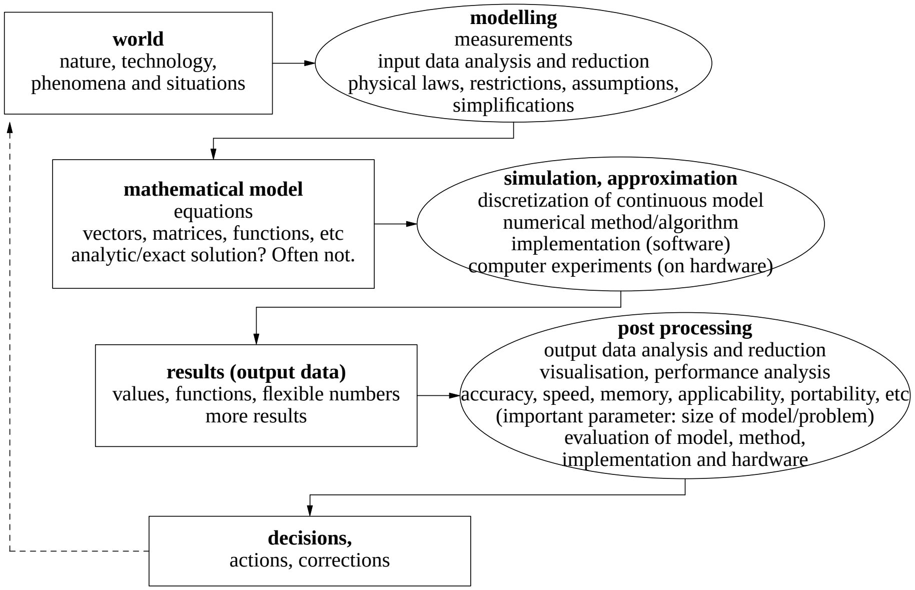
What is Scientific Computing?



world
nature, technology,
phenomena and situations

modelling
measurements
input data analysis and reduction
physical laws, restrictions, assumptions,
simplifications

mathematical model
equations
vectors, matrices, functions, etc
analytic/exact solution? Often not.

simulation, approximation
discretization of continuous model
numerical method/algorithm
implementation (software)
computer experiments (on hardware)

results (output data)
values, functions, flexible numbers
more results

post processing
output data analysis and reduction
visualisation, performance analysis
accuracy, speed, memory, applicability, portability, etc
(important parameter: size of model/problem)
evaluation of model, method,
implementation and hardware

decisions,
actions, corrections

Topics to be covered

- Computer Arithmetic; Data and Computational Errors
  How are numbers represented in the computer?
  How are arithmetic operations and other computations performed?
  Errors and other pitfalls of computations; ways to avoid them or to be alert.
- Solving systems of linear equations
  Gauss elimination; back and forward substitutions; pivoting
  Errors and how to measure them; How to improve accuracy?
  Operation counts; How to improve efficiency?
- Solving nonlinear equations (finding roots)
  Bisection; fixed point iteration and Newton's method; secant method
  Convergence, accuracy and efficiency
- Interpolation
  Approximating functions or data; generating intermediate values
  Polynomial interpolation; piecewise polynomial and spline interpolation
  Convergence, accuracy and efficiency

(Human) Representation of nonnegative integers

Decimal system (base = 10)

$$350 = (350)_{10} = 3 \times 100 + 5 \times 10 + 0 \times 1$$

Digits used: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Binary system (base = 2)

$$350 = (101011110)_2 = 1 \times 256 + 0 \times 128 +$$
$$+ 1 \times 64 + 0 \times 32 + 1 \times 16 + 1 \times 8 + 1 \times 4 + 1 \times 2 + 0 \times 1$$

Digits used: 0, 1

Base b system (base = $b > 0$, b integer)

$$x = (d_n d_{n-1} \cdots d_0)_b =$$
$$= d_n \times b^n + d_{n-1} \times b^{n-1} + \cdots + d_0 \times b^0$$
$$0 \le d_i < b \ , \ i = 0 \ , \cdots , \ n \ , \ x > 0 \ , \ x \in \mathbf{N}$$

Example: hexadecimal system (base = 16)
Digits used: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

Notes

Today's computers are digital. They work with pulses sent by electric components. Thus all computers operate internally in the binary system ($d_k = 0$ or 1). User-wise they operate in decimal, just for our convenience.

Note that, if several binary digits are grouped together, then a base b system, where b is a power of 2, is simulated, (e.g. grouping 4 binary digits together, the hexadecimal system is simulated).

(Human) Representation of integers

To represent nonnegative and negative integers, we use the sign in front of the number digits, then lay out the digits in sequence.
On the computer, the sign can be represented as an extra digit, 0 or 1.

Algorithm for converting base b integers to decimal

- Example 1 : $b = 2$

$$(1101)_2 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 13$$

- Example 2 : $b = 16$

$$(8FA)_{16} = 8 \times 16^2 + 15 \times 16^1 + 10 \times 16^0 = 2298$$

Note: the maximum number of digit positions needs to be known.

