Lecture 1

Numerical Errors

outline

- numerical methods/algorithms
- basics of computer arithmetics
- numerical errors
- examples:
    - evaluation of function value
    - approximation of derivatives with finite differences

Numerical methods are part of applied mathematics and were intensively developed in the second part of XX century along with fast development of computers. However, some numerical algoritms were known long time before, e.g. numerical solutions of ODE & PDE in XIX century by John Couch Adams (discovery of Neptune) or the Runge-Kutta methods for ODE at the turn of 19th and 20th centuries.

Purposes of numerical methods are finding solutions of mathematically defined problems which are

- hardly solved by other analytical means
- can not be solved analytically
- can not be solved analytically in reasonable time

At present numerical methods are utilized everywhere there is a need of processing large number of data and is driven by growing computational power of digital processors, to name some

- processing sounds (music, voices) and images (photos, video)
  → see for FFT or LAPACK packages installed on your computers
- analysis od statistical data for insurance companies, government, banks, CERN
- optimization of industrial processes: transportation, metallurgy, fabrication of devices
- simulations/modelling of complex physical/chemical/biological systems
- generally as vital tools in science and engineering (both use similar methods but for different purposes)

remark 1: computer simulations/modelling are different class of numerical tasks
like e.g. weather prediction, simulations of classic or quantum dynamics etc.,
namely, they are built as more complex models heavily utilizing
the basic numerical methods at intermediate steps

remark 2: during the course we will talk about the basic numerical methods only

Basically each numerical method (M) transforms an input data vector into output data vector,
which mathematically can be written as follows

$$M : R^n \rightarrow R^m : \quad \vec{d}_{out} = M(\vec{d}_{in})$$

immediately we can pose some questions:

- what do we want to do? → we need a defintion of the task, described by more or less complex mathematical definition/model
- how to do it? → which numerical method can solve the task?
- how do we wish to solve this problem? → which numerical algorithm implementing the chosen method will be optimal? (the most accurate? the most efficient? or something between?)

These general considerations let us to make a distinction between:

- numerical task (how to pose a problem),
- numerical method,
- numerical algorithm (computer implementation of the method)

Numerical task – it is a unique general description of our plans concerning the transformation the input data into the output ones

Numerical method – dedicated to given numerical task is the mathematical model usually built up of a sequence of some mathematical operations (intermediate steps) that must be done in order to find the solution

Numerical algorithm – it is a unique computer implementation of given numerical method, which translates the mathematical model into set of instruction performed by computer usually taking into account some optimization factors and/or specification of computer architecture

Example

- define numerical task: find all zeros of a polynomial

$$P_n(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n \quad (M : R^{n+1} \to R^n)$$

- chose iterative numerical method: bissection, Newton method, secant method
  (from the class of methods dedicated to solving nonlinear equations)

- prepare numerical algorithm (in form of a pseudocode)

```
INPUT:        a⃗ = (a_0, a_1, ..., a_n) ∈ R^{n+1}
OUTPUT:       x⃗ = (empty) ∈ R^n

for k=1 to n do
     define starting point: x_start
     iteratively find k-th zero of P_{n-k+1}(x) : x_k
     add zero to OUTPUT vector: x⃗ = (x_1, ..., x_k, *, ..., *)
     deflate polynomial: P_{n-k+1} = (x - x_k) P_{n-k}(x)
end do

OUTPUT: x⃗ = (x_1, x_2, ..., x_n)
```

Computer arithmetic

Neglecting more complex data structures, we may limit our considerations to three number types:

- logical (bool)
- integer
- real

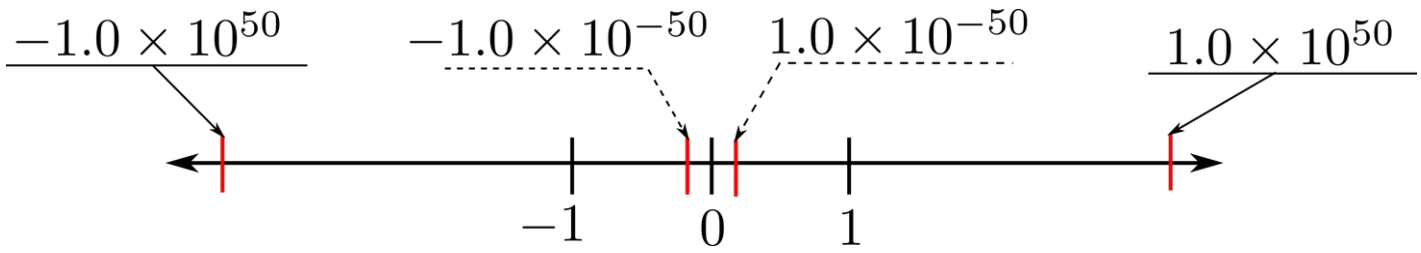
Logical and integer have exact representation in computer memory, so any logical or basic arithmetic operations (excluding division) gives the result of the same type.

The real numbers are represented as floating points in finite-length bit registers.
So only some numbers have exact representations while other are approximated by nearby floating point.

Floating point in scientific notation

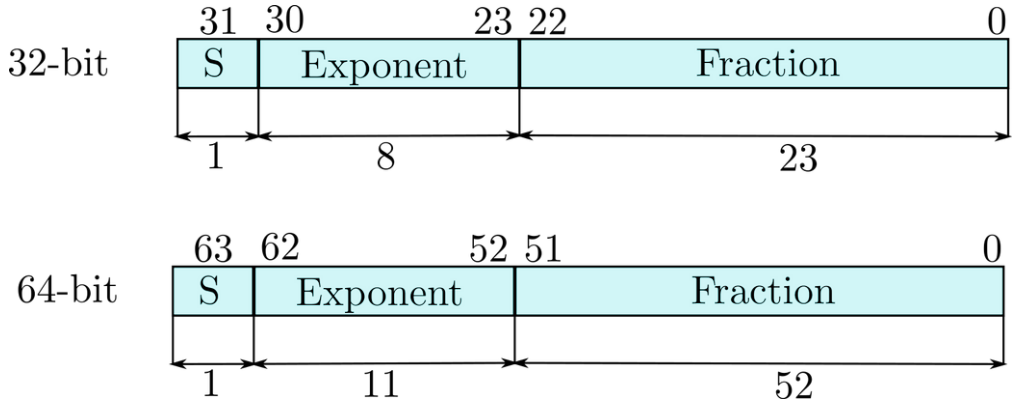
$$Fl(x) = s \cdot 1.F \cdot r^E$$

s - sign
F – fraction
r - radix (base of system, r = 2,8,10,16)
E - exponent



radix – is fixed for given base of the system
Fraction – influence on accuracy of floating-point number,
Exponent – influence on the range of fl number

For each component we must reserve some bits, in computers we use IEEE-754 standard



$type$	$single$	$double$	$quadruple$
$Fraction(bits)$	$23(+1)$	$52(+1)$	$112(+1)$
$Exponent(bits)$	8	11	15
$accuracy$	$2^{-23} \approx 10^{-7}$	$2^{-52} \approx 10^{-15}$	10^{-34}
$range$	$\sim 10^{\pm 38}$	$\sim 10^{\pm 307}$	$10^{\pm 4932}$

(+1) – hidden bit reconstructed when arithmetic operation is conducted

- the use of single precision may easily lead to loss of accuracy and make an algorithm unstable
- double precision gives much better accuracy and enhace stability
- quadruple precison would give the best results, however, because it is used very rarely there are no numerical libraries, moreover on 64-bit computer quadruple precision is ensured by connecting two registers into one what decrease efficency

(similar problem is accounted for double precision computations conducted on desktop GPU Nvidia/AMD 500-600 $, the loss in efficency is 64/16 times with respect to single precision,
while the price of professional GPU with native double precision is about 5,000-6,000 $)

Analysis of rounding errors can be make for simple algorithm, such analysis is based on **Wilkinson lemma**

The rounding errors occuring due to arithmetic operations performed on the floating-point numbers can be equivalently obtain by making these operations in exact arithmetic on slightly perturbed real numbers.

- lets consider 4 basic arithmetic operations:

$$r = x \circ y, \qquad (\circ = +, -, /, *)$$

floating-point arithmetic

$$fl(r) = fl(x) \circ fl(y)$$

how large is the rounding error?

