**Numerical Algorithms**

Computational X is a discipline X that relies on computer simulations

Computational Science and Engineering (CSE) is the umbrella term for all the Computational Xs

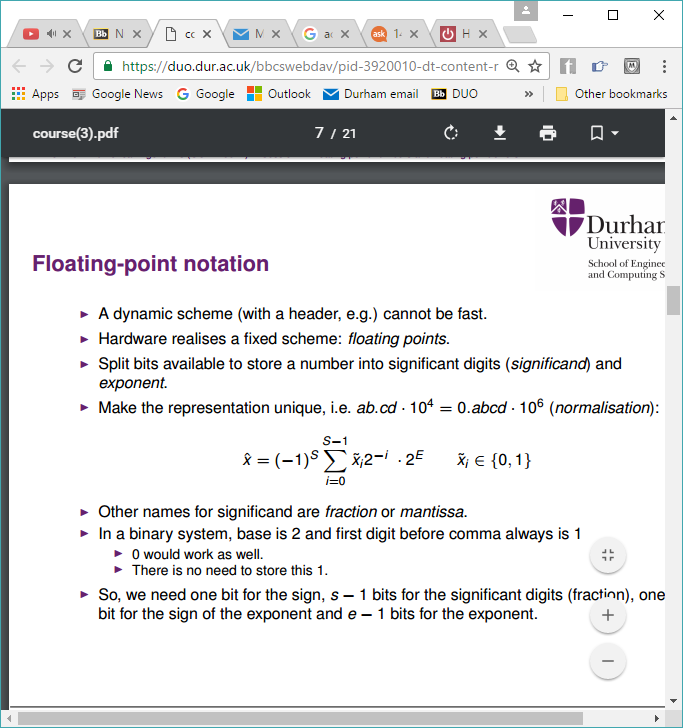
Scientiﬁc Computing is an interdisciplinary (combining maths, CS and application knowledge) discipline tackling methodological questions how to realise Computational X, i.e. how to translate mathematical formulas (theories) into algorithms, which implementations to use for these algorithms, how to program, how to visualise, etc.

Numerical algorithms is a subset of scientific computing focusing on the interplay of mathematics and computer science.

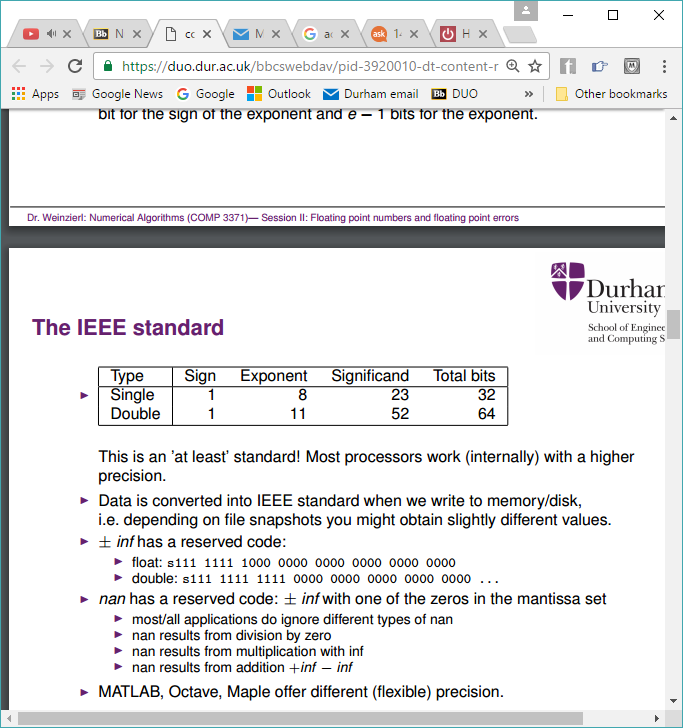
**Numbers in computing**All numbers used in this course come from the set of real numbers. This includes floating point numbers. In a simulation, we must be very careful when dealing with floating point numbers. For example, if there is a simulation in which forces are acting on particles, if two forces of equal magnitude are acting on a particle in opposite directions, the particle should not move. But of course some floating point numbers are irrational – have infinite decimal places. If they are not handled correctly, one force may just be greater than the other because of rounding the floating point number, which causes the simulation to be wrong.

