# LECTURE I:
# Basics (floating-point representation and round-off errors)

Let us start with the following, simple example

$$y = x + a$$
$$z = x - a$$

What is $y - z$ and does it depend on $x$? This is easy to compute

$$y - z = (x + a) - (x - a) = 2a.$$

Hence, the final result $2a$ is independent of $x$. But does this hold in all cases, even when we compute things numerically?

```c
#include <stdio.h>
int main() {
        double x = 1125899973951488;
        double a = 1 ;

        float y, z;
        y = x + a;
        z = x - a;

        printf(”%f \n”, y-z);
}
```

gives 134217728.000000 ... which clearly is not what we expected.

## Number Representation

Let us consider the very simple example of a number presented that is given by 5 digits (inlcuding 2 decimals)

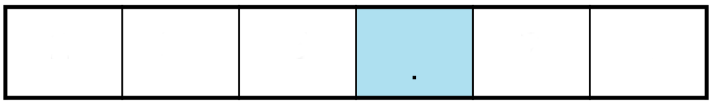

Figure 1: A simple number representation with a total of 5 digits including 2 decimals.

If we want to represent $50\pi = 157.07963...$, we can either use $157.08$ (rounded) or $157.07$ (chopped).

Overall, our representation will allows us to have number ranging between 000.00 and 999.99, no negative number can be represented.

The largest (absolute) error, when representation a number (using rouding), will be 0.005. However, this is only the absolute error, the fractional error is largest for small numbers, e.g., for 0.01 the fractional error is up to 50%. Therefore, one has to be very careful when comparing with zero (for values smaller than 0.005 we can even get larger than 100% relative errors).

To solve some of these issues, the floating-point representation is introduced.

## Floating-point representation

A floating-point representation of a number is given by:

$$\sigma \; m \; b^e \tag{1}$$

with $\sigma$ defining the sign, $m$ being the mantissa/fraction, b the basis of the representation. Note that we are used to think in $b = 10$, but computer use $b = 2$. The exponent is given by $e$.
Our example $50\pi$ gets represented as $\sigma = +$, $m = 1.5708$, $b = 10$, $e = 2$; i.e., $+1.5708 \times 10^2$. In real applications, we will distinguish the following cases:

**single-precision floating-point format (float32)**: This is a number using 32 bits and is often represented as

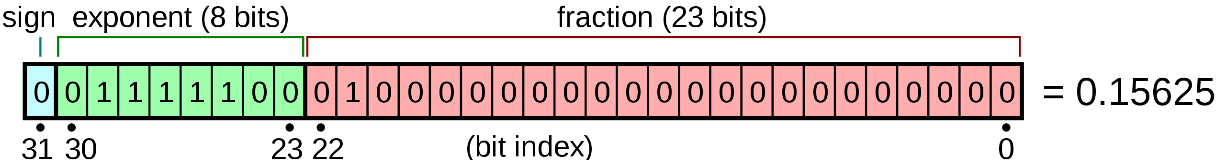

Figure 2: float (single-precision) representation (source: Wikipedia)

**double-precision floating-point format (float64)**: This is a number using 64 bits and is often represented as

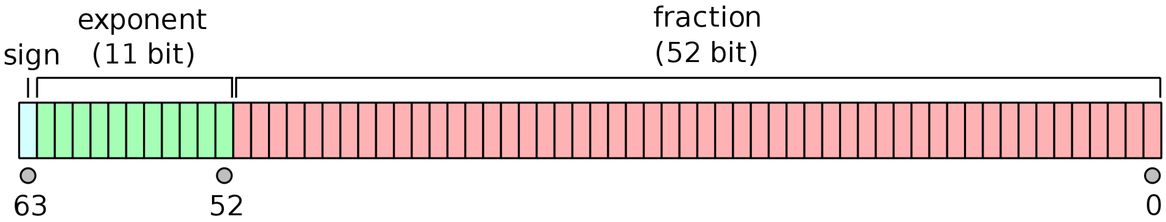

Figure 3: Double (double-precision) representation (source: Wikipedia)

