Programming, numerics and optimization

Lecture B-1: Basics of numerics

Łukasz Jankowski
ljank@ippt.pan.pl

Department of Intelligent Technologies
Institute of Fundamental Technological Research
Room 4.32, Phone +22.8261281 ext. 428

March 23, 2021[1]

[1] Current version is available at http://info.ippt.pan.pl/~ljank.

Outline

Outline

Number representations

In numerical computations, only two kinds of numbers are used:

- integral numbers (integers) and
- real numbers (reals).

However, computers handle not abstract “numbers”, but their *representations* stored in the memory in binary form:

- integers
  1. Integer numbers
- reals
  2. Floating-point numbers
  3. Fixed-point numbers

Integer numbers
Limits

Integer numbers are

- exact binary representations of integers
- within given range.

Ranges of integer types are implementation-dependent. In general, the C/C++ standards specify

- the minimum sizes

char	C: at least 8 bits; C++: exactly 8 bits
int	at least 16 bits
long	at least 16 bits
long long	at least 32 bits

- and (C++) that

```
sizeof(char) <= sizeof(short) <= sizeof(int)
        <= sizeof(long) <= sizeof(long long)
```

Integer numbers
Limits

Each implementation defines its ranges in headers limits and climits (or limits.h).

```
#include <climits>
// ...
cout <<"INT_MAX = " <<INT_MAX <<endl;
cout <<"INT_MIN = " <<INT_MIN <<endl;
// ...
```

```
#include <limits>
// ...
cout <<"INT_MAX = " <<numeric_limits<int >::max() <<endl;
cout <<"INT_MIN = " <<numeric_limits<int >::min() <<endl;
// ...
```

Integer numbers
Limits

Example

integer type	bytes	bits	**unsigned**	**signed**
char	1	8	$0 \ldots 255$	$-128 \ldots 127$
short (int)	2	16	$0 \ldots 2^{16} - 1$	$-2^{15} \ldots 2^{15} - 1$
int	4	32	$0 \ldots 2^{32} - 1$	$-2^{31} \ldots 2^{31} - 1$
long (int)	4	32	$0 \ldots 2^{32} - 1$	$-2^{31} \ldots 2^{31} - 1$
long long (int)	8	64	$0 \ldots 2^{64} - 1$	$-2^{63} \ldots 2^{63} - 1$

Integer numbers
Representations

Implementation of integer types is platform-dependent

- numbers of bytes can be different
- in principle, different number representations can be used for *signed* numbers
  - sign-and-magnitude
  - ones' complement
  - two's complement
  - Excess-N

Virtually all contemporary processors use two's complement for signed integers, however

- In floating-point numbers: *sign-and-magnitude* is used for mantissa, *Excess-N* is used for exponent
- *Ones' complement* was used in older processors, it is also used in checksum algorithms in some Internet protocols

Integer numbers

Representation (2's complements)

- Positive int's and zero are stored in natural binary notation.
- Negative integers are stored as *binary complements to zero* or, in fact, to a large power of two (*two's-complement*).

in 16 bit **short** (range: $0 \dots 65535$ or $-32768 \dots 32767$)

number	**unsigned short**	(**signed**) **short**
0	0000000000000000	0000000000000000
1	0000000000000001	0000000000000001
2	0000000000000010	0000000000000010
32767	0111111111111111	0111111111111111
65535	1111111111111111	—
-1	—	1111111111111111
-2	—	1111111111111110
-32768	—	1000000000000000

Integer numbers

Representation (2's complements)

You can check the (2's) notation yourself

```
typedef int T;        // put your type here
const int size = 8*sizeof(T);
T number = -10;       // put a value here

for(int i=size-1; i>=0; --i)
   cout <<!!(number&T(1)<<i);
cout <<endl;
```

The **typedef** keyword defines an alias for a data type,

&	is the bitwise AND operator
<<	is the bitwise right shift operator
!!a	(double negation) is equivalent to a?1:0 or (a==0)?0:1

Integer numbers

Representation (2's complement)

The two's-complement arithmetic makes the binary summation straightforward

with 8 bit char

number			register							
3	=		0	0	0	0	0	0	1	1
-2	=		1	1	1	1	1	1	1	0
3+(-2)	=	1 ←	0	0	0	0	0	0	0	1

In **signed** integral types, the first bit of the representation conveys information about the sign of the number:

- 0 positive or zero,
- 1 negative.

The first bit is thus called the *sign bit*.