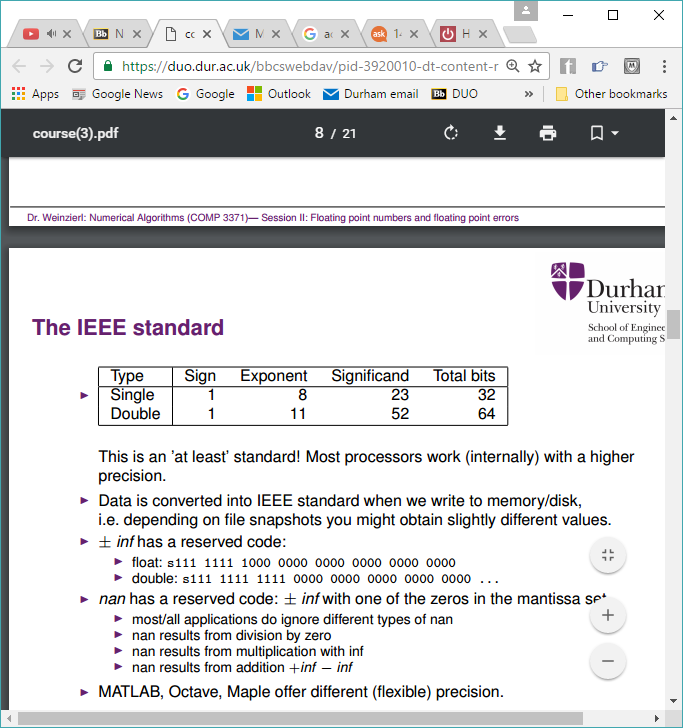
**Floating point notation**A dynamic number notation (i.e. with a header for example) cannot be fast. Hardware needs a fixed scheme – floating points. This splits a number into its significant digits, an exponent, and the sign of the number.

Floating point representation has 3 fields:

* The sign bit – a one bit sign (0 is positive, 1 is negative)
* The exponent – the number of positions the radix point is moved to normalise the number. Normalisation is where the radix point is moved to be in the form 1.XXX… so to convert back to the original number, multiply by 2 to the power of the exponent.
* The significant bits/significand/mantissa) – The actual bits that come after the radix point in the normalised number. Because the normalised number always starts with 1, it does not need to be stored.

Can be described by the formula:

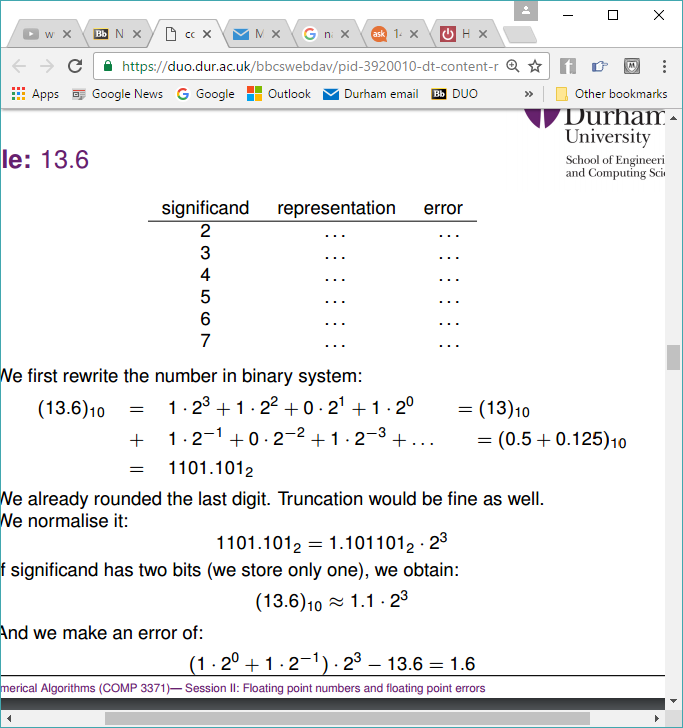
The IEEE standard gives the number of bits used in each part of the floating point representation for various precisions of number:

