ber of places the decimal point has been moved: 2^4, whose representation in excess–127 is $4 + 127 = 131 = 1000\,0011_2$;

(4) write the bit of the sign (0), then the first n_e bits representing the exponent of 2 followed by the fractional part of the mantissa. For $n_e = 8$ and $n_m = 23$ we have: 0 1000 0011 011 0110 0000 0000 0000 0000.

According to the IEEE 754 Standard rational numbers can be represented in *double precision*. In this way they follow the convention described above, using 64 bits, of which 11 are allocated to the exponent. In 2008, the *quadruple precision* representation was defined, as part of the IEEE 754 2008 Standard, using 15 bits for the exponent and 112 bits for the mantissa, for a total length of 128 bits.

Table 1.3.4 shows the minimum and maximum values of numbers representable in the memory of a 32 bit computer.

Number classes	*Minimum*	*Maximum*
integer		
unsigned	0	4294 967 295
signed	−2147 483 647	2147 483 647
two's complement	−2147 483 648	2147 483 647
rational (in modulus)		
in single precision	$1.401\,298 \times 10^{-45}$	$3.402\,823 \times 10^{38}$
in double precision	$4.940\,656 \times 10^{-324}$	$1.797\,693 \times 10^{308}$

1.4 The approximation problem

Since the available number of digits is limited, a computer can only represent those numbers whose fractional part can be expressed as a limited sum of powers of 2 exactly. If this is not the case, it is approximated by the nearest one. While 22.75 can be represented exactly, other cases, such as 0.1, cannot. To ascertain yourself of this try to write the latter number in binary notation considering a 32 bit computer.

The same occurs for non–rational, real numbers (such as $\sqrt{2}$, π, etc.): these numbers have an infinite amount of digits after the decimal point, for which there is not enough space in the computer memory. They can only be approximated by the closest rational number.

This limitation causes a problem which is to be faced each time a computer performs a computation: the rounding error. In practice, all non integer numbers are approximated by the nearest rational number having a finite representation, with a precision of the order of 2^{-n_m}, where n_m is the number of bits reserved for the mantissa. Two numbers that differ less than this quantity from each other are considered to be equal on the computer and a number smaller than the smallest representable number equals 0 (this is the case of an *underflow* error).

Care should be taken with this type of approximation. Though it is harmless in many cases, the error could propagate dramatically in some algorithms, especially the iterative ones, and may become important (a concrete example is given in Chapter 3.7). A frequent cause of such a catastrophic error are either very different or very similar operands. In the first case, the smaller of the two numbers is less precise because the mantissa needs to be represented with a larger exponent. Consider adding two numbers $a = 68\,833\,152$, i.e., 0 1001 1001 000 0011 0100 1001 1111 0000 in the IEEE 754 representation, and $b = 2309\,657\,318\,129\,664$ (0 1011 0010 000 0011 0100 1001 1111 0000). To carry out this addition, we need to organize these numbers in columns. To this purpose, we need to express the smallest of the two (a) with the same exponent of 2 as the biggest (b) in IEEE 754 notation. The exponent of a is 26, while b's is 51. The difference between these two numbers is 25. Thus, we need to rewrite the mantissa of a moving the digits 25 places to the right. As the maximum number of digits of the mantissa is 23, the resulting mantissa of a is represented by a sequence of zeros:

$$\begin{array}{l}
0\ \ 1011\,0010\ \ 000\,0000\,0000\,0000\,0000\,0000 + \ldots \\
\underline{0\ \ 1011\,0010\ \ 000\,0011\,0100\,1001\,1111\,0000} = \\
0\ \ 1011\,0010\ \ 000\,0011\,0100\,1001\,1111\,0000
\end{array}$$

and the sum is $a + b = b$, which obviously is nonsense!

The second type of error often occurs when subtracting two numbers that are close to each other. This is easily understood by considering some examples in the more familiar base 10 (though similar examples can be found in any other base). Suppose we represent floating–point numbers with 3 significant digits after the decimal point. The number 1000 is represented as $a = 1.000 \times 10^3$, while the number 999.8 as $b = 9.998 \times 10^2$.

The difference is $(a - b) = 0.2$. However, to perform this calculation in floating–point representation, the smallest number first needs to be expressed with the same power of 10 as the one of the higher number. This causes a loss of significant digits: $b \to b' = 0.999 \times 10^3$. Consequently, the difference becomes

$$(1.000 - 0.999) \times 10^3 = 10^{-3} \times 10^3 = 1.0\,.$$

Though the precision of each single number is of order 10^{-3}, the approximate value is larger by a factor 5 with respect to the true value! In some cases, this type of error can be avoided (or at least reduced) by reformulating the expression which is to be calculated. Suppose, $x = 3.451 \times 10^0$ and $y = 3.45 \times 10^0$ are given, and we need to calculate $\Delta = \left(x^2 - y^2\right) = 0.006\,901 = 6.901 \times 10^{-3}$. If we first calculate the squares $x^2 = 11.909\,401 = 1.190 \times 10^1$ and $y^2 = 11.9025 = 1.190 \times 10^1$, we find $\Delta = 0$. A disastrous result, especially if Δ is the denominator in some expression. Instead, by reformulating the expression as $\Delta = \left(x^2 - y^2\right) = (x - y)\,(x + y)$, we find $(x - y) = (3.451 - 3.450) \times 10^0 = 0.001 \times 10^0 = 1.000 \times 10^{-3}$, while $(x + y) = (3.451 + 3.450) \times 10^0 = 6.901 \times 10^0$, and thus, $\Delta = 1.000 \times 10^{-3} \times 6.901 \times 10^0 = 6.901 \times 10^{-3}$.

In numerical computation, unlike analytical calculus, the order in which operations are executed is extremely important, as is the order of magnitude of the terms subject to arithmetic operations. Depending on these operations, they should be neither too different from each other, nor be too similar.

1.5 Non-numbers on computers

Performing a task, whether it scientific or not, may require manipulating data other than numbers, such as images, sounds, etc. Today this might seem obvious because we can see images or watch a movie on a computer all the time; we can listen to music by means of the loudspeakers; a microphone allows us to use our voice to communicate with users far away through the Internet; we can use a video camera to organize a video conference.

When the computer era began, in the Forties of last century, their only purpose was to perform calculations. Numbers represented practically the only form of information available. Programs were built by operating switches. Their results were visualized by light bulbs whose status (on/off) indicated the bit sequence in binary. When it became possible to produce written results (on paper, and later, on screen) and to give commands by means of a keyboard, the problem of how to represent other types of information, such as characters, needed to be solved. As more refined techniques