Computer Arithmetic

COE 379

2025

Of the various types of data that one normally encounters, the ones we are concerned with in the context of scientific computing are the numerical types: integers (or whole numbers) $\ldots, -2, -1, 0, 1, 2, \ldots$, real numbers $0, 1, -1.5, 2/3, \sqrt{2}, \log 10, \ldots$ and complex numbers $1 + 2i, \sqrt{3} - \sqrt{5}i, \ldots$. Computer hardware is organized to give only a certain amount of space to represent each number, in multiples of bytes, each containing 8 bits. Typical values are 4 bytes for an integer, 4 or 8 bytes for a real number, and 8 or 16 bytes for a complex number.

Since only a certain amount of memory is available to store a number, it is clear that not all numbers of a certain type can be stored. For instance, for integers only a range is stored. (Languages such as *Python* have arbitrarily *large integers*, but this has no hardware support.) In the case of real numbers, even storing a range is not possible since any interval $[a, b]$ contains infinitely many numbers. Therefore, any *representation of real numbers* will cause gaps between the numbers that are stored. Calculations in a computer are sometimes described as *finite precision arithmetic*, to reflect that in general you not store numbers with infinite precision.

Since many results are not representable, any computation that results in such a number will have to be dealt with by issuing an error or by approximating the result. In this chapter we will look at how finite precision is realized in a computer, and the ramifications of such approximations of the 'true' outcome of numerical calculations.

For detailed discussions,

- the ultimate reference to floating point arithmetic is the *IEEE 754* standard [13];
- for secondary reading, see the book by Overton [19]; it is easy to find online copies of the essay by Goldberg [9];
- for extensive discussions of round-off error analysis in algorithms, see the books by Higham [11] and Wilkinson [21].

1 Bits

At the lowest level, computer storage and computer numbers are organized in *bit*s. A bit, short for 'binary digit', can have the values zero and one. Using bits we can then express numbers in *binary* notation:

$$\texttt{10010}_2 \equiv \texttt{18}_{10} \tag{1}$$

where the subscript indicates the base of the number system, and in both cases the rightmost digit is the least significant one.

The next organizational level of memory is the *byte*: a byte consists of eight bits, and can therefore represent the values $0 \cdots 255$.

Exercise 1. Use bit operations to test whether a number is odd or even.
Can you think of more than one way?

2 Integers

In scientific computing, most operations are on real numbers. Computations on integers rarely add up to any serious computation load, except in applications such as cryptography. There are also applications such as 'particle-in-cell' that can be implemented with bit operations. However, integers are still encountered in indexing calculations.

Integers are commonly stored in 16, 32, or 64 bits, with 16 becoming less common and 64 becoming more and more so. The main reason for this increase is not the changing nature of computations, but the fact that integers are used to index arrays. As the size of data sets grows (in particular in parallel computations), larger indices are needed. For instance, in 32 bits one can store the numbers zero through $2^{32} - 1 \approx 4 \cdot 10^9$. In other words, a 32 bit index can address 4 gigabytes of memory. Until recently this was enough for most purposes; these days the need for larger data sets has made 64 bit indexing necessary.

When we are indexing an array, only positive integers are needed. In general integer computations, of course, we need to accommodate the negative integers too. We will now discuss several strategies for implementing negative integers. Our motivation here will be that arithmetic on positive and negative integers should be as simple as on positive integers only: the circuitry that we have for comparing and operating on bitstrings should be usable for (signed) integers.

There are several ways of implementing negative integers. The simplest solution is to reserve one bit as a *sign bit*, and use the remaining 31 (or 15 or 63; from now on we will consider 32 bits the standard) bits to store the absolute magnitude. By comparison, we will call the straightforward interpretation of bitstrings as *unsigned* integers.

bitstring	$00\cdots0$	$\ldots$	$01\cdots1$	$10\cdots0$	$\ldots$	$11\cdots1$
interpretation as unsigned int	0	$\ldots$	$2^{31}-1$	2^{31}	$\ldots$	$2^{32}-1$
interpretation as signed integer	0	$\cdots$	$2^{31}-1$	-0	$\cdots$	$-(2^{31}-1)$

(2)

This scheme has some disadvantages, one being that there is both a positive and negative number zero. This means that a test for equality becomes more complicated than simply testing for equality as a bitstring. More importantly, in the second half of the bitstring, the interpretation as signed integer decreases, going to the right. This means that a test for greater-than becomes complex; also adding a positive number to a negative number now has to be treated differently from adding it to a positive number.

Another solution would be to let an unsigned number n be interpreted as $n - B$ where B is some plausible bias, for instance 2^{31}.

bitstring	$00\cdots0$	$\ldots$	$01\cdots1$	$10\cdots0$	$\ldots$	$11\cdots1$
interpretation as unsigned int	0	$\ldots$	$2^{31}-1$	2^{31}	$\ldots$	$2^{32}-1$
interpretation as shifted int	-2^{31}	$\ldots$	-1	0	$\ldots$	$2^{31}-1$

(3)

This shifted scheme does not suffer from the ± 0 problem, and numbers are consistently ordered. However, if we compute $n - n$ by operating on the bitstring that represents n, we do not get the bitstring for zero.

To ensure this desired behavior, we instead rotate the number line with positive and negative numbers to put the pattern for zero back at zero. The resulting scheme, which is the one that is used most commonly, is called *2's complement*. Using this scheme, the representation of integers is formally defined as follows.

Definition 1 *Let n be an integer, then the 2's complement 'bit pattern' $\beta(n)$ is a non-negative integer defined as follows:*

- *If* $0 \leq n \leq 2^{31}-1$, *the normal bit pattern for* n *is used, that is*

$$0 \leq n \leq 2^{31}-1 \Rightarrow \beta(n) = n. \tag{4}$$

- *For* $-2^{31} \leq n \leq -1$, n *is represented by the bit pattern for* $2^{32}-|n|$*:*

$$-2^{31} \leq n \leq -1 \Rightarrow \beta(n) = 2^{32}-|n|. \tag{5}$$

We denote the inverse function that takes a bit pattern and interprets as integer with $\eta = \beta^{-1}$.

The following diagram shows the correspondence between bitstrings and their interpretation as 2's complement integer:

bitstring n	$00\cdots0$	$\ldots$	$01\cdots1$	$10\cdots0$	$\ldots$	$11\cdots1$
interpretation as unsigned int	0	$\ldots$	$2^{31}-1$	2^{31}	$\ldots$	$2^{32}-1$
interpretation $\beta(n)$ as 2's comp. integer	0	$\cdots$	$2^{31}-1$	-2^{31}	$\cdots$	-1

(6)

Some observations:

- There is no overlap between the bit patterns for positive and negative integers, in particular, there is only one pattern for zero.
- The positive numbers have a leading bit zero, the negative numbers have the leading bit set. This makes the leading bit act a little like a sign bit; however, you can not make a number change sign by flipping this bit.
- If you have a positive number n, you get $-n$ by flipping all the bits, and adding 1. (What is the inverse operation?)

Exercise 2. For the 'naive' scheme and the 2's complement scheme for negative numbers, give pseudocode for the comparison test $m < n$, where m and n are integers. Be careful to distinguish between all cases of m, n positive, zero, or negative.

2.1 Integer overflow

Adding two numbers with the same sign, or multiplying two numbers of any sign, may lead to a result that is too large or too small to represent. This is called *overflow*; see section 3.4 for the corresponding floating point phenomenon. The following exercise lets you explore the behavior of an actual program.

Exercise 3. Investigate what happens when you perform such a calculation. What does your compiler say if you try to write down a non-representable number explicitly, for instance in an assignment statement?

If you program this in C, it is worth noting that while you probably get an outcome that is understandable, the behavior of overflow in the case of signed quantities is actually *undefined* under the C standard.

2.2 Addition in two's complement

Let us consider doing some simple arithmetic on 2's complement integers. We start by assuming that we have hardware that works on unsigned integers. The goal is to see that we can use this hardware to do calculations on signed integers, as represented in 2's complement.

We consider the computation of $m+n$, where m, n are representable numbers:

$$0 \leq |m|, |n| < 2^{31}. \tag{7}$$

We distinguish different cases.

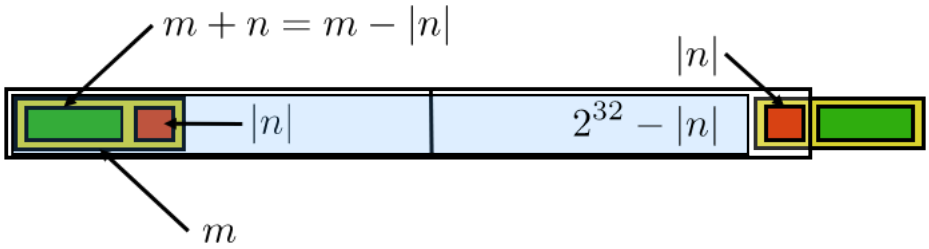

Figure 1: Addition with one positive and one negative number in 2's complement.

- The easy case is $0 < m, n$. In that case we perform the normal addition, and as long as the result stays under 2^{31}, we get the right result. If the result is 2^{31} or more, we have integer overflow, and there is nothing to be done about that.
- Case $m > 0$, $n < 0$, and $m+n > 0$. Then $\beta(m) = m$ and $\beta(n) = 2^{32} - |n|$, so the unsigned addition becomes

$$\beta(m) + \beta(n) = m + (2^{32} - |n|) = 2^{32} + m - |n|. \tag{8}$$

  Since $m - |n| > 0$, this result is $> 2^{32}$. (See figure 1.) However, we observe that this is basically $m + n$ with the 33rd bit set. If we ignore this overflowing bit, we then have the correct result.
- Case $m > 0$, $n < 0$, but $m + n < 0$. Then

$$\beta(m) + \beta(n) = m + (2^{32} - |n|) = 2^{32} - (|n| - m). \tag{9}$$

  Since $|n| - m > 0$ we get

$$\eta\big(2^{32} - (|n| - m)\big) = -\,|(|n| - m)| = m - |n| = m + n. \tag{10}$$

2.3 Subtraction in two's complement

In exercise 2 above you explored comparing two integers. Let us now explore how subtracting numbers in two's complement is implemented. Consider $0 \leq m \leq 2^{31} - 1$ and $1 \leq n \leq 2^{31}$ and let us see what happens in the computation of $m - n$.

Suppose we have an algorithm for adding and subtracting unsigned 32-bit numbers. Can we use that to subtract two's complement integers? We start by observing that the integer subtraction $m - n$ becomes the unsigned addition $m + (2^{32} - n)$.