Integer numbers

Examples (2's complement)

```
unsigned short a=32767, b=32768;
unsigned short c=a+b;
cout <<"a\t= " <<a <<endl;
cout <<"b\t= " <<b <<endl;
cout <<"a+b\t= " <<c <<endl;
```

Console output

```
a   = 32767
b   = 32768
a+b = 65535
```

```
unsigned short a=32768, b=32768;
unsigned short c=a+b;
cout <<"a\t= " <<a <<endl;
cout <<"b\t= " <<b <<endl;
cout <<"a+b\t= " <<c <<endl;
```

Console output

```
a   = 32768
b   = 32768
a+b = 0
```

Integer numbers

Examples (2's complement)

```
unsigned short a=32768, b=32769;
unsigned short c=a+b;
cout <<"a\t= " <<a <<endl;
cout <<"b\t= " <<b <<endl;
cout <<"a+b\t= " <<c <<endl;
```

?

```
unsigned short a=1, b=2;
unsigned short c=a-b;
cout <<"a\t= " <<a <<endl;
cout <<"b\t= " <<b <<endl;
cout <<"a+b\t= " <<c <<endl;
```

?

Integer numbers

Examples (2's complement)

```
unsigned short a=32768, b=32769;
unsigned short c=a+b;
cout <<"a\t= " <<a <<endl;
cout <<"b\t= " <<b <<endl;
cout <<"a+b\t= " <<c <<endl;
```

Console output

```
a    = 32768
b    = 32769
a+b  = 1
```

```
unsigned short a=1, b=2;
unsigned short c=a-b;
cout <<"a\t= " <<a <<endl;
cout <<"b\t= " <<b <<endl;
cout <<"a+b\t= " <<c <<endl;
```

?

Integer numbers

Examples (2's complement)

```
unsigned short a=32768, b=32769;
unsigned short c=a+b;
cout <<"a\t= " <<a <<endl;
cout <<"b\t= " <<b <<endl;
cout <<"a+b\t= " <<c <<endl;
```

Console output

```
a   = 32768
b   = 32769
a+b = 1
```

```
unsigned short a=1, b=2;
unsigned short c=a-b;
cout <<"a\t= " <<a <<endl;
cout <<"b\t= " <<b <<endl;
cout <<"a+b\t= " <<c <<endl;
```

Console output

```
a   = 1
b   = 2
a+b = 65535
```

Floating-point numbers

Floating-point numbers are

- within given range
- unexact exponential binary representations of reals.

General rule:

$$x = \pm 2^e\, m,$$

where

e is the exponent and

m is the mantissa $(1 \leq m < 2)$,

both represented in binary notations.

Floating-point numbers
Example

12.75 represented in typical 32 bit float

$$12.75_{\text{dec}} = +1100.11_{\text{bin}} = +2^{3_{\text{dec}}}\, 1.10011_{\text{bin}}$$

+	$2^{130_{\text{dec}}-127_{\text{dec}}}$								1.100011_{bin}							
0	1	0	0	0	0	0	1	0	1	0	0	1	1	0	...	0
sign	8b: exponent								23b: mantissa (significand)							

- 23 significant binary digits: *relative* representation accuracy $2^{-23} \approx 10^{-7}$ (only < 7 significant decimal digits).

```
#include<limits>
// ...
cout <<numeric_limits<float>::epsilon() <<endl;
```

- Exponent offset of -127 allows for negative exponents (Excess-127 representation).

Floating-point numbers

Representation and limits

Representation of floating-point types is platform-dependent. In general, the C/C++ standards specify that

- three floating-point types must be implemented: **float**, **double**, **long double**
- and that they have certain minimum capacities, e.g.
  - Minimum number of representable *decimal* digits: 6, 10, 10.
  - 10^{-37} is within the range of *normalized* numbers
  - 10^{+37} is within the range of *representable* numbers
  - the difference between 1 and the next representable number must be not greater than 10^{-5}, 10^{-9}, 10^{-9}

The actual values can be checked via the header limits by

```
cout <<numeric_limits<float >::digits10 <<endl;
cout <<numeric_limits<float >::min_exponent10 <<endl;
cout <<numeric_limits<float >::max_exponent10 <<endl;
cout <<numeric_limits<float >::epsilon() <<endl;
```

Virtually all C/C++ implementations use **float** and **double** conforming to the standard IEEE 754-1985.

Floating-point numbers

Standard IEEE 754-1985

name	sign(s)	mantissa(m)	exponent(e)	offset
binary32	1	23	8	127
binary64	1	52	11	1023
binary128	1	112	15	16383

$$x = (-1)^s\,(1.m)\,2^{e-\text{offset}}$$

Special cases for binary32

sign	exponent	mantissa	interpretation
s	0	nonzero	$(-1)^s\,(0.m)\,2^{-126}$ (unnormalized)
	255	nonzero	NaN (Not a Number)
0	255	0	Infinity
1	255	0	-Infinity
0	0	0	0
1	0	0	-0

Floating-point numbers

Standard C/C++ representations: **float**, **double**

Virtually all C/C++ implementations use **float** and **double** conforming to the standard IEEE 754-1985.

C++ name	sign(s)	mantissa(m)	exponent(e)	offset
float	1	23	8	127
double	1	52	11	1023

$$x = (-1)^s \, (1.m) \, 2^{e-\text{offset}}$$

Special cases are handled in an analogous manner.

