Introduction to Scientific Computing

Professor Hanno Rein

Last updated: September 17, 2018

1 Number Representations

In this lecture, we will cover two concept that are important to understand how numbers are represented on a computer. First, we will talk about the binary system. Second, we will cover the finite precision and range of floating point numbers.

1.1 Binary system

Information on a computer is stored in two-state devices such as a flip-flop made of two transistors. Each of these flip-flop devices can either be turned on or off. This is the smallest amount of information a computer can store. We call this unit of information a *bit*. It can be either be 0 or 1. As humans, we normally represent numbers by writing their digits in base 10; after all, we have 10 fingers. However, because computers work with transistors that are either on or off, and do not have ten different levels of *on*, it makes sense to describe numbers in base 2 on a computer.

Let's go over a few examples of converting numbers from decimal to binary and back. You don't need to become an expert in this, but you should be familiar with the issues this conversion can create.

```
Decimal     Binary
0           0
1           1
2           10
3           11
4           100
10          1010
100         1100100
0.5         0.1
0.25        0.01
0.75        0.11
0.1         0.0001001100110011...
0.33333...  0.101010101...
```

Code 1: Example of binary/decimal conversion.

Let's discuss the last two examples in more detail. These illustrate an important fact: some fractions can be represented exactly in base 10 but become an infinite series in base 2. This brings us to the issue of finite precision.

1.2 Bits and bytes

On a computer there is only a certain amount of space available. In fact, the same is true if you write numbers on a piece of paper. We want to make best use of the available space which is why there are

different ways of storing a number on a computer (and on paper). We choose the way that makes most sense for the task we are trying to accomplish.

As you can see in the above example, some of the number cannot be written exactly on any finite piece of memory (or paper). One example is the number 1/3. We had to come up with a completely new way of writing a number, namely as a fraction, to be able to write it down. Some numbers are representable exactly in base 10, but not in base 2, one example is the number $1/10 = 0.1$. Note that computers do (in general) not understand what a fraction is. They use a representation called floating point (see below) as the representation for everything. There are some exceptions to this rule. First of all, you could write a program that understands fractions. Second, there are mathematical packages such as Maple and Mathematica that already come with this functionality. We will not cover those in this lecture, but it is good to keep in mind that solutions already exists if you every work on a problem that makes heavy use of fractions and you want to perform a very accurate calculation.

All types of information on a computer such as numbers, texts, sounds and videos are stored as a collection of bits. A bit is a very small amount of information, so often 8 bits are grouped together and called a byte. The following figure shows other commonly used names for a certain number of bits. You've probably heard *bit* and *byte* before, but maybe not *word* or *nibble*.

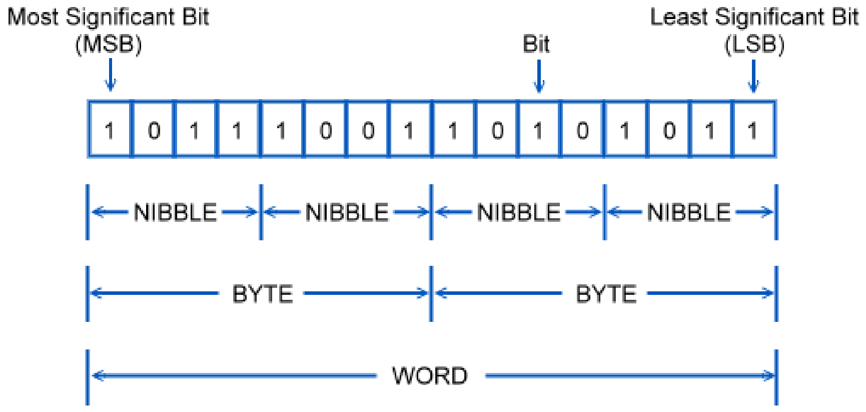

Figure 1: Bits and bytes. Source: `http://www.pcldev.com`.