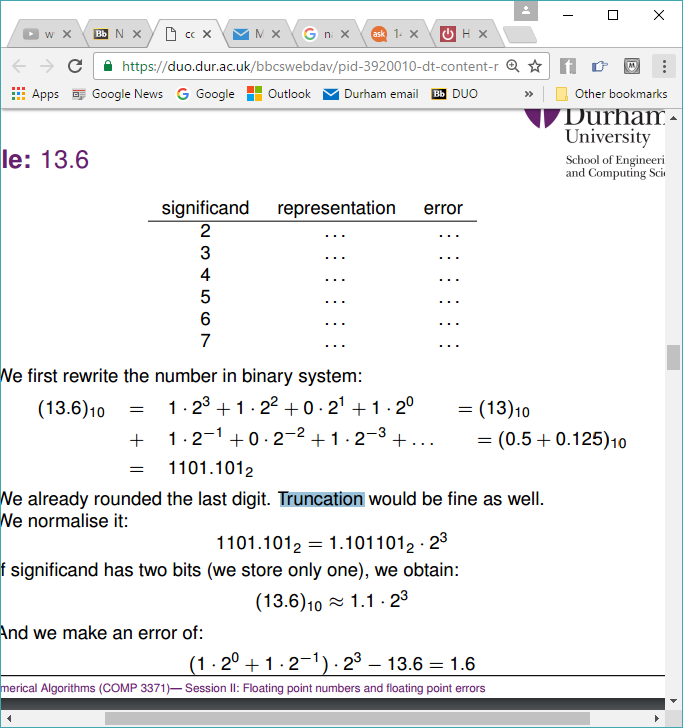
This is “at least” standard. Most processors work (internally) with a higher precision. Data is converted to this standard when written to memory/disk.

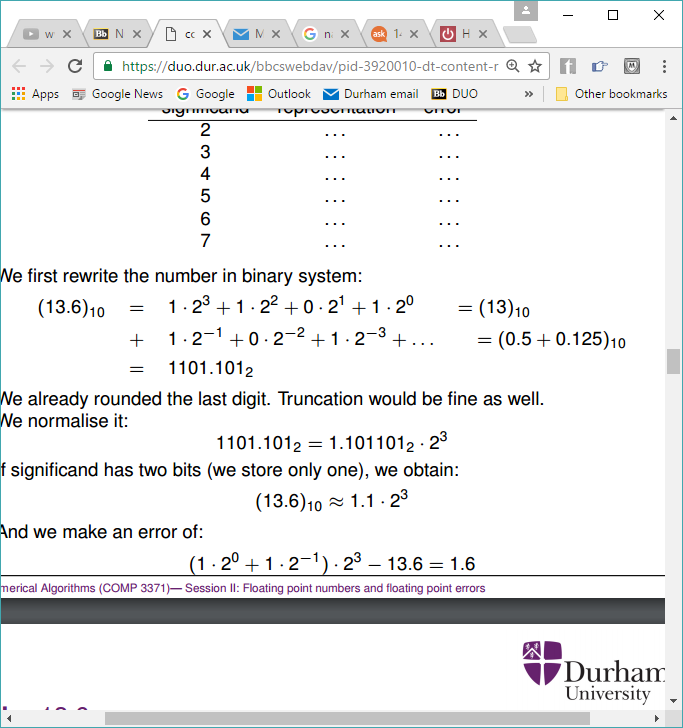
+/- infinity has a reserved code:

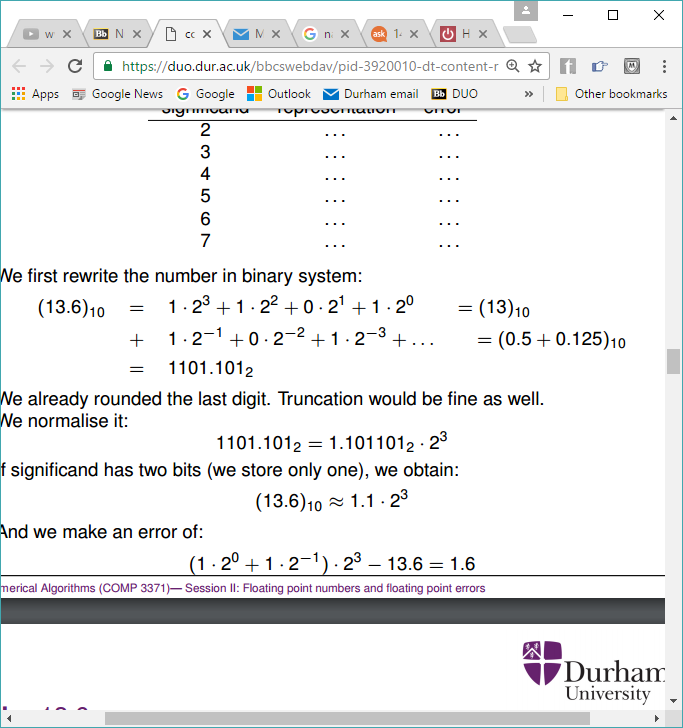
NaN (not a number) represents an undefined or unrepresentable number. It results from division by zero, multiplication with infinity, of from addition +inf – inf. Its reserved code is +/- inf with one of the zeroes in the mantissa set.