Algorithm for converting decimal integers to base b

- Example 1 : $b = 2$

Numerator	Denominator	Quotient	Remainder
350	2	175	0
175	2	87	1
87	2	43	1
43	2	21	1
21	2	10	1
10	2	5	0
5	2	2	1
2	2	1	0
1	2	0	1

$$(350)_{10} = (101011110)_2$$

Algorithm for converting decimal integers to base b

- Example 2 : $b = 16$

Numerator	Denominator	Quotient	Remainder
350	16	21	14
21	16	1	5
1	16	0	1

$$(350)_{10} = (15E)_{16}$$

Notes:

Stop when quotient is 0. Result is formed by writing the remainder digits bottom-up.

Recall that

$$(10)_{10} = (A)_{16} \quad (11)_{10} = (B)_{16} \quad (12)_{10} = (C)_{16}$$
$$(13)_{10} = (D)_{16} \quad (14)_{10} = (E)_{16} \quad (15)_{10} = (F)_{16}$$

(Human) Representation of reals

If $x \in \mathbb{R}$, then $x = \pm(x_I \,.\, x_F)_b = \pm(d_n d_{n-1} \cdots d_0 \,.\, d_{-1} d_{-2} \cdots)_b$, where x_I is the **integral** part, and x_F the **fraction**. The **sign** (+ or −) is represented by one digit, which is 0 or 1. The integral part of a real number is represented as a nonnegative integer. The fraction, which may have infinite number of (nonzero) digits, is represented in a similar way by laying out its digits after the dot.

Examples:

Decimal system (base = 10)

$$.666 \cdots = (.666 \cdots)_{10} = .\bar{6} = 6 \times 10^{-1} + 6 \times 10^{-2} + 6 \times 10^{-3} + \cdots$$

$$.1 = (.1)_{10} = 1 \times 10^{-1}$$

Binary system (base = 2)

$$0.1 = (.000110011 \cdots)_2 = (.0\overline{0011})_2 =$$
$$= 0 \times 2^{-1} + 0 \times 2^{-2} + 0 \times 2^{-3} + 1 \times 2^{-4} + 1 \times 2^{-5} + \cdots$$

Base b system (base = $b > 0$, b integer)

$$x_F = (.d_{-1} d_{-2} d_{-3} \cdots)_b =$$
$$= d_{-1} \times b^{-1} + d_{-2} \times b^{-2} + d_{-3} \times b^{-3} + \cdots = \sum_{k=1}^{\infty} d_{-k} \times b^{-k}$$

Notes

If x_F is a terminating binary fraction with n digits, the corresponding decimal fraction is also terminating in n digits. The opposite is not necessarily true. A terminating decimal fraction may have an equivalent non-terminating binary fraction.

Algorithm for converting base b fractions to decimal

- Example 1 : $b = 2$

$$(.101)_2 = 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} = 0.5 + 0.125 = 0.625$$

- Example 2 : $b = 16$

$$(.A)_{16} = 10 \times 16^{-1} = 10 \times 0.0625 = 0.625$$

Algorithm for converting decimal fractions to base b

- Example 1 : $b = 2$

Multiplier	Base	Product	Integral	Fraction
0.625	2	1.25	1	.25
0.25	2	0.5	0	.5
0.5	2	1.0	1	.0

good example

- Example 2 : $b = 2$

Multiplier	Base	Product	Integral	Fraction
0.1	2	0.2	0	.2
0.2	2	0.4	0	.4
0.4	2	0.8	0	.8
0.8	2	1.6	1	.6
0.6	2	1.2	1	.2
.	.	.	.	.

$$(.625)_{10} = (.101)_2 \quad , \quad (.1)_{10} = (.00011 \cdots)_2$$

Note: Stop when fraction vanishes. Result is formed by writing the integral digits from top to bottom.

Computer representation of numbers

Usually, integers are represented in the computer in the binary system, similarly as described above for the human representation, laying out the digits in sequence, and having an extra digit, 0 or 1, for the sign.

Real numbers are usually represented in the computer as **floating-point** numbers, again often in the binary system.

There are several versions of floating-point numbers, that differ little from each other. Below we give a simplified form, as well as the IEEE (Institute of Electrical and Electronics Engineers) Standard.

Floating-point numbers

A simplified form for representing a floating-point number x in **base** b is the following:

$$x = (f)_b \times b^{(e)_b}$$

$f = \pm(.d_1 d_2 \cdots d_t)_b$ is called the **mantissa** (or **significand**). 0 <= |mantissa| < 1

$e = \pm(c_{s-1} c_{s-2} \cdots c_0)_b$ is an integer called **exponent** (or **characteristic**).

A computer in which numbers are represented as above is said to have t base-b digits **precision**.

0.01 x 10^(e) is not normlaized, 0.1 x 10^(e-1) is.

The floating-point number is **normalised** if $d_1 \neq 0$, or else $d_1 = d_2 = \cdots = d_n = 0$.

Significant digits (of a nonzero floating-point number) are the digits following (including) the first nonzero digit of the mantissa. All the digits of the mantissa of a normalised floating-point number are significant.

Floating-point numbers -- limits

The absolute value of the mantissa is always ≥ 0 and < 1.

The exponent is (also) limited: $E_{\min} \le e \le E_{\max}$.

if b = 16, our E would be FFFF…F

For a set of floating-point numbers in a simplified form, $-E_{\min} = E_{\max} = (aa \cdots a)_b$, where $a = b - 1$. (However, the IEEE Standard uses a slightly different range for the exponent.)

