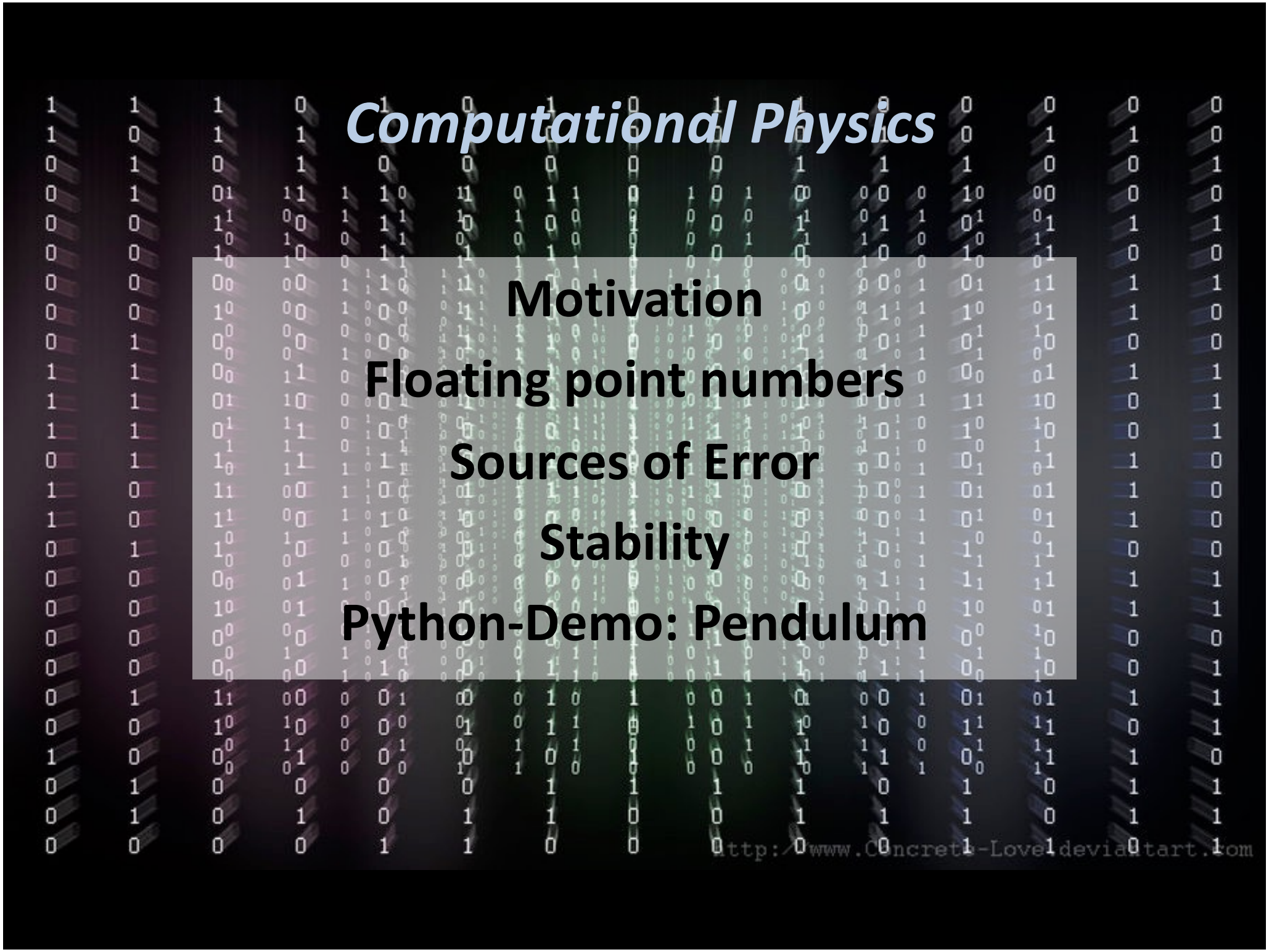

Computational Physics
Motivation
Floating point numbers
Sources of Error
Stability
Python-Demo: Pendulum

Motivation

Computational Physics aims at **solving physical problems by means of numerical methods** developed in the field of numerical analysis, which is concerned with the development and analysis of methods for the numerical solution of practical problems.