Floating-point numbers

Standard C/C++ representations: **long double**

long double representations

C++ name	sign(s)	mantissa(m)	exponent(e)	offset
==**double**	1	52	11	1023
80 bit	1	64	15	16383
binary128	1	112	15	16383
double-double implementation				

$$x = (-1)^s \, (1.m) \, 2^{e-\text{offset}}$$

- The most common representation of **long double** is 80-bit (extended precision), which does not conform to IEEE 754. It is usually stored in 96 or 128 bits (12 or 16 bytes).
- In Microsoft Visual C++, **long double** maps to **double**.
- GNU C on SPARC implements the standard binary128.
- GNU C on PowerPC uses double-double implementation.

Floating-point numbers

Example (a non-standard 6 bit number)

6 bit floating-point numbers (non-standard)

$\square \quad \square\square\square \quad \square\square$ offset $= 3$ $\qquad x = (-1)^s\,(1.m)\,2^{e-\text{offset}}$

s	e	m		interpretation
$\boxed{0}$	$\boxed{1\,\vert\,0\,\vert\,0}$	$\boxed{0\,\vert\,1}$	$=$	$(-1)^0\,1.01_{\text{bin}}\,2^{4-3} = 2.5$
$\boxed{1}$	$\boxed{0\,\vert\,1\,\vert\,1}$	$\boxed{0\,\vert\,0}$	$=$	

Floating-point numbers

Example (a non-standard 6 bit number)

6 bit floating-point numbers (non-standard)

$\square \; \square\square\square \; \square\square$ offset $= 3$ $\qquad x = (-1)^s \, (1.m) \, 2^{e-\text{offset}}$

s	e	m		interpretation
0	1 0 0	0 1	$=$	$(-1)^0 \, 1.01_{\text{bin}} \, 2^{4-3} = 2.5$
1	0 1 1	0 0	$=$	$(-1)^1 \, 1.00_{\text{bin}} \, 2^{3-3} = -1$
1	1 1 1	1 0	$=$	

Floating-point numbers

Example (a non-standard 6 bit number)

6 bit floating-point numbers (non-standard)

$\boxed{}\ \boxed{}\boxed{}\boxed{}\ \boxed{}\boxed{}$ offset $= 3$ $x = (-1)^s\,(1.m)\,2^{e-\text{offset}}$

s	e	m		interpretation
0	1 0 0	0 1	$=$	$(-1)^0\,1.01_{\text{bin}}\,2^{4-3} = 2.5$
1	0 1 1	0 0	$=$	$(-1)^1\,1.00_{\text{bin}}\,2^{3-3} = -1$
1	1 1 1	1 0	$=$	NaN (Not a Number)

Floating-point numbers

Standard IEEE 754-1985: special cases

The standard defines also special cases to handle zero, infinity and exceptional situations

Special cases for binary32 (**float**)

sign	exponent	mantissa	interpretation
s	0	nonzero	$(-1)^s\,(0.m)\,2^{-126}$ (unnormalized)
	255	nonzero	NaN (Not a Number)
0	255	0	Infinity
1	255	0	-Infinity
0	0	0	0
1	0	0	-0

Floating-point numbers

Example (a non-standard 6 bit number)

$\square \; \square\square\square \; \square\square$ offset $= 3$ $\quad x = (-1)^s\,(1.m)\,2^{e-\text{offset}}$

normalized numbers, $e \notin \{0, 7\}$

s 001 00	=	±0.2500
s 001 01	=	±0.3125
s 001 10	=	±0.3750
s 001 11	=	±0.4375
s 010 00	=	±0.5000
s 010 01	=	±0.6250
s 010 10	=	±0.7500
s 010 11	=	±0.8750
s 011 00	=	±1.0000
s 011 01	=	±1.2500
…		…
s 110 11	=	±14.000

unnormalized numbers

s 000 01	=	±0.0625
s 000 10	=	±0.1250
s 000 11	=	±0.1875

other special cases

s 000 00	=	±0
s 111 00	=	±Infinity
s 111 01	=	NaN
s 111 10	=	NaN
s 111 11	=	NaN

Floating-point numbers

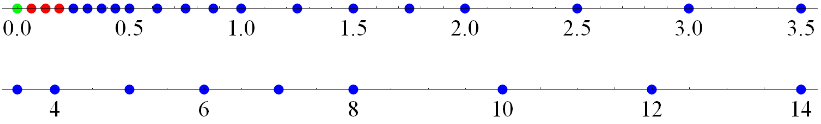
Example (a non-standard 6 bit number)

Note the increasing spacing between the normalized numbers in successive groups (constant *relative* representation accuracy).

normalized numbers ($e = 1, \ldots, 6$)

± 0.2500	± 0.3125	± 0.3750	± 0.4375	$\Delta x = 0.0625$
± 0.5000	± 0.6250	± 0.7500	± 0.8750	$\Delta x = 0.1250$
± 1.0000	± 1.2500	± 1.5000	± 1.7500	$\Delta x = 0.2500$
± 2.0000	± 2.5000	± 3.0000	± 3.5000	$\Delta x = 0.5000$
± 4.0000	± 5.0000	± 6.0000	± 7.0000	$\Delta x = 1.0000$
± 8.0000	± 10.0000	± 12.000	± 14.000	$\Delta x = 2.0000$

The unnormalized numbers maintain constant *absolute* accuracy of representation (but the relative accuracy can be very poor).



