

Chapter 2

Numerical Representation

2.1 Introduction

The numerical system used in the Western World today is a place-value base-10 system inherited from India through the intermediary of Arabic trade; this is why the numbers are often called Arabic numerals or more correctly Indo-Arabic numerals. However, this is not the only numerical system possible. For centuries, the Western World used the Roman system instead, which is a base-10 additive-value system (digits of a number are summed and subtracted from each other to get the value represented), and that system is still in use today, notably in names and titles. Other civilizations experimented with other bases: some precolonial Australian cultures used a base-5 system, while base-20 systems arose independently in Africa and in pre-Columbian America, and the ancient Babylonians used a base-60 counting system. Even today, despite the prevalence of the base-10 system, systems in other bases continue to be used every day: degrees, minutes, and seconds are counted in the base-60 system inherited from Babylonian astrologers, and base-12 is used to count hours in the day and months (or zodiacs) in the year.

When it comes to working with computers, it is easiest to handle a base-2 system with only two digits, 0 and 1. The main advantage is that this two-value system can be efficiently represented by an open or closed electrical circuit that measures 0 or 5 V, or in computer memory by an empty or charged capacitor, or in secondary storage by an unmagnetized or magnetized area of a metal disc or an absorptive or refractive portion of an optical disc. This base-2 system is called *binary*, and a single *binary digit* is called a *bit*.

It should come as no surprise, however, that trying to model our infinite and continuous real world using a computer which has finite digital storage, memory, and processing capabilities will lead to the introduction of errors in our modelling. These errors will be part of all computer results; no amount of technological advancement or upgrading to the latest hardware will allow us to overcome them. Nonetheless, no one would advocate for engineers to give up computers altogether!

It is only necessary to be aware of the errors that arise from using computers and to account for them.

This chapter will look in details at how binary mathematics work and how computers represent and handle numbers. It will then present the weaknesses that result from this representation and that, if ignored, can compromise the quality of engineering work.

2.2 Decimal and Binary Numbers

This section introduces binary numbers and arithmetic. Since we assume the reader to be intimately familiar with decimal (base-10) numbers and arithmetic, we will use that system as a bridge to binary.

2.2.1 Decimal Numbers

Our decimal system uses ten ordered digits 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9 to represent any number as a sequence:

$$d_n d_{n-1} d_{n-2} \dots d_1 d_0 . d_{-1} d_{-2} \dots \quad (2.1)$$

where every d_i is a digit, n is an integer, and $d_n \neq 0$. The sequence is place valued, meaning that the value of d_i in the complete number is the value of its digit multiplied by 10 to the power i . This means the value of the complete number can be computed as

$$\sum_{i=-\infty}^n d_i \times 10^i \quad (2.2)$$

The digits d_0 and d_{-1} , the first digit multiplied by 10 to a negative power, are separated by a point called the decimal point. The digit d_n , which has the greatest value in the total number, is called the *most significant digit*, while the digit d_i with the lowest value of i and therefore the lowest contribution in the total number is called the *least significant digit*.

It is often inconvenient to write numbers in the form of Eq. (2.1), especially when modelling very large or very small quantities. For example, the distance from the Earth to the Sun is 150,000,000,000 m, and the radius of an electron is 0.0000000000000028 m. For this reason, numbers are often represented in *scientific notation*, where the non-zero part is kept, normally with one digit left of the decimal point and a maximum of m digits on the right, and the long string of zeroes is simplified using a multiplication by a power of 10. The number can then be written as

$$d_0.d_{-1}d_{-2}\dots d_{-m}\times 10^n \quad (2.3)$$

or equivalently but more commonly as

$$d_0.d_{-1}d_{-2}\dots d_{-m}en \quad (2.4)$$

The $m+1$ digits that are kept are called the *mantissa*, the value n is called the *exponent*, and the letter “e” in Eq. (2.4) stands for the word “exponent”, and normally $m < n$. Using scientific notation, the distance from the Earth to the Sun is 1.5×10^{11} m and the radius of the electron is 2.8×10^{-15} .

