Lessons in Scientific Computing:
Numerical Mathematics, Computer Technology, and Scientific Discovery
ByNorbert Schorghofer

CHAPTER 3

Roundoff & Number Representation

3.1 NUMBER REPRESENTATION

In a computer every real number is represented by a sequence of bits, often 64 bits (8 bytes). (1 byte is *always* 8 bits.) One bit is for the sign, and the distribution of bits for mantissa and exponent can be platform dependent. Almost universally, however, a 32-bit number will have 8 bits for the exponent and 23 bits for the mantissa (as illustrated in Figure 3.1). In the decimal system this corresponds to a maximum/minimum exponent of ± 38 and approximately 7 decimal digits. The relation between the number of binary bits and decimal accuracy can be worked out as follows. The 23 bits of the mantissa provide $2^{23} \approx 10^7$ different numbers, and therefore about 7 significant digits. The exponent can be negative or positive. Half of the 2^8 numbers can be used for the positive exponent: $2^{2^7} \approx 10^{38.5}$, so the largest number has a decimal exponent of $+38$. The 8 bits, $2^{2^8} \approx 10^{77.1}$, can represent decimal exponents from -38 to $+38$, which are 77 in number.

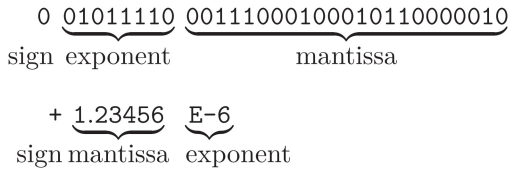

FIGURE 3.1 Typical representation of a real number with 32 bits.

The 7 significant decimal digits are the *typical* precision. There are number ranges where the binary and decimal representations mesh well, and ranges where they mesh poorly. The number of significant digits is at least 6 and at most 9. (This can be figured out with a brute force loop through all floating-point numbers; there are only about 4 billion of them.)

For a 64-bit number (8 bytes) there are 11 bits for the exponent (which translates to decimal exponents of ± 308) and 52 bits for the mantissa, which provides around 16 decimal digits of precision. And here is why: $2^{52} \approx 10^{16}$ and $2^{2^{10}} \approx 10^{+308}$.

Single-precision numbers are typically 4 bytes long. Use of double-precision variables doubles the length of the representation to 8 bytes. On some machines there is a way to extend beyond double, up to quadruple precision (typically 128 bit). Single and double precision variables do not necessarily correspond to the same byte-length on every machine.

The mathematical constant π up to 36 significant decimal digits (usually enough for quadruple precision) is

```
  ← single →
3.14159265  3589793  23846264338327950288
←——      double      ——→
```

Using double-precision numbers is usually barely slower than single-precision, if at all. Some processors always use their highest precision even for single-precision variables, and the extra step to convert between number representations makes single-precision calculations actually slower. Double-precision numbers do, however, take twice as much memory.

Several general-purpose math packages offer arbitrary-precision arithmetic, should this ever be needed. Computationally, arbitrary-precision calculations are disproportionally slow, because they have to be emulated on a software level, whereas single- and double-precision floating-point operations are hardwired into the computer's central processor.

Many fractions have infinitely many digits in decimal representation, e.g.,

$$\frac{1}{6} = 0.1666666\ldots$$

The same is true for binary numbers; only that the exactly represented fractions are fewer. The decimal number 0.5 can be represented exactly as `0.100000...`, but decimal 0.2 is in binary form

```
0.00110011001100110...
```

and hence not exactly representable with a finite number of digits. This causes a *truncation error*. In particular, decimals like 0.1 or 10^{-3} have an infinitely long binary representation. (The fact that binary cannot represent arbitrary decimal fractions exactly is a nuisance for accounting software, where everything needs to match to the cent.) For example, if a value of 9.5 is assigned it will be 9.5 exactly, but 9.1 carries a representation error. One can see this by using the following Python commands, which print numbers to 17 digits after the comma.

```
>>> print '%.17f' % 9.5
9.50000000000000000
>>> print '%.17f' % 9.1
9.09999999999999964
```

(The first `%` symbol indicates that the following is the output format. The second `%` symbol separates this from the number.) For the same reason, $0.1 + 0.1 + 0.1 - 0.3$ is not zero,

```
>>> 0.1+0.1+0.1-0.3
5.551115123125783e-17
```

A discrepancy of this order of magnitude, $\approx 10^{-16}$, incidentally makes clear that by default Python, or at least the implementation used here, represents floating-point numbers as 8-byte variables. Any if condition for a floating-point number needs to include a tolerance, e.g., not `if (a==0.8)`, but `if (abs(a-0.8)<1e-12)`, where 10^{-12} is an empirical choice that got to be safely above a (relative) accuracy of 10^{-16}, for whatever range `a` takes on in this program.

