Numerical Methods
5633

Lecture 2

Michaelmas Term 2018

Marina Krstic Marinkovic
mmarina@maths.tcd.ie

School of Mathematics
Trinity College Dublin

Organisational (Michaelmas Term 2018)

- No MA5533 lecture next week: 26.09.2018 (MA5513 class instead)
- Changed time of the lecture, as of next week

	To appear	Submission DL
Assignment 1	10.10	2.11
Assignment 2	14.11	30.11

Computational Errors

- A numerical method must use a **finite representation** for numbers and thus cannot possibly produce an exact answer for all problems
  - For example, 3.14159 instead of π etc. (also $\sqrt{2}$, 2/3 etc.)
- Sources of error:
  - Truncation error (approximate formulas, including discret. error)
  - Roundoff error (inexact computer arithmetics)
  - Propagated error (errors from input, or previous calc.)
  - Statistical error (stochastic calc.: Monte Carlo; sampling)
- References for this lecture:
  - David Bindel, Jonathan Goodman "Principles of Scientific Computing Sources of Error", Chapter "Sources of Error"
  - Most of the material taken from: http://cims.nyu.edu/~donev/Teaching/NMI-Fall2010/Lecture1.handout.pdf
  - https://cran.r-project.org/doc/manuals/R-intro.pdf

Error propagation and amplification

- Errors can grow as they propagate through a computation, e.g.

```
f1 = . . . ;   // approx of f(x)
f2 = . . . ;   // approx of f(x+h)
fPrimeHat = ( f2 - f1 ) / h ;   // approx of derivative
```

- Three contributions to the error:
  - ➡ Truncation error:
    $$\frac{f(x+h)-f(x)}{h} = f'(x)(1+\epsilon_{\mathrm{tr}})$$
  - ➡ Roundoff error:
    $$\widehat{f'} = \frac{f_2 - f_1}{h}(1+\epsilon_r).$$
  - ➡ Propagated error (using inexact values of f(x) and f(x+h) in the first place):
    $$\frac{f_2 - f_1}{h} = \frac{f(x+h)-f(x)}{h}\left(1 + \frac{e_2 - e_1}{f_2 - f_1}\right) = \frac{f(x+h)-f(x)}{h}(1+\epsilon_{\mathrm{pr}})$$

Consistency, Stability and Convergence

- Discretisation error: $$F(x,d)=0 \quad \longrightarrow \quad \hat{F}_n(\hat{x}_n,\hat{d}_n)=0$$

  ➡ replacing the computational problem with an easier-to-solve approximation

  ➡ for each n there is an algorithm that produces $\hat{x}_n$ given $\hat{d}_n$

- A numerical method is:

  ➡ **consistent** - if the approximation error vanishes as $n \to \infty$

  ➡ **stable** - if propagated errors decrease as the computation progresses

  ➡ **convergent** - if the numerical error can be made arbitrarily small by increasing the computational effort

- Other very important features, determining the choice of NM: **accuracy**, reliability/**robustness**, **efficiency**

Conditioning a Computational Problem

- A generic computational problem:
  - ➡ Find **solution x** that satisfies a **condition F(x,d)=0**,for given **data d**
- **Well posed** problem has a unique solution that depends continuously on the data. Otherwise:**ill-posed** problem (no numerical method will work!)
- Conditioning number (K):

$$K = \sup_{\delta d \neq 0} \frac{||\delta x|| \;/\; ||x||}{||\delta d|| \;/\; ||d||}$$

  - ➡ absolute error: $\hat{x} = x + \delta x$
  - ➡ relative error: $\hat{x} = (1 + \epsilon)x$
- **K** is an important *intrinsic* property of a computational problem.
- **K~1**, problem is **well-conditioned.**
- **Ill-conditioned** problem: a given target accuracy of the solution δx cannot be computed for a given accuracy of the data δd , i.e. condition number **K is large!**

More on Stability:

- Stability analysis in scientific computing: studying the propagation of small changes by a process, to *search for error growth in computations*
  - ➡ focuses on propagated error only (for simplicity)
- A numerical method is:
  - ➡ **stable** - if propagated errors decrease as the computation progresses
  - ➡ **unstable** - if relative errors in the output are much larger than relative errors in the input (ill-conditioned —> unstable)
  - ➡ **backward stable** algorithm - as stable as the condition number allows/ unstable only when the underlying problem is ill-conditioned (many Lin.Alg. algorithms)
  - ➡ **forward stable** algorithm - if it has a forward error (δx) of magnitude similar to some backward stable algorithm (δx/K is small)
  - ➡ **mixed stability:** combines the forward error δx and the backward error δd; if there exists δd such that both δd and δx are small

IEEE 754

- Computers represent everything using bit strings, i.e., integers in base-2. **Integers can thus be exactly represented.** But not real numbers!