We can now define our two basic arithmetic operations. The rule to perform the addition of two decimal numbers written in the form of Eq. (2.1) is to line up the decimal points and add the digits at corresponding positions. If a digit is missing from a position in one of the numbers, it is assumed to be zero. If two digits sum to more than 9, the least significant digit is kept in that position and the most significant digit carries and is added to the digits on the left. An addition of two numbers in scientific notations is done first by writing the two numbers at the same exponent value, then adding the two mantissas in the same way as before. To multiply two decimal numbers written in the form of Eq. (2.1), multiply the first number by each digit d_i of the second number and multiply that partial result by 10^i , then sum the partial results together to get the total. Given two numbers in scientific notation, multiply the two mantissas together using the same method, and add the two exponents together.

2.2.2 Binary Numbers

A binary system uses only two ordered digits (or bits, short for “binary digits”), 0 and 1, to represent any number as a sequence:

$$b_nb_{n-1}b_{n-2}\dots b_1b_0.b_{-1}b_{-2}\dots \quad (2.5)$$

where every b_i is a bit, n is an integer, and $b_n \neq 0$. The sequence is place valued, meaning that the value of b_i in the complete number is the value of its digit multiplied by 2 to the power i . This means the value of the complete number can be computed as

$$\sum_{i=-\infty}^n b_i \times 2^i \quad (2.6)$$

The digits b_0 and b_{-1} , the first digit multiplied by 2 to a negative power, are separated by a point; however, it would be wrong to call it a decimal point now since this is not a decimal system. In binary it is called the *binary point*, and a more

general term for it independent of base is the *radix point*. We can define a binary scientific notation as well, as

$$b_0.b_{-1}b_{-2}\dots b_{-m} \times 2^n \quad (2.7)$$

$$b_0.b_{-1}b_{-2}\dots b_{-m}e_n \quad (2.8)$$

The readers can thus see clear parallels with Eqs. (2.1)–(2.4) which define our decimal system. Likewise, the rules for addition and multiplication in binary are the same as in decimal, except that digits carry whenever two 1s are summed.

Since binary and decimal use the same digits 1 and 0, it can lead to ambiguity as to whether a given number is written in base 2 or base 10. When this distinction is not clear from the context, it is habitual to suffix the numbers with a subscript of their base. For example, the number 110 is ambiguous, but 110_{10} is one hundred and ten in decimal, while 110_2 is a binary number representing the number 6 in decimal. It is not necessary to write that last value as 6_{10} since 6_2 is nonsensical and no ambiguity can exist.

Example 2.1

Compute the addition and multiplication of 3.25 and 18.7 in decimal and of 1011.1 and 1.1101 in binary.

Solution

Following the first addition rule, line up the two numbers and sum the digits as follows:

$$\begin{array}{r} 3.25 \\ +18.7 \\ \hline 21.95 \end{array}$$

The second addition rule requires writing the numbers in scientific notation with the same exponent. These two numbers in scientific notations are 3.25×10^0 and 1.87×10^1 , respectively. Writing them in the same exponent would change the first one to 0.325×10^1 . Then the sum of the mantissa gives

$$\begin{array}{r} 0.325 \\ +1.87 \\ \hline 2.195 \end{array}$$

for a final total of 2.195×10^1 , the same result as before.

(continued)

Example 2.1 (continued)

The multiplication of two numbers in decimal using the first rule is done by summing the result of partial multiplications as follows:

$$\begin{array}{r}
 3.25 \\
 \times 18.7 \\
 \hline
 2.275 \\
 26 \\
 + 32.5 \\
 \hline
 60.775
 \end{array}$$

To perform the operation using the second rule, write the numbers in scientific notation as 3.25×10^0 and 1.87×10^1 , respectively, then multiply the mantissas as before:

$$\begin{array}{r}
 3.25 \\
 \times 1.87 \\
 \hline
 0.2275 \\
 2.6 \\
 + 3.25 \\
 \hline
 6.0775
 \end{array}$$

and sum the exponents ($0 + 1$) to get a final result of 6.0775×10^1 as before.