In terms of more formal mathematics, in the language of algebraic structures, the set of floating-point numbers ($\mathbb{F}$) does not obey the same rules as the set of real numbers ($\mathbb{R}$). For example, $(a + b) + c$ may be different from $a+(b+c)$. The associative property, that the order in which operations are performed does not matter, does not necessarily hold for floating-point numbers. For example,

```
>>> (1.2-1)-0.2
-5.551115123125783e-17
>>> 1.2-(1+0.2)
0.0
```

Some programming languages assume that, unless otherwise instructed, all numbers are double-precision numbers. Others look at the decimal point, so that 5 is an integer, but 5. is a floating-point number. Often that makes no difference in a calculation at all, but sometimes it makes a crucial difference. For example, for an integer division 4/5=0, whereas for a floating-point division 4./5.=0.8. This is one of those notorious situations where the absence of a single period can introduce a fatal bug in a program. Even 4./5 and 4/5. will yield 0.8. For this reason some programmers categorically add periods after integers that are meant to be treated as floating-point numbers, even if they are redundant; that habit helps to avoid this error.

When a number is represented with a fixed number of bits, there is necessarily a maximum and minimum representable number; exceeding them means an "overflow" or "underflow." This applies to floating-point numbers as well as to integers. For floating-point numbers we have already determined these limits. Currently the most common integer length is 4 bytes. Since 1 byte is 8 bits, that provides $2^{4\times 8} = 2^{32} \approx 4 \times 10^9$ different integers. The C language and numerous other languages also have *unsigned* integers, so all bits can be used for positive integers, whereas the regular 4-byte integer goes from about -2×10^9 to about $+2 \times 10^9$. It is prudent not to use loop counters that go beyond 2×10^9.

Some very high-level programming languages, such as Python and Mathematica, use *variable* byte lengths for integers, in contrast to the fixed byte length representations we have discussed so far. If we type `10**1000` at a Python prompt, which would overflow a 4-byte integer, and even overflow the maximum exponent of an 8-byte floating-point, then it automatically switches to a longer integer representation.

The output of `10**1000` in Python 2 will be appended with the letter `L`, which indicates a "long" integer, that would not fit within an integer of regular byte length. Similarly, floating-point numbers are sometimes written as `1e6` or `1d6` to distinguish single-precision from double-precision numbers. The numbers `1e6` and `1d6` are exactly the same, but `1e-6` and `1d-6` are not, because of truncation error. For example, in Fortran the assignment `a=0.1` is less accurate than `a=0.1d0`, even when `a` is declared as a double-precision variable in both cases. Letters that indicate the variable type within a number, such as `e`, `d`, and `L`, are known as "literals". Only some languages use them.

3.2 IEEE STANDARDIZATION

The computer arithmetic of floating-point numbers is defined by the IEEE 754 standard (originally 754-1985, then revised by the 854-1987 and 754-2008). It standardizes number representation, roundoff behavior, and exception handling. All three components are will be described in this chapter.

TABLE 3.1 Specifications for number representation according to the IEEE 754 standard.

	single	double
bytes	4	8
bits for mantissa	23	52
bits for exponent	8	11
significant decimals	6–9	15–17
maximum finite	3.4E38	1.8E308
minimum normal	1.2E-38	2.2E-308
minimum subnormal	1.4E-45	4.9E-324

Table 3.1 summarizes the IEEE standardized number representations, partly repeating what is described above. What matters for the user are the number of significant decimals, and the maximum and minimum representable exponent. When the smallest (most negative) exponent is reached, the mantissa can be gradually filled with zeros, allowing for even smaller numbers to be represented, albeit at less precision. Underflow is hence gradual. These numbers are referred to as "subnormals" in Table 3.1.

As a curiosity, $\tan(\pi/2)$ does not overflow with standard IEEE 754 numbers, neither in single nor double precision. This is straightforward to demonstrate. If π is not already defined intrinsically, assign it with enough digits,

given above, divide by two, and take the tangent; the result will be finite. Or, if π is already defined, type

```
>>> tan(pi/2)
1.633123935319537e+16
```

In fact the tangent does not overflow for any argument.

