Introduction to numerical Methods

Marcin Chrząszcz, Danny van Dyk
mchrzasz@cern.ch,
danny.van.dyk@gmail.com

University of Zurich[UZH]

Numerical Methods,
19 September, 2016

Course plan

$\Rightarrow$ The course consists of 2 hours lecture and 2 hours of exercise.

$\Rightarrow$ TA will explain to you how you will get points for the class exercise (you will be coding the algorithms) described on the lecture and applying them to some problems.

$\Rightarrow$ After passing the class exercise ($> 66\%$ of points) you will have an exam (oral).

$\Rightarrow$ The final mark:

$$\text{mark} = 0.5 \, \text{exam} + 0.5 \, \text{class}$$

$\Rightarrow$ The final exam will be average of exam and class marks.

Mark	Range
6	$84 - 100\ \%$
5	$67 - 83\ \%$
4	$50 - 67\ \%$
3	$33 - 50\ \%$
2	$16 - 33\ \%$
1	$0 - 16\ \%$

Course plan

1. Numerical precision, floating point representation.
2. Numerical stability.
3. Function interpolation, multidim. function interpolation.
4. Function approximation.
5. Linear equation solving with elimination methods and iteration methods.
6. Non-linear system of equation solving.
7. Numerical integration.
8. Differential equation eq. solving.
9. Chaos theorem.

Why do we need numerical methods?

Prose:

⇒ Most of the problems that one tackles on daily basis cannot be computed analytically! During the university studies the problems you were solving were tailor suited to be analytically solvable!

⇒ Even if the problem is solvable analytically in daily work one needs to repeat the calculations many times changing some conditions. Solving n^{th} time a similar integral on paper can have deadly consequences on your sanity.

⇒ Let's face is computers are just much much faster then us in this kind of computations.

Cons:

⇒ One disadvantage of numerical methods is the fact that they don't give you an exact solution. The reason for that is that the computers don't operate on real numbers but just on their representation which we will learn during today's lecture.

Errors, fast reminder

⇛ As you already know there are two types of errors:

- Absolute error Δ.
- Relative error δ.

$$\Delta = |A - a| \qquad \delta = |\frac{A - a}{A}|, \quad A \neq 0,$$

where A is the exact number, a its approximation.

Errors, fast reminder

$\Rightarrow$ As you already know there are two types of errors:

• Absolute error Δ.

• Relative error δ.

$$\Delta = |A - a| \qquad \delta = |\frac{A - a}{A}|, \quad A \neq 0,$$

where A is the exact number, a its approximation.

$\Rightarrow$ Now since we know what are errors lets look into the sources of errors:

$\Rightarrow$ Input errors.

$\Rightarrow$ Cut-off errors.

$\Rightarrow$ Rounding errors.

Input Errors

$\Rightarrow$ Input errors are errors that are associated with the inputs to the computer. We can have different types of this errors:

- Mathematical model errors. A canonical example of such simplification is the pendulum, where we make an approximation of $\sin x = x$ for small x:

$$l\frac{d^2x}{dt^2} + gx = 0$$

- Errors associated to a given numerical algorithm: a choice of a numerical algorithm is indeed an very important step of solving a given problem. As we will see during the course the simplest models that are fast to implement to execute on the machine can lead to large numerical errors.
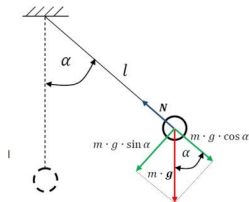
- Input data errors. For many computations one needs tome inputs: constants, starting points, etc. There are errors that are associated to those as well. This kind of errors are super the easiest to control and usually are negligible

Cut-off Errors

⇒ This kind of errors erase where the true mathematical problem have some kind of infinite sum.

⇒ Computers are stupid creatures and they don't understand what is ∞ so we need to cut of computations at some point. Example:

Maclaurin series:

We know that an e^x function can be Taylor expanded:

$$e^x = \sum_{n=0}^{n=\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + ...$$

We can replace the infinite sum with a finite one:

$$e^x \approx= 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + ... + \frac{x^N}{N!}$$

⇒ This kind of errors are unfortunately very common in the field. We will have to deal with them every time we have some kind of $\lim$, $\sum^{\infty}$, etc.

Rounding Errors

⇛ This kind of errors occur during calculations on machines.

⇛ Machine represents the number with a final precision.

⇛ During the calculations the errors accumulate with every operation.

⇛ This errors can be avoided or reduced by proper algorithm or by changing the precision of computations.

Rounding Errors

$\Rightarrow$ This kind of errors occur during calculations on machines.