Most importantly for us, each number on a computer is represented by a finite number of bits (or bytes). This has dramatic consequences for the algorithms we will implement. For that reason, we will discuss in detail how numbers are stored. There are many conceptually different ways to store numbers. We'll talk about the two most commonly used ones: integers and floating point numbers next.

1.3 Integers

Integers are one of the most fundamental datatypes in computer science. Typically an integer consists of at least two bytes on a modern computer. With two bytes (16 bits) we can store numbers from 0 (including) up to

$$2^{16} - 1 = 65535.$$

However, often we also want to be able to store negative integers. We need to reserve one bit for the sign of the integer, thus leaving us 15 bits for the magnitude. The range of a signed two-byte integer is therefore

$$-2^{15} = -32768 \dots 32767 = 2^{15} - 1.$$

Let's look at what this finite range of possible integers means in practice. If we start with an integer set to 0 and add 1 many times, eventually we will reach the maximum. When this happens, we will get a negative number! Python 3 is a high level language and actually pretty smart. It automatically increases the number of bytes for the integer if you try to add 1 to the maximum allowed value for the integer. In that case, the integers becomes what is called a *long integer*, or just *long*. In many other programming languages, this is not done automatically for you and you need to be extremely careful about having integers which are larger then the maximum allowed size.

1.4 Floats

Most numbers in scientific calculations will be represented by something called a *float* or a *floating point number* on the computers that we use. The name comes from the fact that the decimal point will shift (float) and is not fixed. You can consider the integer datatype as a fixed precision datatype (i.e. no information about the digits after the decimal point is stored). In python and almost all other programming languages, the industry standard of representing such a number is IEEE-754. There is a single and double precision version of this standard. We use the *double precision* version. It uses 8 bytes, i.e. 64 bits. On some special hardware, for example GPUs, a datatype with less precision, *single precision* is used. So how do we represent a number x where the decimal point can shift? We describe the number as a combination of three other numbers:

$$x = \text{sign} \times \text{mantissa} \times 2^{\text{exponent}}$$

The sign is just a single bit, either 0 or 1, describing whether the number is positive or negative. The mantissa (sometimes called fraction) consists of 52 bits. These are the digits after the decimal point. The exponent contains 11 bits, which is just an integer. The following figure illustrates how a double floating point number is stored in memory.

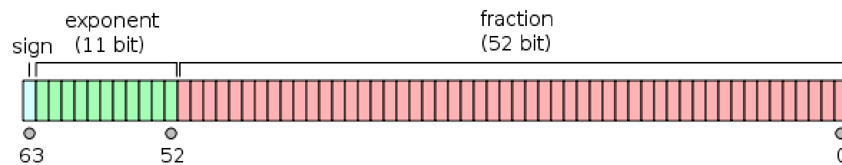

Figure 2: Double floating point number in memory. Source: `http://en.wikipedia.org`.

Putting this all together we can calculate the number x from the sign, mantissa and exponent bits using the following formula:

$$x = (-1)^{\text{sign}} \left(1 + \sum_{i=1}^{52} b_{52-i} 2^{-i}\right) 2^{e-1023}$$

This formula is crucial to understand how a floating point number is constructed. Floating point numbers have important limitations for both the precision and the range of possible numbers.

Let's look at the precision first. There are only 52 bits available in the mantissa. Thus the binary representation of a number is truncated after 52 binary digits. 52 binary digits correspond to approximately 16 decimal digits. Any digits after that cannot be represented and are simply ignored. Let's look at 1/10 and 2/10. Both of these numbers cannot represented exactly as a binary fraction (see above). For that reason, the addition of the two numbers will also not exactly give us 3/10 (which also can not be represented exactly in the first place). Have a careful look at the python snippet in Code 2 and make sure you understand why $0.2 + 0.1$ is not the same number as 0.3.

```
>>> print("%.32f"%(0.1))
0.10000000000000000555111512312578270211815
>>> print("%.32f"%(0.2))
0.20000000000000001110223024625156540423631
>>> print("%.32f"%(0.2+0.1))
0.30000000000000004440892098500626161694526
>>> print("%.32f"%(0.3))
0.29999999999999998889776975374843459576368
```

Code 2: Python listing testing the floating point precision.