As part of the IEEE 754 standard, a few bit patterns have special meaning and serve as "exceptions". There is a bit pattern for numbers exceeding the maximum representable number: `Inf` (infinity). There are also bit patterns for `-Inf` and `NaN` (not a number). For example, 1./0. will produce `Inf`. An overflow is also an `Inf`. There is a positive and a negative zero. If a zero is produced as an underflow of a tiny negative number it will be $-0.$, and $1./(-0.)$ produces `-Inf`. A `NaN` is produced by expressions like $0./0.$, $\sqrt{-2.}$, or `Inf-Inf`. Exceptions are intended to propagate through the calculation, without need for any exceptional control, and can turn into well-defined results in subsequent operations, as in `1./Inf` or in `if (2.<Inf)`. If a program aborts due to exceptions in floating-point arithmetic, which can be a nuisance, it does not comply with the standard. IEEE 754 floating-point arithmetic is algebraically complete; every algebraic operation produces a well-defined result.

Roundoff under the IEEE 754 standard is as good as it can be for a given precision. The standard requires that the result must be as if it was first computed with infinite precision, and then rounded. This is a terrific accomplishment; we get numbers that are literally accurate to the last bit. The error never exceeds half the gap of the two machine-representable numbers closest to the exact result. (There are actually several available rounding modes, but, for good reasons, this is the default rounding mode.) This applies to the elementary operations ($+$, $-$, $/$, $\times$) as well as to the remainder (in many programming languages denoted by %) and the square root $\sqrt{}$. Halfway cases are rounded to the nearest even (`0` at the end) binary number, rather than always up or always down, because rounding in the same direction would be more likely to introduce a statistical bias, as minuscule as it may be.

The IEEE 754 standard for floating-point arithmetic represents the as-good-as-it-can-be case. But to what extent are programming platforms compliant with the standard? The number representation, that is, the partitioning of the bits, is nowadays essentially universally implemented on platforms one would use for scientific computing. That means for an 8-byte number, relative accuracy is about 10^{-16} and the maximum exponent is $+308$ nearly always. Roundoff behavior and exception handling are often available as options, because obeying the standard rigorously comes with a penalty on speed. Compilers for most languages provide the option to enable or disable the roundoff and exception behavior of this IEEE standard. Certainly for C and Fortran, ideal rounding and rigorous handling of exceptions can be enforced on most machines. Many general-purpose computing environments also comply with the IEEE 754 standard. Pure Python stops upon division by zero—which is a violation of the standard, but the NumPy module is IEEE compliant.

Using exactly representable numbers allows us to do calculations that incur no roundoff at all, at least when IEEE 754 is enabled. Of course every integer, even when defined as a floating-point number, is exactly representable. For example, addition of 1 or multiplication by 2 does not incur any roundoff at all. Normalizing a number to avoid an overflow is better done by dividing by a power of 2 than by a power of 10, due to truncation error. Factorials can be calculated, without loss of precision, using floating-point numbers; when the result is smaller than $\sim 2 \times 10^9$ it will provide the exact same answer as a calculation with integers would have, and if it is larger it will be imprecise, but at least it will not overflow as integers would.

NaNs as floating point numbers can be tricky. For example,

```
if (x >= 0) { printf("x is positive or zero\n"); }
   else { printf("x is negative\n"); }
```

would identify x=NaN as negative, although it is not. It is more robust to replace the `else` statement with `{if (x<0) printf("x is negative\n"); }`. The result of a comparison can only be true or false (a boolean/logical variable cannot have a NaN value), and all comparisons with NaN return false. Whereas the result of NaNs in elementary operations is clear (if one argument is NaN, then the result is NaN), its impact upon other functions can be ambiguous. The meaning of NaN (Not a Number or Not any Number) can be twofold. One is as a number of unknown value; the other is as a missing value. The result of `max(1,2,NaN)` should be `NaN` in the former case and 2 in the latter.

To summarize part of the IEEE 754 number representation standard: there are three special values: NaN, +Inf, and −Inf, and two zeros, +0 and −0. It has signed infinities and signed zeros. It allows for five exceptions: invalid operation (produces NaN), overflow (produces +/-Inf), underflow (produces +/-0), division by zero (produces +/−Inf), and inexact. The last of these exceptions, which had not been mentioned yet, tells us whether any rounding has occurred at all, a case so common that no further attention is usually paid to it. IEEE 754 applies to floating point numbers, and not to other types of variable; so if an integer overflows, we may really be in trouble.

The rigor with which roundoff is treated is wonderfully illustrated with the following example: The numerical example of a chaotic iteration in chapter 1 is extremely sensitive to the initial condition and subsequent roundoff. Nevertheless, these numbers, even after one thousand iterations, can be reproduced *exactly* on a different computer and with a different programming language (Exercise 3.1). Of course, given the sensitivity to the initial value, the result is quantitatively incorrect on all computers; after many iterations it is entirely different from a calculation using infinitely many digits.