- **IEEE Standard for floating-point arithmetic** (est. 1985):

  - ➡ Formats for representing and encoding real numbers using bit strings (single and double precision)

  - ➡ Rounding algorithms for performing accurate arithmetic operations (e.g. addition,subtraction,division,multiplication) and conversions (e.g. single to double precision)

  - ➡ Exception handling for special situations (e.g. division by zero and overflow)

- **R programming:**

Some info on the implementation of the IEEE 754 standard in R:
https://stat.ethz.ch/R-manual/R-devel/library/base/html/double.html

Floating Point Representation

- Assume we have a computer that represents numbers using a given (decimal) number system
- Representing real numbers, with N available digits:
  - **Fixed-point** representation:
    $$x = (-1)^s [a_{N-2} a_{N-3} \dots a_k a_{k-1} \dots a_0]$$
    - Problem with representing large/small numbers: 9.872 but 0.009
  - **Floating-point** representation:
    $$x = (-1)^s \cdot [0 a_1 a_2 \dots a_t] \cdot \beta^e \quad = (-1)^s \cdot m \cdot \beta^{e-t}$$
    - Similar to the common scientific representation: $0.9872 \cdot 10^{1}$ and $0.9872 \cdot 10^{-2}$
- A floating-point number in base β is represented using:
  - one **sign** bit s = 0 or 1 (positive or negative nr.)
  - integer **exponent** giving its order of magnitude
  - t-digit integer **mantissa** specifying actual digits of the number

IEEE Standard Representations

- **IEEE representation example (single precision example):**

Take the number x = 2752 = 0.2752 • 10^4

1. Converting 2752 to the binary:

$$x = 2^{11} + 2^9 + 2^7 + 2^6 = (101011000000)_2 = 2^{11} \cdot (1.01011)_2$$
$$= (-1)^0 2^{138-127} \cdot (1.01011)_2 = (-1)^0 2^{(10001010)_2 - 127} \cdot (1.01011)_2$$

2. On the computer:

$$x = (-1)^s \cdot 2^{p-127} \cdot (1.f)_2$$

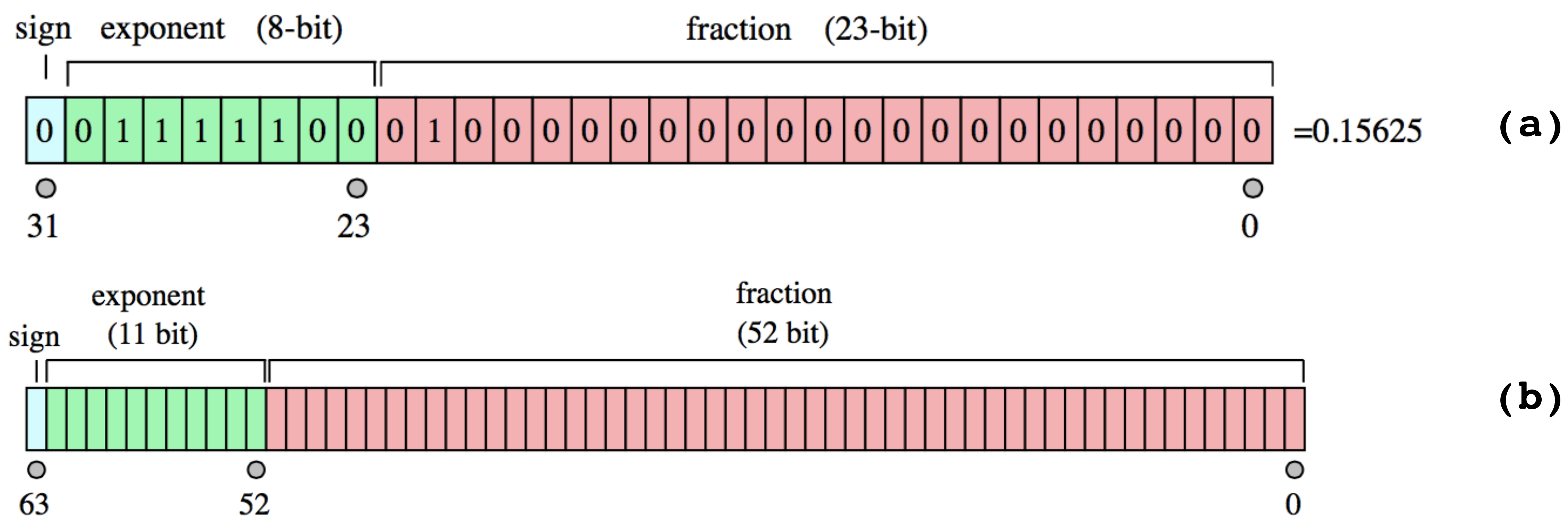
$$x = [s\ |p\ |\ f]$$
$$= [0 \mid 100, 0101, 0 \mid 010, 1100, 0000, 0000, 0000, 0000]$$
$$= (452c0000)_{16}$$