- Case: $m < |n|$. In this case, $m - n$ is negative and $1 \leq |m - n| \leq 2^{31}$, so the bit pattern for $m - n$ is

$$\beta(m - n) = 2^{32} - (n - m). \tag{11}$$

  Now, $2^{32} - (n - m) = m + (2^{32} - n)$, so we can compute $m - n$ in 2's complement by adding the bit patterns of m and $-n$ as unsigned integers:

$$\eta\big(\beta(m) + \beta(-n)\big) = \eta\big(m + (2^{32} - |n|)\big) = \eta\big(2^{32} + (m - |n|)\big) = \eta\left(2^{32} - \big|m - |n|\big|\right) = m - |n| = m + n. \tag{12}$$

- Case: $m > n$. Here we observe that $m + (2^{32} - n) = 2^{32} + m - n$. Since $m - n > 0$, this is a number $> 2^{32}$ and therefore not a legitimate representation of a negative number. However, if we store this number in 33 bits, we see that it is the correct result $m - n$, plus a single bit in the 33-rd position. Thus, by performing the unsigned addition, and ignoring the *overflow bit*, we again get the correct result.

In both cases we conclude that we can perform the subtraction $m - n$ by adding the unsigned number that represent m and $-n$ and ignoring overflow if it occurs.

2.4 Binary coded decimal

Decimal numbers are not relevant in scientific computing, but they are useful in financial calculations, where computations involving money absolutely have to be exact. Binary arithmetic is at a disadvantage here, since numbers such as $1/10$ are repeating fractions in binary. With a finite number of bits in the mantissa, this means that the number $1/10$ can not be represented exactly in binary. For this reason, *binary-coded-decimal* schemes were used in old IBM mainframes, and are in fact being standardized in revisions of *IEEE 754* [13, 12]; see also section 4.

In BCD schemes, one or more decimal digits are encoded in a number of bits. The simplest scheme would encode the digits $0 \ldots 9$ in four bits. This has the advantage that in a BCD number each digit is readily identified; it has the disadvantage that about $1/3$ of all bits are wasted, since 4 bits can encode the numbers $0 \ldots 15$. More efficient encodings would encode $0 \ldots 999$ in ten bits, which could in principle store the numbers $0 \ldots 1023$. While this is efficient in the sense that few bits are wasted, identifying individual digits in such a number takes some decoding. For this reason, BCD arithmetic needs hardware support from the processor, which is rarely found these days; one example is the IBM Power architecture, starting with the *IBM Power6*. Intel had 'BCD opcodes' that stored a decimal digit in 4 bits, so a byte could represent the numbers $0 \cdots 99$. These opcodes were removed in 64-bit mode.

3 Real numbers

In this section we will look at the principles of how real numbers are represented in a computer, and the limitations of various schemes. Section 4 will discuss the specific IEEE 754 solution, and section 5 will then explore the ramifications of this for arithmetic involving computer numbers.

3.1 They're not really real numbers

In the physical sciences, we usually work with real numbers, so it's convenient to pretend that computers can do this too. However, since numbers in a computer have only a finite number of bits, most real numbers can not be represented exactly. In fact, even many fractions can not be represented exactly, since they repeat; for instance, $1/3 = 0.333\ldots$, which is not representable in either decimal or binary. An illustration of this is given below in exercise 6.

Exercise 4. Some programming languages allow you to write loops with not just an integer, but also with a real number as 'counter'. Explain why this is a bad idea. Hint: when is the upper bound reached?

Whether a fraction repeats depends on the number system. (How would you write $1/3$ in ternary, or base 3, arithmetic?) In binary computers this means that fractions such as $1/10$, which in decimal arithmetic are terminating, are repeating. Since decimal arithmetic is important in financial calculations, some people care about the accuracy of this kind of arithmetic; see section 2.4 for a brief discussion of how this was realized in computer hardware.

Exercise 5. Show that each binary fraction, that is, a number of the form 0.01010111001_2, can exactly be represented as a terminating decimal fraction. What is the reason that not every decimal fraction can be represented as a binary fraction?

Exercise 6. Above, it was mentioned that many fractions are not representable in binary. Illustrate that by dividing a number first by 7, then multiplying it by 7 and dividing by 49. Try as inputs 3.5 and 3.6.

3.2 Representation of real numbers

Real numbers are stored using a scheme that is analogous to 'scientific notation', where a number is represented as a *significand* and an *exponent*, for instance $6.022 \cdot 10^{23}$, which has a significand `6022` with a *radix point* after the first digit, and an exponent `23`. This number stands for

$$6.022 \cdot 10^{23} = \left[6 \times 10^0 + 0 \times 10^{-1} + 2 \times 10^{-2} + 2 \times 10^{-3}\right] \cdot 10^{23}. \tag{13}$$

For a general approach, we introduce a *base* β, which is a small integer number, 10 in the preceding example, and 2 in computer numbers. With this, we write numbers as a sum of t terms:

$$\begin{aligned} x &= \pm 1 \times \left[d_1\beta^0 + d_2\beta^{-1} + d_3\beta^{-2} + \cdots + d_t\beta^{-t+1}b \right] \times \beta^e \\ &= \pm \Sigma_{i=1}^{t} d_i\beta^{1-i} \times \beta^e \end{aligned} \tag{14}$$

where the components are

- the *sign bit*: a single bit storing whether the number is positive or negative;
- β is the base of the number system;
- $0 \leq d_i \leq \beta - 1$ the digits of the *mantissa* or *significand* – the location of the radix point (decimal point in decimal numbers) is implicitly assumed to be immediately following the first digit d_1;
- t is the length of the mantissa;
- $e \in [L, U]$ exponent; typically $L < 0 < U$ and $L \approx -U$.

Note that there is an explicit sign bit for the whole number, but the sign of the exponent is handled differently. For reasons of efficiency, e is not a signed number; instead it is considered as an unsigned number in *excess* of a certain minimum value. For instance, with 8 bits for the exponent we can represent the numbers $0 \cdots 256$, but with an excess of -127 this is interpreted as $-127 \cdots 128$. Additionally, the highest and lowest exponent values can be used to express special values such as infinity.

3.2.1 Some examples

Let us look at some specific examples of floating point representations. Base 10 is the most logical choice for human consumption, but computers are binary, so base 2 predominates there. Old IBM mainframes grouped bits to make for a base 16 representation.

	β	t	L	U
IEEE single precision (32 bit)	2	23	-126	127
IEEE double precision (64 bit)	2	53	-1022	1023
Old Cray 64 bit	2	48	-16383	16384
IBM mainframe 32 bit	16	6	-64	63
Packed decimal	10	50	-999	999

(15)

Of these, the single and double precision formats are by far the most common. We will discuss these in section 4 and further; for packed decimal, see section 2.4.

3.3 Normalized and unnormalized numbers

The general definition of floating point numbers, equation (14), leaves us with the problem that numbers can have more than one representation. For instance, $.5 \times 10^2 = .05 \times 10^3$. Since this would make computer arithmetic needlessly complicated, for instance in testing equality of numbers, we use *normalized floating point numbers*. A number is normalized if its first digit is nonzero. The implies that the mantissa part is $1 \leq x_m < \beta$; in binary $1 \leq x_m < 2$.

A practical implication in the case of binary numbers is that the first digit is always 1, so we do not need to store it explicitly. This means that every normalized floating point number is of the form

$$1.d_1d_2 \ldots d_t \times 2^e \tag{16}$$

and only the digits $d_1d_2 \ldots d_t$ are stored.

Exercise 7. With quantities t, e, what is the smallest positive normalized number?

Unnormalized numbers do exist, but only for numbers smaller than the smallest positive normalized number. See section 3.4.3.

3.4 Limitations: overflow and underflow

Since we use only a finite number of bits to store floating point numbers, not all numbers can be represented. The ones that can not be represented fall into two categories: those that are too large or too small (in some sense), and those that fall in the gaps.

The second category, where a computational result has to be rounded or truncated to be representable, is the basis of the field of round-off error analysis. We will study this in some detail below.

Numbers can be too large or too small in the following ways.

3.4.1 Overflow

The largest number we can store has every digit equal to β:

	unit	fractional			exponent
digit	$\beta-1$	$\beta-1$	$\cdots$	$\beta-1$	
value	1	β^{-1}	$\cdots$	$\beta^{-(t-1)}$	U

(17)

which adds up to

$$(\beta-1)\cdot 1+(\beta-1)\cdot\beta^{-1}+\cdots+(\beta-1)\cdot\beta^{-(t-1)}=\beta-\beta^{-(t-1)}, \tag{18}$$

and the smallest number (that is, the most negative) is $-\left(\beta-\beta^{-(t-1)}\right)$; anything larger than the former or smaller than the latter causes a condition called *overflow*.

3.4.2 Underflow

The number closest to zero is $\beta^{-(t-1)}\cdot\beta^{L}$.

	unit	fractional			exponent
digit	0	0	$\cdots$	$0,\beta-1$	$\emptyset$
value	0	0	$\cdots$	$0,\beta^{-(t-1)}$	L

(19)

A computation that has a result less than that (in absolute value) causes a condition called *underflow*.

The above 'smallest' number can not actually exist if we work with normalized numbers. Therefore, we also need to look at 'gradual underflow'.

3.4.3 Gradual underflow

The normalized number closest to zero is $1\cdot\beta^{L}$.

	unit	fractional			exponent
digit	1	0	$\cdots$	0	$\emptyset$
value	1	0	$\cdots$	0	L

(20)

Trying to compute a number less than that in absolute value is sometimes handled by using *subnormal* (or *denormalized* or *unnormalized*) numbers, a process known as *gradual underflow*. In this case, a special value of the exponent

indicates that the number is no longer normalized. In the case of IEEE standard arithmetic (see figure 3 for single precision) this is done through a zero exponent field.

However, this is typically tens or hundreds of times slower than computing with regular floating point numbers[1] . Not all processors have hardware support for gradual underflow. Ones that do include the *IBM Power6* (and up), *NVidia Fermi* (and up), but processors can have their default set to apply *flush-to-zero*. Section **??** explores performance implications.

3.4.4 What happens with over/underflow?

The occurrence of overflow or underflow means that your computation will be 'wrong' from that point on. Underflow will make the computation proceed with a zero where there should have been a nonzero; overflow is represented as `Inf`, short for 'infinite'.

Exercise 8. For real numbers x, y, the quantity $g = \sqrt{(x^2 + y^2)/2}$ satisfies

$$g \leq \max\{|x|, |y|\} \tag{21}$$

so it is representable if x and y are. What can go wrong if you compute g using the above formula? Can you think of a better way?

Computing with `Inf` is possible to an extent: adding two of those quantities will again give `Inf`. However, subtracting them gives `NaN`: 'not a number'. (See section 4.2.1.)

In none of these cases will the computation end: the processor will continue. However, you can query whether such an exception has occurred; see section 7.4.

3.4.5 Gradual underflow

Another implication of the normalization scheme is that we have to amend the definition of underflow (see section 3.4 above): any number less than $1 \cdot \beta^L$ now causes underflow.