Example of convertinf to floating point representation:

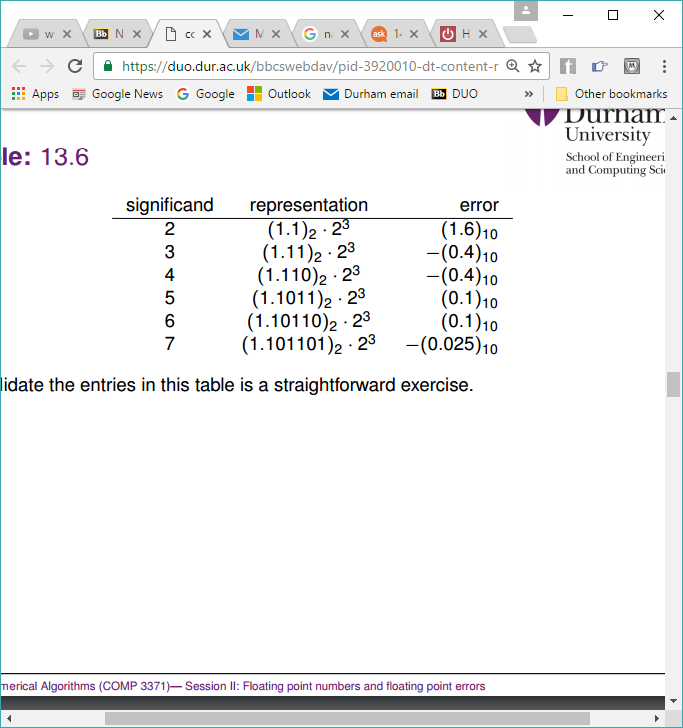
Rewrite in binary (if not already):

The last digit has been rounded. Could also truncate (limit number of digitsa after radix point). Number is then normalised:

Depending on the precision of the format, the length of the significand is altered. So if the specified precision is 2, we have the 1 before the radix point, and the first digit from the significand:



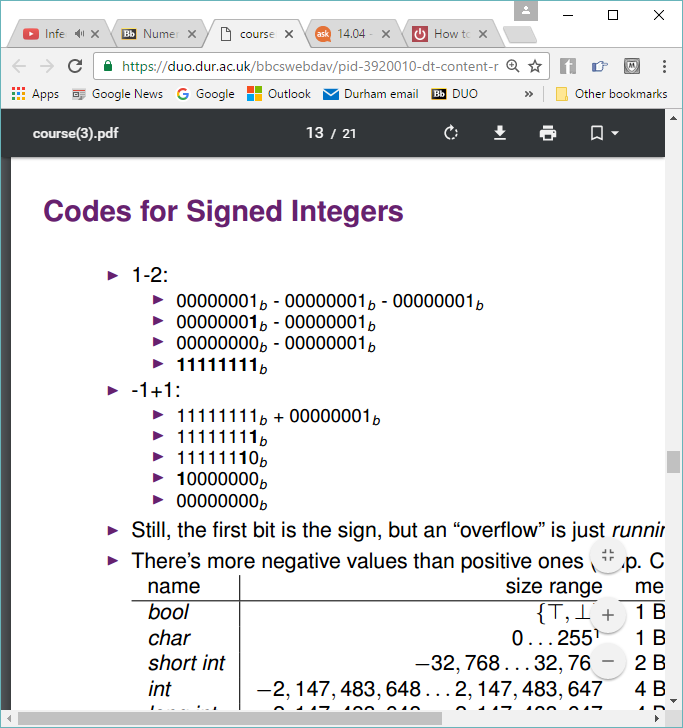
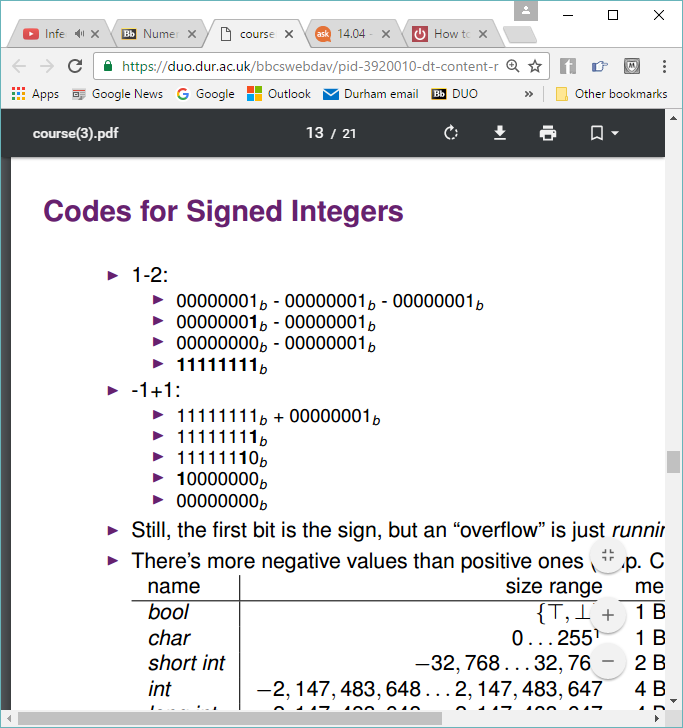
Hence there is an errror of:

Clearly the longer the significand, the less the error:

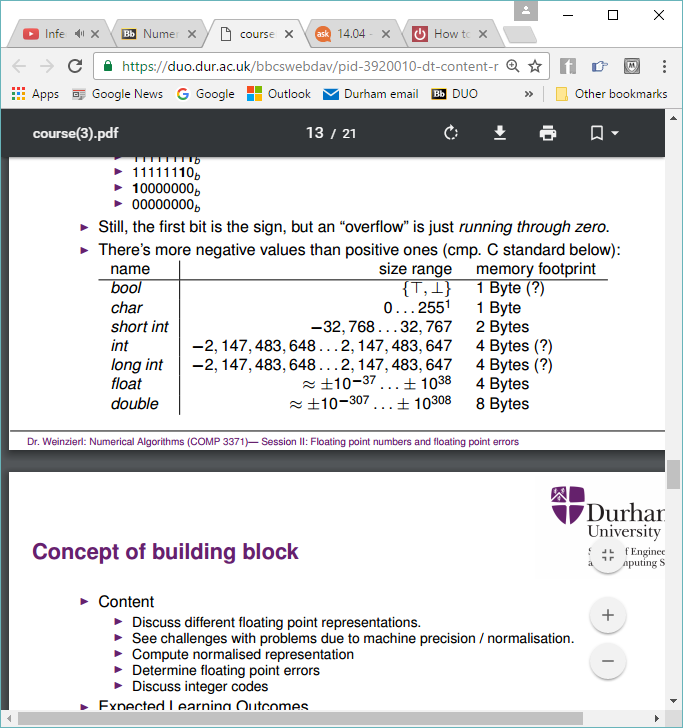
**Overflows**All interpreted languages (like Java and C#) check for overflows. C++ does not.

For floating point numbers, overflows are identified in the exponent and lead to +-inf.  
For floating point umbers, underflows lead to denormalised numbers i.e. they can not be represented as normalised numbers anymore. Processors can be configured to work with denormalised numbers or to avoid these by casting these numbers to zero.