$\Rightarrow$ Machine represents the number with a final precision.

$\Rightarrow$ During the calculations the errors accumulate with every operation.

$\Rightarrow$ This errors can be avoided or reduced by proper algorithm or by changing the precision of computations.

Google   399999999999999-399999999999998   Search
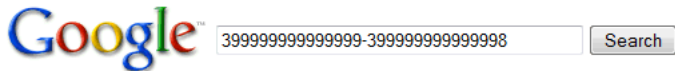
Web   Show options...

399 999 999 999 999 - 399 999 999 999 998 = 0

More about calculator.

Integer numbers on our PC

$\Rightarrow$ The integer numbers are very easy to represent on PC:

$$a = \pm(c_1 N^0 + c_2 N^1 + ... + c_d N^{d-1}),$$

where N is the base of the system. Computers of course use 2.

$\Rightarrow$ For example:

$$101011 \rightleftarrows 1 \cdot 2^0 + 1 \cdot 2^1 + 0 \cdot 2^2 + 1 \cdot 2^3 +$$
$$0 \cdot 2^4 + 1 \cdot 2^5$$
$$= 43$$

$\Rightarrow$ Now the first bit can be used for sign determination (0 plus 1 minus) int or can be used to extend the range of numbers unsigned

$\Rightarrow$ int - $-2147483648 - 2147483647$

$\Rightarrow$ unsigned - $0 - 4294967295$

There are only **10** kinds of people.
Those who understand binary
and those who don't.

Floating numbers on our PC

⇒ Computers are using so-called "classical floating point representation":

$$a = \pm M \cdot N^C, \ a \neq 0,$$

- M - mantissa or significand
- N - base of the system
- C - exponent

⇒ In such system the mantissa is always normalized:

$$M \in \left[\frac{1}{N}, 1\right)$$

⇒ Using the definition we can write the mantissa and exponent in the following way:

$$M = (m_1 N^{-1} + m_2 N^{-2} + ... + m_t N^t)$$
$$C = \pm(c_1 N^0 + c_2 N^1 + c_3 N^2 + ... + c_d N^{d-1})$$

⇒ The most often know base are $2, 8, 10, 16$.

Numbers on our PC: binary system

$\Rightarrow$ For example let's construct numbers using binary system:

$$M = \pm(m_1 2^{-1} + m_2 2^{-2} + ... + m_t 2^t)$$
$$C = \pm(c_1 2^0 + c_2 2^1 + c_3 2^2 + ... + c_d 2^{d-1})$$

where:

- t - length of mantissa, d - length of exponent
- m_i - mantissa digits; $m_i \in \{0, 1\}$
- c_i - exponent digits; $c_i \in \{0, 1\}$

Example: 11110001 $(s)mmmm(s)cc$

Lets say we are representing a number with a byte. First 5 digits are the mantissa the next 3 are exponent.

$$M = -(1 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdot 2^{-3} + 0 \cdot 2^{-4} = -\frac{7}{8}$$
$$C = +(0 \cdot 2^0 + 1 \cdot 2^1) = 2$$
$$x = -\frac{7}{8} \cdot 2^2 = -3.5$$

IEEE754-2008

⇒ The rules of the binary systems are defined by the `ISO-IEEE754` standard (updated in 2008 of the 1985 standard).

Name	Common name	Base	Digits	Decimal digits	Exponent bits	Decimal E max	Exponent bias[6]	E min	E max	Notes
binary16	Half precision	2	11	3.31	5	4.51	2^4-1 = 15	−14	+15	not basic
binary32	Single precision	2	24	7.22	8	38.23	2^7-1 = 127	−126	+127	
binary64	Double precision	2	53	15.95	11	307.95	$2^{10}-1$ = 1023	−1022	+1023	
binary128	Quadruple precision	2	113	34.02	15	4931.77	$2^{14}-1$ = 16383	−16382	+16383	
binary256	Octuple precision	2	237	71.34	19	78913.2	$2^{18}-1$ = 262143	−262142	+262143	not basic
decimal32		10	7	7	7.58	96	101	−95	+96	not basic
decimal64		10	16	16	9.58	384	398	−383	+384	
decimal128		10	34	34	13.58	6144	6176	−6143	+6144	

⇒ There is a minimum and maximum number we can represent in the current system.
⇒ There are some tricks you can play: use subnormal
⇒ In general we can represent numbers from: $\langle -a, -b \rangle \cup \{0\} \cup \langle b, a \rangle$
⇒ The standard also defines the rounding procedures for numbers.

Error propagation: sum

$\Rightarrow$ Let's consider we have two numbers x and y.