3.5 Representation error

Let us consider a real number that is not representable in a computer's number system.

An unrepresentable number is approximated either by ordinary *rounding*, rounding up or down, or *truncation*. This means that a machine number $\tilde{x}$ is the representation for all x in an interval around it. With t digits in the mantissa, this is the interval of numbers that differ from $\bar{x}$ in the $t+1$st digit. For the mantissa part we get:

$$\begin{cases} x \in \left[\tilde{x}, \tilde{x} + \beta^{-t+1}\right) & \text{truncation} \\ x \in \left[\tilde{x} - \frac{1}{2}\beta^{-t+1}, \tilde{x} + \frac{1}{2}\beta^{-t+1}\right) & \text{rounding} \end{cases} \tag{22}$$

If x is a number and $\tilde{x}$ its representation in the computer, we call $x - \tilde{x}$ the *representation error* or *absolute representation error*, and $\frac{x-\tilde{x}}{x}$ the *relative representation error*.

Often we are not interested in the sign of the error, so we may apply the terms error and relative error to $|x - \tilde{x}|$ and $|\frac{x-\tilde{x}}{x}|$ respectively.

Often we are only interested in bounds on the error. If ϵ is a bound on the error, we will write

$$\tilde{x} = x \pm \epsilon \underset{\text{D}}{\equiv} |x - \tilde{x}| \leq \epsilon \Leftrightarrow \tilde{x} \in [x - \epsilon, x + \epsilon] \tag{23}$$

1. In real-time applications such as audio processing this phenomenon is especially noticeable; see http://phonophunk.com/articles/pentium4-denormalization.php?pg=3.

For the relative error we note that

$$\tilde{x} = x(1+\epsilon) \Leftrightarrow \left| \frac{\tilde{x} - x}{x} \right| \leq \epsilon \tag{24}$$

Let us consider an example in decimal arithmetic, that is, $\beta = 10$, and with a 3-digit mantissa: $t = 3$. The number $x = 1.256$ has a representation that depends on whether we round or truncate: $\tilde{x}_{\text{round}} = 1.26$, $\tilde{x}_{\text{truncate}} = 1.25$. The error is in the 4th digit: if $\epsilon = x - \tilde{x}$ then $|\epsilon| < \beta^{-(t-1)}$.

Exercise 9. The number in this example had no exponent part. What are the error and relative error if there had been one?

Exercise 10. In binary arithmetic the unit digit is always 1 as remarked above. How does that change the representation error?

3.6 Machine precision

Often we are only interested in the order of magnitude of the representation error, and we will write $\tilde{x} = x(1+\epsilon)$, where $|\epsilon| \leq \beta^{-t}$. This maximum relative representation error is called the *machine precision*, or sometimes *machine epsilon*. Typical values are:

$$\begin{cases} \epsilon \approx 10^{-7} & \text{32-bit single precision} \\ \epsilon \approx 10^{-16} & \text{64-bit double precision} \end{cases} \tag{25}$$

(After you have studied the section on the IEEE 754 standard, can you derive these values from the number of bits in the mantissa?)

Machine precision can be defined another way: ϵ is the smallest number that can be added to 1 so that $1 + \epsilon$ has a different representation than 1. A small example shows how aligning exponents can shift a too small operand so that it is effectively ignored in the addition operation:

$$\begin{array}{rll} & 1.0000 & \times 10^0 \\ + & 1.0000 & \times 10^{-5} \\ \hline \end{array} \quad \Rightarrow \quad \begin{array}{rll} & 1.0000 & \times 10^0 \\ + & 0.00001 & \times 10^0 \\ \hline = & 1.0000 & \times 10^0 \end{array} \tag{26}$$

Yet another way of looking at this is to observe that, in the addition $x + y$, if the ratio of x and y is too large, the result will be identical to x.

The machine precision is the maximum attainable accuracy of computations: it does not make sense to ask for more than 6-or-so digits accuracy in single precision, or 15 in double.

Exercise 11. Write a small program that computes the machine epsilon. Does it make any difference if you set the *compiler optimization levels* low or high? (If you speak C++, can you solve this exercise with a single templated code?)

Exercise 12. The number $e \approx 2.72$, the base for the natural logarithm, has various definitions. One of them is

$$e = \lim_{n \to \infty} (1 + 1/n)^n. \tag{27}$$

Write a single precision program that tries to compute e in this manner. (Do not use the `pow` function: code the power explicitly.) Evaluate the expression for an upper bound $n = 10^k$ for some k. (How far do you let k range?) Explain the output for large n. Comment on the behavior of the error.

Exercise 13. The exponential function e^x can be computed as

$$e = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots \tag{28}$$

Code this and try it for some positive x, for instance $x = 1, 3, 10, 50$. How many terms do you compute? Now compute e^{-x} and from it e^x. Use for instance the same x values and number of iterations as for e^x. What do you observe about the e^x versus e^{-x} computation? Explain.

4 The IEEE 754 standard for floating point numbers

Some decades ago, issues such as the length of the mantissa and the rounding behavior of operations could differ between computer manufacturers, and even between models from one manufacturer. This was obviously a bad situation from a point of portability of codes and reproducibility of results. The *IEEE 754* standard [13] codified all this in

sign	exponent	mantissa
p	$e = e_1 \cdots e_8$	$s = s_1 \ldots s_{23}$
31	$30 \cdots 23$	$22 \cdots 0$
$\pm$	2^{e-127} (except $e = 0, 255$)	$2^{-s_1} + \cdots + 2^{-s_{23}}$

sign	exponent	mantissa
s	$e_1 \cdots e_{11}$	$s_1 \ldots s_{52}$
63	$62 \cdots 52$	$51 \cdots 0$
$\pm$	2^{e-1023} (except $e = 0, 2047$)	$2^{-s_1} + \cdots + 2^{-s_{52}}$

(29)

Figure 2: Single and double precision definition.

1985, for instance stipulating 24 and 53 bits (before normalization) for the mantissa in single and double precision arithmetic, using a storage sequence of sign bit, exponent, mantissa; see figure 2.

Remark *The full name of the 754 standard is 'IEEE Standard for Binary Floating-Point Arithmetic (ANSI/IEEE Std 754-1985)'. It is also identical to IEC 559: 'Binary floating-point arithmetic for microprocessor systems', superseded by ISO/IEC/IEEE 60559:2011. IEEE 754 is a standard for binary arithmetic; there is a further standard,* IEEE 854, *that allows decimal arithmetic.*

Remark *'It was remarkable that so many hardware people there, knowing how difficult p754 would be, agreed that it should benefit the community at large. If it encouraged the production of floating-point software and eased the development of reliable software, it would help create a larger market for everyone's hardware. This degree of altruism was so astonishing that MATLAB's creator Dr. Cleve Moler used to advise foreign visitors not to miss the country's two most awesome spectacles: the Grand Canyon, and meetings of IEEE p754.' W. Kahan,* `http://www.cs.berkeley.edu/~wkahan/ieee754status/754story.html`.

The standard also declared the rounding behavior to be *correct rounding*: the result of an operation should be the rounded version of the exact result. See section 5.1.

4.1 Definition of the floating point formats

An inventory of the meaning of all bit patterns in *IEEE 754 single precision* is given in figure 3. Recall from section 3.3 above that for normalized numbers the first nonzero digit is a 1, which is not stored, so the bit pattern $d_1 d_2 \ldots d_t$ is interpreted as $1.d_1 d_2 \ldots d_t$.

The highest exponent is used to accommodate `Inf` and `Nan`; see section 4.2.1. The lowest exponent is used to accommodate unnormalized numbers; see section 3.4.3.

Exercise 14. Every programmer, at some point in their life, makes the mistake of storing a real number in an integer or the other way around. This can happen for instance if you call a function differently from how it was defined.

```
void a(float x) {....}
int main() {
  int i;
  .... a(i) ....
}
```

exponent	numerical value	range
$(0\cdots0)=0$	$\pm 0.s_1\cdots s_{23}\times 2^{-126}$	$s=0\cdots01 \Rightarrow 2^{-23}\cdot 2^{-126}=2^{-149}\approx 10^{-50}$
		$\cdots$
		$s=1\cdots11 \Rightarrow (1-2^{-23})\cdot 2^{-126}$
$(0\cdots01)=1$	$\pm 1.s_1\cdots s_{23}\times 2^{-126}$	$s=0\cdots00 \Rightarrow 1\cdot 2^{-126}\approx 10^{-37}$
$(0\cdots010)=2$	$\pm 1.s_1\cdots s_{23}\times 2^{-125}$	
	$\cdots$	
$(01111111)=127$	$\pm 1.s_1\cdots s_{23}\times 2^{0}$	$s=0\cdots00 \Rightarrow 1\cdot 2^{0}=1$
		$\cdots$
		$s=0\cdots01 \Rightarrow 1+2^{-23}\cdot 2^{0}=1+\epsilon$
		$\cdots$
		$s=1\cdots11 \Rightarrow (2-2^{-23})\cdot 2^{0}=2-\epsilon$
$(10000000)=128$	$\pm 1.s_1\cdots s_{23}\times 2^{1}$	
	$\cdots$	
$(11111110)=254$	$\pm 1.s_1\cdots s_{23}\times 2^{127}$	
$(11111111)=255$	$s_1\cdots s_{23}=0 \Rightarrow \pm\infty$ $s_1\cdots s_{23}\neq 0 \Rightarrow NaN$	

(30)

Figure 3: Interpretation of single precision numbers depending on the exponent bit pattern.

What happens when you print `x` in the function? Consider the bit pattern for a small integer, and use the table in figure 3 to interpret it as a floating point number. Explain that it will be a subnormal number. (This is one of those errors you won't forget after you make it. In the future, whenever you see a number on the order of 10^{-305} you'll recognize that you probably made this error.)

These days, almost all processors adhere to the IEEE 754 standard. Early generations of the *NVidia Tesla Graphics Processing Unit (GPU)*s were not standard-conforming in single precision. The justification for this was that single precision is more likely used for graphics, where exact compliance matters less. For many scientific computations, double precision is necessary, since the precision of calculations gets worse with increasing problem size or runtime. This is true for Finite Difference (FD) calculations (chapter **??**), but not for others such as the *Lattice Boltzmann Method (LBM)*.

4.2 Floating point exceptions

Various operations can give a result that is not representable as a floating point number. This situation is called an *exception*, and we say that an exception is *raised*. (Note: these are not C++ or python exceptions.) The result depends on the type of error, and the computation progresses normally. (It is possible to have the program be interrupted: section **??**.)

4.2.1 Not-a-Number

Apart from overflow and underflow, there are other exceptional situations. For instance, what result should be returned if the program asks for illegal operations such as $\sqrt{-4}$? The IEEE 754 standard has two special quantities for this: `Inf` for 'infinity' and `NaN` for 'not a number'. Whereas Infinity is the result of overflow or dividing by zero, not-a-number is the result of, for instance, subtracting infinity from infinity.