Next, let's look at the finite range. Let's start with the number $x = 1$ and multiply it with the number 2 multiple times. Eventually we will run over the maximum range of what double precision floating point numbers can represent. This happens when we run out of bits in the exponent. Let's see when this occurs in python. A simple test is implemented in Code 3. The code outputs the largest number before we reach infinity. Here, infinite means that the computer has noticed that we ran above the maximum range. It's not actually infinity, just a number larger than what the computer can represent. In python (using the default double precision) we get a value around 10^{307}. Note that python outputs this in *scientific notation* with a syntax involving `..e+...` Can you explain why we get the number 307 (roughly)?

```
>>> x = 1.
>>> while x<x*2.:
...     x=2.*x
...     print(x)
...
2.0
4.0
8.0
16.0
[...]
4.49423283716e+307
8.98846567431e+307
inf
```

Code 3: Python listing testing the floating point maximum.

1.5 Key things to remember

- Finite amount of storage on a computer
- Two number formats most often used: integers + floating point numbers
- Know how to convert numbers from binary to decimal and vice versa
- Binary fractions
- Double precision uses 8 bytes, 64 bits
- 1 bit for sign, 11 for exponent, 52 for fraction
- Smallest number $\sim 5 \cdot 10^{-324}$ (the numbers from $\sim 10^{-308}$ to $\sim 10^{-324}$ are a special case)

- Largest number $\sim 2.8 \cdot 10^{308}$
- Scientific notation
- Special numbers for 0, infinity, and *not a number*
- Know which numbers can be represented exactly, approximately, or not at all.
- Try to understand the following. First column shows the number we want to represent, second shows the actual floating point number on the computer, third shows the binary representation.

```
0.74000    0.73999999999999999112    0 01111111110 0111101011100001010001111010111000010100011110101110
0.87500    0.87500000000000000000    0 01111111110 1100000000000000000000000000000000000000000000000000
0.75000    0.75000000000000000000    0 01111111110 1000000000000000000000000000000000000000000000000000
0.75000    0.75000000000000000000    0 01111111110 1000000000000000000000000000000000000000000000000000
0.50000    0.50000000000000000000    0 01111111110 0000000000000000000000000000000000000000000000000000
0.87500    0.87500000000000000000    0 01111111110 1100000000000000000000000000000000000000000000000000
1.25000    1.25000000000000000000    0 01111111111 0100000000000000000000000000000000000000000000000000
1.25000    1.25000000000000000000    0 01111111111 0100000000000000000000000000000000000000000000000000
0.87500    0.87500000000000000000    0 01111111110 1100000000000000000000000000000000000000000000000000
0.75000    0.75000000000000000000    0 01111111110 1000000000000000000000000000000000000000000000000000
```

```
5.0e+309          inf    0 11111111111 0000000000000000000000000000000000000000000000000000
9.0e+302    9.000e+302   0 11111101101 0100111111111001100100110011100010000110011101011110
1.0e-324    0.000e+00    0 00000000000 0000000000000000000000000000000000000000000000000000
5.0e+305    5.000e+305   0 11111110110 0110110010001110010111001010001000111001000000101000
8.0e-321    7.984e-321   0 00000000000 0000000000000000000000000000000000000000011001010000
2.0e-328    0.000e+00    0 00000000000 0000000000000000000000000000000000000000000000000000
4.0e+305    4.000e+305   0 11111110110 0010001110100101000101101110100000101101100110111010
9.0e+306    9.000e+306   0 11111111010 1001101000100000001010000110110100000000001000101110
4.0e+301    4.000e+301   0 11111101000 1101110111010100101110101010000000001001001100000011
1.0e-328    0.000e+00    0 00000000000 0000000000000000000000000000000000000000000000000000
```