Wilkinson lemma

$$fl(x) = x(1+\varepsilon_x)$$

$$fl(y) = y(1+\varepsilon_y)$$

$$r(1+\varepsilon_r) = x(1+\varepsilon_x) \circ y(1+\varepsilon_y)$$

- it is better to calculate the relative errors – the dependence of error on magnitude of operands disappears

multiplication: $$\frac{fl(x)fl(y) - xy}{xy} = (1+\varepsilon_x)(1+\varepsilon_y) - 1 = \varepsilon_x + \varepsilon_y + \varepsilon_x\varepsilon_y \approx \varepsilon_x + \varepsilon_y$$

division: $$\frac{fl(x)/fl(y) - x/y}{x/y} = \frac{1+\varepsilon_x}{1+\varepsilon_y} - 1 \approx \varepsilon_x - \varepsilon_y$$

addition/subtraction: $$\frac{fl(x) \pm fl(y) - (x \pm y)}{x \pm y} = \frac{x\varepsilon_x \pm y\varepsilon_y}{x \pm y} = \frac{x}{x \pm y}\varepsilon_x - \frac{y}{x \pm y}\varepsilon_y$$

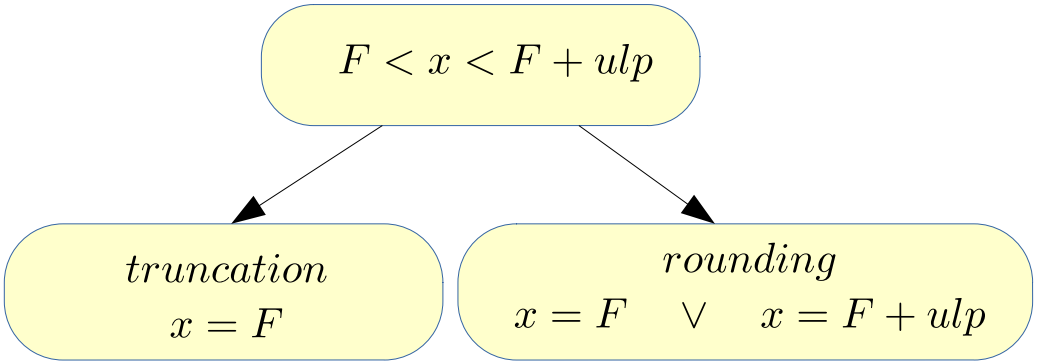
rounding errors would be largely amplified for: $x \approx y \quad \Rightarrow \quad \varepsilon_x - \varepsilon_y$

Conclusion: subtracting large numbers of similar values may generate large rounding errors

Rounding errors

- during the course of sequence of arithmetic operations carried over floating-point numbers we get new numbers which have no floating-point representation before saving them in memory these must be prescribed to one of two neighbours (smaller or greater)

- such projection is done by simple truncating (always shift to smaller value – deficient numbers) or by rounding (both are considered)



ulp – **u**nit in the **l**ast **p**lace,
is the maximum error for stored number
(10^{-7} for single & 10^{-15} for double)

$$\pi = 3.1415927$$

$$\pi = 3.14159265358979\mathbf{3}$$

- each result is perturbed on the scale of present **ulp** value, this loss of information is called **the rounding errors**

- rounding of „real number” is conducted automatically by the compiler directives, these can be made more or less strict increasing computational efficiency in the latter case by the price of less accurate rounding (danger)

  e.g. option: **-ffast-math** for GNU compilers

- it is better to assume the rounding erros are always present in computations and these are the cause which really hinders us from finding the solution

- rounding errors do not cancel mutually but rather accumulate during the course of computation

Truncation errors (due to evaluation of function values)

- Very often in scientific computations we must compute the values of elementary and special functions like e.g.