Working in binary, the rules apply in the same way. The binary sum, using the first rule, gives

$$\begin{array}{r}
 1011.1 \\
 + 1.1101 \\
 \hline
 1101.0101
 \end{array}$$

Writing the numbers in scientific notation gives 1.0111×2^3 and 1.1101×2^0 , respectively. Using the second rule, the multiplication of the mantissas gives

$$\begin{array}{r}
 1.0111 \\
 \times 1.1101 \\
 \hline
 0.00010111 \\
 0. \\
 0.010111 \\
 0.10111 \\
 + 1.0111 \\
 \hline
 10.10011011
 \end{array}$$

and the sum of the exponents gives 3, for a total of 10.10011011×2^3 , better written as 1.010011011×2^4 .

2.2.3 Base Conversions

The conversion from binary to decimal can be done simply by computing the summation from Eq. (2.6).

The conversion from decimal to binary is much more tedious. For a decimal number N , it is necessary to find the largest power k of 2 such that $2^k \leq N$. Add that power of 2 to the binary number and subtract it from the decimal number, and continue the process until the decimal number has been reduced to 0. This will yield the binary number as a summation of the form of Eq. (2.6).

Example 2.2

Convert the binary number 101.101 to decimal, and then convert the result back to binary.

Solution

Convert the number to decimal by writing it in the summation form of Eq. (2.6) and computing the total:

$$\begin{aligned} 101.101 &= 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} \\ &= 4 + 0 + 1 + 0.5 + 0 + 0.125 \\ &= 5.625 \end{aligned}$$

Converting 5.625 back to binary requires going through the algorithm steps:

step 1 :	$N = 5.625$	$k = 2$	$2^k = 4 \leq N$	$N - 2^k = 1.625$
step 2 :	$N = 1.625$	$k = 0$	$2^k = 1 \leq N$	$N - 2^k = 0.625$
step 3 :	$N = 0.625$	$k = -1$	$2^k = 0.5 \leq N$	$N - 2^k = 0.125$
step 4 :	$N = 0.125$	$k = -3$	$2^k = 0.125 \leq N$	$N - 2^k = 0$

$$\begin{aligned} 5.625 &= 4 + 1 + 0.5 + 0.125 \\ &= 1 \times 2^2 + 1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-3} \\ &= 101.101 \end{aligned}$$

2.3 Number Representation

It is unreasonable and often impossible to store numbers in computer memory to a maximum level of precision by keeping all their digits. One major issue is that the increasing number of digits that results from performing complex calculations will quickly consume the computer's memory. For example, the product of the two 11-digit numbers $1.2345678901 \times 2.3456789012$ equals the 21-digit number 2.89589985190657035812, and the sum of two 9-digit numbers

$123456789.0 + 0.123456789$ equals the 18-digit 123456789.123456789 where the two summands each have 9 significant digits but the sum requires 18 significant digits. Moreover, certain numbers, such as π , have an infinite number of digits that would need to be stored in memory for a maximum level of precision, which is quite simply impossible to do. Numbers in the natural world may be infinite, but computer memory is not.

It is therefore necessary for modern computers to truncate the numbers stored in their memory. This truncation will naturally cause a loss in precision in the values stored and will be a source of errors for any computer model of the real world. And this problem will be unavoidable on any computer, no matter how advanced and powerful (at least, until someone invents a computer with infinite memory and instantaneous processing). Nonetheless, the way numbers are represented and stored in the computer will lead to different levels of seriousness of these errors, and one would be wise to pick a representation scheme that minimizes problems.