According to the above simplified form,

in this case, b = 16, N would be: (.EEEE…E) x 16^(EE…E)

the largest floating-point number (often called **overflow level**, OFL) is
$N_{\max} = (.\,aa \cdots a)_b \times b^{(aa \cdots a)_b}$, where $a = b - 1$, while
the smallest in absolute value nonzero floating-point number (often called **underflow level**, UFL) is
$N_{\min} = (.\,100 \cdots 0)_b \times b^{-(aa \cdots a)_b}$ if normalised, or
$N_{\min} = (.\,00 \cdots 01)_b \times b^{-(aa \cdots a)_b}$ else.
From now on, we assume normalised numbers (mantissae), unless otherwise stated.

Set of floating-point numbers

Notation: $\mathbb{R}_b(t, s)$ denotes the set of all base b floating-point numbers that can be represented by t b-digits mantissa (incl. sign), and s b-digits exponent (incl. sign).
Note: $\mathbb{R}_b(t, s) \subset [-OFL, -UFL] \cup \{0\} \cup [UFL, OFL]$.

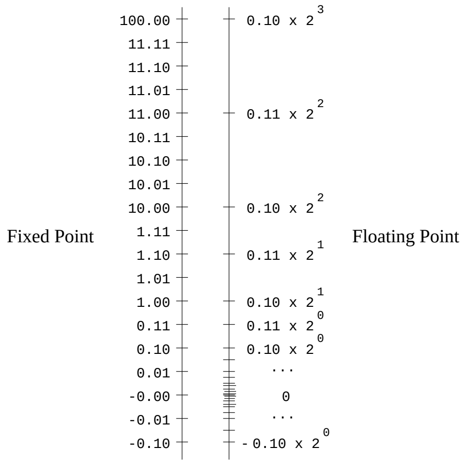
Overflow occurs whenever a floating-point number greater than $N_{\max}$ or smaller than $-N_{\max}$ has to be stored in the computer.
Underflow occurs whenever a nonzero floating-point number in the range $(-N_{\min}, N_{\min})$ has to be stored in the computer. (Some exceptions may apply in IEEE Standard.)

Notes:

- $\mathbb{R}_b(t, s)$ is finite, while $\mathbb{R}$ is infinite.
- $\mathbb{R}$ is dense, while $\mathbb{R}_b(t, s)$ is not. There are many "holes" in the real axis, which are not uniform, and are not represented on the computer.
- The representable numbers are concentrated towards 0.

越接近0，越密集；越远离0，“洞”越多

Comparing floating-point to fixed-point numbers



Fixed-point numbers look like integers multiplied by some (negative) power of the base. The number of digits to the left and to the right of the dot is fixed and there is no exponent.

Approximation of real numbers by floating-point numbers

A real number $x = \pm(x_I . x_F)_b = \pm(d_k d_{k-1} \cdots d_0 . d_{-1} \cdots)_b$ can be easily represented in a floating-point form (assuming no overflow or underflow occurs). First normalise the mantissa:

$$x = (d_k \cdots d_0 . d_{-1} \cdots)_b = (.\,D_1 D_2 \cdots)_b \times b^{k+1}$$

Then, there are two common ways to convert $x \in \mathbb{R}$ to a floating-point number $fl(x) \in \mathbb{R}_b(t, s)$.

(a) **Chopping**: chop after digit t of the mantissa.

(b) **Rounding (traditional)**: chop after digit t, then round D_t up or down, depending on whether $D_{t+1} \ge b/2$ or $D_{t+1} < b/2$ respectively, or in other words, add $b/2$ to D_{t+1}, then chop.

“四舍五入”：对于10进制来说；如果是16进制那就是“七舍八入” etc.

(c) **Rounding (proper or perfect)**: same as traditional, except when $D_{t+1} = b/2$ and $D_{t+2} = D_{t+3} = \cdots = 0$; then round D_t up or down, to the nearest even.

0.85 round to 1-significant digit, then should be 0.8 instead of 0.9

Approximation of real numbers by floating-point numbers

- Examples (in 2 decimal digits) (*):

x		chop	trad. round	proper round
2/3	.666...	.66	.67	.67
-3.05	$-.305 \times 10^1$	$-.30 \times 10^1$	$-.31 \times 10^1$	$-.30 \times 10^1$
-3.15	$-.315 \times 10^1$	$-.31 \times 10^1$	$-.32 \times 10^1$	$-.32 \times 10^1$
-3.155	$-.3155 \times 10^1$	$-.31 \times 10^1$	$-.32 \times 10^1$	$-.32 \times 10^1$
-3.055	$-.3055 \times 10^1$	$-.30 \times 10^1$	$-.31 \times 10^1$	$-.31 \times 10^1$

- Examples (in 2 binary digits) (*):

x		chop	trad. round	proper round
.101	$.101 \times 2^0$	$.10 \times 2^0$	$.11 \times 2^0$	$.10 \times 2^0$
.111	$.111 \times 2^0$	$.11 \times 2^0$	$.10 \times 2^1$	$.10 \times 2^1$

* We assume no overflow or underflow while rounding.

The IEEE Standard

In 1985, the Institute of Electrical and Electronics Engineers (IEEE) approved the so-called **IEEE Standard** for binary floating-point numbers representation and arithmetic operations. According to the IEEE Standard, the form for representing floating-point number is slightly more sophisticated than the above simplified form.

