1. Numbers

We think in generalities, but we live in details.
–Alfred North Whitehead

Have you ever wondered how computers, which operate in a realm of zeros and ones, manage to perform mathematical calculations with real numbers? The secret lies in approximation.

In this chapter and the next we will investigate the foundations that allow a computer to do mathematical calculations at all. How can it store real numbers? How can it calculate the values of mathematical functions? We will understand that the computer can do these things only approximately and will thus always make errors. Numerical Analysis is all about keeping these errors as small as possible while still being able to do efficient calculations.

We will meet the two kinds of errors that a computer makes: **rounding errors** and **truncation errors**. Rounding errors arise from the way the computer needs to approximate real numbers by binary floating point numbers, which are the numbers it know how to add, subtract, multiply and divide. We'll discuss this in this chapter. Truncation errors arise from the way the computer needs to reduce all calculations to a finite number of these four basic arithmetic operations. We will see that for the first time in Chapter 2 when we discuss how computers approximate functions by power series and then have to truncate these at some finite order.

Let's start with a striking example of how bad computers actually are at doing even simple calculations:

Exercise 1.1. By hand (no computers!) compute the first few terms of the sequence

$$x_{n+1} = \begin{cases} 2x_n, & x_n \in [0, \frac{1}{2}) \\ 2x_n - 1, & x_n \in [\frac{1}{2}, 1] \end{cases} \tag{1.1}$$

with the initial condition $x_0 = 1/10$. Calculate enough terms so that you can see a pattern.

Exercise 1.2. Now use a spreadsheet to do the computations. Do you get the same answers?

Exercise 1.3. Finally, solve this problem with Python. Some starter code is given to you below.

```
x = 1.0/10
for n in range(50):
    if x< 0.5:
        # put the correct assignment here
    else:
        # put the correct assigment here
    print(x)
```

(Even if you don't know Python, you should be able to do this exercise after having read up to Section A.2.1 in the chapter on Essential Python.)

In the previous exercise it seems like the computer has failed you! What do you think happened on the computer and why did it give you a different answer? What, do you suppose, is the cautionary tale hiding behind the scenes with this problem?

Exercise 1.4. Now calculate the sequence that you get when starting with $x_0 = 1/8$? Do you again get a different answer from the computer than when doing the calculations by hand?

1.1. Binary Numbers

A computer circuit knows two states: on and off. As such, anything saved in computer memory is stored using base-2 numbers. This is called a binary number system. To fully understand a binary number system it is worthwhile to pause and reflect on our base-10 number system for a few moments.

What do the digits in the number "735" really mean? The position of each digit tells us something particular about the magnitude of the overall number. The number 735 can be represented as a sum of powers of 10 as

$$735 = 700 + 30 + 5 = 7 \times 10^2 + 3 \times 10^1 + 5 \times 10^0 \tag{1.2}$$

and we can read this number as 7 hundreds, 3 tens, and 5 ones.

Now let us switch to the number system used by computers: the binary number system. In a binary number system the base is 2 so the only allowable digits are 0 and 1 (just like in base-10 the allowable digits were 0 through 9). In binary (base-2), the number "101,101" can be interpreted as

$$101,101_2 = 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \tag{1.3}$$

(where the subscript "2" indicates the base). If we put this back into base 10, so that we can read it more comfortably, we get

$$101,101_2 = 32 + 0 + 8 + 4 + 0 + 1 = 45_{10}.$$

(The commas in the numbers are only to allow for greater readability – we can easily see groups of three digits and mentally keep track of what we are reading.)

Exercise 1.5. Express the following binary numbers in base-10.

1. 111_2
2. $10,101_2$
3. $1,111,111,111_2$

For the last one you can save yourself some work by noticing that $1,111,111,111_2 = 10,000,000,000_2 - 1$. This is a generally useful trick that will help you again in some later exercises.