- **IEEE non-normalised numbers**

value	power p	fraction f
±0	0	0
denormal (subnormal)	0	>0
±∞(inf)	255	0
Not a number (NaN)	255	>0

IEEE Standard Representations

- **Representation of single(a) and double(b) precision numbers:**



[Illustrations: By Codekaizen - Own work, GFDL, https://commons.wikimedia.org/w/index.php?curid=3595583]

- **See wikipedia article on IEEE:**

  https://en.wikipedia.org/wiki/IEEE_754-1985

IEEE Standard Representations

- **R-script for conversion of integer to binary**

```
# function for converting integer to binary numbers
binary<-function(p_number)
{
bsum<-0
bexp<-1
while (p_number > 0) {
digit<-p_number %% 2
p_number<-floor(p_number / 2)          #predefined math function floor
                                       #floor(x) gives [x]
bsum<-bsum + digit * bexp
bexp<-bexp * 10
}
return(bsum)
}

p_number<-readline("Decimal number: ")  #reading a number decimal representation
p_number<-as.numeric(p_number)         #converts the input to integer
bsum<-binary(p_number)                 #calls function to perform conversion to binary
cat("Binary: ", bsum)                  #prints binary number
```

```
> source(“binary.sh”)
Decimal number: 2752
Binary:  1.01011e+11>
```

Important Facts about Floating-Point

- Not all real numbers **x**, or even integers, can be represented exactly as a floating-point number, instead, they must be rounded to the nearest floating point number

- The relative spacing or gap between a floating-point x and the nearest other one is at most $\boldsymbol{\varepsilon = 2^{-N_f}}$, sometimes called **ulp** (unit of least precision). In particular, $\boldsymbol{1 + \varepsilon}$ is the first floating-point number larger than 1

- Floating-point numbers have a relative rounding error that is smaller than the **machine precision** or **roundoff-unit** u. The rule of thumb is that single precision gives 7-8 digits of precision and double 16 digits

- Do not compare floating point numbers (especially for loop termination), or more generally, do not rely on logic from pure mathematics!

Floating-Point Exceptions

- Computing with floating point values may lead to exceptions, which may be trapped and halt the program:

  - **Divide-by-zero,** the result is $\pm\infty$
  - **Invalid** if the result is a NaN
  - **Overflow** if the result is too large to be represented
  - **Underflow** if the result is too small to be represented

- Numerical software needs to be careful about avoiding exceptions where possible

  ➡ Do not compare floating point numbers (especially for loop termination), or more generally, do not rely on logic from pure mathematics!

Numerical Cancellation

- If **x** and **y** are close to each other, **x - y** can have reduced accuracy due to cancellation of digits.
- Note: If gradual underflow is not supported **x - y** can be zero even if **x** and **y** are not exactly equal
- **Benign cancellation:** subtracting two **exactly-known** IEEE numbers results in a relative error of no more than an **ulp.** The result is **precise**
- **Catastrophic cancellation** occurs when subtracting two nearly equal **inexact** numbers and leads to loss of accuracy and a large relative error in the result
- For example, 1.1234 - 1.1223 = 0.0011 which only has 2 significant digits instead of 4. The result is **not accurate**

Avoiding Cancellation

- Rewriting in mathematically-equivalent but numerically-preferred form is the first try
  - ➡ For example

$$\sqrt{x+\delta}-\sqrt{x} \quad \longrightarrow \quad \frac{\delta}{\sqrt{x+\delta}+\sqrt{x}}$$

  - ➡ to avoid catastrophic cancellation. But what about the extra cost?

- Sometimes one can use Taylor series or other approximation to get an approximate but stable result

$$\sqrt{x+\delta}-\sqrt{x} \approx \frac{\delta}{2\sqrt{x}} \quad \text{for} \quad \delta \ll x$$

Summary

- A numerical method needs to control the various computational errors (approximation, roundoff …) while balancing computational cost
- The IEEE standard (attempts to) standardises the single and double precision floating-point formats, their arithmetic, and exceptions. It is widely implemented (R, Matlab, C, …)
- Numerical overflow, underflow and cancellation need to be carefully considered and may be avoided
- Mathematically-equivalent forms are not numerically-equivalent
- Never compare floating point numbers! Especially for loop termination, or more generally, do not rely on logic from pure mathematics
- Some disastrous things might happen due to applying numerical methods in an incorrect way