Without getting into details, IEEE Standard floating-point numbers are of the form

$$(-1)^q \times (d_0 . d_1 d_2 \cdots d_{t-1})_b \times 2^e$$

where $q = 0$ or 1, $(-1)^q$ is the sign of the number,
$d_i = 0$ or 1, for $i = 0, \cdots, t-1$, and
$E_{\min} \le e \le E_{\max}$.

There are numbers of **single precision** (binary32), **double precision** (binary64), and **quadruple precision** (binary128), with the characteristics as given in the table below.

The IEEE Standard

type of number	no. of bits	t	$E_{\min}$	$E_{\max}$	$\varepsilon_{\text{mach}}$
single precision, binary32	32	24 (=23+1)	-126	+127	1.2×10^{-7}
double precision, binary64	64	53 (=52+1)	-1022	+1023	2.2×10^{-16}
quadruple prec., binary128	128	113 (=112+1)	-16382	+16383	1.9×10^{-34}

The mantissa is normalised: $d_0 = 1$ (or else the number is 0); thus, d_0 need not be stored, and a t-bit mantissa needs $t-1$ storage bits.
The IEEE Standard uses proper rounding.

The IEEE Standard also includes some "special" numbers, for example +∞, -∞, NaN (Not-a-Number), to handle indeterminate values that may arise in some computations.

The IEEE Standard is adopted today by many modern computer systems.

The respective characteristics in a decimal system are given (approximately) below:

type of number	t	$E_{\min}$	$E_{\max}$
single precision, binary32	7	-38	+38
double precision, binary64	16	-308	+308
quadruple prec., binary128	34	-4928	+4928

About $\varepsilon_{\text{mach}}$, later.

Round-off error (representation error)

The difference between x and $fl(x)$ is called the **round-off** error. It is roughly proportional to x, so we can write $fl(x) - x = x\delta$, or $fl(x) = x(1+\delta)$, where δ is the **relative round-off error**. Note that δ may depend on x but there is a bound for it, independent of x, called **unit round-off** and often denoted by μ or u.

- $|\delta| \le b^{1-t}$ if normalised numbers and chopping are assumed,
- $|\delta| \le \frac{1}{2} b^{1-t}$ if normalised numbers and rounding are assumed.

calculation is shown in tut02.pdf

An approximation $\hat{x}$ to x is said to be **correct in r significant b-digits**, if $\left| \frac{x - \hat{x}}{x} \right| \le \frac{1}{2} b^{1-r}$ ($= 5b^{-r}$ if $b = 10$).

Absolute and relative errors

If x is a number and $\hat{x}$ an approximation to x, then
the **(absolute) error** in the approximation is $x - \hat{x}$, and
the **relative error** in the approximation is $\frac{x - \hat{x}}{x}$.

Often, we are interested only in the magnitude of the absolute and relative errors, in which case we take the **absolute value** of the respective errors.

Computer arithmetic

Let $x, y \in \mathbb{R}_b(t, s)$ and $o \in \{+, -, \times, /\}$. Then, for the computer's floating-point operation $\bar{o}$ corresponding to o, the following holds in general:

$$\underline{x \; o \; y \neq fl(x \; \bar{o} \; y)}$$

The operation $\bar{o}$ may vary from machine to machine, but, in general, it is constructed so that

$$x \; \bar{o} \; y = fl(x \; o \; y)$$

This is achieved with the use of one (or two) extra temporary digits (bits).

Thus, **the error of a computation is the round-off error of the correct result**.
In other words, computer operations return the correctly rounded result (i.e. the number closest to the correct answer in the representation being used).

Computer arithmetic

- Examples (in 2 decimal digits, proper rounding)

$$x = .2 \times 10^1, \; y = .51 \times 10^{-5}, \; x + y = .2000051 \times 10^1$$
$$x \mathbin{\bar{+}} y = fl(x + y) = .2 \times 10^1$$

$$x = .2 \times 10^1, \; y = .51 \times 10^{-2}, \; x + y = .2051 \times 10^1$$
$$x \mathbin{\bar{+}} y = fl(x + y) = .21 \times 10^1$$

$$x = .2 \times 10^1, \; y = .50 \times 10^{-2}, \; x + y = .2050 \times 10^1$$
$$x \mathbin{\bar{+}} y = fl(x + y) = .2 \times 10^1$$

$$x = .21 \times 10^1, \; y = .50 \times 10^{-2}, \; x + y = .2150 \times 10^1$$
$$x \mathbin{\bar{+}} y = fl(x + y) = .22 \times 10^1$$

Note 1: $\mathbb{R}_b(t, s)$ is not closed w.r.t addition (nor w.r.t other mathematical operations)!
Note 2: The phenomenon in which a non-zero number is added to another and the latter is left unchanged is often referred to as **saturation**.

Computation of functions

Similarly, the computer application $\bar{f}(x)$ of a function f to a number $x \in \mathbb{R}_b(t, s)$ is constructed so that

just like computer arithmetic

$$\bar{f}(x) = fl(f(x))$$

that is, the error of computing a function value is the round-off error of the correct result.