To be precise, processors will represent as `NaN` ('not a number') the result of:

- Addition $\texttt{Inf} - \texttt{Inf}$, but note that $\texttt{Inf} + \texttt{Inf}$ gives `Inf`;
- Multiplication $0 \times \texttt{Inf}$;
- Division $0/0$ or $\texttt{Inf}/\texttt{Inf}$;
- Remainder $\texttt{Inf} \, rem \, x$ or $x \, rem \, \texttt{Inf}$;
- $\sqrt{x}$ for $x < 0$;
- Comparison $x < y$ or $x > y$ where any operand is `NaN`.

Since the processor can continue computing with such a number, it is referred to as a 'quiet `NaN`'. By contrast, some `NaN` quantities can cause the processor to generate an *interrupt* or *exception*. This is called a 'signaling `NaN`'.

If `NaN` appears in an expression, the whole expression will evaluate to that value. The rule for computing with `Inf` is a bit more complicated [9].

There are uses for a signaling NaN. You could for instance fill allocated memory with such a value, to indicate that it is uninitialized for the purposes of the computation. Any use of such a value is then a program error, and would cause an exception.

The *2008 revision* of IEEE 754 suggests using the most significant bit of a NaN as the `is_quiet` bit to distinguish between quiet and signaling NaNs.

See https://www.gnu.org/software/libc/manual/html_node/Infinity-and-NaN.html for treatment of `NaN` in the GNU compiler.

Remark *NaN and Inf do not always propagate:*

- *A finite number divided by Inf is zero;*
- *Various transcendental functions treat Inf correctly, such as* $\tan^{-1}(\mathrm{Inf}) = \pi/2$.

The 2008 version of the standard treated the quiet NaN *as 'missing data', meaning that the minimum of 1 and a quiet* `NaN` *was 1. This led to min/max being non-associative. This was fixed in the 2019 standard where* `NaN`*s always propagate.*

4.2.2 Divide by zero

Division by zero results in `Inf`.

4.2.3 Overflow

This exception is raised if a result is not representable as a finite number.

4.2.4 Underflow

This exception is raised if a number is too small to be represented.

4.2.5 Inexact

This exception is raised for inexact results such as square roots, or overflow if that is not trapped.

5 Round-off error analysis

Numbers that are too large or too small to be represented, leading to overflow and underflow, are uncommon: usually computations can be arranged so that this situation will not occur. By contrast, the case that the result of a computation (even something as simple as a single addition) is not exactly representable is very common. Thus, looking at the implementation of an algorithm, we need to analyze the effect of such small errors propagating through the computation. This is commonly called *round-off error analysis*.

5.1 Correct rounding

The IEEE 754 standard, mentioned in section 4, does not only declare the way a floating point number is stored, it also gives a standard for the accuracy of operations such as addition, subtraction, multiplication, division. The model for arithmetic in the standard is that of *correct rounding*: the result of an operation should be as if the following procedure is followed:

- The exact result of the operation is computed, whether this is representable or not;
- This result is then rounded to the nearest computer number.

In short: the representation of the result of an operation is the rounded exact result of that operation. Of course, after two operations it no longer needs to hold that the computed result is the exact rounded version of the exact result.

If this statement sounds trivial or self-evident, consider subtraction as an example. In a decimal number system with two digits in the mantissa we could have a computation:

$$\begin{aligned} 1.0 - 9.4 \cdot 10^{-1} &= \\ 1.0 - 0.94 &= 0.06 \\ &= 0.6 \cdot 10^{-2} \end{aligned} \tag{31}$$

Note that in an intermediate step the mantissa $.094$ appears, which has one more digit than the two we declared for our number system. The extra digit is called a *guard digit*. Without a guard digit, this operation would have proceeded as $1.0 - 9.4 \cdot 10^{-1}$, where $9.4 \cdot 10^{-1}$ would be normalized to 0.9, giving a final result of 0.1, which is almost double the correct result.

Exercise 15. Consider the computation $1.0 - 9.5 \cdot 10^{-1}$, and assume again that numbers are rounded to fit the 2-digit mantissa. Why is this computation in a way a lot worse than the example?

One guard digit is not enough to guarantee correct rounding. An analysis that we will not reproduce here shows that three extra bits are needed [8].

Remark *The* 2008 revision *of the 754 standard declares that mathematical library functions should also be correctly rounded. However, not all compilers obey this.*

5.1.1 Mul-Add operations

In 2008, the IEEE 754 standard was revised to include the behavior of *Fused Multiply-Add (FMA)* operations, that is, operations of the form

$$c \leftarrow a * b + c. \tag{32}$$

This operation has a twofold motivation.

First, the FMA is potentially more accurate than a separate multiplication and addition, since it can use higher precision for the intermediate results, for instance by using the 80-bit *extended precision* format; section 8.3.

The standard here defines correct rounding to be that the result of this combined computation should be the rounded correct result. A naive implementation of this operations would involve two roundings: one after the multiplication and one after the addition[2].

Exercise 16. Can you come up with an example where correct rounding of an FMA is considerably more accurate than rounding the multiplication and addition separately? Hint: let the `c` term be of opposite sign as `a*b`, and try to force cancellation in the subtraction.

Secondly, FMA instructions are a way of getting higher performance: through pipelining we asymptotically get two operations per cycle. An FMA unit is then cheaper to construct than separate addition and multiplication units. Fortunately, the FMA occurs frequently in practical calculations.

2. On the other hand, if the behavior of an application was 'certified' using a non-FMA architecture, the increased precision breaks the certification. Chip manufacturers have been known to get requests for a *double rounding* FMA to counteract this change in numerical behavior.

Exercise 17. Can you think of some linear algebra operations that features FMA operations?

See section **??** for historic use of FMA in processors.

5.2 Addition

Addition of two floating point numbers is done in a couple of steps.

1. First the exponents are aligned: the smaller of the two numbers is written to have the same exponent as the larger number.
2. Then the mantissas are added.
3. Finally, the result is adjusted so that it again is a normalized number.

As an example, consider $1.00 + 2.00 \times 10^{-2}$. Aligning the exponents, this becomes $1.00 + 0.02 = 1.02$, and this result requires no final adjustment. We note that this computation was exact, but the sum $1.00 + 2.55 \times 10^{-2}$ has the same result, and here the computation is clearly not exact: the exact result is 1.0255, which is not representable with three digits to the mantissa.

In the example $6.15 \times 10^1 + 3.98 \times 10^1 = 10.13 \times 10^1 = 1.013 \times 10^2 \rightarrow 1.01 \times 10^2$ we see that after addition of the mantissas an adjustment of the exponent is needed. The error again comes from truncating or rounding the first digit of the result that does not fit in the mantissa: if x is the true sum and $\tilde{x}$ the computed sum, then $\tilde{x} = x(1+\epsilon)$ where, with a 3-digit mantissa $|\epsilon| < 10^{-3}$.

Formally, let us consider the computation of $s = x_1 + x_2$, and we assume that the numbers x_i are represented as $\tilde{x}_i = x_i(1+\epsilon_i)$. Then the sum s is represented as

$$\begin{aligned} \tilde{s} &= (\tilde{x}_1 + \tilde{x}_2)(1+\epsilon_3) \\ &= x_1(1+\epsilon_1)(1+\epsilon_3) + x_2(1+\epsilon_2)(1+\epsilon_3) \\ &\approx x_1(1+\epsilon_1+\epsilon_3) + x_2(1+\epsilon_2+\epsilon_3) \\ &\approx s(1+2\epsilon) \end{aligned} \tag{33}$$

under the assumptions that all ϵ_i are small and of roughly equal size, and that both $x_i > 0$. We see that the relative errors are added under addition.

5.3 Multiplication

Floating point multiplication, like addition, involves several steps. In order to multiply two numbers $m_1 \times \beta^{e_1}$ and $m_2 \times \beta^{e_2}$, the following steps are needed.

- The exponents are added: $e \leftarrow e_1 + e_2$.
- The mantissas are multiplied: $m \leftarrow m_1 \times m_2$.
- The mantissa is normalized, and the exponent adjusted accordingly.

For example: $1.23 \cdot 10^0 \times 5.67 \cdot 10^1 = 0.69741 \cdot 10^1 \rightarrow 6.9741 \cdot 10^0 \rightarrow 6.97 \cdot 10^0$.

Exercise 18. Analyze the relative error of multiplication.

5.4 Subtraction

Subtraction behaves very differently from addition. Whereas in addition errors are added, giving only a gradual increase of overall roundoff error, subtraction has the potential for greatly increased error in a single operation.

For example, consider subtraction with 3 digits to the mantissa: $1.24 - 1.23 = 0.01 \rightarrow 1.00 \cdot 10^{-2}$. While the result is exact, it has only one significant digit[3]. To see this, consider the case where the first operand 1.24 is actually the

3. Normally, a number with 3 digits to the mantissa suggests an error corresponding to rounding or truncating the fourth digit. We say that such a number has 3 *significant digits*. In this case, the last two digits have no meaning, resulting from the normalization process.

rounded result of a computation that should have resulted in 1.235:

$$\begin{array}{rll} .5 \times 2.47 - 1.23 & = 1.235 - 1.23 & = 0.005 \quad \text{exact} \\ & \downarrow & = 5.0 \cdot 10^{-3} \\ 1.24 - 1.23 & = 0.01 = 1. \cdot 10^{-2} \quad \text{compute} & \end{array} \tag{34}$$

Now, there is a 100% error, even though the relative error of the inputs was as small as could be expected. Clearly, subsequent operations involving the result of this subtraction will also be inaccurate. We conclude that subtracting almost equal numbers is a likely cause of numerical roundoff.

There are some subtleties about this example. Subtraction of almost equal numbers is exact, and we have the correct rounding behavior of IEEE arithmetic. Still, the correctness of a single operation does not imply that a sequence of operations containing it will be accurate. While the addition example showed only modest decrease of numerical accuracy, the cancellation in this example can have disastrous effects. You'll see an example in section 6.1.

Exercise 19. Consider the iteration

$$x_{n+1} = f(x_n) = \begin{cases} 2x_n & \text{if } 2x_n < 1 \\ 2x_n - 1 & \text{if } 2x_n \ 1 \end{cases} \tag{35}$$

Does this function have a fixed point, $x_0 \equiv f(x_0)$, or is there a cycle $x_1 = f(x_0)$, $x_0 \equiv x_2 = f(x_1)$ et cetera?
Now code this function. Is it possible to reproduce the fixed points? What happens with various starting points x_0. Can you explain this?