elementary: $\sin(x),\ \cos(x)\ \tan(x), \sinh(x), \cosh(x), \tanh(x), \ldots$
$arc\sin(x), arc\cos(x), arc\tan(x), arc\sinh(x), arc\cosh(x), arc\tanh(x), \ldots$
$e^x, ln(x), sqrt(x)$

special: $erf(x), J_n(x), Y_n(x), \Gamma(x), \ldots$

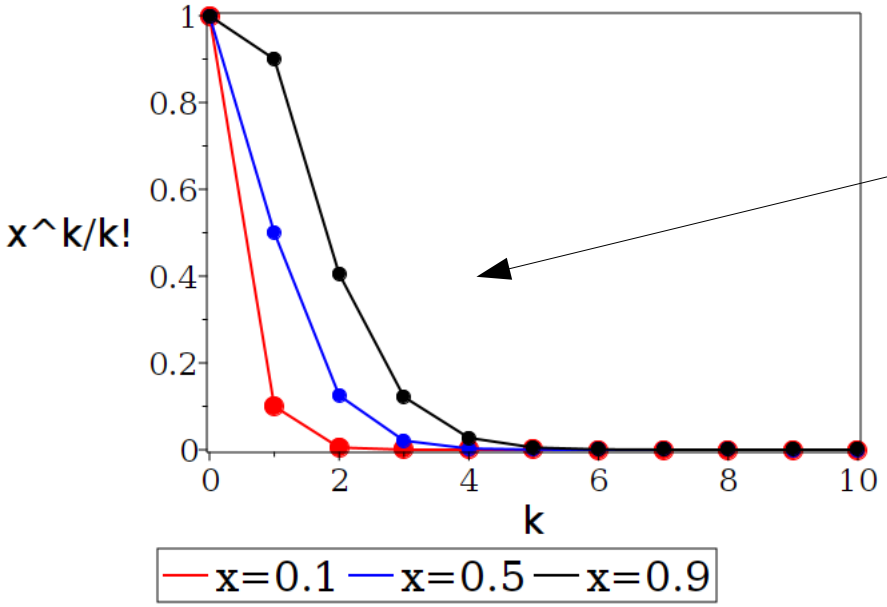
- many of them are provided by the compiler natively, other we must compute oureselves

- all of these functions are connected by common factor, they are usually expressed in form of infinite series e.g. based on Talyor series and usually summation is performed for up to several terms giving approximated value, neglecting the rest terms introduce the truncation errors

Example: compute value of exp(w)

$$exp(w) = ?, \quad w = n + x, \quad n \in Z,\ x \in (0, 1)$$

$$e^w = e^n \cdot e^x = e^n \sum_{k=0}^{\infty} \frac{x^k}{k!}$$
$$= e^n \left(1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \ldots + \frac{x^p}{p!}\right) + R_{p+1}$$



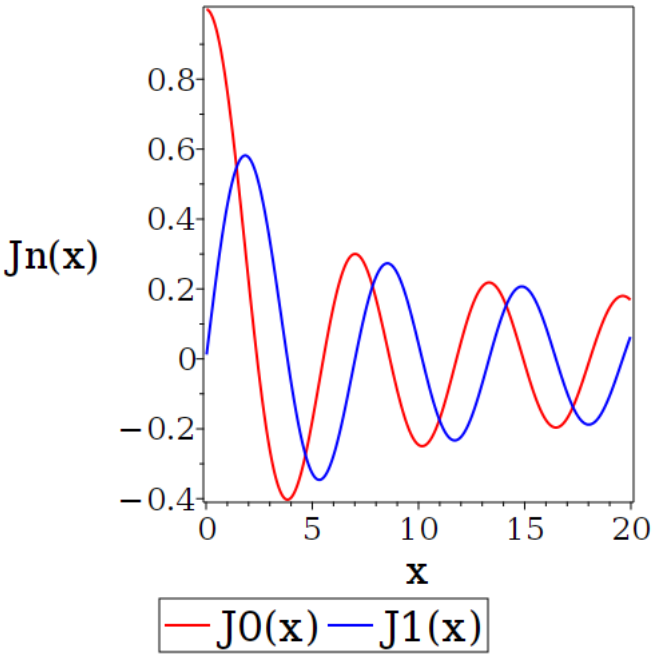
here we get fast convergence, but it is not a rule

Example: value of Bessel function $J_0(x)$

- Bessel functions ($J_n(x)$ & $Y_n(x)$) are solutions of differential Bessel equation

$$x^2y'' + xy' + (x^2 - n^2)y = 0, \quad x \in R, \quad n \in N$$

these functions are used as basis functions to solve more complex differential problems or we need values of $J_n(x)$ since it can be result of some analytical results