Fixed-point numbers

Floating-point numbers are flexible, but have disadvantages:

- much more complicated than integers, hence operations involving them are much slower.
- require a dedicated FPU (Floating Point Unit), hence their hardware implementation is costly.

An intermediate type between integer and floating-point types for representation of reals are fixed-point types. A fixed-point number has a fixed number of binary digits before and after the binary point (no exponential notation).

- cheaper hardware implementation
- quicker operations
- much smaller data range (no exponential progress)
- constant spacing between successive numbers, so accuracy measured not relatively but in absolute terms

Outline

Arithmetic errors

The errors in computer arithmetic can be of three basic types

- An overflow error occurs when the number to be represented in a given type is *too large* for it (happens with both integer and floating-point types).
- An underflow error occurs when the number to be represented in a given floating-point type is *too small* (too small exponent) and has to be represented by zero (floating-point only).
- A round-off error occurs each time a real number cannot be exactly represented in a given floating-point type and *has to be rounded* to the nearest floating-point number (floating-point only).

Arithmetic errors

Round-off error

Relative accuracy of floating-point numbers

C++ name	mantissa bits (m_b)	significant dec. digits
float	23 bits	~ 6
double	52 bits	~ 15
long double (80 bit)	64 bits	~ 19
long double (128 bit)	112 bits	~ 33

Exact number	v	
Representation	$v\,(1 \pm \rho)$,	$\lvert\rho\rvert \le 2^{-m_b}$

Arithmetic errors

Common error situations

- Division by a number which is very close to zero (represented by an unnormalized number)

```
#include <cmath>
// ...
cout <<float(pow(10.,-45)) <<endl;
cout <<1./float(pow(10.,-45)) <<endl;
        // the result is a double
        // floating-point number
```

Computed result: 7.13624e+44.
Exact value: 1e+45.

This is a relative error level of 29% in a single operation.

Arithmetic errors

Common error situations

- Subtraction of two very similar numbers (cancellation of terms)

$$\begin{aligned} \textbf{float}(1+10^{-6}) \text{ - } \textbf{float}(1) &= 0.954\,10^{-6} \\ \textbf{float}(1+10^{-7}) \text{ - } \textbf{float}(1) &= 1.192\,10^{-7} \\ \textbf{float}(1+10^{-8}) \text{ - } \textbf{float}(1) &= 0 \end{aligned}$$

This can happen e.g. when solving a quadratic equation $x^2 - 2px + q = 0$ with $p^2 >= q$. Direct application of the standard formula,

$$x = p \pm \sqrt{p^2 - q},$$

can yield a very inaccurate result in one of the roots if $p^2 >> q$.

Arithmetic errors

Common error situations

$x^2 - 1000x + 0.1 = 0$

	exact (6 digits)	computed in **float**	relative error
x_1	1000	1000	0 %
x_2	10e-5	9.15527e-5	9 %

A modified version of the algorithm

```
if(p>=0) x1 = p+sqrt(p*p-q);
else x1 = p-sqrt(p*p-q);
x2 = q/x1;
```

avoids the pitfall. It is valid, because $x_1 x_2 = q$ (as well as $x_1 + x_2 = 2p$).

Arithmetic errors

Common error situations

- Two floating points are practically never equal:

```
float x = 1./3;    // notice the dot (1.)
cout <<(4*x-1==x) <<endl;   // NOT equal!
```

- Never rely on exact comparison of floating point numbers:

```
for(float x=0; x<=1; x+=0.1)
    cout <<"x = " <<x <<endl;
```

An even worse stop condition would be x==1 (infinite loop).

Arithmetic errors

Common error situations

- Error propagation happens when the errors accumulate over many steps with rounding at each of them

$$x_0 = 1/3 \qquad x_{n+1} = 4x_n - 1$$

step no.	exact	**float**	**double**	**long double**
0	0.333333	0.333333	0.333333	0.333333
5	0.333333	0.333344	0.333333	0.333333
10	0.333333	0.34375	0.333333	0.333333
15	0.333333	11	0.333333	0.333333
20	0.333333	10923	0.333313	0.333333
25	0.333333	$\sim 10^7$	0.3125	0.333344
30	0.333333	$\sim 10^{10}$	21	0.34375

Arithmetic errors

Examples

Patriot system, failed Scud interception (Gulf War, Feb. 1991)

The system used an integer timing register which was incremented at intervals of 0.1 s. However, the integers were converted to decimal numbers by multiplying by the *binary* approximation of 0.1,

$$d = 0.00011001100110011001100_{\text{bin}} = 0.09999990463\ldots_{\text{dec}}.$$