- Example (in 2 decimals)

$\sin(1) = 0.84147098\ldots, \quad \overline{\sin}(1) = fl(\sin(1)) = fl(0.84147098\ldots) = 0.84$

Machine epsilon

The smallest (non-normalised) floating-point number $\varepsilon_{\text{mach}}$ with the property $1 + \varepsilon_{\text{mach}} > 1$, is called the **machine epsilon** (in short, **mach-eps**).
$\varepsilon_{\text{mach}} = b^{1-t}$ if chopping,
$\varepsilon_{\text{mach}} = \frac{1}{2} b^{1-t}$ if traditional rounding and

Imptt

$\frac{1}{2} b^{1-t} < \varepsilon_{\text{mach}} \le b^{1-t}$, ($\varepsilon_{\text{mach}} = \frac{1}{2} b^{1-t} + b^{-t}$) if proper rounding.
Thus, $-\varepsilon_{\text{mach}} \le \delta \le \varepsilon_{\text{mach}}$.

For IEEE Standard numbers, $\varepsilon_{\text{mach}} = 2^{1-t}$, i.e.
for single precision, $\varepsilon_{\text{mach}} = 2^{-23} \approx 1.19209 \times 10^{-7}$, and
for double precision, $\varepsilon_{\text{mach}} = 2^{-52} \approx 2.22045 \times 10^{-16}$.
for quadruple precision, $\varepsilon_{\text{mach}} = 2^{-112} \approx 1.9259 \times 10^{-34}$.

Note 1: There are alternative definitions of $\varepsilon_{\text{mach}}$ that result in slightly different values, still of the same order.

Note 2: We have $0 < \text{UFL} < \varepsilon_{\text{mach}} < \text{OFL}$.

Error propagation in simple arithmetic calculations

Recall that the relative round-off error δ in the floating-point representation $fl(x)$ of a number x (i.e. $\delta = \frac{fl(x) - x}{x}$) satisfies $-\varepsilon_{\text{mach}} \le \delta \le \varepsilon_{\text{mach}}$.

As soon as an error (round-off or by a floating-point operation) rises, it may then be amplified or reduced in subsequent operations.

反正就不相等了

Let $x, y \in \mathbb{R}$. Then $fl(x) = x(1+\delta_1)$, $fl(y) = y(1+\delta_2)$

(a) Multiplication

Assume we want to multiply x and y. Then the computer computes

$$\begin{aligned} fl(x) \,\bar{\times}\, fl(y) &= fl(fl(x) \times fl(y)) = (x(1+\delta_1)y(1+\delta_2))(1+\delta_3) = \\ &= xy(1+\delta_1+\delta_2+\delta_1\delta_2)(1+\delta_3) = \\ &= xy(1+\delta_1+\delta_2+\delta_1\delta_2+\delta_3+\delta_1\delta_3+\delta_2\delta_3+\delta_1\delta_2\delta_3) \approx \\ &\approx xy(1+\delta_1+\delta_2+\delta_3) = xy(1+\delta_\times) \end{aligned}$$

where $|\delta_\times| \le 3\varepsilon_{\text{mach}}$

(b) Division Similar (do it!).

Error propagation in simple arithmetic calculations

(c) Addition

Assume we want to add x and y. Then the computer computes

$$\begin{aligned} fl(x) \,\bar{+}\, fl(y) &= fl(fl(x) + fl(y)) = (x(1+\delta_1) + y(1+\delta_2))(1+\delta_3) = \\ &= x(1+\delta_1)(1+\delta_3) + y(1+\delta_2)(1+\delta_3) \approx x(1+\delta_1+\delta_3) + y(1+\delta_2+\delta_3) = \\ &= (x+y)(1 + \frac{x}{x+y}(\delta_1+\delta_3) + \frac{y}{x+y}(\delta_2+\delta_3)) = (x+y)(1+\delta_+) \end{aligned}$$

where

$$\begin{aligned} |\delta_+| &\le \left|\frac{x}{x+y}\right| \cdot 2\varepsilon_{\text{mach}} + \left|\frac{y}{x+y}\right| \cdot 2\varepsilon_{\text{mach}} = \frac{|x|+|y|}{|x+y|} \cdot 2\varepsilon_{\text{mach}} \\ &= 2\varepsilon_{\text{mach}} \quad \text{when} \quad xy > 0 \\ &= 2\varepsilon_{\text{mach}} \frac{|x-y|}{|x+y|} \quad \text{when} \quad xy < 0 \end{aligned}$$

Consider the case $x \approx -y$. The (relative) error blows up.

Example of catastrophic cancellation

- Example (in 8 decimal digits with chopping)

Consider the real numbers
$x = .123456790 \times 10^0$ and
$y = .123456789 \times 10^0$.
Then compute the difference

$$fl(fl(x) - fl(y)) = .00000001 \times 10^0 = .10000000 \times 10^{-7},$$

while

$$x - y = .000000001 \times 10^0 = .10000000 \times 10^{-8}.$$

In the above computed $fl(fl(x) - fl(y))$ no significant decimal digits are correct. Although the (relative) error in the floating-point representation of y is at the level of 10^{-8}, the (relative) error in $fl(fl(x) - fl(y))$ is too high (at the level of 10^0).

$$\frac{.1 \times 10^{-8} - .1 \times 10^{-7}}{.1 \times 10^{-8}} = \frac{-.9 \times 10^{-8}}{.1 \times 10^{-8}} = -9 \times 10^0$$

Important note

Adding nearly opposite (or subtracting nearly equal) numbers may result in having no correct digits at all. In some cases though, we are able to eliminate this **cancellation phenomenon** (often referred to as **catastrophic cancellation**).