Exercise 1.6. Explain the joke: *There are 10 types of people. Those who understand binary and those who do not.*

Exercise 1.7. Discussion: With your group, discuss how you would convert a base-10 number into its binary representation, without using a calculator or computer. Once you have a proposed method put it into action on the number 237_{10} to show that the base-2 expression is $11,101,101_2$.

Exercise 1.8. By hand, using the methods you developed above, convert the following numbers from base 10 to base 2 or visa versa.

- Write 12_{10} in binary
- Write 11_{10} in binary
- Write 23_{10} in binary
- Write 11_2 in base 10
- What is 100101_2 in base 10?

The icons indicate that you should enter your answers into the feedback quiz. This gives you feedback on whether or not your answer is correct, but just as importantly, it lets your lecturer know how you are progressing with the material. The feedback quizzes are not used for assessment purposes, solely for feedback.

Exercise 1.9. Write down an explanation of the technique that your group has come up with to do the conversion from base 10 to base 2.

Next we will work with fractions and decimals.

Example 1.1. Let us take the base 10 number 5.341_{10} and expand it out to get

$$5.341_{10} = 5 + \frac{3}{10} + \frac{4}{100} + \frac{1}{1000} = 5 \times 10^0 + 3 \times 10^{-1} + 4 \times 10^{-2} + 1 \times 10^{-3}.$$

The position to the right of the decimal point is the negative power of 10 for the given position.

We can do a similar thing with binary decimals.

Exercise 1.10. The base-2 number $1,101.01_2$ can be expanded in powers of 2. Fill in the question marks below and observe the pattern in the powers.

$$1,101.01_2 = ? \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + ? \times 2^0 + 0 \times 2^{?} + 1 \times 2^{-2}.$$

Example 1.2. Convert 11.01011_2 to base 10.
Solution:

$$\begin{aligned} 11.01011_2 &= 2 + 1 + \frac{0}{2} + \frac{1}{4} + \frac{0}{8} + \frac{1}{16} + \frac{1}{32} \\ &= 1 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} + 1 \times 2^{-4} + 1 \times 2^{-5} \\ &= 3.34375_{10}. \end{aligned}$$

Exercise 1.11. Convert the base 10 decimal 0.15625_{10} to binary using the following steps.

1. Multiply 0.15625 by 2. The whole number part of the result is the first binary digit to the right of the decimal point.
2. Take the result of the previous multiplication and ignore the digit to the left of the decimal point. Multiply the remaining decimal by 2. The whole number part is the second binary decimal digit.
3. Repeat the previous step until you have nothing left.

Explain to each other in the group why each step gives the binary digit that it does.

Exercise 1.12. Convert the base 10 fraction 0.1 into binary. Use this to explain why the computer made errors when it calculated with this number in Exercise 1.3.

1.2. Floating Point Numbers

In this section we will discuss how a computer actually stores a number. More specifically, since computers only have finite memory, we would really like to know the full range of numbers that are possible to store in a computer. Clearly, given the uncountable nature of the real numbers, there will be gaps between the numbers that can be stored. We would like to know what gaps in our number system to expect when using a computer to store and do computations on numbers. For this it is important to know that computers store numbers in a way that is similar to how we write numbers in scientific notation.

Example 1.3. Let us start the discussion with a very concrete example. Consider the number $x = -123.15625$ (in base 10). As we have seen this number can be converted into binary. Indeed

$$x = -123.15625_{10} = -1111011.00101_2$$

(you should check this).

If a computer needs to store this number then first they put in the binary version of scientific notation. In this case this will be

$$x = -1.11101100101_2 \times 2^6.$$

This is the **floating point representation** of the number.

Definition 1.1. For any non-zero base-2 number x the **floating point representation** is given by

$$x = (-1)^s \times (1 + m) \times 2^E$$

where $s \in \{0, 1\}$, m is a binary number such that $0 \leq m < 1$, and E is an integer.