After 100 hours ($3.6\ 10^6$ ticks), an error of $3.6\ 10^6\,(0.1 - d) \approx 0.34$ s had accumulated. This discrepancy caused the Patriot system to continuously recycle itself instead of targeting properly. As a result, an Iraqi Scud missile could not be targeted and was allowed to detonate on a barracks, killing 28 people.[a]

[a] Weisstein, Eric W. "Roundoff Error." From MathWorld–A Wolfram Web Resource. `http://mathworld.wolfram.com/RoundoffError.html`

Arithmetic errors

Examples

Ariane rocket

A notorious example is the fate of the Ariane rocket launched on June 4, 1996 (European Space Agency 1996). In the 37th second of flight, the inertial reference system attempted to convert a 64-bit floating-point number to a 16-bit number, but instead triggered an overflow error which was interpreted by the guidance system as flight data, causing the rocket to veer off course and be destroyed.[a]

[a] Weisstein, Eric W. "Roundoff Error." From MathWorld–A Wolfram Web Resource. `http://mathworld.wolfram.com/RoundoffError.html`

Outline

Problems and algorithms

The accuracy of the computed solution to a numerical problem is affected by two fundamental distinct factors:

1. “difficulty level” of the problem itself and
2. quality of the algorithm used to solve it.

Problems

A problem F is a functional relation between input and output data, formulated as

$$\text{given } x, \quad \text{find } F(x).$$

A problem does not specify the way (algorithm) to compute y.

- Given x, find $\sqrt{x}$.
- Given a_0, a_1, a_2, find the roots of $a_0 + a_1 x + a_2 x^2 = 0$.
- Given initial conditions and a differential equation, find the the solution.

Well-posed problem (Hadamard, 1902)

Existence, uniqueness and stability (Liptschitz continuity) of the solution.

Algorithms

An algorithm is a recipe (A) to solve a given problem F in a well-defined, specific way:

$$\text{given } x, \quad \text{compute } F_A(x).$$

There can be several different algorithms to solve one problem.

- Given x, find $\sqrt{x}$.
  1. Taylor expansion around e.g. 1.
  2. Recurrence $r_{n+1} = \frac{1}{2}(r_n + x/r_n)$ with $r_0 = (x < 1)?1 : x$ being the first approximation (Heron's algorithm).
- Given a_0, a_1, a_2, find the roots of $a_0 + a_1 x + a_2 x^2 = 0$.
  1. Direct application of the standard/modified formulas.
  2. bisection, Brent's, Newton, secant methods, ...
- Given initial conditions, solve a differential equation.
  1. Euler, Runge-Kutta, midpoint, multistep methods, ...

Sources of errors

Problem (well-posed)

given x, find $F(x)$

Algorithm

given x, compute $F_A(x)$

There are two fundamental sources of errors:

- Problem conditioning: sensitivity of the problem to unaccuracy of the initial data. Instead of exact x, only its approximation (due to round-off errors, unexact measurements, etc.) $\tilde{x}$ is available. Hence, even having an ideal algorithm it is possible to compute not $F(x)$, but only $F(\tilde{x})$.
- Algorithm stability: algorithms themselves introduce internal errors, hence even having exact input data x, it is possible to compute not $F(x)$, but only $F_A(x)$.

As a result, not $F(x)$, but merely $F_A(\tilde{x})$ is computed.

Outline

Problem posedness and conditioning

Well-posed problem (Hadamard, 1902)

A problem is well-posed for a given input data, if its solution

1. exists,
2. is unique and
3. is stable (Liptschitz continuous) with respect to the input.

Conditioning of a problem

A problem:	given x, find $y = F(x)$
Conditioning:	How sensitive is $F(x)$ to errors in x?

Problem posedness and conditioning

- A problem can be well-posed, but significantly ill-conditioned.
- A problem that is not Lipschitz-continuous is ill-posed. Such a problem must be extremely ill-conditioned.
- Continuous ill-posed problems often yield extremely ill-conditioned discretized versions.

The notions of posedness and conditioning are important as

1. many important physical problems are ill-posed (especially inverse problems)
2. real-world data are always approximate (whether measurements or simulations)
    - measurement errors
    - round-off arithmetic errors

Conditioning

1D differentiable problems

Assume $F : \mathbb{R} \to \mathbb{R}$ is differentiable

Exact data: x

Rounded-off data: $\tilde{x} = x + \Delta x$

Absolute error: $|\Delta y| = |F(\tilde{x}) - F(x)| \approx |F'(x)||\Delta x|$

Relative error: $\dfrac{|\Delta y|}{|y|} = \dfrac{|F(\tilde{x}) - F(x)|}{|F(x)|} \approx \dfrac{|F'(x)|}{|F(x)|}|\Delta x|$

Condition number: $\dfrac{|\Delta y|}{|y|} \Big/ \dfrac{|\Delta x|}{|x|} = \dfrac{|F'(x)|}{|F(x)|}|x|$

Conditioning

1D differentiable problems

Condition number κ

If $F\colon \mathbb{R} \to \mathbb{R}$ is differentiable and $F(x) \neq 0$, a condition number κ at x can be computed as

$$\kappa(x) = \frac{|F'(x)|}{|F(x)|}|x|.$$

- The condition number is a measure of amplification of relative error between input and output data.
- It is a property of the problem, which (independent of any algorithm) can be well- or ill-conditioned.
- If a problem is ill-conditioned for specific input data, an algorithm can give better results only by chance. The errors in input data propagate to the output data.