5.5 Associativity

The implementation of floating point number implies that simple arithmetic operation such as adding or multiplying no longer are *associative*, they way they are in mathematics. That is,

$$(a + b) + c \neq a + (b + c).$$

Let's consider a simple example, showing how associativity is affected by the rounding behavior of floating point numbers. Let floating point numbers be stored as a single digit for the mantissa, one digit for the exponent, and one guard digit; now consider the computation of $4 + 6 + 7$. Evaluation left-to-right gives:

$$\begin{array}{lll} (4 \cdot 10^0 + 6 \cdot 10^0) + 7 \cdot 10^0 & \Rightarrow 10 \cdot 10^0 + 7 \cdot 10^0 & \text{addition} \\ & \Rightarrow 1 \cdot 10^1 + 7 \cdot 10^0 & \text{rounding} \\ & \Rightarrow 1.0 \cdot 10^1 + 0.7 \cdot 10^1 & \text{using guard digit} \\ & \Rightarrow 1.7 \cdot 10^1 & \\ & \Rightarrow 2 \cdot 10^1 & \text{rounding} \end{array} \tag{36}$$

On the other hand, evaluation right-to-left gives:

$$\begin{array}{lll} 4 \cdot 10^0 + (6 \cdot 10^0 + 7 \cdot 10^0) & \Rightarrow 4 \cdot 10^0 + 13 \cdot 10^0 & \text{addition} \\ & \Rightarrow 4 \cdot 10^0 + 1 \cdot 10^1 & \text{rounding} \\ & \Rightarrow 0.4 \cdot 10^1 + 1.0 \cdot 10^1 & \text{using guard digit} \\ & \Rightarrow 1.4 \cdot 10^1 & \\ & \Rightarrow 1 \cdot 10^1 & \text{rounding} \end{array} \tag{37}$$

The conclusion is that the sequence in which rounding and truncation is applied to intermediate results makes a difference.

Exercise 20. The above example used rounding. Can you come up with a similar example in an arithmetic system using truncation?

Usually, the evaluation order of expressions is determined by the definition of the programming language, or at least by the compiler. In section 6.5 we will see how in parallel computations the associativity is not so uniquely determined.

6 Examples of round-off error

From the above, the reader may got the impression that roundoff errors only lead to serious problems in exceptional circumstances. In this section we will discuss some very practical examples where the inexactness of computer arithmetic becomes visible in the result of a computation. These will be fairly simple examples; more complicated examples exist that are outside the scope of this book, such as the instability of matrix inversion. The interested reader is referred to [11, 21].

6.1 Cancellation: the 'abc-formula'

As a practical example of cancellation, consider the quadratic equation $ax^2 + bx + c = 0$ which has solutions $x = \frac{-b \pm \sqrt{b^2-4ac}}{2a}$. Suppose $b > 0$ and $b^2 \gg 4ac$ then $\sqrt{b^2 - 4ac} \approx b$ and the '+' solution will be inaccurate. In this case it is better to compute $x_- = \frac{-b-\sqrt{b^2-4ac}}{2a}$ and use $x_+ \cdot x_- = c/a$.

For an illustration, we consider the equation

$$f(x) = \epsilon x^2 - (1 + \epsilon^2)x + \epsilon,$$

which has, for ϵ small, the roots

$$x_+ \approx \epsilon^{-1}, \qquad x_- \approx \epsilon.$$

We have two problems here. First of all, for ϵ less than machine precision we have $b \approx 1$, $b^2 - 4ac = 1$, and $x_- = 0$, which means that this solution is totally wrong. Even if the *discriminant* $b^2 - 4ac > 1$, we get problems with cancellation, and the function value is grossly inaccurate: for the textbook solution, $f(x_-) \approx \epsilon$, while for the solution without cancellation $f(x_-) \approx \epsilon^3$.

	textbook		accurate	
ϵ	x_-	$f(x_-)$	x_-	$f(x_-)$
10^{-3}	$1.000 \cdot 10^{-03}$	$-2.876 \cdot 10^{-14}$	$1.000 \cdot 10^{-03}$	$-2.168 \cdot 10^{-19}$
10^{-4}	$1.000 \cdot 10^{-04}$	$5.264 \cdot 10^{-14}$	$1.000 \cdot 10^{-04}$	0.000
10^{-5}	$1.000 \cdot 10^{-05}$	$-8.274 \cdot 10^{-13}$	$1.000 \cdot 10^{-05}$	$-1.694 \cdot 10^{-21}$
10^{-6}	$1.000 \cdot 10^{-06}$	$-3.339 \cdot 10^{-11}$	$1.000 \cdot 10^{-06}$	$-2.118 \cdot 10^{-22}$
10^{-7}	$9.992 \cdot 10^{-08}$	$7.993 \cdot 10^{-11}$	$1.000 \cdot 10^{-07}$	$1.323 \cdot 10^{-23}$
10^{-8}	$1.110 \cdot 10^{-08}$	$-1.102 \cdot 10^{-09}$	$1.000 \cdot 10^{-08}$	0.000
10^{-9}	0.000	$1.000 \cdot 10^{-09}$	$1.000 \cdot 10^{-09}$	$-2.068 \cdot 10^{-25}$
10^{-10}	0.000	$1.000 \cdot 10^{-10}$	$1.000 \cdot 10^{-10}$	0.000

(38)

Exercise 21. Write a program that computes the roots of the quadratic equation, both the 'textbook' way, and as described above.

- Let $b = -1$ and $a = -c$, and $4ac \downarrow 0$ by taking progressively smaller values for a and c.
- Print out the computed root, the root using the stable computation, and the value of $f(x) = ax^2 + bx + c$ in the computed root.

Now suppose that you don't care much about the actual value of the root: you want to make sure the residual $f(x)$ is small in the computed root. Let x^* be the exact root, then

$$f(x^* + h) \approx f(x^*) + hf'(x^*) = hf'(x^*). \tag{39}$$

Now investigate separately the cases $a \downarrow 0, c = -1$ and $a = -1, c \downarrow 0$. Can you explain the difference?

Exercise 22. Consider the functions

$$\begin{cases} f(x) = \sqrt{x+1} - \sqrt{x} \\ g(x) = 1/\left(\sqrt{x+1} + \sqrt{x}\right) \end{cases} \tag{40}$$

- Show that they are the same in exact arithmetic; however:
- Show that f can exhibit cancellation and that g has no such problem.
- Write code to show the difference between f and g. You may have to use large values for x.
- Analyze the cancellation in terms of x and machine precision. When are $\sqrt{x+1}$ and $\sqrt{x}$ less than ϵ apart? What happens then? (For a more refined analysis, when are they $\sqrt{\epsilon}$ apart, and how does that manifest itself?)
- The inverse function of $y = f(x)$ is

$$x = (y^2 - 1)^2/(4y^2) \tag{41}$$

  Add this to your code. Does this give any indication of the accuracy of the calculations?

Make sure to test your code in single and double precision. If you speak python, try the `bigfloat` package.

6.2 Summing series

The previous example was about preventing a large roundoff error in a single operation. This example shows that even gradual buildup of roundoff error can be handled in different ways.

Consider the sum $\sum_{n=1}^{10000} \frac{1}{n^2} = 1.644834\ldots$ and assume we are working with single precision, which on most computers means a machine precision of 10^{-7}. The problem with this example is that both the ratio between terms, and the ratio of terms to partial sums, is ever increasing. In section 3.6 we observed that a too large ratio can lead to one operand of an addition in effect being ignored.

If we sum the series in the sequence it is given, we observe that the first term is 1, so all partial sums ($\sum_{n=1}^{N}$ where $N < 10000$) are at least 1. This means that any term where $1/n^2 < 10^{-7}$ gets ignored since it is less than the machine precision. Specifically, the last 7000 terms are ignored, and the computed sum is 1.644725. The first 4 digits are correct.

However, if we evaluate the sum in reverse order we obtain the exact result in single precision. We are still adding small quantities to larger ones, but now the ratio will never be as bad as one-to-ϵ, so the smaller number is never ignored. To see this, consider the ratio of two terms subsequent terms:

$$\frac{n^2}{(n-1)^2} = \frac{n^2}{n^2 - 2n + 1} = \frac{1}{1 - 2/n + 1/n^2} \approx 1 + \frac{2}{n} \tag{42}$$

Since we only sum 10^5 terms and the machine precision is 10^{-7}, in the addition $1/n^2 + 1/(n-1)^2$ the second term will not be wholly ignored as it is when we sum from large to small.

Exercise 23. There is still a step missing in our reasoning. We have shown that in adding two subsequent terms, the smaller one is not ignored. However, during the calculation we add partial sums to the next term in the sequence. Show that this does not worsen the situation.

The lesson here is that series that are monotone (or close to monotone) should be summed from small to large, since the error is minimized if the quantities to be added are closer in magnitude. Note that this is the opposite strategy from the case of subtraction, where operations involving similar quantities lead to larger errors. This implies that if an application asks for adding and subtracting series of numbers, and we know a priori which terms are positive and negative, it may pay off to rearrange the algorithm accordingly.

Exercise 24. The sine function is defined as

$$\begin{aligned} \sin(x) &= x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots \\ &= \sum_{i\geq 0}^{\infty} (-1)^i \frac{x^{2i+1}}{(2i+1)!}. \end{aligned} \tag{43}$$

Here are two code fragments that compute this sum (assuming that `x` and `nterms` are given):

```
double term = x, sum = term;
for (int i=1; i<=nterms; i+=2) {
  term *=
    - x*x / (double)((i+1)*(i+2));
  sum += term;
}
printf("Sum: %e\n\n",sum);
```

```
double term = x, sum = term;
double power = x, factorial = 1., factor =
     1.;
for (int i=1; i<=nterms; i+=2) {
  power *= -x*x;
  factorial *= (factor+1)*(factor+2);
  term = power / factorial;
  sum += term; factor += 2;
}
printf("Sum: %e\n\n",sum);
```

- Explain what happens if you compute a large number of terms for $x > 1$.
- Does either code make sense for a large number of terms?
- Is it possible to sum the terms starting at the smallest? Would that be a good idea?
- Can you come with other schemes to improve the computation of $\sin(x)$?

6.3 Unstable algorithms

We will now consider an example where we can give a direct argument that the algorithm can not cope with problems due to inexactly represented real numbers.

Consider the recurrence $y_n = \int_0^1 \frac{x^n}{x-5} dx = \frac{1}{n} - 5y_{n-1}$, which is monotonically decreasing; the first term can be computed as $y_0 = \ln 6 - \ln 5$.

Performing the computation in 3 decimal digits we get:

computation		correct result
$y_0 = \ln 6 - \ln 5 = .182\|322 \times 10^1 \ldots$		1.82
$y_1 = .900 \times 10^{-1}$		.884
$y_2 = .500 \times 10^{-1}$		.0580
$y_3 = .830 \times 10^{-1}$	going up?	.0431
$y_4 = -.165$	negative?	.0343

We see that the computed results are quickly not just inaccurate, but actually nonsensical. We can analyze why this is the case.