In general, most arithmetic operations do not give rise to huge errors. Adding nearly opposite (or subtracting nearly equal) numbers is an exception.

Error propagation in computation: conditioning of problems

Consider some computation with input $x \in \mathbb{R}$ and output $f(x) \in \mathbb{R}$.
Let $fl(x) = x(1+\delta_x)$, and assume $f(x)$ is twice differentiable in a neighbourhood of x. Then, if $f(x) \neq 0$, using Taylor's series and ignoring terms of second or higher order ($O(\delta_x^2)$), we get

$$f(fl(x)) \approx f(x)(1+\delta_f) \text{ with } \delta_f = \frac{xf'(x)}{f(x)}\delta_x.$$

If $|\delta_x| \le \varepsilon_{\text{mach}}$, we get

$$|\delta_f| \le \left|\frac{xf'(x)}{f(x)}\right|\varepsilon_{\text{mach}}.$$

The factor $\kappa_f = \left|\frac{xf'(x)}{f(x)}\right|$ is called **(relative) condition number** of $f(x)$ and is a measure of the relative sensitivity of the computation of $f(x)$ on relatively small changes in the input x, or in other words a measure of how the relative error in x propagates in $f(x)$.

A computation is called **well-conditioned** if relatively small changes in the input, produce relatively small changes in the output, otherwise it is called **ill-conditioned**.

Note: Once $f(fl(x))$ is computed, it is stored/represented as $fl(f(fl(x))) = fl(f(x)(1+\delta_f)) = f(x)(1+\delta_f)(1+\delta) \approx f(x)(1+\delta_f+\delta)$, where $|\delta| \le \varepsilon_{\text{mach}}$.

Error propagation in computation: conditioning of problems

- Example: Let $f(x) = \sqrt{x}$. Then, the condition number of $f(x)$ is $\left|\frac{xf'(x)}{f(x)}\right| = \frac{1}{2}$. For this function, the condition number happens to be small (and independent of x). Thus the computation of the square root of a number is a well-conditioned computation.

- Example: Let $f(x) = e^x$. Then, the condition number of $f(x)$ is $\left|\frac{xf'(x)}{f(x)}\right| = |x|$. For this function, the condition number depends (actually linearly) on x. For small $|x|$, the condition number is small, for large $|x|$, it is large. However, for large $|x|$, we will have overflow (note: $e^{700} \approx 10^{304}$), thus we can say that the computation of the exponential is well-conditioned for all acceptable values of x.
- Note 1: If, for some function, $\kappa_f > \varepsilon_{\text{mach}}{}^{-1}$, we risk having no correct digits at all in the computed result $\overline{f(x)}$.
- Note 2: The condition number of a function is a property inherent to the function itself, and not to the way the function is computed.

Error propagation in computation: stability of algorithms

Stability is a concept similar to conditioning, but it refers to a numerical algorithm, i.e. to the particular way a certain computation is carried out. A numerical algorithm is **stable** if small changes in the algorithm input parameters have a small effect on the algorithm output, otherwise it is called **unstable**.

- Example (in 3 decimal digits with rounding)

$$(a-b)^2 = a^2 - 2ab + b^2$$

Let $a = 15.6$, $b = 15.7$

$$(a-b)^2 = (-.1)^2 = .01 \to .1 \times 10^{-1}$$

$$a^2 - 2ab + b^2 = 243.36 - 2 \times 244.92 + 246.49$$

$$\to 243 - 490 + 246 = 489 - 490 = -1 = -.1 \times 10^1$$

Computing the rhs we don't even get the sign correct! In this case, computing the lhs is a more stable algorithm to compute the square of the difference of two numbers, than computing the rhs.

Moral: Mathematically equivalent expressions are not necessarily computationally equivalent.

Error propagation in computation: stability of algorithms

General considerations

Mathematically equivalent expressions are not necessarily computationally equivalent.

There is no general rule to pick the most stable algorithm for a certain computation. However, we make an effort

Imptt

- to avoid adding nearly opposite (or subtracting nearly equal) numbers,
- to minimize the number of operations,
- when adding several numbers, to add them starting from the smallest and proceeding to the largest.
- to be alert when adding numbers of very different scales.

No rule, though, guarantees success. Each case may be special.

Also, keep in mind, that picking a stable algorithm for a certain computation does not improve the condition number of the computation.

Forward and backward errors

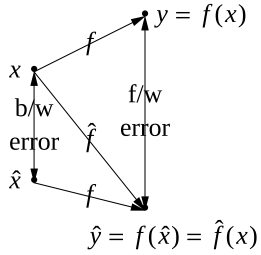
Let x be a number and let $y = f(x)$ be the desired result of some computation with input x.
Assume that instead of y, we compute $\hat{y}$ due to various errors (e.g. round-off error in x, propagation of error).
Then, $y - \hat{y}$ is called **forward error**. The forward error may include initial data error in x, as well as propagation of error in computations.
If all errors were due to computations, one can view $\hat{y}$ as being the result of inexact computations on exact input: $\hat{y} = \hat{f}(x)$.
If all errors were due to initial data error, one can view $\hat{y}$ as being the result of exact computations on inexact input: $\hat{y} = f(\hat{x})$.
Thus $\hat{f}(x) = f(\hat{x})$.
Assume the inverse function f^{-1} of f exists. The relation $\hat{f}(x) = f(\hat{x})$ defines $\hat{x}$, i.e. $\hat{x} = f^{-1}(\hat{f}(x))$.
Then, $x - \hat{x}$ is called **backward error**.