Since all number representation schemes are not equal, it is necessary to begin by defining four important requirements to compare the different schemes by:

1. A scheme must represent numbers using a fixed amount of memory. This is an unavoidable requirement of modern computers.
2. A scheme must allow the user to represent a range of values both very large and very small, in order to accommodate models of any aspect of the world. The real world has an infinite range of values, which is something that is impossible to represent given the requirement of using a fixed amount of memory. However, the greater the range of values that a scheme can be represented, the better.
3. A scheme must be able to represent numbers, within the range it can handle, with a small relative error. Truncation to accommodate a fixed amount of memory will necessarily lead to errors, but keeping these errors to a minimum is always preferable. This requirement does not take into account the error on numbers outside the range of values the scheme can represent, as this error can be infinite.
4. A scheme must allow software to efficiently test for equality and relative magnitude between two numbers. This is a computing requirement rather than a memory requirement. Number comparisons are the fundamental building block of software, and given two otherwise equivalent representation scheme, one that allows more efficient comparisons will lead to more efficient software and runtime performances.

2.3.1 Fixed-Point Representation

Perhaps the easiest method of storing a real number is by storing a fixed number of digits before and after the radix point, along with its sign (0 or 1 to represent a positive or negative number respectively). For the sake of example, we will assume three digits before the point and three after, thus storing a decimal

number $\pm d_2 d_1 d_0 . d_{-1} d_{-2} d_{-3}$. For example, the constant π would be stored as 0003142. This representation is called *fixed-point representation*. It clearly satisfies the first requirement of using a fixed amount of memory. Moreover, the fourth requirement of efficiently comparing two numbers for relative magnitude can be achieved by simply comparing the two numbers digit by digit from left to right and stopping as soon as one is greater than the other.

Unfortunately, fixed-point representation does not perform well on the other two requirements. For the second requirement, the range of values that this notation can represent is very limited. The largest value that can be stored is ± 999.999 and the smallest one is ± 0.001 , which are neither very large nor very small. And for the third requirement, the relative error on some values within the range can be very large. For instance, the value 0.0015 is stored as 0.002 with a staggering relative error of 0.33.

2.3.2 Floating-Point Representation

Using the same amount of memory as fixed-point representation, it would be a lot more efficient to store numbers in scientific notation and use some of the stored digits to represent the mantissa and some to represent the exponent. Using the same example as before of storing six digits and a sign, this could give the decimal number $\pm d_0 . d_{-1} d_{-2} d_{-3} \times 10^{E_0 E_1 - 49}$. For reasons that will become clear shortly, the exponent digits are stored before the mantissa digits, so in this representation, the constant π would be stored as 0493142. This representation is called *floating-point representation*. The value 49 subtracted in the exponent is called the *bias* and is half the maximum value the exponent can take.

Floating-point representation satisfies all four requirements better than fixed-point representation. The first requirement is clearly satisfied by using exactly as much memory as fixed-point representation. The range of values covered to satisfy the second requirement is much larger than before: a two-digit exponent can cover 100 orders of magnitude, and thanks to the introduction of the bias, numbers can have exponents ranging from the minuscule 10^{-49} to the massive 10^{50} . As for the third requirement, the maximum relative error of any real number in the interval $[1.000 \times 10^{-49}, 9.999 \times 10^{50}]$ is $1/2001 \approx 0.0005$. Finally, by adding the requirement that the first digit of the mantissa d_0 must be different from zero and thanks to the fact the exponent digits are stored first, it is still possible to efficiently compare the relative magnitude of two numbers by comparing them digit by digit from left to right and stopping as soon as one is greater than the other. The added requirement that the first mantissa bit be different from zero is required; otherwise it would always be necessary to read all digits of the numbers and to perform some computations to standardize them, as the same real number could be stored in several different ways. For instance, the number 3 could be stored as 0493000,

0500300, 0510030, or 0520003, and only by subtracting the bias and shifting the mantissa appropriately does it become evident that all four values are the same. The requirement that the first bit of the mantissa be non-zero insures that only the first of these four representations is legal.

The requirement that the first bit of the mantissa must be non-zero introduces a surprising new problem: representing the real value 0 in floating-point representation is a rule-breaking special case. Moreover, given that each floating-point value has a sign, there are two such special values at 0000000 and 1000000. Floating-point representation uses this to its advantage by actually defining two values of zero, a positive and a negative one. A positive zero represents a positive number smaller than the smallest positive number in the range, and a negative zero represents a negative number greater than the greatest negative number in the range.