Conditioning

Multidimensional differentiable problems (see Lecture B-4)

The condition number of a differentiable multidimensional problem

$$F\colon \mathbb{R}^n \to \mathbb{R}^m$$

in a given point $\mathbf{x} \in \mathbb{R}^n$ is usually computed as the condition number of the corresponding linearized problem

$$F(\mathbf{x} + \Delta\mathbf{x}) \approx F(\mathbf{x}) + J(\mathbf{x})\Delta\mathbf{x} \qquad = \mathbf{b} + \mathbf{A}\mathbf{x},$$

where $J(\mathbf{x})$ is the Jacobian. The condition number of such a problem is usually (in the spectral norm) computed as

$$\kappa(\mathbf{A}) = \frac{\sigma_{\text{max}}(\mathbf{A})}{\sigma_{\text{min}}(\mathbf{A})},$$

that is as a ratio of the maximum and the minimum singular value of $\mathbf{A}$. For square normal matrices

$$\kappa(\mathbf{A}) = \left|\frac{\lambda_{\text{max}}(\mathbf{A})}{\lambda_{\text{min}}(\mathbf{A})}\right|.$$

Conditioning

Noncontinuous problems (see Lecture B-5)

If F is noncontinuous at x (ill-posed), no algorithm can provide an accurate result. Small inaccuracies of the input data x propagate to discontinuous jumps of the output data $F(x)$.

- In most practical finite-dimensional problems, F is piecewise continuous/differentiable, thus only a limited number of non-continuity points has to be studied in detail.
- Infinite-dimensional problems can be noncontinuous everywhere. For example, it happens to Fredholm integral equations of the first kind with a continuous kernel $K(t,s)$,

$$y(t) = \int_a^b K(t,s)x(s)ds = (\mathcal{K}x)(t),$$

since then $\mathcal{K}$ is a compact operator and so $x = \mathcal{K}^{-1}y$ is everywhere non-continuously dependent on the function y.

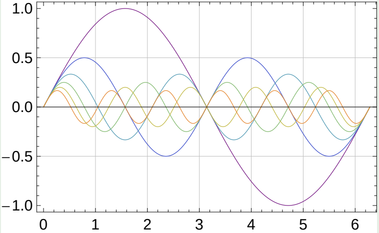
Conditioning

Noncontinuous problems (see Lecture B-5)

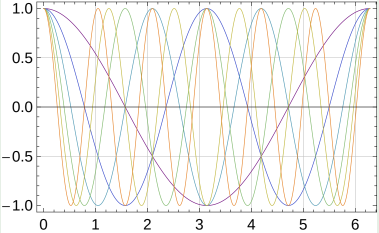
$$(\mathcal{K}x)(t) = \int_0^t x(t)\,dt \qquad\qquad (\mathcal{K}^{-1}y)(t) = y'(t)$$

Both $\mathcal{K}$ and its inverse $\mathcal{K}^{-1}$ are linear. However, the inverse is everywhere non-continuous.

$$y_n(t) = \frac{\sin\ nt}{n} \qquad\qquad (\mathcal{K}^{-1}y_n)(t) = \cos\ nt$$



uniform convergence to 0



non-convergent

Conditioning — Examples

$F(x) = \frac{1}{x}$

- Absolute error can be huge $\sim \frac{\Delta x}{x^2}$
- Condition number (relative error) $\kappa(x) = 1$

$F(x, y) = x - y$

Exact data:	x, y
Rounded-off data:	$\tilde{x} = x(1+\delta_x), \quad \tilde{y} = y(1+\delta_y)$
Exact result:	$x - y$
Computed result:	$\tilde{x} - \tilde{y} = (x-y) + (x\delta_x - y\delta_y)$
Relative error:	$\dfrac{(\tilde{x}-\tilde{y}) - (x-y)}{x-y} = \dfrac{x\delta_x - y\delta_y}{x-y}$

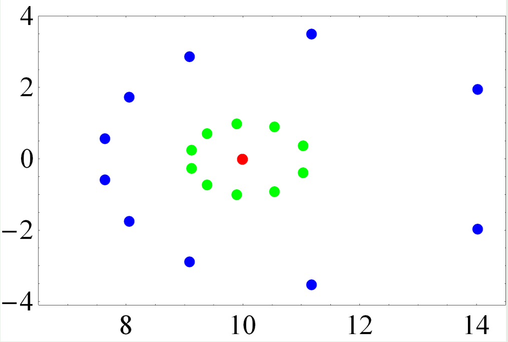
If $x \approx y$, even small δ_x and δ_y can result in a large relative error.