Forward and backward errors

For both forward and backward errors, the respective relative versions are

relative forward error = $\dfrac{y - \hat{y}}{y}$,

relative backward error = $\dfrac{x - \hat{x}}{x}$.



The following relation holds:
| relative forward error | ≈ condition number × | relative backward error|

Forward and backward errors -- example

Let $x = 2$ and $f(x) = \sqrt{x}$. Let $\hat{y} = 1.4$. Notice that $\sqrt{1.96} = 1.4$, thus $\hat{x} = 1.96$.

Forward error = $1.4142... - 1.4 = 0.0142...$
Relative forward error = $(1.4142... - 1.4)/1.4142... = 0.0142.../1.4142... \approx 0.01$
Backward error = $2 - 1.96 = 0.04$
Relative backward error = $(2 - 1.96)/2 = 0.04/2 = 0.02$
Condition number = $1/2$

Truncation (discretization) and rounding errors

We have seen that the computer representation of numbers may involve some round-off (data) error, which may be propagated through simple arithmetic operations or various other computations (e.g. function evaluations).

Sometimes, certain mathematical expressions are approximated by other (possibly more convenient for calculations) expressions. The latter expressions can be viewed as algorithms for approximating the original expressions. The error in these approximations is referred to as **truncation** or **discretization** error.

In addition, the evaluation of the approximate expressions is not performed in exact but in finite arithmetic. Therefore, it may involve additional error, referred to as **rounding** error.

The truncation and the rounding errors together form the **computational** error.

Total error

Let the input data be x, its computer representation be $\hat{x}$, the target computation be $f(x)$. Let's approximate $f(x)$ by $g(x)$, while the computer evaluates $\hat{g}(x)$ instead. Then the total error is

$f(x) - \hat{g}(\hat{x}) = (f(x) - f(\hat{x})) + [(f(\hat{x}) - g(\hat{x})) + (g(\hat{x}) - \hat{g}(\hat{x}))]$

= (propagated data error) + [computation error]

= (propagated data error) + [(truncation error) + (rounding error)].

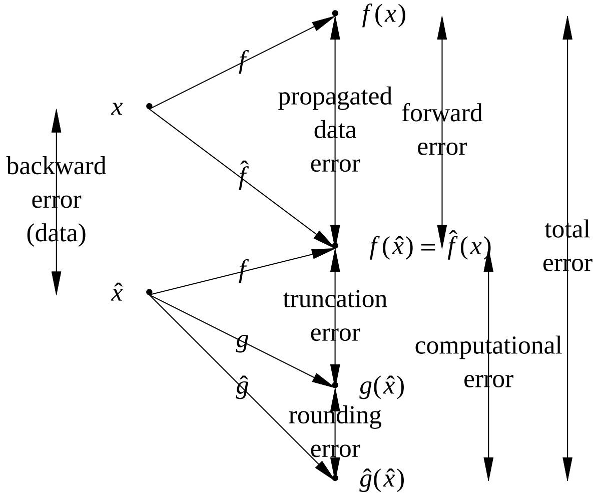
Often, one of these errors is dominant, but there is no general rule that tells us which.

Total error

Total error $= f(x) - \hat{g}(\hat{x}) = (f(x) - f(\hat{x})) + [(f(\hat{x}) - g(\hat{x})) + (g(\hat{x}) - \hat{g}(\hat{x}))]$

= (propagated data error) + [computation error]

= (propagated data error) + [(truncation error) + (rounding error)].



Total error -- example

Assume we want to compute $\sin(\frac{\pi}{8})$. (Note: $\sin(\frac{\pi}{8}) = 0.38268343\ldots$).

Let π be approximated by 3 ($x = \pi$, $\hat{x} = 3$), $\sin(x)$ be approximated by $x - \frac{x^3}{3!}$ ($f(x) = \sin(x)$, $g(x) = x - \frac{x^3}{3!}$), and the computation be done in 3-decimal digit floating-point arithmetic. The result will be $fl(fl(\frac{3}{8}) - fl[fl(\frac{1}{3!})\, fl(fl(\frac{3}{8})^3)]) \approx 0.366$, where all intermediate computations are performed in 3 decimal digits.

The initial data error is $x - \hat{x} = \pi - 3 \approx 0.1415926536$,
the propagated data error is $f(x) - f(\hat{x}) = \sin(\pi/8) - \sin(3/8) \approx 0.0164109$,
the truncation error is $f(\hat{x}) - g(\hat{x}) = \sin(3/8) - 3/8 + (3/8)^3/6 \approx 0.0000615915865$ and
the rounding error is approximately $g(\hat{x}) - \hat{g}(\hat{x}) =$
$3/8 - (3/8)^3/6 - fl(3/8 - (3/8)^3/6) = 0.3662109375 - 0.366 = 0.0002109375$.
Total error $f(x) - \hat{g}(\hat{x}) = 0.38268343\ldots - 0.366 = 0.01668343\ldots$.

General conclusions

Mathematical operations are not always equivalent to the respective computational operations.

Not all the mathematical formulae and other properties hold necessarily when computer arithmetic is used.

In most cases, errors arise, which are usually small.