If we define the error ϵ_n in the n-th step as

$$\tilde{y}_n - y_n = \epsilon_n, \tag{44}$$

then

$$\tilde{y}_n = 1/n - 5\tilde{y}_{n-1} = 1/n + 5n_{n-1} + 5\epsilon_{n-1} = y_n + 5\epsilon_{n-1} \tag{45}$$

so $\epsilon_n \geq 5\epsilon_{n-1}$. The error made by this computation shows exponential growth.

6.4 Linear system solving

Sometimes we can make statements about the numerical precision of a problem even without specifying what algorithm we use. Suppose we want to solve a linear system, that is, we have an $n \times n$ matrix A and a vector b of size n, and we want to compute the vector x such that $Ax = b$. (We will actually considering algorithms for this in chapter **??**.) Since the vector b will the result of some computation or measurement, we are actually dealing with a vector $\tilde{b}$, which is some perturbation of the ideal b:

$$\tilde{b} = b + \Delta b. \tag{46}$$

The perturbation vector Δb can be of the order of the machine precision if it only arises from representation error, or it can be larger, depending on the calculations that produced $\tilde{b}$.

We now ask what the relation is between the exact value of x, which we would have obtained from doing an exact calculation with A and b, which is clearly impossible, and the computed value $\tilde{x}$, which we get from computing with A and $\tilde{b}$. (In this discussion we will assume that A itself is exact, but this is a simplification.)

Writing $\tilde{x} = x + \Delta x$, the result of our computation is now

$$A\tilde{x} = \tilde{b} \tag{47}$$

or

$$A(x + \Delta x) = b + \Delta b. \tag{48}$$

Since $Ax = b$, we get $A\Delta x = \Delta b$. From this, we get (see appendix **??** for details)

$$\left\{ \begin{array}{ll} \Delta x & = A^{-1}\Delta b \\ Ax & = b \end{array} \right\} \Rightarrow \left\{ \begin{array}{rl} \|A\|\|x\| & \geq \|b\| \\ \|\Delta x\| & \leq \|A^{-1}\|\|\Delta b\| \end{array} \right. \Rightarrow \frac{\|\Delta x\|}{\|x\|} \leq \|A\|\|A^{-1}\| \frac{\|\Delta b\|}{\|b\|} \tag{49}$$

The quantity $\|A\|\|A^{-1}\|$ is called the *condition number* of a matrix. The bound (49) then says that any perturbation in the right hand side can lead to a perturbation in the solution that is at most larger by the condition number of the matrix A. Note that it does not say that the perturbation in x *needs* to be anywhere close to that size, but we can not rule it out, and in some cases it indeed happens that this bound is attained.

Suppose that b is exact up to machine precision, and the condition number of A is 10^4. The bound (49) is often interpreted as saying that the last 4 digits of x are unreliable, or that the computation 'loses 4 digits of accuracy'.

Equation (49) can also be interpreted as follows: when we solve a linear system $Ax = b$ we get an approximate solution $x + \Delta x$ which is the *exact* solution of a perturbed system $A(x + \Delta x) = b + \Delta b$. The fact that the perturbation in the solution can be related to the perturbation in the system, is expressed by saying that the algorithm exhibits *backwards stability*.

The analysis of the accuracy of linear algebra algorithms is a field of study in itself; see for instance the book by Higham [11].

6.5 Roundoff error in parallel computations

As we discussed in section 5.5, and as you saw in the above example of summing a series, addition in computer arithmetic is not *associative*. A similar fact holds for multiplication. The lack of associativity has an interesting consequence for parallel computations: the way a computation is spread over parallel processors influences the result.

As a very simple example, consider computing the sum of four numbers $a+b+c+d$. On a single processor execution proceeds left-to-right, corresponding to the following associativity:

$$((a + b) + c) + d. \tag{50}$$

On the other hand, spreading this computation over two processors, where processor 0 has a, b and processor 1 has c, d, corresponds to

$$((a+b)+(c+d)). \tag{51}$$

Generalizing this, we see that reduction operations will most likely give a different result on different numbers of processors. (The MPI standard declares that two program runs on the same set of processors should give the same result.) It is possible to circumvent this problem by replace a reduction operation by a *gather* operation to all processors, which subsequently do a local reduction. However, this increases the memory requirements for the processors.

There is an intriguing other solution to the parallel summing problem. If we use a mantissa of 4000 bits to store a floating point number, we do not need an exponent, and all calculations with numbers thus stored are exact since they are a form of fixed-point calculation [16, 17]. While doing a whole application with such numbers would be very wasteful, reserving this solution only for an occasional inner product calculation may be the solution to the reproducibility problem.

7 Computer arithmetic in programming languages

Different languages have different approaches to declaring integers and floating point numbers. Here we study some of the issues.

7.1 C/C++ data models

The C/C++ language has `short`, `int`, `float`, `double` types (see section 8.6 for complex), plus the `long` and `unsigned` modifiers on these. (Using `long` by itself is synonymous to `long int`.) The language standard gives no explicit definitions on these, but only the following restrictions:

- A short int is at least 16 bits;
- An integer is at least 16 bits, which was the case in the old days of the *DEC PDP-11*, but nowadays they are commonly 32 bits;
- A long integer is at least 32 bits, but often 64;
- A `long long` integer is at least 64 bits.
- If you need only one byte for your integer, you can use a `char`.

Additionally, pointers are related to unsigned integers; see section 7.2.3.

There are a number of conventions, called *data model*s, for the actual sizes of the various integer types.

- *LP32* (or '2/4/4') has 16-bit `int`s, and 32 bits for `long int` and pointers. This is used in the *Win16* API.
- *ILP32* (or '4/4/4') has 32 bits for `int`, `long`, and pointers. Only `long long` is 64 bits. This is used in the *Win32* API and on most 32-bit Unix or Unix-like systems.
- *LLP64* (or '4/4/8') has 32 bits for `int` and `long` (!!!); 64 bits are used for `long long`, `size_t` and pointers. This is used in the *Win64* API.
- *LP64* (or '4/8/8') has 32-bit `int`s, and 64 bits for `long`, `long long`, `size_t`, and pointers.
- *SILP64* (or '8/8/8') uses 64 bits for all of `short`, `int`, `long`, `long long`, `size_t`, and pointers. This only existed on some early Unix systems such as *Cray UNICOS*. The related *ILP64* used 16 bits for `short`.

7.2 C/C++

Certain type handling is common to the C and C++ languages; see below for mechanisms that are exclusive to C++.

7.2.1 Bits

The C logical operators and their bit variants are:

	boolean	bitwise
and	`&&`	`&`
or	`||`	`|`
not	!	
xor		`^`

The following *bit shift* operations are available:

left shift	<<
right shift	>>

You can do arithmetic with bit operations:

- Left-shift is multiplication by 2:

```
i_times_2 = i<<1;
```

- Extract bits:

```
i_mod_8 = i & 7
```

  (How does that last one work?)

Exercise 25. Bit shift operations are normally applied to unsigned quantities. Are there extra complications when you use bitshifts to multiply or divide by 2 in 2's-complement?

The following code fragment is useful for printing the bit pattern of numbers:

MISSING SNIPPET cprintbits in codesnippetsdir=snippets.code

Sample usage:

MISSING SNIPPET printbits5 in codesnippetsdir=snippets.code

7.2.2 Printing bit patterns

While C++ has a `hexfloat` format, this is not an intuitive way of displaying the bit pattern of a binary number. Here is a handy routine for displaying the actual bits.

Code:

```
// 754/bitprint.cxx
void format(const std::string  &s)
{
  // sign bit
  std::cout << s.substr(0,1)  << ' ';
  // exponent
  std::cout << s.substr(1,8);
  // mantissa in groups of 4
  for(int walk=9;walk<32;walk+=4)
    std::cout << ' ' << s.substr(walk,4);
  // newline
  std::cout << "\n";
}
    /* ... */
  uint32_t    u;
  std::memcpy(&u,&d,sizeof(u));
  std::bitset<32> b{u};
  std::stringstream s;
  s << std::hexfloat << b << '\n';
  format(s.str());
```

Output:
[code/754] bitprint:

```
Binary output of 3.14:

hexfloat:0x1.91eb86p+1
S eeeeeeee mmmm mmmm mmmm mmmm
    ↪mmmm mmm
0 10000000 1001 0001 1110 1011
    ↪1000 011
```

7.2.3 Integers and floating point numbers

The `sizeof()` operator gives the number of bytes used to store a datatype.

Floating point types are specified in *float.h*.

C integral types with specified storage exist: constants such as *int64_t* are defined by *typedef* in *stdint.h*.

The constant *NAN* is declared in `math.h`. For checking whether a value is *NaN*, use *isnan()*.

7.2.4 Changing rounding behavior

As mandated by the 754 standard, *rounding behavior* should be controllable. In C99, the API for this is contained in *fenv.h* (or for C++ *cfenv*):

```
#include <fenv.h>

int roundings[] =
  {FE_TONEAREST, FE_UPWARD, FE_DOWNWARD, FE_TOWARDZERO};
rchoice = ....
int status = fesetround(roundings[rchoice]);
```

Setting the rounding behavior can serve as a quick test for the stability of an algorithm: if the result changes appreciably between two different rounding strategies, the algorithm is likely not stable.

7.3 Limits

For C, the *numerical ranges* of C integers are defined in *limits.h*, typically giving an upper or lower bound. For instance, *INT_MAX* is defined to be 32767 or greater.

In C++ you can still use the C header `limits.h` or *climits*, but it's better to use *std::numeric_limits*, which is templated over the types. (See *Introduction to Scientific Programming book, section 24.2* for details.)

For instance

```
std::numerical_limits<int>.max();
```

7.4 Exceptions

The C/C++ languages allow access to floating point behavior through the *fenv.h* or *cfenv* header. This can be used to set rounding behavior, or to query *floating point exception*, such as overflow or divide by zero.

7.4.1 Testing

The header defines symbolic names for exceptions to be tested, such as *FE_DIVBYZERO*, *FE_OVERFLOW*, *FE_UNDERFLOW*. The actual testing is done through the function *fetestexcept*:

Code:

```
// fenv/divzero.cpp
float result = 1.f/zero;
cout << "result: " << result << '\n';
if (std::fetestexcept(FE_DIVBYZERO))
  std::cout << "division by zero reported\n";
else
  std::cout
    << "division by zero not reported\n";
```

Output:

```
[code/fenv] divzero:

result: inf
division by zero reported
```

7.4.2 Changing behavior

Another functionality offered by the header is to change rounding behavior, with values *FE_DOWNWARD*, *FE_TONEAREST*, *FE_TOWARDZERO*, *FE_UPWARD*.

Code:

```
// fenv/round.cpp
fesetround(FE_UPWARD);
float thirdup = one/3.f;
fesetround(FE_DOWNWARD);
float thirddn = one/3.f;
cout << "difference: "
     << thirdup-thirddn << '\n';
```

Output:

```
[code/fenv] round:

difference: 2.98023e-08
```

(Can you relate the output of this code to the value of machine epsilon?)