It is also possible to include an additional exception to the rule that the first digit of the mantissa must be non-zero, in the special case where a number is so small that it cannot be represented while respecting the rule. In the six-digit example, this would be the case, for example, for the number 1.23×10^{-50} , which could be represented as 0000123 but only with a zero as the first digit of the mantissa. Allowing this type of exception is very attractive; it would increase the range of values that can be represented by several orders of magnitude at no cost in memory space and without making relative comparisons more expensive. But this is no free lunch: the cost is that the mantissa will have fewer digits, and thus the relative error on the values in this range will be increased. Nonetheless, this trade-off can sometimes be worthwhile. A floating-point representation that allows this exception is called *denormalized*.

Example 2.3

Represent $10!$ in the six-digit floating-point format.

Solution

First, compute that $10! = 3628800$, or 3.6288×10^6 in scientific notation. The exponent is thus 55 to take into account the bias of -49 , the mantissa rounded to four digits is 3.629, and the positive sign is a 0, giving the representation 0553629.

Example 2.4

What number is represented, using the six-digit floating-point format, by 1234567?

Solution

The leading 1 indicates that it is a negative number, the exponent is 23 and the mantissa is 4.567. This represents the number $-4.567 \times 10^{23-49} = -4.567 \times 10^{-26}$.

2.3.3 Double-Precision Floating-Point Representation

The representation most commonly used in computers today is *double*, short for “double-precision floating-point format,” and formally defined in the IEEE 754 standard. Numbers are stored in binary (as one would expect in a computer) over a fixed amount of memory of 64 bits (8 bytes). The name comes from the fact this format uses double the amount of memory that was allocated to the original floating-point format (float) numbers, a decision that was made when it was found that 4 bytes was not enough to allow for the precision needed for most scientific and engineering calculations.

The 64 bits of a double number comprise, in order, 1 bit for the sign (0 for positive numbers, 1 for negative numbers), 11 bits for the exponent, and 52 bits for the mantissa. The maximum exponent value that can be represented with 11 bits is 2047, so the bias is 1023 (01111111111₂), allowing the representation of numbers in the range from 2^{-1022} to 2^{1023} . And the requirement defined previously, that the first digit of the mantissa cannot be 0, still holds. However, since the digits are now binary, this means that the first digit of the mantissa must always be 1; it is consequently not stored at all, and all 52 bits of the mantissa represent digits after the radix point following an implied leading 1. This means also that double cannot be a denormalized number representation.

For humans reading and writing 64-bit-long binary numbers can be tedious and very error prone. Consequently, for convenience, the 64 bits are usually grouped into 16 sets of 4 bits, and the value of each set of 4 bits (which will be between 0 and 15) is written using a single hexadecimal (base-16) digit. The following Table 2.1 gives the equivalences between binary, hexadecimal, and decimal.

Table 2.1 Binary, hexadecimal, and decimal number conversions

Binary	Hexadecimal	Decimal
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	a	10
1011	b	11
1100	c	12
1101	d	13
1110	e	14
1111	f	15

Converting a decimal real number to double format is done by converting it to scientific notation binary, rounding the mantissa to 52 bits after the radix point, and adding the bias of 1023 to the exponent. The double number is the assembled, starting with the sign bit of 0 or 1 if the number is positive or negative, respectively, then the exponent, and finally the 52 bits of the mantissa after the radix point, discarding the initial 1 before the radix point. On the other hand, converting a double number into a decimal real is done first by splitting it into three parts, the sign, exponent, and mantissa. The bias of 1023 is subtracted from the exponent, while a leading one is appended at the beginning of the mantissa. The real number is then computed by converting the mantissa to decimal, multiplying it by 2 to the correct exponent, and setting the appropriate positive or negative sign based on the value of the sign bit.

Example 2.5
Convert the double c066f40000000000 into decimal.