We need to be alert for cases where the error is amplified.

Error analysis, especially the study of the relation between the input data error and the final result error, helps in understanding the source of large errors and possibly avoiding them.

The use of maximum available precision is suggested.

Taylor's Theorem

Let $k \geq 1$ be an integer, and $a \in \mathbb{R}$. Assume $f : \mathbb{R} \to \mathbb{R}$ be $k+1$ times differentiable on the open interval and continuous on the closed interval between a and $x \in \mathbb{R}$. Then, there exists ξ between x and a, such that

$$f(x) = f(a) + f'(a)(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \cdots + \frac{f^{(k)}(a)}{k!}(x-a)^k + \frac{f^{(k+1)}(\xi)}{(k+1)!}(x-a)^{k+1} \quad (1)$$

All but the last term in the right side of (1) is the so-called *Taylor polynomial* $t_k(x)$ of degree k, i.e.

$$t_k(x) = f(a) + f'(a)(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \cdots + \frac{f^{(k)}(a)}{k!}(x-a)^k \quad (2)$$

while the last term $R_{k+1}(x) \equiv \frac{f^{(k+1)}(\xi)}{(k+1)!}(x-a)^{k+1}$ is referred to as the *remainder*. When x is close to a (so that $|x-a|$ is small), the remainder is small, and the Taylor's polynomial is a good approximation to $f(x)$. Thus,

$$f(x) = t_k(x) + R_{k+1}(x) \approx t_k(x) \quad (3)$$

Taylor's theorem has many mathematical and computational applications. The most frequent computional application is that it provides a way to approximate a function $f(x)$ in a neighbourhood of a point a, by the respective Taylor polynomial $t_k(x)$.

Taylor's theorem, expansion, series

Relation (1) (repeated here for convenience)

$$f(x) = f(a) + f'(a)(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \cdots + \frac{f^{(k)}(a)}{k!}(x-a)^k + \frac{f^{(k+1)}(\xi)}{(k+1)!}(x-a)^{k+1} \quad (1)$$

is often presented as

$$f(x+h) = f(x) + f'(x)h + f''(a)\frac{h^2}{2!} + \cdots + f^{(k)}(a)\frac{h^k}{k!} + f^{(k+1)}(\xi)\frac{h^{k+1}}{(k+1)!} \quad (4)$$

as the roles of x and a in (1) are played by $x+h$ and x in (4) (and the role of $x-a$ in (1) is played by $x+h-x=h$ in (4)).
Relation (1) is also referred to as Taylor's *expansion* of $f(x)$ about point a, and relation (4) as Taylor's expansion of $f(x+h)$ about x.

Assume now that f is infinitely differentiable. The *infinite Taylor series* of $f(x)$ is given by

$$f(a) + f'(a)(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \cdots + \frac{f^{(k)}(a)}{k!}(x-a)^k + \frac{f^{(k+1)}(a)}{(k+1)!}(x-a)^{k+1} + \cdots \quad (5)$$

$$= \sum_{k=0}^{\infty} \frac{f^{(k)}(a)}{k!}(x-a)^k$$

If the infinite series converges to $f(x)$ for x close enough to a, then $f(x)$ is called *analytic* function.

Typical analytic functions and respective Taylor's series

$$e^x = 1 + x + \frac{x^2}{2!} + \cdots = \sum_{k=0}^{\infty} \frac{x^k}{k!} \quad (6)$$

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \cdots = \sum_{k=0}^{\infty} (-1)^k \frac{x^{2k+1}}{(2k+1)!} \quad (7)$$

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \cdots = \sum_{k=0}^{\infty} (-1)^k \frac{x^{2k}}{(2k)!} \quad (8)$$

$$\log(x) = (x-1) - \frac{(x-1)^2}{2} + \frac{(x-1)^3}{3} - \cdots = \sum_{k=1}^{\infty} (-1)^{k+1} \frac{(x-1)^k}{k} \quad (9)$$

$$\log(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \cdots = \sum_{k=1}^{\infty} (-1)^{k+1} \frac{x^k}{k} \quad (10)$$

Note that (6), (7) and (8) converge for any x, while (9) converges for $0 < x \leq 2$, and (10) for $-1 < x \leq 1$.

Note also that (6), (7) and (8) arise from Taylor's expansions of the respective functions about 0 ($a = 0$), while (9) from Taylor's expansion of $\log(x)$ about 1 ($a = 1$).

Approximating functions by Taylor polynomials

Recall the approximation of a function $f(x)$ by its Taylor polynomial $t_k(x)$:

$$f(x) = t_k(x) + R_{k+1}(x) \approx t_k(x) \quad (3)$$

In this approximation, the *truncation error* is the remainder $R_{k+1}(x)$.

When we approximate a function $f(x)$ by its Taylor polynomial $t_k(x)$, there are some questions that need answers:
How many terms in the Taylor expansion (how high a degree Taylor polynomial) should we use

- to keep the truncation error below a certain tolerance?
- to keep the computational error below a certain tolerance?

There are several sophisticated techniques that attempt to answer these questions.
Most frequently used techniques:
To answer the first question, we study (mathematically) for which k the remainder term (in abs. value) becomes bounded by the tolerance, for all x and ξ of interest.
To answer the second question, we keep adding more terms until the difference in the resulting approximation to the function is small enough (or until it makes no difference, i.e. sum to saturation).