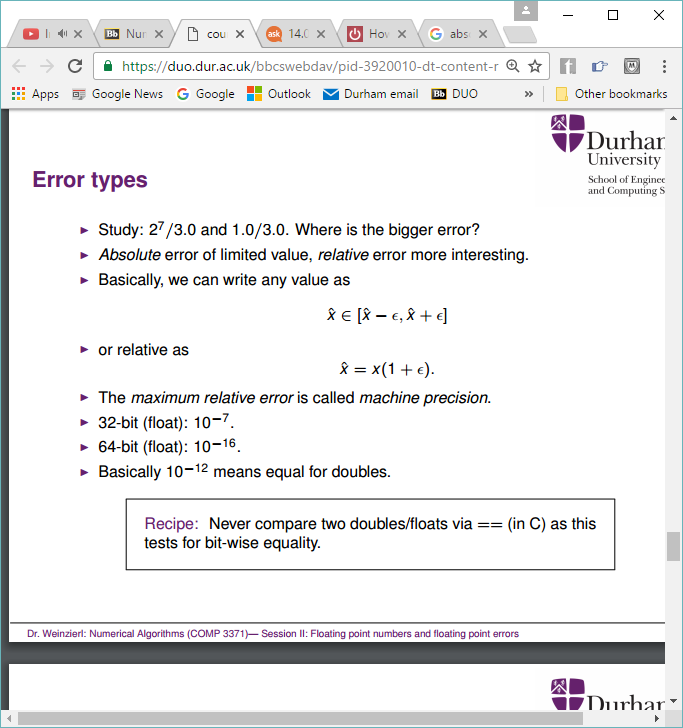
**Signed integers**In signed integers, the first bit represents the sign. This is because for a number that is n bits long, there are 2n possible values, half of which start with a 0, half start with a 1. To prevent overflow and to enable easy addition when going from negative to positive, the smallest 2n-1 represent the positive numbers, and the rest are the negative numbers. This allos for arithemetic such as:

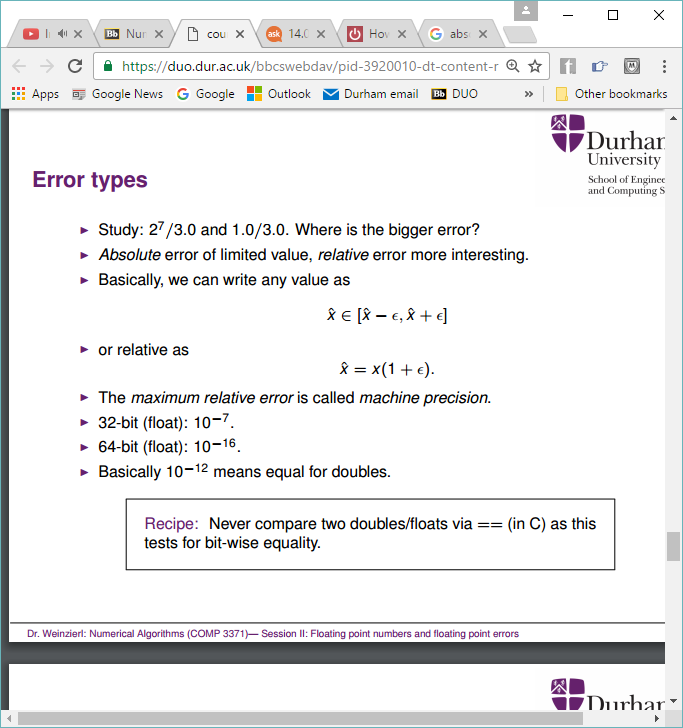


So effetively “overflow” is just running through zero.

There are more negative values than positive values:

**Error types**Absolute error is the magnitude of the difference between the exact value and the approximation. In terms of approximating values (e.g. 1.0/3.0), this has limited value.

Relative error is the absolute error divided by the magnitude of the exact value. This is more interesting.

Basically, any value can be written as:

Or relative as:

The maximum relative error is called machine precision. For 32-bit float this is 10-7 , for a 64-bit float this is 10-6.

Never compare two doubles/floats with == (in C) as this tests for bit-wise equality

Error example:

P2 – 2q2 with p = 665857, q = 470832

P2 = 4.43365x1011

2q2 = p2 – 1

Using double precision the answer is 1. Using float precision the answer is 0. As can be seen, it is not a great idea to subtract two values of similar size from each other. This subtraction of nearly 2 equal numbers such that the number of accurate digits is too low/zero is known as cancellation.