4 The real numbers

4.1 The reals and scientific notation

Rationals are not very useful as a numeric datatype because numerator and denominator tend to grow large very rapidly as the computation progresses. Another problem is that not every mathematically interesting function yields rational results, for example, $\sqrt{2}$ is *irrational*, that is, it can not be represented as a fraction (a fact already known to the ancient Greeks).

In mathematics we address this problem by extending the number system once again, from the rationals to the reals. We think of the reals as infinite expansions

$$d_n d_{n-1} \ldots d_1 d_0 \bullet d_{-1} d_{-2} \ldots$$

which are a shorthand for *infinite sums* of powers of the base

$$d_n \times b^n + d_{n-1} \times b^{n-1} + \ldots + d_1 \times b + d_0 + d_{-1} \times b^{-1} + d_{-2} \times b^{-2} \ldots$$

This is a good representation except that there are real numbers which have more than one representation: $1.000\ldots$ is the same real number as $0.999\ldots$, but these identifications are not as irritating as the ones between fractions. The set of all real numbers is denoted with $\mathbb{R}$. Like $\mathbb{Q}$, the real numbers form a field.

In practice we don't have the patience to write out all the infinitely many digits of a real number, so we have to make do with finitely many of them, for example, we work with 3.1416 as an approximation to π. It may be tempting to think of such a finite approximation as representing a decimal fraction, for example, $3.1416 = \frac{31416}{10000}$, but this would obscure the fact that we obtained 3.1416 from the real π by rounding. If we don't know the actual π but trust that someone did the rounding properly, then we know that π lies somewhere between 3.14155 and 3.14165. Thus we should think of finite decimal expansions as representing *intervals* of reals, not actual numbers.

Every finite decimal expansion (except zero) can be written in the form

$$d_0 \bullet d_{-1} d_{-2} \ldots d_{-m} \ \times \ b^n$$

where $d_0 \neq 0$ and n is an integer. This is often shortened to

$$d_0 \bullet d_{-1} d_{-2} \ldots d_{-m} \, \mathrm{E}n$$

For example, Avogadro's constant is 6.022 E23. This way of writing (approximations to) real numbers is called **scientific notation** where the part in front of E is called **mantissa** and the part following E is the **exponent**.

4.2 Floating point numbers

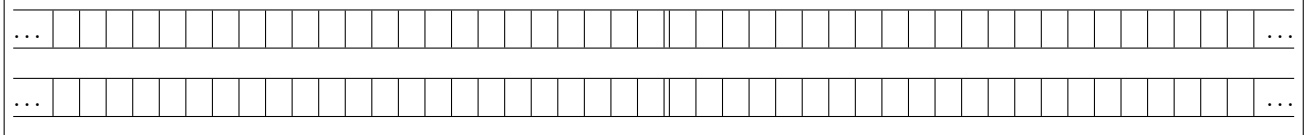
It is now a small step from scientific notation to floating point numbers stored in registers of 32 bits:

- We use base 2 for both mantissa and exponent.
- Since always $d_0 \neq 0$, we know that $d_0 = 1$, so this part of the mantissa does not need to be stored, only the digits following the binary point. (We have to find a representation for zero, though.)
- We fix the number of binary digits after the binary point to 23.
- We use 8 bits for the exponent which we interpret as an unsigned integer from which 127 has to be subtracted. (In other words, we don't use the clever encoding of integers that is employed for `int` variables.)
- We use 1 bit to indicate the sign.
- We pack the three parts in the order sign–exponent–mantissa into 32 bit registers.

There are more aspects to this representation, which you can read up on by searching for IEEE-754, the universally accepted standard for floating point representation. Rather than delving into these details, I want to present a visualisation for floating point numbers. For this we imagine a fixed scale of 256 positions in the middle of which we place the binary point:

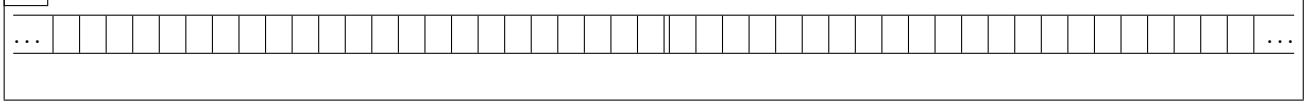
The mantissa always has 24 bits; where those 24 bits are located on the scale is determined by the exponent. However, to make the pictures more manageable, let's pretend that the mantissa has only 8 bits. As an example, we fill in the floating point approximation to $\frac{1}{5}$, and below it the same for $8 \times \frac{1}{5}$:

28

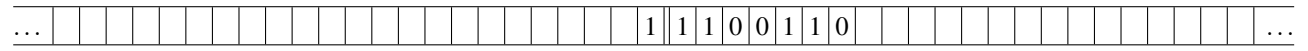


If we apply our usual algorithm for addition, then we get

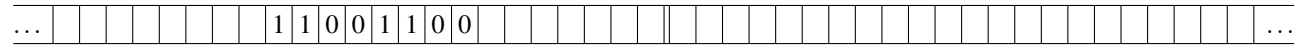
29



but now there are *too many digits*. We must cut back to the maximum that is allowed for a mantissa, in our illustration 8 bits. This is done by rounding:

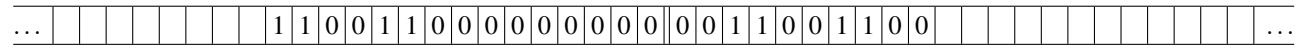


Almost every arithmetic operation requires rounding at the end. Even worse, if a is a large number and b is small, then their sum equals a again since all the digits of b are lost to rounding. Here is an illustration of this:



plus

equals



which rounds to

As a consequence, almost *no arithmetic laws* are valid for the floating point numbers.

Rather than focus on the *digit representation* of floating point numbers, we can also try to visualise their *value* on the number line. Given a fixed exponent n, recall that the mantissa can hold 2^{23} different patterns, representing the numbers

$$1.000\ldots0\mathrm{E}n \quad \text{to} \quad 1.111\ldots1\mathrm{E}n$$

The "next" floating point number after $1.111\ldots1\mathrm{E}n$ would be $10.000\ldots0\mathrm{E}n$ but we drop the last zero and write this as $1.000\ldots0\mathrm{E}n+1$. This means that the 2^{23} patterns in the mantissa represent numbers that lie between 2^n and 2^{n+1}, evenly spaced. The next 2^{23} patterns similarly represent numbers evenly spaced between 2^{n+1} and 2^{n+2}, and so on. The key observation is that the step from 2^{n+1} to 2^{n+2} is *twice* as large as that from 2^n to 2^{n+1}. And from 2^{n+2} to 2^{n+3} it is again twice what it was before.

At the other end, if we look at the largest floating point number just below $1.000\ldots0\mathrm{E}n$ we get $1.111\ldots1\mathrm{E}n-1$ and the numbers with the exponent $n-1$ are squeezed into *half* the space, i.e., they lie between 2^{n-1} and 2^n.

To summarise, as we go towards zero, the floating point numbers are closer and closer together. As we go towards infinity, they are further and further apart. Let's try to draw a picture:

30

In Java the datatype `float` corresponds to 32 bit floating point numbers as described above, the datatype `double` corresponds to 64 bit numbers. For the latter the mantissa has 52 bits (where again the leading one is kept implicit) and the exponent occupies 11 bits. In graphics co-processors one also finds half-precision floating point numbers which require only 16 bits, but Java does not support these.

Summary. This concludes our discussion of the basic number types in mathematics and computer science. We summarise the operations and laws in the following table:

	$+$	$\times$	$-$	$(\ )^{-1}$	add. canc.	mult. canc.
$\mathbb{N}$	✓	✓			✓	✓
$\mathbb{Z}$	✓	✓	✓		✓	✓
$\mathbb{Z}_m$	✓	✓	✓		✓	
$\mathbb{Z}_p$, $\mathbb{Q}$, $\mathbb{R}$	✓	✓	✓	✓	✓	✓

Exercises

1. Which field laws would you expect to hold for floating point numbers and which may fail?
2. Why should we never use the comparison "`x = y`" in a Java program where `x` and `y` are `float` variables?
3. A Java `float` variable can hold (about) 2^{32} different bit patterns. How many of these denote numbers between -1 and 1?

Practical advice

In the exam, I expect you to be able to

- draw practical conclusions from the finite precision afforded by floating point numbers.

I will also expect you to know

- the notation $\mathbb{R}$ for the reals.

I will *not* ask you about the details of the IEEE-754 standard.

More information

You may ask, can't we do better? Indeed, every real number can be thought of as an infinite *stream* of digits which contains precisely one decimal/binary point and we can write computer programs that read and produce such streams, digit by digit. Such programs can be said to implement **exact real arithmetic**, but it is not at all obvious how one might implement efficiently even elementary functions such as multiplication. In fact, exact real arithmetic is still an active area of research. If you are interested, have a look at `https://www.cs.bham.ac.uk/~mhe/papers/index.html` or `https://github.com/norbert-mueller/iRRAM`