Solution
First, express the number into binary, by replacing each hexadecimal digit with its binary equivalent:

c	0	6	6	f	4	0	0	0	0	0	0	0	0	0	0
1100	0000	0110	0110	1111	0100	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000

Next, consider the three components of a double number:

1. The first bit is the sign. It is 1, meaning the number is negative.
2. The next 11 bits are the exponent. They are $10000000110_2 = 1030_{10}$. Recall also that the bias in double is 1023_{10} , which means the number is to the power $2^{1030-1023} = 2^7$.
3. The remaining 52 bits are for the mantissa. They are 0110111101 000. Adding in the implied leading 1 before the radix point and discarding the unnecessary trailing zeros, this represents the mantissa $1.0110111101_2 = 1.4345703125_{10}$.

The number in decimal is thus $-1.4345703125 \times 2^7 = -183.625$.

Example 2.6
Find the double representation of the integer 289.

Solution
First, note that the number is positive, so the sign bit is 0.
Next, convert the number to binary: $289 = 256 + 32 + 1 = 2^8 + 2^5 + 2^0 = 100100001_2$. In scientific notation, this becomes $1.00100001_2 \times 2^8$ (the radix point must move eight places to the left). The exponent for the number

(continued)

Fig. 2.1 C code generating the double special values

```
double MaxVal = 1.8E307;

double MinVal = 5E-324;

double PlusInf = MaxVal * 10;

double MinusInf = MaxVal * -1 * 10;

double PlusZero = MinVal / 10;

double MinusZero = MinVal * -1 / 10;
```

The name “overflow” comes from a figurative imagery of the problem. Picture the largest number that can be represented in 8-bit binary, 1111111_2 . Adding 1 to that value causes a chain of carry-overs: the least significant bit flips to 0 and a 1 carries over to the next position and causes that 1 to flip and carry a 1 over to the next position, and so on. In the end the most significant bit flips to zero and a 1 is carried over, except it has no place to carry to since the value is bounded to 8 bits; the number is said to “over flow”. As a result, instead of 10000000_2 , the final result of the addition is only 00000000_2 . This problem will be very familiar to the older generation of gamers: in many 8-bit RPG games, players who spent too much time levelling up their characters might see a level-255 (1111111_2) character gain a level and fall back to its starting level. This problem was also responsible for the famous kill screen in the original *Pac-Man* game, where after passing level 255, players found themselves in a half-formed level 00.

Overflow and underflow are well-known problems, and they have solutions defined in the IEEE 754 standard. That solution is to define four special values: a positive infinity as $7ff0000000000000_2$, a negative infinity as $fff0000000000000_2$, a positive zero as 0000000000000000_2 , and a negative zero as 8000000000000000_2 (both of which are different from each other and from an actual value of zero). Converting these to binary will show the positive and negative infinity values to be the appropriate sign bit with all-1 exponent bits and all-zero mantissa bits, while the positive and negative zero values are again the appropriate sign bit with all-zero exponent and mantissa bits. Whenever a computation gives a result that falls beyond one of the four edges of the double range, it is replaced by the appropriate special value. The sample code in the next Fig. 2.1 is an example that will generate all four special values.

2.4.2 Subtractive Cancellation

Consider the following difference: $3.523 - 3.537 = 0.014$. Using the six-digit floating-point system introduced previously, these numbers are represented by 0493523, 0493537, and 0471400, respectively. All three numbers appear to have the same

precision, with four decimal digits in the mantissa. However, 3.523 is really a truncated representation of any number in the range $[3.5225, 3.5235]$, as any number in that five-digit range will round to the four-digit representation 3.523. Likewise, the second number 3.537 represents the entire range $[3.5365, 3.5375]$. The maximum relative error on any of these approximations is 0.00014, so they are not a problem. However, when considering the ranges, the result of the subtraction is not 0.014 but actually could be any value in the range $[0.013, 0.015]$. The result 0.014 has no significant digits. Worse, as an approximation of the range of results, 0.014 has a relative error of 0.071, more than 500 times greater than the error of the initial values.

This phenomenon where the subtraction of similar numbers results in a significant reduction in precision is called *subtractive cancellation*. It will occur any time there is a subtraction of two numbers which are almost equal, and the result will always have no significant digits and much less precision than either initial numbers.