**quadrupole-precision floating-point format (float128)**: This is a number using 128 bits and is often represented as

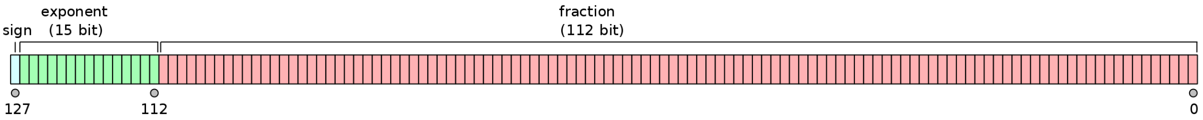

Figure 4: Quadrupole-precision representation (source: Wikipedia)

Note: If you use a normalized representation, then the first bit has to be non-zero. This way, you don't need to store this bit, since you know it is 1. This gives you an addition, free, ”hidden” bit.

Furthermore, we also have the following data types (you can use them, but I suggest that you don't do it!):

- INF: infinity
- - INF: negative infinity
- NaN: Not a Number

## Floating-point operations

For simplicity, we will assume here that we have a ‘simple’ representation with sign (1), exponent (1), and fraction (5); no hidden bit.

### Addition & Subtraction:

$$\begin{aligned}
123453.7 + \quad & 104.7654 = 123558.4654 = 1.2356 \cdot 10^5 \\
1.234537 \cdot 10^5 + \quad & 1.047654 \cdot 10^2 \\
1.2345 \cdot 10^5 + \quad & 1.0477 \cdot 10^2 \\
1.2345 \cdot 10^5 + \quad & 0.0010477 \cdot 10^5 \\
1.2345 \cdot 10^5 + \quad & 0.0010 \cdot 10^5 = 1.2355 \cdot 10^5
\end{aligned}$$

### Multiplication:

$$4734.612 \times 541724.2 = 2564853898.0104 = 2.5649 \cdot 10^9$$
$$4.7346 \cdot 10^3 \times 5.4171 \cdot 10^5 \to 25.64827512 \cdot 10^8 \to 25.648 \cdot 10^8 \to 2.5648 \cdot 10^9$$

## ‘Real-world’ applications

We will consider 3 different examples:

- part 1: compute $x^n - 1/(1/x^n)$
- part 2: compute $\frac{1}{1+x^n-\frac{1}{1/x^n}} - 1$
- part 3: computation of the power-law decay rates when solving the Teukolsky Equation

### Part 1

```
#include<stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{

   double in1, res, res2;
   int in2;

   sscanf(argv[1],”%lf”,&in1);
   sscanf(argv[2],”%d”,&in2);

   res = in1;
   res2 = 1./in1;


```

```c
  for (int i = 0; i <in2; ++i)
  {

    res   *= in1;
    res2  *= 1./in1;

    printf("%f^%d = %le \n", in1, i+2, res);
    printf("error: %le \n", res-1./res2);
    printf("rel. error: %le \n \n", (res-1./res2)/res);
  }

  return 0;
}
```

We find that the absolute error increases very slowly, but the ‘relative error’ $(A - A')/A$ with $A = x^n$ and $A' = -1/(1/x^n)$ increases noticeably.

**Part 2**

```c
#include<stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{

  double in1;
  int in2;
  double res, res2, erg;


  sscanf(argv[1],"%lf",&in1);
  sscanf(argv[2],"%d",&in2);

  res  = in1;
  res2 = 1./in1;

  for (int i = 0; i <in2; ++i)
  {

    res   *=  in1;
    res2  *=  1./in1;
    erg    =  1./(1.+(res-1./res2))-1.;

    printf("%d:     %le \n", i , erg);
  }

  return 0;
}
```

In slight contrast with the previous example, we find that the ‘relative error’ $(A - A')/A$ with $A = \frac{1}{1+x^n-\frac{1}{1/x^n}}$ and $A' = 1$ increases noticeably.