Problem conditioning — examples

Roots of (higher order) polynomials…

… can be very sensitive to even small errors in the coefficients

$$F(x) = (x-10)^{10} + \delta x^{10}$$
$$= (1+\delta)x^{10} - 100x^9 + 4500x^8 - \ldots - 10^{10}x + 10^{10}$$



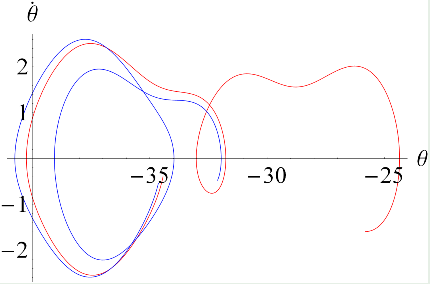
$\delta = 0$
$\delta = 10^{-10}$
$\delta = 10^{-5}$

Problem conditioning — examples

Damped pendulum with harmonic excitation

Non-linearized equation of motion

$$\ddot{\theta} + \gamma\dot{\theta} + \sin\theta = A\cos\omega t$$



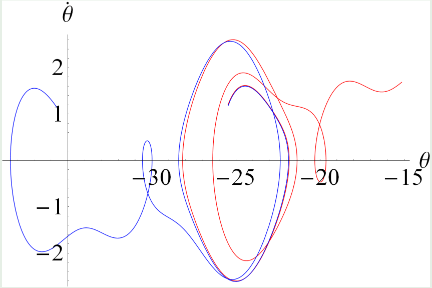
$$\begin{array}{lcl} A & = & 1.5 \\ \gamma & = & 0.5 \\ \omega & = & 2/3 \\ t & \in & [70, 90] \\ \dot{\theta}_0 & = & 0 \\ \hline \color{red}{\theta_0} & \color{red}{=} & \color{red}{0} \\ \color{blue}{\theta_0} & \color{blue}{=} & \color{blue}{10^{-3}} \end{array}$$

Problem conditioning — examples

Damped pendulum with harmonic excitation

Non-linearized equation of motion

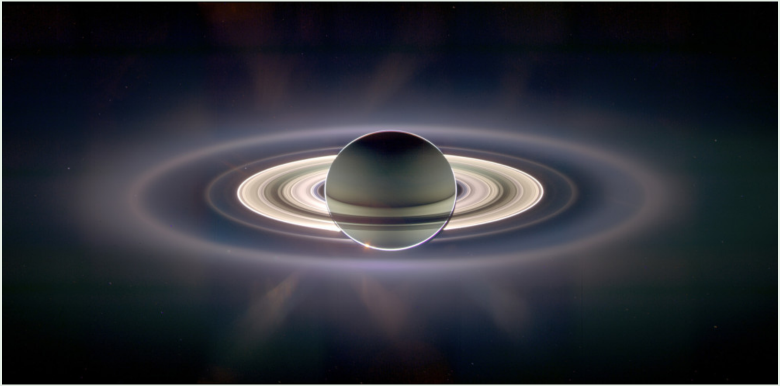
$$\ddot{\theta} + \gamma\dot{\theta} + \sin\theta = A\cos\omega t$$

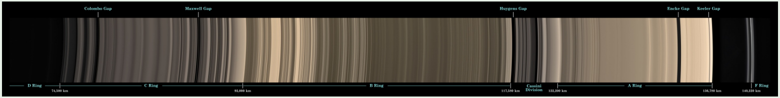


$$\begin{array}{lcl} A & = & 1.5 \\ \gamma & = & 0.5 \\ \omega & = & 2/3 \\ t & \in & [160, 190] \\ \dot{\theta}_0 & = & 0 \\ \hline \color{red}{\theta_0} & \color{red}{=} & \color{red}{0} \\ \color{blue}{\theta_0} & \color{blue}{=} & \color{blue}{10^{-6}} \end{array}$$

Problem conditioning — examples

Saturn ring gaps & unstable orbits (orbital resonances with moons)





Outline

Algorithm stability

A problem:	given x, find $F(x)$
An algorithm:	given x, compute $F_A(x)$
Stability:	How large is the error introduced by the algorithm?

Algorithms introduce internal errors, hence having even exact input data x, it is possible to compute not $F(x)$ but rather $F_A(x)$.

- Even the best algorithm will not solve exactly a severely ill-conditioned problem (as conditioning is problem-specific).
- A stable algorithm will not introduce considerable more error than the error resulting from the problem conditioning and approximate input.