Unlike overflow and underflow, double format does not substitute the result of such operations with a special value. The result of 0.014 in the initial example will be stored and used in subsequent computations as if it were a precise value rather than a very inaccurate approximation. It is up to the engineers designing the mathematical software and models to check for such situations in the algorithms and take steps to avoid them.

Example 2.8

Consider two approximations of π using the six-digit floating-point representation: 3.142 and 3.14. Subtract the second from the first. Then, compute the relative error on both initial values and on the subtraction result.

Solution

$$3.142 - 3.14 = 0.002$$

However, in six-digit floating-point representation, 3.142 (0493142) represents any value in $[3.1415, 3.1425]$ and 3.14 (0493140) represents any value in the range $[3.1395, 3.1405]$. Their difference is any number in the range $[0.001, 0.003]$. The result of 0.002 has no significant digits.

Compared to $\pi = 3.141592654 \dots$, the value 3.142 has a relative error of 0.00013 and the value 3.14 has a relative error of 0.0051. The correct result of the subtraction is $\pi - 3.14 = 0.001592654 \dots$, and compared to that result, 0.002 has a relative error of 0.2558, 50 times greater than the relative error of 3.14.

2.4.3 *Non-associativity of Addition*

In mathematics, *associativity* is a well-known fundamental property of additions which states that the order in which additions are done makes no difference on the final result. Formally

$$(a + b) + c = a + (b + c) \quad (2.9)$$

This property no longer holds in floating-point number representation. Because of the truncation required to enforce a fixed amount of memory for the representation, larger numbers will dominate smaller numbers in the summation, and the order in which the larger and smaller numbers are introduced into the summation will affect the final result.

To simplify, consider again the six-digit floating-point representation. For an example of a large number dominating a smaller one, consider the sum $5592 + 0.7846 = 5592.7846$. In six-digit floating-point representation, these numbers are written 0525592 and 0487846, respectively, and there is no way to store the result of their summation entirely as it would require ten digits instead of six. The result stored is actually 0525593, which corresponds to the value 5593, a rounded result that maintains the more significant digits of the larger number and discards the less significant digits of the smaller number. The larger number has eclipsed the smaller one in importance. The problem becomes even worse in the summation $5592 + 0.3923 = 5592.3923$. The result will again need to be cropped to be stored in six digits, but this time it will be rounded to 5592. This means that the summation has left the larger number completely unchanged!

These two rounding issues are at the core of the problem of *non-associativity* when dealing with sequence of sums as in Eq. (2.9). The problem is that not just the final result, but also every partial result summed in the sequence, must be encoded in the same representation and will therefore be subject to rounding and loss.

Consider the sum of three values, $5592 + 0.3923 + 0.3923 = 5592.7846$. As before, every value in the summation can be encoded in the six-digit floating-point format, but the final result cannot. However, the order in which the summation is computed will affect what the final result will be. If the summation is computed as

$$5592 + (0.3923 + 0.3923) = 5592 + 0.7846 = 5592.7846 \quad (2.10)$$

then there is no problem in storing the partial result 0.7846, and only the final result needs to be rounded to 5593. However, if the summation is computed as

$$(5592 + 0.3923) + 0.3923 = 5592.3923 + 0.3923 = 5592.7846 \quad (2.11)$$

then there is a problem, as the partial result 5592.3923 gets rounded to 5592, and the second part of the summation then becomes $5592 + 0.3923$ again, the result of which again gets rounded to 5592. The final result of the summation has changed

Fig. 2.2 C++ code
suffering from
non-associativity

```
double sum = 0.0;
for ( int i = 0; i < 100000; i++ ) {
    sum += 0.1;
}
```

because of the order in which the partial summations were computed, in clear violation of the associativity property.

Example 2.9

Using three decimal digits of precision, add the powers of 2 from 0 to 17 in the order from 0 to 17 and then in reverse order from 17 to 0. Compute the relative error of the final result of each of the two summations.