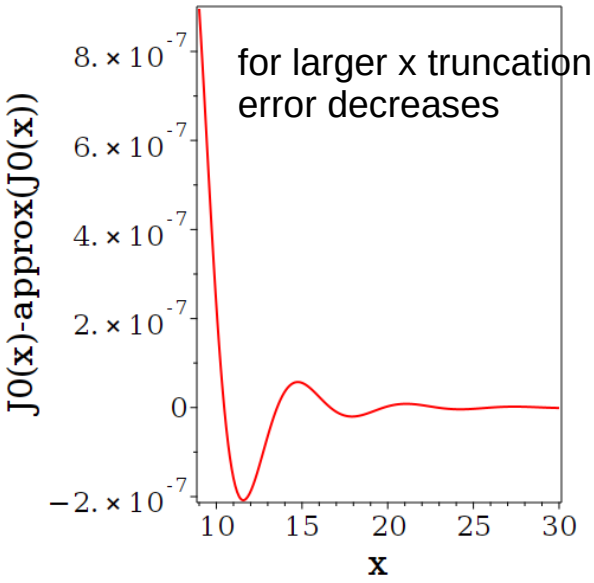
- $J_n(x)$ can be expressed as power series

$$y(x) = J_n(x) = \sum_{m=0}^{\infty} \frac{(-1)^m (x/2)^{n+2m}}{m!(n+m)!}$$

but …. direct use of this series for large x»1 is impractical as summation must include hundreds or so terms

- There are many approches which modify this series into more compact expression, one of them is the following (x»1)

$$\tilde{J}_0(x) = \sqrt{\frac{2}{\pi x}} \left(1 - \frac{1}{16x^2} + \frac{53}{512x^4}\right) \cos\left(x - \frac{\pi}{4} - \frac{1}{8x} + \frac{25}{384x^3}\right)$$



Finite differences and truncation errors

- In computational physics we are often faced the problem defined in language of differential equations ODE/PDE

Newton's equations of motion: $$F = \frac{dp}{dt} = m\frac{d^2x}{dt^2} \quad \Rightarrow \quad x(t) = ?$$

Schrodinger equation: $$i\hbar\frac{\partial\psi}{\partial t} = -\frac{\hbar^2}{2m}\frac{\partial^2\psi}{\partial x^2} + U(x,t)\psi \quad \Rightarrow \quad \psi(x,t) = ?$$

Poisson equation: $$\frac{d^2V}{dx^2} = -\frac{1}{\varepsilon_0\varepsilon}\rho(x) \quad \rightarrow \quad V(x) = ?$$

- Our aim is then by first translate differential equations into other form (usually algebraic), and secondly, solve the transformed problem numerically.

- the simplest method rely on replacing the derivatives with their approximations: **finite differences** which are get after some manipulations of Taylor series

$$f(x \pm h) = \sum_{k=0}^{\infty} (\pm 1)^k \frac{d^k f(x)}{dx^k}\frac{h^k}{k!}, \quad h << 1$$

In the following we use an abbrieviation fo k-th order derivative

$$f^{(k)}(x) = \frac{d^k f(x)}{dx^k}$$

and write down the Taylor series for f(x±h)

$$f(x+h) = f(x) + f^{(1)}\frac{h^1}{1!} + f^{(2)}\frac{h^2}{2!} + f^{(3)}\frac{h^3}{3!} + f^{(4)}\frac{h^4}{4!} + O(h^5) \qquad (1)$$

$$f(x-h) = f(x) - f^{(1)}\frac{h^1}{1!} + f^{(2)}\frac{h^2}{2!} - f^{(3)}\frac{h^3}{3!} + f^{(4)}\frac{h^4}{4!} + O(h^5) \qquad (2)$$

We see that by combining (1) and (2) expressions we may express first or second derivative by the sum of other terms.

calculating [(1)-(2)]/2h gives 1-st derivative

$$f^{(1)} = \frac{f(x+h) - f(x-h)}{2h} - f^{(3)}\frac{h^2}{3!} + \dots$$

$$f^{(1)} \approx \frac{f(x+h) - f(x-h)}{2h} + O(h^2)$$

1-st order finite difference

truncation error

calculating [(1)+(2)]/h² gives 2-nd derivative

$$f^{(2)} \approx \frac{f(x-h) - 2f(x) + f(x+h)}{h^2} - f^{(4)}\frac{h^2}{12}$$

$$f^{(2)} \approx \frac{f(x-h) - 2f(x) + f(x+h)}{h^2} + O(h^2)$$

2-nd order finite difference

truncation error

- The lowest 1-st and 2-nd order symmetric finite diffreneces approximate derivatives with maximal truncation error of order 2 because the term h^2 has the largest contribution to truncation.