3.3 ROUNDOFF SENSITIVITY

Using the rules of error propagation, or common sense, we recognize situations that are sensitive to roundoff. If x and y are real numbers of the same sign, their sum $x + y$ has an absolute error that adds the two individual absolute errors, and the relative error is at most as large as the relative error of x or y. Hence, adding them is insensitive to roundoff. On the other hand, $x - y$ has increased relative error (Exercise 2.4). The relative error of a product of two numbers is the sum of the relative errors of its factors, so multiplication is also not roundoff sensitive. The relative error of a ratio of two numbers is also the sum of relative errors, so division is not sensitive to roundoff either. To go through at least one of these error propagations:

$$\frac{x}{y}(1+\epsilon_{x/y}) = \frac{x(1+\epsilon_x)}{y(1+\epsilon_y)} \approx \frac{x}{y}(1+\epsilon_x)(1-\epsilon_y) \approx \frac{x}{y}(1+\epsilon_x-\epsilon_y)$$

and therefore the relative error of the ratio is $|\epsilon_{x/y}| \leq |\epsilon_x| + |\epsilon_y|$. For divisions, we only need to worry about overflows or underflows, in particular division by zero. Among the four elementary operations only subtraction of numbers of equal sign or addition of numbers of opposite sign increase the relative error.

An instructive example is solving a quadratic equation $ax^2 + bx + c = 0$ numerically. In the familiar solution formula

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

a cancellation effect will occur for one of the two solutions if ac is small compared to b^2. The remedy is to compute the smaller root from the larger. For a quadratic polynomial the product of its two roots equals $x_1 x_2 = c/a$, because $ax^2 + bx + c = a(x - x_1)(x - x_2)$. If b is positive then one solution is obtained by the equation above, $x_1 = -q/(2a)$, with $q = b + \sqrt{b^2 - 4ac}$, but the other solution is obtained as $x_2 = c/(ax_1) = -2c/q$. This implementation of the solution of quadratic equations requires no extra line of code; the common term q could be calculated only once and stored in a temporary variable, and the sign of b can be accommodated by using the sign function $\mathrm{sgn}(b)$, $q = b + \mathrm{sgn}(b)\sqrt{b^2 - 4ac}$. To perfect it, a factor of $-1/2$ can be absorbed into q to save a couple of floating-point operations.

```
!straight-forward version  ! roundoff-robust version
d = sqrt(b**2-4*a*c)       q=-(b+sgn(b)*sqrt(b**2-4*a*c))/2
x1 = (-b + d)/(2*a)        x1 = q/a
x2 = (-b - d)/(2*a)        x2 = c/q
```

We usually do not need to bother writing an additional line to check whether a is zero. If a division by zero does occur, a modern computer will either complain or it is properly taken care of by the IEEE standard, which would produce an `Inf` and continue with the calculation in a consistent way.

Sometimes an expression can be recast to avoid cancellations that lead to

increased sensitivity to roundoff. For example, $\sqrt{1+x^2}-1$ leads to cancellations when x is close to zero, but the equivalent expression $x^2/(\sqrt{1+x^2}+1)$ has no such problem. A basic example of an alternating series whose cancellation error can be avoided is

$$1-\frac{1}{2}+\frac{1}{3}-\frac{1}{4}+-...=\frac{1}{1\cdot 2}+\frac{1}{3\cdot 4}+...$$

The version to the left involves cancellations, the one to the right does not. There is no need to evaluate this infinite series numerically—it sums up to $\ln 2$ —, but it illustrates the concept.

An example of unavoidable cancellations is finite-difference formulae, like $f(x+h)-f(x)$, where the value of a function at point x is subtracted from the value of a function at a nearby point $x+h$. (An illustration of the combined effect of discretization and roundoff errors in a finite-difference expression will be given in Figure 6.1.)

A potential problem also arises for acos(x) and asin(x), where the argument has to be between -1 and $+1$. If x is slightly above 1 due to roundoff, it will lead to a NaN. An extra line may be necessary to prevent that. (It happens so that IEEE 754 guarantees that $-1 \le x/\sqrt{x^2+y^2} \le 1$, unless the denominator underflows.)