7.4.3 Exceptions defined

Exceptions defined:

- *FE_DIVBYZERO* pole error occurred in an earlier floating-point operation.
- *FE_INEXACT* inexact result: rounding was necessary to store the result of an earlier floating-point operation.
- *FE_INVALID* domain error occurred in an earlier floating-point operation.
- *FE_OVERFLOW* the result of the earlier floating-point operation was too large to be representable.
- *FE_UNDERFLOW* the result of the earlier floating-point operation was subnormal with a loss of precision.
- *FE_ALL_EXCEPT* bitwise OR of all supported floating-point exceptions .

Usage:

```
std::feclearexcept(FE_ALL_EXCEPT);
if(std::fetestexcept(FE_UNDERFLOW)) { /* ... */ }
```

In C++, `std::numeric_limits<double>::quiet_NaN()` is declared in the `limits` header, which is meaningful if `std::numeric_limits::has_quiet_NaN` is true, which is the case if `std::numeric_limits::is_iec559` is true. (ICE 559 is essentially IEEE 754; see section 4.)

The same module also has `infinity()` and `signaling_NaN()`.

For checking whether a value is `NaN`, use `std::isnan()` from `cmath` in C++.

See further http://en.cppreference.com/w/cpp/numeric/math/nan.

7.4.4 Compiler-specific behavior

Trapping exceptions can sometimes be specified by the compiler. For instance, the *gcc* compiler can *trap* exceptions by the flag `-ffpe-trap=list`; see https://gcc.gnu.org/onlinedocs/gfortran/Debugging-Options.html.

7.5 Compiler flags for fast math

Various compilers have an option for *fast math* optimizations.

- GCC and Clang: `-ffast-math`
- Intel: `-fp-model=fast` (default)
- MSVC: `/fp:fast`

This typically covers the following cases:

- Finite math: assume that `Inf` and `NaN` don't occur. This means that the test `x==x` is always true.
- Associative math: this allows rearranging the order of operations in an arithmetic expression. This is known as *re-association*, and is for instance beneficial for vectorization. However, as you saw in section 5.5, this can change the result of a computation. Also, it makes *compensated summation* impossible; section 8.1.
- Flushing subnormals to zero.

Extensive discussion: https://simonbyrne.github.io/notes/fastmath/

7.6 Fortran

7.6.1 Variable 'kind's

Fortran has several mechanisms for indicating the precision of a numerical type.

```
integer(2) :: i2
integer(4) :: i4
integer(8) :: i8

real(4) :: r4
real(8) :: r8
real(16) :: r16

complex(8) :: c8
complex(16) :: c16
complex*32 :: c32
```

This often corresponds to the number of bytes used, **but not always**. It is technically a numerical *kind selector*, and it is nothing more than an identifier for a specific type.

```
integer, parameter :: k9 = selected_real_kind(9)
real(kind=k9) :: r
r = 2._k9; print *, sqrt(r) ! prints 1.4142135623730
```

The 'kind' values will usually be 4,8,16 but this is compiler dependent.

7.6.2 Rounding behavior

In Fortran2003 the function `IEEE_SET_ROUNDING_MODE` is available in the `IEEE_ARITHMETIC` module.

7.6.3 C99 and Fortran2003 interoperability

Recent standards of the C and Fortran languages incorporate the C/Fortran interoperability standard, which can be used to declare a type in one language so that it is compatible with a certain type in the other language.

7.7 Round-off behavior in programming

From the above discussion it should be clear that some simple statements that hold for mathematical real numbers do not hold for floating-point numbers. For instance, in floating-point arithmetic

$$(a+b)+c \neq a+(b+c). \tag{52}$$

This implies that a compiler can not perform certain optimizations without it having an effect on round-off behavior[4]. In some codes such slight differences can be tolerated, for instance because the method has built-in safeguards. For instance, the stationary iterative methods of section **??** damp out any error that is introduced.

On the other hand, if the programmer has written code to account for round-off behavior, the compiler has no such liberties. This was hinted at in exercise 11 above. We use the concept of *value safety* to describe how a compiler is allowed to change the interpretation of a computation. At its strictest, the compiler is not allowed to make any changes that affect the result of a computation.

Compilers typically have an option controlling whether optimizations are allowed that may change the numerical behavior. For the Intel compiler that is `-fp-model=...`. On the other hand, options such as `-Ofast` are aimed at performance improvement only, and may affect numerical behavior severely. For the Gnu compiler full 754 compliance takes the option `-frounding-math` whereas `-ffast-math` allows for performance-oriented compiler transformations that violate 754 and/or the language standard.

These matters are also of importance if you care about *reproducibility* of results. If a code is compiled with two different compilers, should runs with the same input give the same output? If a code is run in parallel on two different processor configurations? These questions are very subtle. In the first case, people sometimes insist on *bitwise reproducibility*, whereas in the second case some differences are allowed, as long as the result stays 'scientifically' equivalent. Of course, that concept is hard to make rigorous.

Here are some issues that are relevant when considering the influence of the compiler on code behavior and reproducibility.

Re-association Foremost among changes that a compiler can make to a computation is *re-association*, the technical term for grouping $a+b+c$ as $a+(b+c)$. The *C language standard* and the *C++ language standard* prescribe strict left-to-right evaluation of expressions without parentheses, so re-association is in fact not allowed by the standard. The *Fortran language standard* has no such prescription, but there the compiler has to respect the evaluation order that is implied by parentheses.

A common source of re-association is *loop unrolling*; see section **??**. Under strict value safety, a compiler is limited in how it can unroll a loop, which has implications for performance. The amount of loop unrolling, and whether it's performed at all, depends on the compiler optimization level, the choice of compiler, and the target platform.

A more subtle source of re-association is parallel execution; see section 6.5. This implies that the output of a code need not be strictly reproducible between two runs on different parallel configurations.

4. This section borrows from documents by Microsoft http://msdn.microsoft.com/en-us/library/aa289157(vs.71).aspx and Intel http://software.intel.com/sites/default/files/article/164389/fp-consistency-122712_1.pdf; for detailed discussion the reader is referred to these.

Constant expressions It is a common compiler optimization to compute constant expressions during compile time. For instance, in

```
float one = 1.;
...
x = 2. + y + one;
```

the compiler change the assignment to `x = y+3.`. However, this violates the re-association rule above, and it ignores any dynamically set rounding behavior.

Expression evaluation In evaluating the expression $a+(b+c)$, a processor will generate an intermediate result for $b+c$ which is not assigned to any variable. Many processors are able to assign a higher *precision of the intermediate result*. A compiler can have a flag to dictate whether to use this facility.

Behavior of the floating point unit Rounding behavior (truncate versus round-to-nearest) and treatment of *gradual underflow* may be controlled by library functions or compiler options.

Library functions The IEEE 754 standard only prescribes simple operations; there is as yet no standard that treats sine or log functions. Therefore, their implementation may be a source of variability.

For more discussion, see [18].

8 More about floating point arithmetic

8.1 Kahan summation

The example in section 5.5 made visible some of the problems of computer arithmetic: rounding can cause results that are quite wrong, and very much dependent on evaluation order. A number of algorithms exist that try to compensate for these problems, in particular in the case of addition. We briefly discuss *Kahan summation*[15], named after *William Kahan*, which is one example of a *compensated summation* algorithm.

```
sum ← 0
correction ← 0
while there is another input do
    oldsum ← sum
    input ← input − correction
    sum ← oldsum + input
    correction ← (sum − oldsum) − input
```

Exercise 26. Go through the example in section 5.5, adding a final term 3; that is compute $4+6+7+3$ and $6+7+4+3$ under the conditions of that example. Show that the correction is precisely the 3 undershoot when 17 is rounded to 20, or the 4 overshoot when 14 is rounded to 10; in both cases the correct result of 20 is computed.

8.2 Other computer arithmetic systems

Other systems have been proposed to dealing with the problems of inexact arithmetic on computers. One solution is *extended precision* arithmetic, where numbers are stored in more bits than usual. A common use of this is in the calculation of inner products of vectors: the accumulation is internally performed in extended precision, but returned

as a regular floating point number. Alternatively, there are libraries such as GMPlib [7] that allow for any calculation to be performed in higher precision.

Another solution to the imprecisions of computer arithmetic is 'interval arithmetic' [14], where for each calculation interval bounds are maintained. While this has been researched for considerable time, it is not practically used other than through specialized libraries [2].

There have been some experiments with *ternary arithmetic* (see http://en.wikipedia.org/wiki/Ternary_computer and http://www.computer-museum.ru/english/setun.htm), however, no practical hardware exists.

8.3 Extended precision

When the IEEE 754 standard was drawn up, it was envisioned that processors could have a whole range of precisions. In practice, only single and double precision as defined have been used. However, one instance of *extended precision* still survives: Intel processors have 80-bit registers for storing intermediate results. (This goes back to the *Intel 8087 co-processor*.) This strategy makes sense in *FMA* instructions, and in the accumulation of inner products.

These 80-bit registers have a strange structure with an *significand integer* bit that can give rise to bit patterns that are not a valid representation of any defined number [20].

8.4 Reduced precision

You can ask 'does double precision always give benefits over single precision' and the answer is not always 'yes' but rather: 'it depends'.

8.4.1 Lower precision in iterative refinement

In iterative linear system solving (section **??**, the accuracy is determined by how precise the residual is calculated, not how precise the solution step is done. Therefore, one could do operations such as applying the preconditioner (section **??**) in reduced precision [3]. This is a form of *iterative refinement*; see section **??**.

8.4.2 Lower precision in Deep Learning

IEEE 754-2008 has a definition for the *binary16* half precision format, which has a 5-bit exponent and 11-bit mantissa.

In *Machine Learning (ML)* it is more important to express the range of values than to be precise about the exact value. (This is the opposite of traditional scientific applications, where close values need to be resolved.) This has led to the definition of the *bfloat16* 'brain float' format [1], which is a 16-bit floating point format. It uses 8 bits for the exponent and 7 bits for the mantissa. This means that it shares the same exponent range as the IEEE single precision format; see figure 4.

First of all, conversion between bfloat16 and fp32 is simple.

- Since bfloat16 and fp32 have the same structure in the first two bytes, a bfloat16 number can be derived from an fp32 number by truncating the third and fourth byte. However, rounding may give better results in practice.
- Conversely, casting a bloat16 to fp32 only requires filling the final two bytes with zeros.

A second argument for using bfloat16 is that it effectively doubles the available bandwidth. Thus, we see proposals for using this format where bandwdith is a limiting factor, and converting to regular formats elsewhere.