Algorithm stability

Examples

$F(x, y) = x^2 - y^2$

The problem is ill-conditioned for $x^2 \approx y^2$. But assume the input data x and y are known exactly.

$$F_A(x, y) = x^2 - y^2$$
$$F_B(x, y) = (x - y)(x + y)$$

$$\begin{aligned} F_A(x, y) &= \left[x^2(1 + \delta_1) - y^2(1 + \delta_2)\right](1 + \delta_3) \\ &\approx (x^2 - y^2)(1 + \delta_3 + \frac{x^2\delta_1 - y^2\delta_2}{x^2 - y^2}) \\ F_B(x, y) &= [(x - y)(1 + \delta_1)(x + y)(1 + \delta_2)](1 + \delta_3) \\ &\approx (x^2 - y^2)(1 + \delta_1 + \delta_2 + \delta_3) \end{aligned}$$

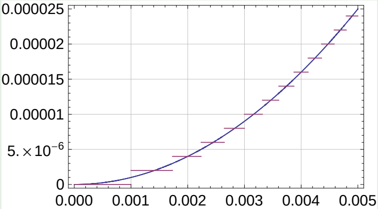
If $x^2 \approx y^2$, a serious loss of accuracy can occur within algorithm A.

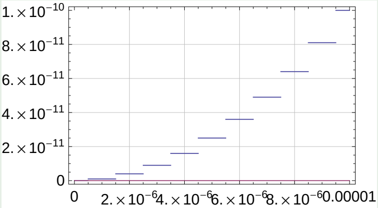
Algorithm stability

Examples

Assume that $\sin x$ and $\cos x$ are computed with 6 accurate digits ($\sim$**float**) and chop the rest off.

$$F_A(x) = \sin^2 x \qquad F_B(x) = 1 - \cos^2 x$$

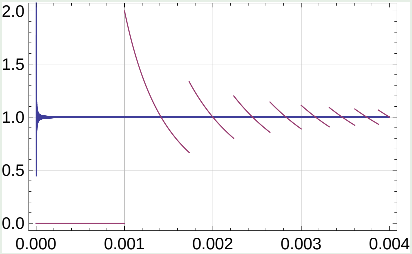




Algorithm stability
Examples

Assume that $\sin x$ and $\cos x$ are computed with 6 accurate digits (∼**float**) and chop the rest off. All the other operations are accurate.

$$F_A(x) = \frac{\sin^2 x}{x^2} \qquad F_B(x) = \frac{1-\cos^2 x}{x^2}$$



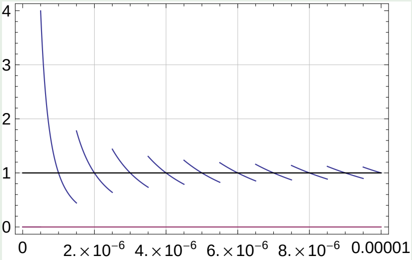
$$\boxed{\begin{aligned} F(0.0012345) &= 0.99999 \\ F_A(0.0012345) &= 0.99919 \\ F_B(0.0012345) &= 1.31234 \end{aligned}}$$

Algorithm stability

Examples

Assume that $\sin x$ and $\cos x$ are computed with 6 accurate digits ($\sim$**float**) and chop the rest off. All the other operations are accurate.

$$F_A(x) = \frac{\sin^2 x}{x^2} \qquad F_B(x) = \frac{1-\cos^2 x}{x^2}$$



exact
$F_A(x)$
$F_B(x)$

Outline

Homework 4 (10 points)

Basics of numerics

Let $S(n) = \sum_{i=1}^{n} \frac{1}{i}$.

In accurate arithmetic, the following three formulas are equivalent[2]:

$$s_1 = \frac{1}{1} + \frac{1}{2} + \ldots + \frac{1}{n}$$

$$s_2 = \frac{1}{n} + \frac{1}{n-1} + \ldots + \frac{1}{1}$$

$$s_3 = \frac{1}{n} + \frac{1}{n}\frac{n}{n-1} + \frac{1}{n}\frac{n}{n-1}\frac{n-1}{n-2} + \ldots + \frac{1}{n}\frac{n}{n-1}\cdots\frac{3}{2}\frac{2}{1}$$

[2] Assume that "+" is left-associative: $a+b+c$ is interpreted as $(a+b)+c$.

Homework 4 (10 points)

Basics of numerics

Consider the following implementations in single (**float**) arithmetic:

```
float s1 = 0;
for(int i=1; i<=n; ++i)
    s1 += 1/i;
```

```
float s2 = 0;
for(int i=n; i>=1; --i)
    s2 += 1/i;
```

```
float d = 1/n, s3 = d;
for(int i=n-1; i>=1; --i) {
    d *= (i+1)/i;
    s3 += d;
}
```

Homework 4 (10 points)

Basics of numerics

Compute $S(10^8)$ using all three algorithms.

1. The results are quite unexpected. Find and explain the reason.
2. Correct the error and repeat the computations. Explain the differences between the results.
3. How can you compute the exact value (up to five or six significant decimal digits)? What is it?

E-mail the answers to `ljank@ippt.pan.pl`.