- Do we get more accurate approximations?

  Yes, but we must involve more information about the behaviour of a function around the central point x, we need Taylor series for f(x ± 2h)

$$f(x+h) = f(x) + f^{(1)}\frac{h^1}{1!} + f^{(2)}\frac{h^2}{2!} + f^{(3)}\frac{h^3}{3!} + f^{(4)}\frac{h^4}{4!} + O(h^5) \quad (1)$$

$$f(x-h) = f(x) - f^{(1)}\frac{h^1}{1!} + f^{(2)}\frac{h^2}{2!} - f^{(3)}\frac{h^3}{3!} + f^{(4)}\frac{h^4}{4!} + O(h^5) \quad (2)$$

$$f(x+2h) = f(x) + f^{(1)}\frac{2h^1}{1!} + f^{(2)}\frac{4h^2}{2!} + f^{(3)}\frac{8h^3}{3!} + f^{(4)}\frac{16h^4}{4!} + O(h^5) \quad (3)$$

$$f(x-2h) = f(x) - f^{(1)}\frac{2h^1}{1!} + f^{(2)}\frac{4h^2}{2!} - f^{(3)}\frac{8h^3}{3!} + f^{(4)}\frac{16h^4}{4!} + O(h^5) \quad (4)$$

8(1-2)-(3-4)/12h gives 4-th order finite difference for $f^{(1)}$

$$f^{(1)} \approx \frac{f(x-2h) - 8f(x-h) + 8f(x+h) - f(x+2h)}{12h} + O(h^4)$$

16(1+2)-(3+4)/12h^2 gives 4-th order finite difference for $f^{(2)}$

$$f^{(2)} \approx \frac{-f(x-2h) + 16f(x-h) - 30f(x) + 16f(x+h) - f(x+2h)}{12h^2} + O(h^4)$$

Stability of numerical method/algorithm

We say the numerical method/algorithm is stable if the norm of the output data vector is finite, otherwise it is unstable

$$M : R^n \to R^m, \quad \vec{d}_{out} = M(\vec{d}_{in}), \quad \|\vec{d}_{out}\| < \infty$$

Stability is an inherent feature of the method, it depends on:

- the type of transformation,
- input data,
- numerical errors

For simple methods their stability can be assessed theoretically by means of **numerical analysis**.
Stability is the basic prerequiste feature we require from the numerical method.

Accuracy

From numerical method/algorithm we anticipate to provide „numerically accurate" results
i.e. the numerical outcomes (in floating point arithmetic) should differ from the exact ones „only slightly"

$$\text{exact arithmetic :} \quad \vec{d}_{out} = M(\vec{d}_{in})$$

$$\text{inexact arithmetic :} \quad Fl\left\{\vec{d}_{out}\right\} = Fl\left\{M(\vec{d}_{in})\right\}$$

$$\left\|\vec{d} - Fl\left\{\vec{d}_{out}\right\}\right\| \approx ulp$$

simetimes it happens, but not very often, therefore we shall carefully check correctness of numerical results

stability & accuracy – final remarks

- both, accuracy and stability may strongly depend on numerical errors due to large sensitivity of the intermediate results on unavoidable perturbations, such behaviour becomes typical for increasing number of processed data

- general recommendation is to use the stronger arithmetic (if possible): 32 bit → 64 bit → 128 bit (?) ….

- don’t forget: in scientific and engineering computations the double precision is preferred

- if the given method can not guarantee required accuracy/stability in double precision then

  1) try to change the method/algorithm (if possible, it is the fastest way)

  2) change the mathematical model (the major change), it is time-consuming but may help