**Part 3**

This work is one of the few examples, where a numerical code requires quadrupole precision to provide correct results. Without introducing quadrupole precision, it was possible to compute something like power-law tails, but these were dominated by round-off errors and completely wrong.

However, it is worth pointing out that changing a code to quadrupole precision is a hard endeavor, in particular for already long and complex codes. For example, it was planned

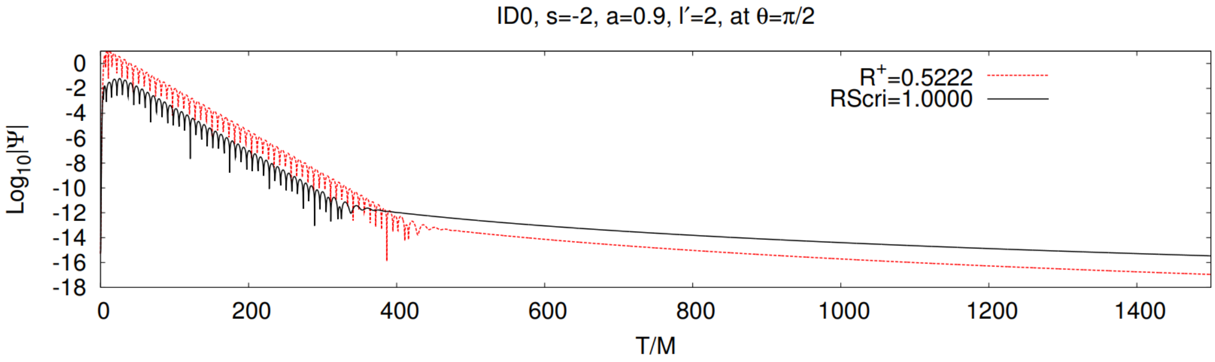

Figure 5: Quasi-normal mode ringdown and power-law tail of perturbed black hole; (Harms et al., arxiv: 1301.1591).

in the SpEC code to switch to quadrupole precision for the simulation scalar fields on a Kerr Spacetime background, however, it was not possible (even after several attempts) to substitute all doubles with quadrupole precision.

## Possible problems with floats

**Loss of significance:** An operation on two numbers which increases the relative error substantially, e.g., subtracting two nearly equal numbers.
One possibility of avoiding this is to use:

$$x - y = \frac{(x-y)(x+y)}{(x+y)} = \frac{x^2 - y^2}{x+y},$$

e.g., for small $\delta$

$$1 - \sqrt{1-\delta} = \frac{\delta}{1+\sqrt{1-\delta}}$$

**Absorption:** Addition or subtraction of two numbers of very different size. In this case the larger number is not changed by the smaller number and accuracy is lost.

**Arithmetic underflow:** Arithmetic underflow occurs when the result of an operation is smaller (in magnitude) than the smallest value representable as a normal floating point number in the target datatype.

**Violation of the associative and distributive property:** Note that the associative law $(x+y)+z \neq x+(y+z)$ and the distributive law $x \cdot (y+z) \neq (x \cdot y) + (x \cdot z)$ do not hold.

**Conversion:** Issue when representing numbers in human-readable format (i.e., using basis 10).

**Representation:** Often already simple numbers such as 0.1 can not be presented to full precision when using binary format.

### Equalities/Inequalities

```
#include <stdio.h>
```

```

int main ( void ) {
  if (0.362 * 10.0 == 3.62)
    printf(”True \n ”);
  else
    printf(”False \n”);


  if (0.362 * 100.0 == 36.2)
    printf(”True\n”);
  else
    printf(”False\n”);

  if (0.362 * 100.0/100.0 == 0.362)
    printf(”True\n”);
  else
    printf(”False \n”);


  return 0;
}
```

Despite the fact that all equalities seem to be true, this is not the case. Equalities are not properly interpreted due the reformulation into binary format.

Another possible issue might arise if you compare $+0$ and $-0$, which indeed is considered to be equal, but if you use this inside a substitution, this could lead to $\frac{1}{\pm 0} = \pm\infty$, where $+\infty$ and $-\infty$ are not equal.