Advanced Programming for Scientific Computing (PACS)
Lecture title: A brief introduction to floating point numbers

Luca Formaggia

MOX
Dipartimento di Matematica
Politecnico di Milano

A.A. 2022/2023

Floating point number system

A normalized floating point number system F is formed by *zero* and real numbers of the form

$$y = \pm m \times \beta^{e-t}$$

where

β	basis (typically 2)		t	precision
m	mantissa, $\beta^{t-1} \leq m < \beta^t$		e	exponent, $e_{min} \leq e \leq e_{max}$

Or,

$$y = 0.d_1 d_2 \dots d_t \times \beta^e \quad d_1 \neq 0, \quad 0 \leq d_i < \beta$$

we use all the space for the digits, we dont i.e. lose precision

All floating point numbers representable in a computer belong to $F^* = F \cup F_s$ where F_s is the set of *subnormal numbers* of the form

$$y = \pm m \times \beta^{e_{min}-t}, \quad 0 < m < \beta^{t-1}$$

The values of t, e_{min} and e_{max} characterize the type of floating point number and is defined by the IEEE 754 standard.

Round off

A real number $x \in \text{range}(F^*)$ is approximated in a computer by $\hat{x} = fl(x) \in F^*$ and $e_r = (x - \hat{x})/x$ is the (relative) rounding error.

If $x \in \text{range}(F^*)$ then $\hat{x} = fl(x) \in F^*$ and $e_r = 0$. In general,

$$|e_r| \leq u = \frac{1}{2}\beta^{1-t}$$

where u is the roundoff unit. The machine epsilon ϵ_M is the smallest positive floating point number by which $fl(1 + \epsilon_M) \neq 1$. We have:

$$u = \frac{1}{2}\epsilon_M$$

IEEE arithmetic

The IEEE 754 has been defined in 1985 (and amended various times since then) and defines the floating point arithmetic system normally implemented in modern processors. There are two main types of floating point numbers

Type	Size	t	e	u	Range
float	32	$23+1$	8	2^{-24}	$10^{\pm 38}$
double	64	$52+1$	11	2^{-53}	$10^{\pm 308}$

The value in the table indicate the number of bits used. Moreover, the implementation of IEEE arithmetic system should satisfy the standard model: if $x, y \in F$ then — ie x and y are real ($\mathbb{R}$) numbers

$$fl(x \,\mathrm{op}\, y) = (x \,\mathrm{op}\, y)(1+\delta) \quad \boxed{|\delta| \leq u,} \quad \mathrm{op} = + \; - \; \times \; /$$

$\Rightarrow$ we have a control about the precision in the computation

Special numbers

The IEEE standard prescribes that

$$fl(x) = 0 \quad \text{if } |x| < F_{min} \quad \text{(UNDERFLOW)}$$

$F_{min} \in F^*$ being the smallest positive floating point number.

$$fl(x) = \text{sign}(x) Inf \quad \text{if } |x| > F_{max} \quad \text{(OVERFLOW)}$$

$F_{max} \in F$ being the maximum floating point number.
Moreover, $x/0 = \pm Inf$ if $x \neq 0$, $Inf + Inf = Inf$ and $x/Inf = 0$ if $x \neq 0$.

Finally, the special number NaN (Not-a-Number) indicates the result of an invalid operation ($0/0$, $\log(-1)$ etc) and we have $x \,\text{op}\, NaN = NaN$, $Inf - Inf = NaN$ e $Inf/Inf = NaN$.

Floating point exceptions

The standard prescribes that in normal situations an invalid operation or an over/underflow do not stop computations, but produces the special numbers illustrated before.

However, the processor may record if an invalid floating point operation (normally called floating point exception) has occurred, so that the user may trap it.

We will discuss this issue into more detail later in the course, showing how you can capture floating point exceptions.

Forward and backward error

Let $f : \mathbb{R} \to \mathbb{R}$ and $\hat{f} : F \to F$ the corresponding expression on floating point numbers. Let $y = f(x)$ e $\hat{y} = \hat{f}(\hat{x})$. The analysis of the forward error aims to find a δ such that

$$e_f = \frac{|y - \hat{y}|}{|y|} \leq \delta$$

The relative *backward error* Δ is defined by $\hat{y} = f(x(1 + \Delta))$. If $f \in C^2$ we have that

$$\frac{|y - \hat{y}|}{|y|} = c(x)|\Delta| + O(\Delta^2)$$

with $c(x) = |xf'(x)/f(x)|$ called the relative condition number of f. In general, one looks for an estimate such that

$$e_f \leq C(x)|\Delta|$$

A simple example: cancellation

Let us consider $y = f(a, b) = a - b$ e $\hat{y} = f(a(1+\Delta), b(1+\Delta))$. It is easy to find out that

$$|\frac{y-\hat{y}}{y}| = \frac{|a\Delta - b\Delta|}{|a-b|} \leq \frac{|a|+|b|}{|a-b|}|\Delta| \Rightarrow C = \frac{|a|+|b|}{|a-b|}$$

We have that $C \to \infty$ when $|a-b| \to 0$: the subtraction of almost equal floating point values causes a big round-off error. We should avoid it whenever possible!

A thorough analysis of floating point errors for common mathematical operations is found in the book by N.J. Higham Accuracy and stability of numerical algorithms, SIAM, ISBN: 978-0-89871-521-7, 2002.