Solution

Recall that with any such system, numbers must be rounded before and after any operation is performed. For example, $2^{10} = 1024 = 1020$ after rounding to three significant digits. Thus, the partial sums in increasing order are

1 3 7 15 31 63 127 255 511 1020 2040 4090 8190 16400 32800 65600 131000 262000

while in decreasing order, they are

131000 196000 229000 245000 253000 257000 259000 260000 261000 261000 261000 261000 261000 261000 261000 261000 261000

The correct value of this sum is $2^{18} - 1 = 262,143$. The relative error of the first sum is 0.00055, while the relative error of the second sum is 0.0044. So not only are the results different given the order of the sum, but the sum in increasing order gives a result an order of magnitude more accurate than the second one.

Like with subtractive cancellation, there is no special value in the double format to substitute in for non-associative results, and it is the responsibility of software engineers to detect and correct such cases when they happen in an algorithm. In that respect, it is important to note that non-associativity can be subtly disguised in the code. It can occur, for example, in a loop that sums a small value into a total at each increment of a long process, such as in the case illustrate in Fig. 2.2. As the partial total grows, the small incremental addition will become rounded off in later iterations, and inaccuracies will not only occur, they will become worse and worse as the loop goes on.

The solution to this problem is illustrated in Example 2.9. It consists in sorting the terms of summations in order of increasing value. Two values of the same magnitude summed together will not lose precision, as the digits of neither number will be rounded off. By summing together smaller values first, the precision of the partial total is maintained, and moreover the partial total grows in magnitude and can then be safely added to larger values. This ordering ensures that the cumulative sum of many small terms is still present in the final total.

2.5 Summary

Modern engineering modelling is done on computers, and this will continue to be the case long into the foreseeable future. This means that the first source of errors in any model is not in the algorithms used to compute and approximate it, but in the format used by the computer to store the very values that are modelled. It is simply impossible to store the infinite continuous range of real numbers using the finite set of discrete values available to computers, and this fundamental level of approximation is also the first source of errors that must be considered in any modelling process.

In engineering, a less accurate result with a predictable error is better than a more accurate result with an unpredictable error. This was one of the main reasons behind standardizing the format of floating-point representations on computers. Without standardization, the same code run on many machines could produce different answers. IEEE 754 standardized the representation and behaviour of floating-point numbers and therefore allowed better prediction of the error, and thus, an algorithm designed to run within certain tolerances will perform similarly on all platforms. Without standardization, a particular computation could have potentially very different results when run on different machines. Standardization allows the algorithm designer to focus on a single standard, as opposed to wasting time fine-tuning each algorithm for each different machine.

The properties of the double are specified by the IEEE 754 technical standard. For additional information, there are a few excellent documents which should be read, especially “Lecture Notes on the Status of IEEE Standard 754 for Binary Floating-Point Arithmetic” by Prof W. Kahan and “What Every Computer Scientist Should Know about Floating-Point Arithmetic” by David Goldberg.

2.6 Exercises

1. Represent the decimal number 523.2345 in scientific notation.
2. What is the problem if we don't use scientific notation to represent 1.23×10^7 ?
3. Add the two binary integers 100111_2 and 1000110_2 .
4. Add the two binary integers 1100011_2 and 10100101101_2 .
5. Add the two binary numbers 100.111_2 and 10.00110_2 .
6. Add the two binary numbers 110.0011_2 and 10100.101101_2 .
7. Multiply the two binary integers 100111_2 and 1010_2 .
8. Multiply the two binary integers 1100011_2 and 10011_2 .
9. Multiply the two binary numbers 10.1101_2 by 1.011_2 .
10. Multiply the two binary numbers 100.111_2 and 10.10_2 .
11. Multiply the two binary numbers 1100.011_2 and 10.011_2 .
12. Convert the binary number 10010.0011_2 to decimal.
13. Convert the binary number -111.111111_2 to decimal.

Numerical Methods and Modelling for Engineering

Khoury, R.; Harder, D.W.

2016, XVII, 332 p. 131 illus., 77 illus. in color.,

Hardcover

ISBN: 978-3-319-21175-6