The number $1 + m$ is called the **significand**, s is known as the **sign bit**, and E is known as the **exponent**. We will refer to m, the fractional part of the significand that actually contains the information, as the **mantissa**, but this use is not universal.

To allow for both very large and very small numbers, the exponent E can be positive or negative. However, inside the computer it is efficiently stored as an unsigned integer e called the **biased exponent**. The true exponent E is obtained by subtracting a fixed **bias** B from e:

$$E = e - B.$$

The bias is chosen to be roughly half of the maximum possible value of the stored exponent e.

Example 1.4. What are the mantissa, sign bit, and unbiased exponent for the numbers $7_{10}, -7_{10}$, and $(0.1)_{10}$?
Solution:

- For the number $7_{10} = 111_2 = 1.11 \times 2^2$ we have $s = 0, m = 0.11$ and $E = 2$.
- For the number $-7_{10} = 111_2 = -1.11 \times 2^2$ we have $s = 1, m = 0.11$ and $E = 2$.
- For the number $\frac{1}{10} = 0.000110011001100\cdots = 1.100110011\cdots \times 2^{-4}$ we have $s = 0, m = 0.100110011\cdots$, and $E = -4$.

In the last part of the previous example we saw that the number $(0.1)_{10}$ is actually a repeating decimal in base-2. This means that in order to completely represent the number $(0.1)_{10}$ in base-2 we need infinitely many decimal places. Obviously that cannot happen since we are dealing with computers with finite memory. Each number can only be allocated a finite number of bits. Thus the number needs to be rounded to the nearest number that can be represented with that number of bits. That leads to an error called the **rounding error** (sometimes also called *roundoff error*). We'll look into these in more detail in Section 1.3 below.

Definition 1.2. Machine precision is the gap between the number 1 and the next larger floating point number. Often it is represented by the symbol ϵ. To clarify: the number 1 can always be stored in a computer system exactly and if ϵ is machine precision for that computer then $1 + \epsilon$ is the next largest number that can be stored with that machine.

For all practical purposes the computer cannot tell the difference between two numbers if the relative difference is smaller than machine precision. It is important to remember this when you want to check the equality of two numbers in a computer.

Exercise 1.13. To make all of these ideas concrete let us play with a small computer system where each number is stored in the following format, using 6 bits:

$$s\, e_1\, e_2\, b_1\, b_2\, b_3$$

The first bit is for the sign ($0 = +$ and $1 = -$). The next two bits, e_1 and e_2 are for the biased exponent, and we will assume in this example that the bias is $B = 1$. The three bits on the right represent the significand of the number. Hence, every number in this number system takes the form

$$(-1)^s \times (1 + 0.b_1b_2b_3) \times 2^{e_1e_2-1}$$

- What is the smallest positive number that can be represented in this form?
- What is the largest positive number that can be represented in this form?
- What is the machine precision in this number system?

Over the course of the past several decades there have been many systems developed to properly store numbers. The IEEE standard that we now use is the accumulated effort of many computer scientists, much trial and error, and deep scientific research. We now have two standard precisions for storing numbers on a computer: single and double precision. The double precision standard is what most of our modern computers use.

Definition 1.3. According to the IEEE 754 standard:

- A **single-precision** number consists of 32 bits, with 1 bit for the sign, 8 for the exponent, and 23 for the mantissa. The bias is $B = 127$.
- A **double-precision** number consists of 64 bits with 1 bit for the sign, 11 for the exponent, and 52 for the mantissa. The bias is $B = 1023$.

Exercise 1.14. What are the largest numbers that can be stored in single and double precision?

Exercise 1.15. What is machine precision for the single and double precision standard?

Exercise 1.16. What is the gap between 2^n and the next largest number that can be stored in double precision?

Exercise 1.17. Computers contain hardware that can perform the basic operations of addition, subtraction, multiplication, and division on floating point numbers. To get a feel for what goes on under the hood, figure out how to add the numbers $x = 1.010_2 \times 2^3$ and $y = 1.110_2 \times 2^1$. How do you deal with the different exponents? What do you do so that the result is in the correct floating point format?