An example: numerical differentiation

If $f(x) : \Omega \to \mathbb{R}$ is sufficiently regular we have

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + O(h^2), \quad \forall x \in B_h(x).$$

Therefore the centered formula $\delta_f(x) = (f(x+h) - f(x-h))/2h$ is a second'order approximation of the derivative. However, the quantity actually computed is

$$\widehat{\delta}_f(x) = \frac{f[(x+h)(1+\Delta_1)] - f[(x-h)(1+\Delta_2)]}{2h},$$

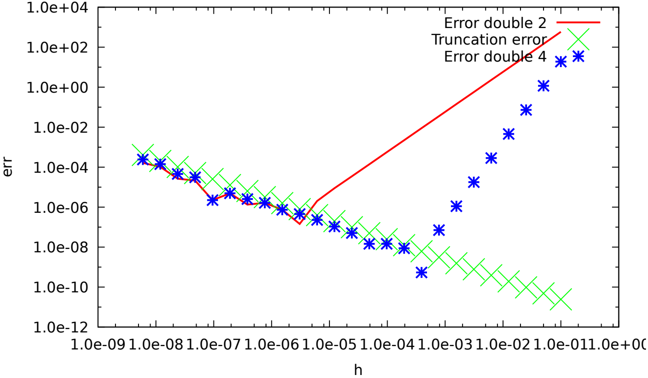
with $|\Delta_1| \leq u$ and $|\Delta_2| \leq u$. If $f'(x) \neq 0$ we may obtain the estimate

$$|\widehat{\delta}_f(x) - \delta_f(x)| \leq \frac{C}{2h}$$

with $C \leq |x||f'(x)|u$.

In the example Examples/src/FloatingPoint/FinDiff we also implement the fourth order formula $(\frac{1}{12}f(x-2h)-\frac{2}{3}f(x-h)-\frac{1}{12}f(x+2h)+\frac{2}{3}f(x+h)/h$ and we compute the numerical derivative of $100e^x$ at the point $x=3$ with the two formulas and plot the error $|\widehat{\delta_f}(x)-f'(x)|$ for different values of h, together with the estimated forward truncation error. We repeat the example using single, double and extended precision

The result for double precision



You may note that for $h \lesssim 10^{-6}$ the truncation error dominates the second order formula (Error double 2), while it already spoils the result for $h \lesssim 10^{-3}$ if we use the fourth order approximation!

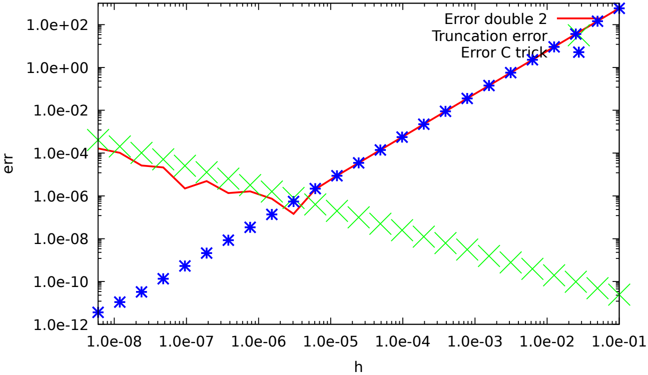
Differentiation with(out) a difference

Sometimes one may use non-standard techniques if a great precision is needed. One may for instance note that if f is extended to a complex function and if the extension is analytic we may write (we need to apply Cauchy-Riemann equation and note that $f(x)$ is real)

$$f(\overbrace{x+ih}^{\substack{\text{complex}\\ \text{variable}}}) = f(x) + ihf'(x) + O(h^2).$$

So $\mathrm{Im}(\frac{f(x+ih)}{h})$ is a second order approximation of $f'(x)$ and its computation does not imply taking differences!.
Indeed the result obtained for small h may be very precise. In FloatingPoint/FinDiff we have implemented also this formula. We show the result.

The result for double precision



You may note that with this trick (blue stars) we can reduce considerably the size of h. For larger h, as expected, we get results similar to the 2nd order formula.

Other nasty and less nasty examples

Some examples of floating point failures are in the FloatingPoint directory of the examples, with a file containing the explanation.

Some of them they are specially hand-crafted examples to highlight some possible unwanted side-effects of floating point operations. Normally the situation is not that bad!

For instance, in FloatingPoint/QuadraticRoot, where we show how the classic formula for the zero of a quadratic $ax^2 + bx + c$ may give incorrect results when $|b| >> ac$, because of cancellation errors.

Finally, in FloatingPoint/FPFailure you find an example that shows some "floating point failures".

Numeric limits

C++ allows you to interrogate the characteristics of the numeric types as implemented in your machine.
Not just floating points, but also integral types. In particular, we may look for the machine epsilon or the maximal representable number.
We need to use the Standard Library header `<limits>`:

```
#include <limits>
float eps=numeric_limits<float>::epsilon();
```

Since S++20 including the header `<numbers>` we have some mathematical constants in C++ style
See a complete example in Examples/src/Numeric_Limits

Beware of floating point comparisons!

Floating points have discrete values. So comparing them can be very critical. The statement

```
double a,b;
... // many computations involving a and b
if(a==b)....
```

is dangerous. Maybe the test is never satisfied. A stupid example

```
 double a = std::pow(3.0,1./5);
 double b = a*a*a*a*a;
 double c =3.0;
if(b==3.0) .. // IS FALSE!
```

It is better to write

```
if(std::abs(a-b)<tol)...
```

where `tol` is a well chosen tolerance (unfortunately it's not always evident what "well chosen" means.)