 15 others, *Architecture and software support in IBM S/390 Parallel Enterprise Servers for IEEE Floating-Point arithmetic*, IBM Journal of Research and Development **43**(5/6) 723–760 (1999), `https://doi.org/10.1147/rd.435.0723`. This article has a good description of IBM's algorithm for exact decimal-to-binary conversion, complementing earlier ones by Clinger and others.
- Nelson H. F. Beebe, *The Mathematical-Function Computation Handbook: Programming Using the MathCW Portable Software Library*, Springer (2017), ISBN 3-319-64109-3 (hardcover), 3-319-64110-7 (e-book) (xxxvi + 1114 pages), `https://doi.org/10.1007/978-3-319-64110-2`. This book describes portable implementations of a large superset of the mathematical functions available in many programming languages, extended to a future 256-bit format (70 decimal digits), for both binary and decimal floating point. It includes a substantial portion of the functions described in the famous *NIST Handbook of Mathematical Functions*, Cambridge (2018), ISBN 0-521-19225-0. See `http://www.math.utah.edu/pub/mathcw` for compilers and libraries.
- William D. Clinger, *How to Read Floating Point Numbers Accurately*, ACM SIGPLAN

Notices **25**(6) 92–101 (June 1990), https://doi.org/10.1145/93548.93557. See also the papers by Steele & White.

- William D. Clinger, *Retrospective: How to read floating point numbers accurately*, ACM SIGPLAN Notices **39**(4) 360–371 (April 2004), http://doi.acm.org/10.1145/989393.989430. Reprint of 1990 paper, with additional commentary.
- I. Bennett Goldberg, *27 Bits Are Not Enough For 8-Digit Accuracy*, Communications of the ACM **10**(2) 105–106 (February 1967), http://doi.acm.org/10.1145/363067.363112. This paper, and its companions by David Matula, address the base-conversion problem, and show that the naive formulas are wrong by one or two digits.
- David Goldberg, *What Every Computer Scientist Should Know About Floating-Point Arithmetic*, ACM Computing Surveys **23**(1) 5–58 (March 1991), corrigendum **23**(3) 413 (September 1991), https://doi.org/10.1145/103162.103163. This paper has been widely distributed, and reissued in vendor programming-language documentation. It is well worth reading, and then rereading from time to time.
- Norbert Juffa and Nelson H. F. Beebe, *A Bibliography of Publications on Floating-Point Arithmetic*, http://www.math.utah.edu/pub/tex/bib/fparith.bib. This is the largest known bibliography of publications about floating-point, and also integer, arithmetic. It is actively maintained, and in early 2024, contains almost 7500 references to original research papers, reports, theses, books, and Web sites on the subject matter. It can be used to locate the latest research in the field, and the historical coverage dates back to 1703 and 1712 papers on binary arithmetic, a 1726 paper on signed-digit arithmetic, an 1837 paper by Charles Babbage (the intellectual father of mechanical computers), and an 1862 paper on hexadecimal arithmetic. The entries for the Abbott, Clinger, and Steele & White papers cited earlier contain pointers to several other important related papers on the base-conversion problem.
- William Kahan, *Branch Cuts for Complex Elementary Functions, or Much Ado About Nothing's Sign Bit*, (1987), http://people.freebsd.org/~das/kahan86branch.pdf. This Web document about the fine points of complex arithmetic also appears in the volume edited by A. Iserles and M. J. D. Powell, *The State of the Art in Numerical Analysis: Proceedings of the Joint IMA/SIAM Conference on the State of the Art in Numerical Analysis held at the University of Birmingham, 14–18 April 1986*, Oxford University Press (1987), ISBN 0-19-853614-3 (xiv + 719 pages). Its author is the famous chief architect of the IEEE 754 arithmetic system, and one of the world's greatest experts in the field of floating-point arithmetic. An entire generation of his students at the University of California, Berkeley, have gone on to careers in academic and industry, spreading the knowledge of how to do floating-point arithmetic right.
- Donald E. Knuth, *A Simple Program Whose Proof Isn't*, in *Beauty is our business: a birthday salute to Edsger W. Dijkstra*, W. H. J. Feijen, A. J. M. van Gasteren, D. Gries, and J. Misra (eds.), Springer (1990), ISBN 1-4612-8792-8, https://doi.org/10.1007/978-1-4612-4476-9. This book chapter supplies a correctness proof of the decimal to binary, and binary to decimal, conversions in fixed-point arithmetic in the TeX typesetting system. The proof evaded its author for a dozen years.
- David W. Matula, *In-and-out conversions*, Communications of the ACM **11**(1) 57–50 (January 1968), https://doi.org/10.1145/362851.362887.
- David W. Matula, *The Base Conversion Theorem*, Proceedings of the American Math-

ematical Society **19**(3) 716–723 (June 1968). See also other papers here by this author, and by I. Bennett Goldberg.

- David W. Matula, *A Formalization of Floating-Point Numeric Base Conversion*, IEEE Transactions on Computers **C-19**(8) 681–692 (August 1970), `https://doi.org/10.1109/T-C.1970.223017`.
- Jean-Michel Muller and eight others, *Handbook of Floating-Point Arithmetic*, Birkhäuser-Boston (2010), ISBN 0-8176-4704-X (xxiii + 572 pages), `https://doi.org/10.1007/978-0-8176-4704-9`. This is a comprehensive treatise from a French team who are among the world's greatest experts in floating-point arithmetic, and among the most prolific writers of research papers in that field. They have much to teach, and their book deserves a place on the shelves of every serious numerical programmer.
- Jean-Michel Muller and eight others, *Handbook of Floating-Point Arithmetic*, Second edition, Birkhäuser-Boston (2018), ISBN 3-319-76525-6 (xxv + 627 pages), `https://doi.org/10.1007/978-3-319-76526-6`. This is a new edition of the preceding entry.
- Michael Overton, *Numerical Computing with IEEE Floating Point Arithmetic, Including One Theorem, One Rule of Thumb, and One Hundred and One Exercises*, SIAM (2001), ISBN 0-89871-482-6 (xiv + 104 pages), `http://www.ec-securehost.com/SIAM/ot76.html`. This is a small volume that can be covered in a few hours.
- Guy L. Steele Jr. and Jon L. White, *How to Print Floating-Point Numbers Accurately*, ACM SIGPLAN Notices **25**(6) 112–126 (June 1990), `https://doi.org/10.1145/93548.93559`. See also the papers by Clinger.
- Guy L. Steele Jr. and Jon L. White, *Retrospective: How to Print Floating-Point Numbers Accurately*, ACM SIGPLAN Notices **39**(4) 372–389 (April 2004), `http://doi.acm.org/10.1145/989393.989431`. Reprint of 1990 paper, with additional commentary.
- Pat H. Sterbenz, *Floating Point Computation*, Prentice-Hall (1974), ISBN 0-13-322495-3 (xiv + 316 pages). This often-cited book provides solid coverage of what floating-point arithmetic was like *before* the introduction of IEEE 754 arithmetic.