$\Rightarrow$ Their representation are not exact so: $x = \overline{x} + \epsilon_x$ and $y = \overline{y} + \epsilon_y$

$\Rightarrow$ Now if we want to sum them:

$$x + y = \overline{x} + \overline{y} + \epsilon_x + \epsilon_y = \overline{x} + \overline{y} + \epsilon$$

$\Rightarrow$ If each of the ϵ_i numbers have a constant distribution on $\left[-\frac{1}{2}10^{-d}, \frac{1}{2}10^{-d}\right]$, where d is the precision.

$\Rightarrow$ Then ϵ has a triangular distribution on $\left[-10^{-d}, 10^{-d}\right]$

$\Rightarrow$ Repeating N times this we will approach the Gaussian distribution with a width of $\sim \sqrt{N}10^{-d}$

Error propagation: multiplication, division

$\Rrightarrow$ Let's consider we have two numbers x and y.

$\Rrightarrow$ Their representation are not exact so: $x = \overline{x} + \epsilon_x$ and $y = \overline{y} + \epsilon_y$

$\Rrightarrow$ Now if we want to multiply them (we can neglect $\epsilon_x \epsilon_y$ terms):

$$x \cdot y = \overline{x} \cdot \overline{y} + \epsilon_x \cdot \overline{y} + \epsilon_y \cdot \overline{x}$$

$\Rrightarrow$ If $|\overline{x}| \gg (\ll)|\overline{y}|$ then the error might explode.

$\Rrightarrow$ Now if we want to divide them:

$$x/y = \overline{x}/\overline{y} + \frac{\epsilon_x}{\overline{y}} + \frac{\epsilon_y \overline{x}}{\overline{y}^2}$$

$\Rrightarrow$ If $|y| \ll |x|$ then the division is will have large errors.

Application

$\Rightarrow$ When we implement a current algorithm we need to ensure that our algorithm is "backward stable", "well-conditioned" and "numerically stable".

Numerical stability

We say that the algorithm is numerically stable if the the results does not change if we increase the computation precision.

Backward stability

Backward stability means that our algorithm will gives us the true answer if we move to infinite precision of the machine. In practice we look if we can conserve the errors of the representation.

Well-posed

Each algorithm has some input parameters. If we introduce a slight difference in those parameters (of the order of the representation precision), the results should not change significantly.

$\Rightarrow$ The numerical precision lead to discovery of chaos. If we have time we will discuss this during our lectures.

Rounding error

$\Rightarrow$ Imagine we want to calculate:

$$1.0000 + \sum_{i=1}^{10} 0.0001$$

$\Rightarrow$ Now imagine we will carry the calculations with 4 digit precision.
$\Rightarrow$ If we calculate the:

$$1.0000 + \sum_{i=1}^{10} 0.0001 = 1.000$$

$\Rightarrow$ If we calculate the:

$$\sum_{i=1}^{10} 0.0001 + 1.0000 = 1.001$$

$\Rightarrow$ The order matters!

Examples 1

$\Rightarrow$ Let's calculate the π number accordingly to the following formula for couple of i:

$$G_p = \frac{1}{p} \left[10^p \left(1 + p\pi 10^{-p} \right) - 10^p \right]$$

with a simple program one can see:

Example 1

⇒ How to fix this?!
⇒ Change `double` to `long double`


```
mchrzasz-ThinkPad-W530% ./ex1.x 10
1   3.1415926535897931 0
2   3.1415926535897967 3.5527136788005009e-15
3   3.1415926535898202 2.708944180085382e-14
4   3.1415926535901235 3.3040237212844659e-13
5   3.1415926535904877 6.9455552420549793e-13
6   3.1415926535846666 -5.1265658385091228e-12
7   3.1415926537343433 1.4455014962777568e-10
8   3.1415926534682512 -1.2154188766544394e-10
9   3.1415926615397134 7.9499202776389666e-09
10  3.1415927886962893 1.3510649621295556e-07
1   3.1415926535897931 0
2   3.1415926535897931 0
3   3.1415926535897931 0
4   3.1415926535897931 0
5   3.141592653589794 8.8817841970012523e-16
6   3.1415926535897918 -1.3322676295501878e-15
7   3.1415926535898637 7.0610184366159956e-14
8   3.1415926535901235 3.3040237212844659e-13
9   3.1415926535911343 1.341149413747189e-12
10  3.1415926535613834 -2.8409719021738056e-11
```

Summary

$\Rrightarrow$ Computers use only representations of the floating points!

$\Rrightarrow$ Numerical methods suffer from computer precision!

$\Rrightarrow$ It's YOUR responsibility to ensure that what you are doing will not explode the error.

Backup