The limited precision of bfloat16 is probably enough to represent quantities in *Deep Learning (DL)* applications, but in order not to lose further precision it is envisioned that FMA hardware uses 32-bit numbers internally: the product of two bfloat16 number is a regular 32-bit number. In order to compute inner products (which happens as part of matrix-matrix multiplication in *DL*), we then need an FMA unit as in figure 5.

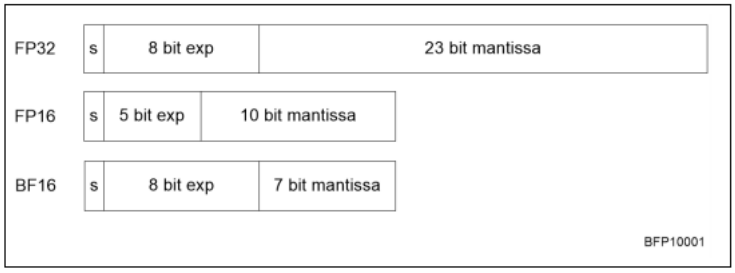

Figure 4: Comparison of fp32, fp16, and bfloat16 formats. (Illustration from [4].)

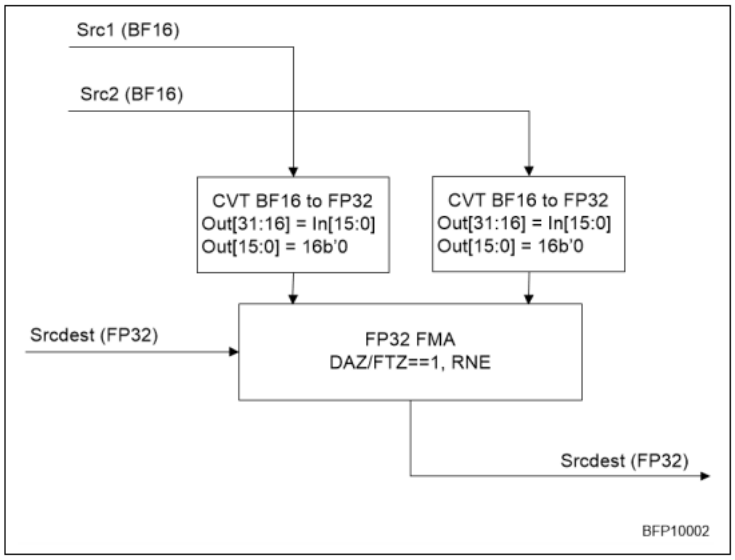

Figure 5: An FMA unit taking two bloat16 and one fp32 number. (Illustration from [4].)

- The *Intel Knights Mill*, based on the *Intel Knights Landing*, has support for reduced precision.
- The *Intel Cooper Lake* implements the *bfloat16* format [4].

Even further reduction to 8-bit was discussed in [5].

8.5 Fixed-point arithmetic

A fixed-point number (for a more thorough discussion than found here, see [22]) can be represented as $\langle N, F\rangle$ where $N \geq \beta^0$ is the integer part and $F < 1$ is the fractional part. Another way of looking at this, is that a fixed-point number is an integer stored in $N + F$ digits, with an implied decimal point after the first N digits.

Fixed-point calculations can overflow, with no possibility to adjust an exponent. Consider the multiplication $\langle N_1, F_1\rangle \times \langle N_2, F_2\rangle$, where $N_1 \geq \beta^{n_1}$ and $N_2 \geq \beta^{n_2}$. This overflows if $n_1 + n_2$ is more than the number of positions available for the integer part. (Informally, the number of digits of the product is the sum of the number of digits of the operands.) This means that, in a program that uses fixed-point, numbers will need to have a number of leading zero digits, if you are ever going to multiply them, which lowers the numerical accuracy. It also means that the programmer has to think harder about calculations, arranging them in such a way that overflow will not occur, and that numerical accuracy is still preserved to a reasonable extent.

So why would people use fixed-point numbers? One important application is in embedded low-power devices, think a battery-powered digital thermometer. Since fixed-point calculations are essentially identical to integer calculations, they do not require a floating-point unit, thereby lowering chip size and lessening power demands. Also, many early video game systems had a processor that either had no floating-point unit, or where the integer unit was considerably faster than the floating-point unit. In both cases, implementing non-integer calculations as fixed-point, using the integer unit, was the key to high throughput.

Another area where fixed point arithmetic is still used is in signal processing. In modern CPUs, integer and floating point operations are of essentially the same speed, but converting between them is relatively slow. Now, if the sine function is implemented through table lookup, this means that in $\sin(\sin x)$ the output of a function is used to index the next function application. Obviously, outputting the sine function in fixed point obviates the need for conversion between real and integer quantities, which simplifies the chip logic needed, and speeds up calculations.

8.6 Complex numbers

Some programming languages have *complex numbers* as a built-in data type, others not, and others are in between. For instance, in Fortran you can declare

```
COMPLEX z1,z2, z(32)
COMPLEX*16 zz1, zz2, zz(36)
```

A complex number is a pair of real numbers, the real and imaginary part, allocated adjacent in memory. The first declaration then uses 8 bytes to store to `REAL*4` numbers, the second one has `REAL*8`s for the real and imaginary part. (Alternatively, use `DOUBLE COMPLEX` or in Fortran90 `COMPLEX(KIND=2)` for the second line.)

By contrast, the C language does not directly have complex numbers, but both `C99` and `C++` have a `complex.h` header file[5] . This defines as complex number as in Fortran, as two real numbers.

Storing a complex number like this is easy, but sometimes it is computationally not the best solution. This becomes apparent when we look at arrays of complex numbers. If a computation often relies on access to the real (or imaginary) parts of complex numbers exclusively, striding through an array of complex numbers, has a stride two, which is disadvantageous (see section **??**). In this case, it is better to allocate one array for the real parts, and another for the imaginary parts.

Exercise 27. Suppose arrays of complex numbers are stored the Fortran way. Analyze the memory access pattern of pairwise multiplying the arrays, that is, $\forall_i\colon c_i \leftarrow a_i \cdot b_i$, where `a(), b(), c()` are arrays of complex numbers.

Exercise 28. Show that an $n \times n$ linear system $Ax = b$ over the complex numbers can be written as a $2n \times 2n$ system over the real numbers. Hint: split the matrix and the vectors in their real and imaginary parts. Argue for the efficiency of storing arrays of complex numbers as separate arrays for the real and imaginary parts.

9 Conclusions

Computations done on a computer are invariably beset with numerical error. In a way, the reason for the error is the imperfection of computer arithmetic: if we could calculate with actual real numbers there would be no problem. (There would still be the matter of measurement error in data, and approximations made in numerical methods; see the next chapter.) However, if we accept roundoff as a fact of life, then various observations hold:

- Mathematically equivalent operations need not behave identically from a point of stability; see the 'abc-formula' example.

5. These two header files are not identical, and in fact not compatible. Beware, if you compile C code with a C++ compiler [6].

- Even rearrangements of the same computations do not behave identically; see the summing example.

Thus it becomes imperative to analyze computer algorithms with regard to their roundoff behavior: does roundoff increase as a slowly growing function of problem parameters, such as the number of terms evaluated, or is worse behavior possible? We will not address such questions in further detail in this book.

10 Review questions

Exercise 29. True or false?

- For integer types, the 'most negative' integer is the negative of the 'most positive' integer.
- For floating point types, the 'most negative' number is the negative of the 'most positive' one.
- For floating point types, the smallest positive number is the reciprocal of the largest positive number.

Contents

References

[1] https://en.wikipedia.org/wiki/Bfloat16_floating-point_format. [Cited on page 27.]

[2] BOOST interval arithmetic library. http://www.boost.org/libs/numeric/interval/doc/interval.htm. [Cited on page 27.]

[3] A. Buttari, J. Dongarra, J. Langou, P. Luszczek, and J. Kurzak. Mixed precision iterative refinement techniques for the solution of dense linear systems. *International Journal of High Performance Computing Applications*, 21:457–466, 2007. [Cited on page 27.]

[4] Intel Corporation. bfloat16 - hardware numerics defintion. https://software.intel.com/sites/default/files/managed/40/8b/bf16-hardware-numerics-definition-white-paper.pdf, 2018. Document Number: 338302-001US. [Cited on page 28.]

[5] Tim Dettmers. 8-bit approximations for parallelism in deep learning. 11 2016. Proceedings of ICLR 2016. [Cited on page 28.]

[6] Dr. Dobbs. Complex arithmetic: in the intersection of C and C++. http://www.ddj.com/cpp/184401628. [Cited on page 29.]

[7] GNU multiple precision library. http://gmplib.org/. [Cited on page 27.]

[8] D. Goldberg. Compuer arithmetic. Appendix in [10]. [Cited on page 13.]

[9] David Goldberg. What every computer scientist should know about floating-point arithmetic. *Computing Surveys*, March 1991. [Cited on pages 1 and 12.]

[10] John L. Hennessy and David A. Patterson. *Computer Architecture, A Quantitative Approach*. Morgan Kaufman Publishers, 3rd edition edition, 1990, 3rd edition 2003. [Cited on page 31.]

[11] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second edition, 2002. [Cited on pages 1, 16, and 19.]

[12] IEEE 754: Standard for binary floating-point arithmetic. http://grouper.ieee.org/groups/754. [Cited on page 5.]

[13] IEEE 754-2019 standard for floating-point arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, 2019. [Cited on pages 1, 5, and 10.]

[14] Interval arithmetic. http://en.wikipedia.org/wiki/Interval_(mathematics). [Cited on page 27.]

[15] W. Kahan. Pracniques: Further remarks on reducing truncation errors. *Commun. ACM*, 8(1):40–, January 1965. [Cited on page 26.]

[16] Ulrich Kulisch. Very fast and exact accumulation of products. *Computing*, 91(4):397–405, April 2011. [Cited on page 20.]

[17] Ulrich Kulisch and Van Snyder. The exact dot product as basic tool for long interval arithmetic. *Computing*, 91(3):307–313, 2011. [Cited on page 20.]

[18] Stevel Lionel. Improving numerical reproducibility in C/C++/Fortran, 2013. [Cited on page 26.]

[19] Michael L. Overton. *Numerical Computing with IEEE Floating Point Arithmetic*. SIAM, Philadelphia PA, 2001. [Cited on page 1.]

[20] Siddhesh Poyarekar. Mostly harmless: An account of pseudo-normal floating point numbers. https://developers.redhat.com/blog/2021/05/12/mostly-harmless-an-account-of-pseudo-normal-floating-point-numbers/, 2021. [Cited on page 27.]

[21] J.H. Wilkinson. *Rounding Errors in Algebraic Processes*. Prentice-Hall, Englewood Cliffs, N.J., 1963. [Cited on pages 1 and 16.]

[22] Randy Yates. Fixed point: an introduction. http://www.digitalsignallabs.com/fp.pdf, 2007. [Cited on page 28.]