We conclude this section by applying our knowledge of roundoff to the problem of calculating the distance on a sphere. The great circle distance $\Delta\sigma$ on a sphere is

$$\cos(\Delta\sigma)=\sin\phi_1\sin\phi_2+\cos\phi_1\cos\phi_2\cos(\Delta\lambda)$$

which follows from one or the other law of spherical trigonometry. Here, $\Delta\sigma$ is the angular distance along a great circle, ϕ_1 and ϕ_2 are the latitudes of the two points, and $\Delta\lambda=\lambda_2-\lambda_1$ is the difference in longitudes. This expression is prone to rounding errors when the distance is small. This is clear from the left-hand side alone. For $\Delta\sigma \approx 0$, the cosine is almost 1. Since the Taylor expansion of the cosine begins with $1-\Delta\sigma^2/2$, the result will be dominated by the truncation error at $\Delta\sigma^2/2 \approx 10^{-16}$ for 8-byte floating-point numbers. The volumetric mean radius of the Earth is 6371 km, so the absolute distance comes out to $2\pi \times 6371 \times 10^3 \times \sqrt{2\times 10^{-16}} \approx 0.6$ m. At distances much larger than that, the formula given above does not have significant rounding errors. For small distances we could use the Pythagorean relation $\Delta\sigma=\sqrt{\Delta\phi^2+\Delta\lambda^2\cos^2\phi}$. The disadvantage of this approach is that there will be a small discontinuity when switching between the roundoff-sensitive exact equation and the roundoff-insensitive approximate equation. Alternatively the equation can be reformulated as

$$\sin^2\left(\frac{\Delta\sigma}{2}\right)=\sin^2\left(\frac{\Delta\phi}{2}\right)+\cos\phi_1\cos\phi_2\sin^2\left(\frac{\Delta\lambda}{2}\right)$$

This no longer has a problem for nearby points, because $\sin(\Delta\sigma/2) \approx \Delta\sigma/2$, so

the left-hand side is small for small $\Delta\sigma$, and there is no cancellation between the two terms on the right-hand side, because they are both positive. This expression becomes roundoff sensitive when the sine is nearly 1 though, that is, for nearly antipodal points. Someone has found a formula that is roundoff insensitive in neither situation, which involves more terms. The lesson from this example is that *if* 16 digits are not enough, the problem can often be fixed mathematically.

Interval arithmetic. Generally, we want rounding to the nearest number, but other rounding modes are available: round always up or always down. An advanced technique called "interval arithmetic" takes advantage of these directed roundings. Every result is represented not by one value of unknown accuracy, but by two that straddle the exact result. An upper and a lower bound are determined at every step of the calculation. Although the interval may vastly overestimate the actual uncertainty, it provides mathematically rigorous bounds. Interval arithmetic can sometimes turn a numerical calculation into a mathematically rigorous result.

Recommended Reading: The "father" of the IEEE 754 standard, William Kahan, posts roundoff-related notes online at http://people.eecs.berkeley.edu/~wkahan/. The links include a plain-language description of the standard, http://people.eecs.berkeley.edu/~wkahan/ieee754status/IEEE754.PDF. A technical summary is provided by David Goldberg, *What every computer scientist should know about floating point arithmetic.* The document is readily available online, for example at https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html.

EXERCISES

3.1 The iteration $x_{n+1} = 4x_n(1 - x_n)$ is extremely sensitive to the initial value as well as to roundoff. Yet, thanks to the IEEE 754 standard, it is possible to reproduce the exact same sequence on many platforms. Calculate the first 1010 iterations with initial conditions $x_0 = 0.8$ and $x_0 = 0.8 + 10^{-15}$. Use double (8-byte) precision. Submit the program as well as the values $x_{1000} \ldots x_{1009}$ to 9 digits after the decimal point. We will compare the results in class.

3.2 The expression for gravitational acceleration is GMr_i/r^3, where G is the gravitational constant 6.67×10^{-11} m^3/kg s, M is the mass of the sun 2×10^{30} kg, $r = 150 \times 10^9$ m is the distance between the Sun and Earth, and $i = 1, 2, 3$ indexes the three spatial directions. We cannot anticipate in which order the computer will carry out these floating-point operations. What are the worst possible minimum and maximum

exponents of an intermediate result? Could 4-byte IEEE or 8-byte IEEE floating-point numbers overflow or underflow?

3.3 *hypot function:* The hypotenuse of a triangle is given by

$$c = \sqrt{x^2 + y^2}$$

A problem with this expression is that $x^2 + y^2$ might overflow or underflow, even when c does not. Find a way to calculate c that circumvents this problem.

3.4 The following formulae may incur large roundoff errors: a) $x - \sqrt{x}$, and b) $\cos^2 x - \sin^2 x$. Identify values of x for which they are sensitive to roundoff, and suggest an alternative formula for each which avoids this problem.