Much more can be said about floating point numbers such as how we store infinity, how we store NaN, and how we store 0. The Wikipedia page for floating point arithmetic might be of interest for the curious reader. It is beyond the scope of this module to go into all of those details here.

The biggest takeaway points from this section and the previous are:

- Real numbers are stored with finite precision in a computer.
- Nice rational numbers like 0.1 are sometimes not machine representable in binary.
- Machine precision is the gap between 1 and the next largest number that can be stored.
- The gap between one number and the next grows in proportion to the number.

1.3. Rounding Errors

When the binary representation of a real number has too many binary digits to be represented faithfully by a floating point number, we need to round it to the nearest floating point number that can be represented. That introduces rounding errors. We have seen above that the gap between two consecutive floating point numbers grows in proportion to the number. This means that the relative error of rounding is bounded by $\epsilon/2$ where ϵ is the machine precision.

The rounding rule that is used is "**round to nearest, ties to even**", which means that if the number is exactly halfway between two numbers that can be represented then we round the mantissa to an even binary number, i.e., to a mantissa that ends in 0.

Example 1.5. If we want to store the number $1.625 = 1.101_2$ in a floating point number system where the mantissa has only 2 bits then we round to $1.10_2 = 1.5_{10}$ because 1.101_2 is exactly halfway between 1.100_2 and 1.110_2 and the rounding rule is "round to nearest, ties to even".

To dive a little deeper into what happened in Exercise 1.3, simplify the detailed analysis by working with only a 4 bit mantissa:

Exercise 1.18. Write down how the number 1/10 is represented in a floating point number system where the mantissa has only 4 bits. Then calculate the first 10 terms of the sequence

$$x_{n+1} = \begin{cases} 2x_n, & x_n \in [0, \frac{1}{2}] \\ 2x_n - 1, & x_n \in (\frac{1}{2}, 1] \end{cases} \quad \text{with} \quad x_0 = \frac{1}{10} \tag{1.4}$$

using this number system.

Exercise 1.19. (This problem is modified from (Greenbaum and Chartier 2012))
Sometimes floating point arithmetic does not work like we would expect (and hope) as compared to exact mathematics. In each of the following problems we have a mathematical problem that the computer gets wrong. Explain why the computer is getting these wrong.

1. Mathematically we know that $\sqrt{5}^2$ should just give us 5 back. In Python type `np.sqrt(5)**2 == 5`. What do you get and why do you get it?
2. Mathematically we know that $49 \cdot \left(\frac{1}{49}\right)$ should just be 1. In Python type `49*(1/49) == 1`. What do you get and why do you get it?

3. Mathematically we know that $e^{\log(3)}$ should just give us 3 back. In Python type `np.exp(np.log(3)) == 3`. What do you get and why do you get it?
4. Create your own example of where Python gets something incorrect because of floating point arithmetic.

1.4. Loss of Significant Digits

As we have discussed, when representing real numbers by floating point numbers in the computer, rounding errors will usually occur. When doing a calculation with double-precision floating point numbers then the rounding error is only a tiny fraction of the actual number, so one might think that they really don't matter. However, calculations usually involve a number of steps, and we saw in Exercise 1.3 that the rounding errors can accumulate and become quite noticeable after a large number of steps.

But the problem is even worse. If we are not careful, then the rounding errors can get magnified already after very few steps if we perform the steps in an unfortunate way. The following examples and exercises will illustrate this.

Example 1.6. Consider the expression

$$(10^{10} + 0.123456789) - 10^{10}.$$

Mathematically, this is strictly equal to

$$(10^{10} - 10^{10}) + 0.123456789 = 0.123456789.$$

However, let us evaluate this in Python:

```
0.12345695495605469
```

Only the first six digits after the decimal point were preserved, the other digits were replaced by something seemingly random. The reason should be clear. The computer makes a rounding error when it tries to store the 10000000000.123456789. This is known as the loss of significant digits. It occurs whenever you subtract two almost equal numbers from each other.

Exercise 1.20. Consider these two mathematically equivalent ways to compute the same thing:

1) $(a + b) - c$
2) $a + (b - c)$

a) Why might these give different results in floating-point arithmetic?
b) If a is very small compared to b and c, which form would you expect to be more accurate? Why?

Exercise 1.21. Consider the trigonometric identity

$$2\sin^2(x/2) = 1 - \cos(x).$$

It gives us two different methods to calculate the same quantity. Ask Python to evaluate both sides of the identity when $x = 0.0001$. Hint: as described in Section A.2.8, use `import math` so that you can then use `math.cos()` and `math.sin()`. Also remember that exponentiation in Python is represented by `**`.

What do you observe? If you want to calculate $1 - \cos(x)$ with the highest precision, which expression would you use? Discuss.

Exercise 1.22. You know how to find the solutions to the quadratic equation

$$ax^2 + bx + c = 0.$$

You know the quadratic formula. For the larger of the two solutions the formula is

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

Let's assume that the parameters are given as

$$a = 1, \quad b = 1000000, \quad c = 1.$$

Use the quadratic formula to find the larger of the two solutions, by coding the formula up in Python. You should get a solution slightly smaller than -10^{-6}. Hint: use `math.sqrt()` to code up the square root.

Then check whether your value for x really does solve the quadratic equation by evaluating $ax^2 + bx + c$ with your value of x. You will notice that it does not work. Discuss the cause of the error.

Now, on a piece of paper, rearrange the quadratic formula for the larger solution by multiplying both the numerator and denominator by $-b - \sqrt{b^2 - 4ac}$ and then simplify by multiplying out the resulting numerator. This should give you the alternative formula

$$x = \frac{2c}{-b - \sqrt{b^2 - 4ac}}.$$

Can you see why this expression will work better for the given parameter values? Again evaluate x with Python and then check it by substituting into the quadratic expression. What do you find?

Exercise 1.23. Google the term "catastrophic cancellation" to find more examples of this phenomenon of loss of significant digits.

These exercises will give much material for in-class discussion. The aim is to make you sensitive to the issue of loss of significant figures and the fact that expressions that are mathematically equal are not always computationally equal.

1.5. Exam-Style Question

Consider a hypothetical "8-bit Mini-Float" system based on the IEEE 754 standard. The 8 bits are allocated as follows:

- **Sign bit (s):** 1 bit (Bit 7)
- **Exponent (e):** 3 bits (Bits 6-4), using a bias of 3.
- **Mantissa (m):** 4 bits (Bits 3-0), normalized with an implied leading 1.

The value of a number in this system is given by: $x = (-1)^s \times (1.m)_2 \times 2^{e-3}$.

(a) How is the **machine precision** ϵ defined. Give its value for this floating-point system. How is machine precision related to rounding errors? [3 marks]

(b) Convert the decimal number **13.5** into this 8-bit floating-point representation. Write your final answer as an 8-bit binary pattern (e.g., `0 101 1010`). [4 marks]

(c) Using this specific floating-point system, perform the addition of the number **13.5** (from part b) and **0.25**.

- Write 0.25 in this floating point system.

- Perform the addition simulating the hardware: align exponents, add significands.
- Apply the rounding rule ("Round to Nearest, Ties to Even") to determine the final stored bits.
- What is the final decimal value stored by the system, and what is the absolute error compared to the exact mathematical sum? [4 marks]

(d) A student attempts to calculate the function $f(x) = \sqrt{x^2 + 1} - x$ for a very large value $x = 10^8$.

- Explain why the result computed by a standard computer might be inaccurate (specifically naming the type of error).
- Propose an algebraically equivalent formula for $f(x)$ that avoids this error. [3 marks]