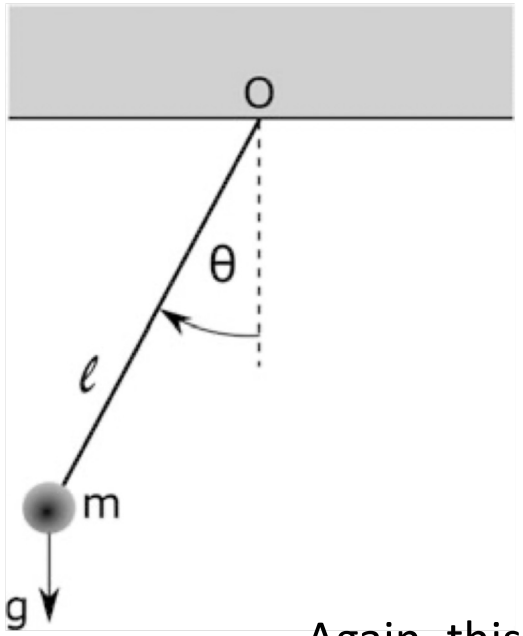
Example of a problem without analytical solution (the error "function"):

$$\int_a^b \mathrm{d}x \exp\left(-x^2\right)$$

Typical problem to find probability to get values in an interval [a,b] $\in \mathbb{R}$ for a normal distribution.
No analytical solution [unless erf(x) is considered a solution] in contrast to e.g.

$$\int_a^b \mathrm{d}x \exp\left(x\right) = \exp\left(b\right) - \exp\left(a\right)$$

Example of a physics problem, not-analytically solvable* is the pendulum, described by the equation of motion:



$$\ddot{\theta} + \frac{g}{\ell}\sin(\theta) = 0$$

Usually combined with the initial conditions:

$$\begin{cases} \theta(0) = \theta_0 \ , \\ \dot{\theta}(0) = 0 \ . \end{cases}$$

Again, this is in contrast to the much simpler harmonic oscillator, which is obtained for $\theta_0 << 1$:

$$\ddot{\theta} + \frac{g}{\ell}\theta = 0$$

which has the solution: $\theta(t) = \theta_0 \cos(\omega t)$ with $\omega = \sqrt{g/\ell}$ → $\tau = 2\pi\sqrt{\frac{\ell}{g}}$

* Some properties can be described by the elliptic integral of first kind (e.g. $\tau = 4\sqrt{\frac{\ell}{g}}K_1(k)$).

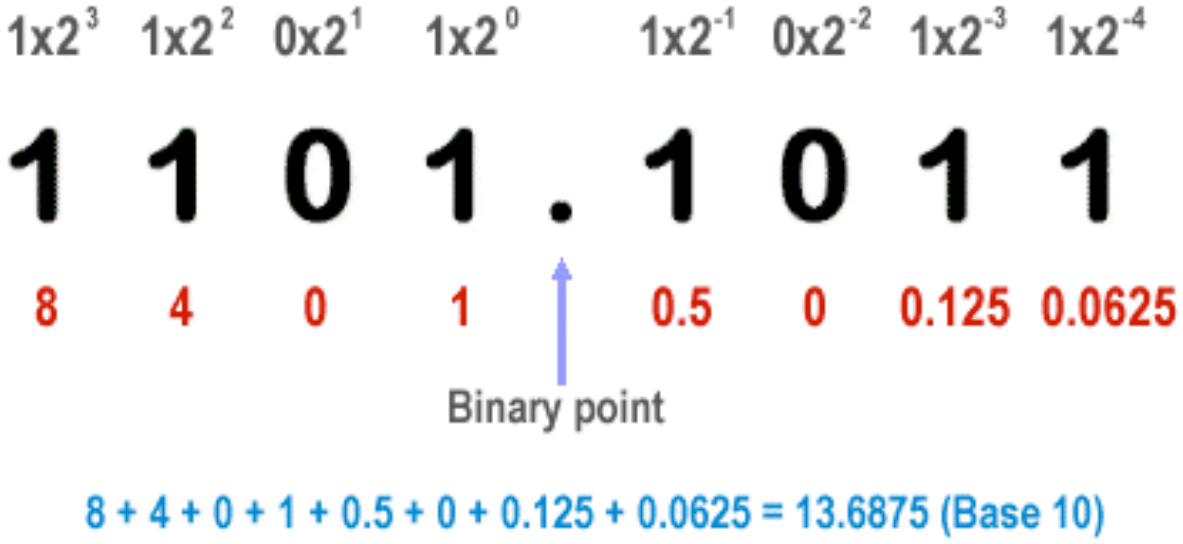
But before we solve these problems:

Basics of numerical calculations

Binary representation

$$x = \pm(\alpha_n 2^n + \alpha_{n-1} 2^{n-1} + \cdots + \alpha_0 2^0 + \alpha_{-1} 2^{-1} + \alpha_{-2} 2^{-2} + \cdots)$$

α_i=0 or 1



8 + 4 + 0 + 1 + 0.5 + 0 + 0.125 + 0.0625 = 13.6875 (Base 10)

Hexadecimal numbers (base 16): {0,1,…,15} = {0,1,…,9,a,b,..,f}

Floating point numbers

Definitions

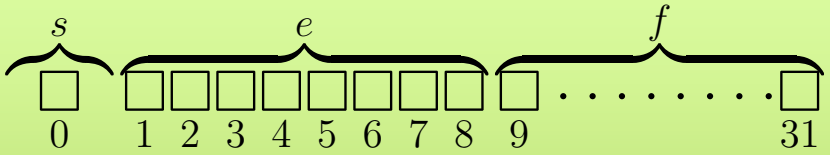
Bit = 0 or 1

Byte = 8bits

Word = Reals: 4 bytes (single precision)
8 bytes (double precision)
Integers: 1, 2, 4, or 8 byte signed
1, 2, 4, or 8 byte unsigned

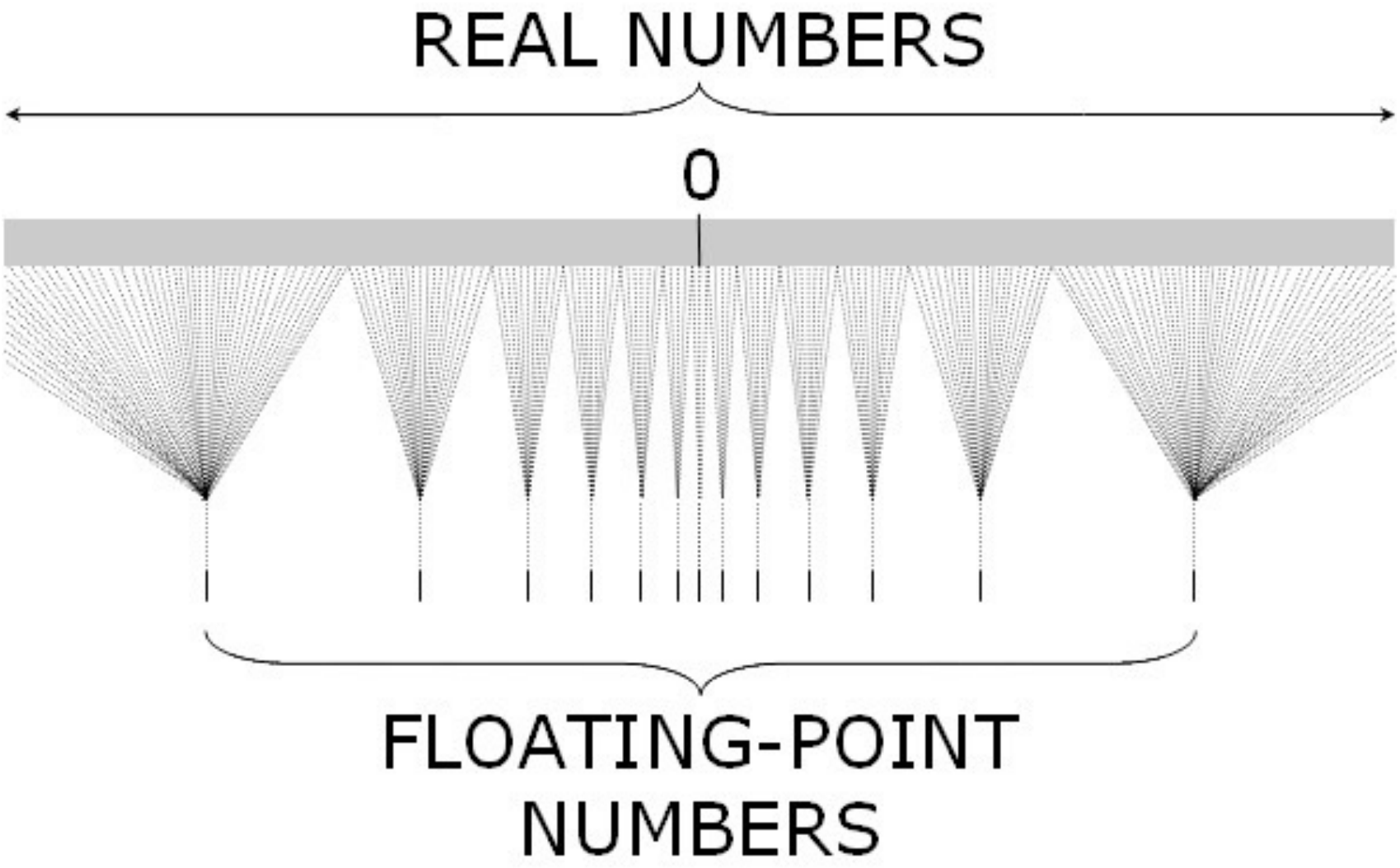
IEEE single precision format:

"float="



$$x = (-1)^s \times 2^{e-127} \times 1.f$$

s – sign, e – biased exponent, 1.f – mantissa/significand


REAL NUMBERS
0
FLOATING-POINT
NUMBERS

Single precision, special numbers

Smallest exponent:	e = 0000 0000, represents denormal numbers (1.f $\rightarrow$ 0.f)	unless f=0
Largest exponent:	e = 1111 1111, represents $\pm\infty$, if f = 0	
	e = 1111 1111, represents NaN, if f $\neq$ 0	
Number Range:	e = 1111 1111 = 2^8 - 1 = 255	reserved
	e = 0000 0000 = 0	reserved
so,	p = e - 127 is	
	1 - 127 $\leq$ p $\leq$ 254-127	
	-126 $\leq$ p $\leq$ 127	

Smallest positive normal number

= 1.0000 0000 $\cdots\cdots$ 0000$\times$ 2^{-126}

$\simeq 1.2 \times 10^{-38}$

bin: 0000 0000 1000 0000 0000 0000 0000 0000

hex: 00800000

MATLAB: realmin('single')

Largest positive number

= 1.1111 1111 $\cdots\cdots$ 1111$\times$ 2^{127}

$= (1 + (1 - 2^{-23})) \times 2^{127}$

$\simeq 2^{128} \simeq 3.4 \times 10^{38}$

bin: 0111 1111 0111 1111 1111 1111 1111 1111

hex: 7f7fffff

MATLAB: realmax('single')

Zero

bin: 0000 0000 0000 0000 0000 0000 0000 0000

hex: 00000000

Subnormal numbers

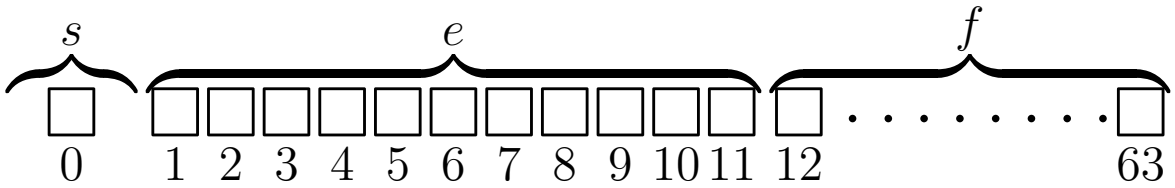
Allow 1.f $\rightarrow$ 0.f (in software)

Smallest positive number = 0.0000 0000 $\cdots\cdots$ 0001 $\times$ 2^{-126}

$= 2^{-23} \times 2^{-126} \simeq 1.4 \times 10^{-45}$

Double precision

"double="



$$x = (-1)^s \times 2^{e-1023} \times 1.f$$

On average, on a PC of year 2012 build, calculations with double precision are 1.1–1.6 times slower than with single precision.

Max(double)=$(1 + (1 - 2^{-52})) \times 2^{1023} \approx 1.7976931348623157 \times 10^{308}$

Min(double>0)=$2^{-1022} \approx 2.2250738585072014 \times 10^{-308}$

Subnormal, min =$2^{-1022-52} \approx 4.9406564584124654 \times 10^{-324}$

Between 2^{52}=4,503,599,627,370,496 and 2^{53}=9,007,199,254,740,992 the representable numbers are exactly the integers. For the next range, from 2^{53} to 2^{54}, everything is multiplied by 2, so the representable numbers are the even ones, etc. Conversely, for the previous range from 2^{51} to 2^{52}, the spacing is 0.5, etc.

*Definition "**Machine epsilon**"* (ϵ_{mach}): the distance between 1 and the next largest number.

$$= \min_{\delta} \left\{ \delta > 0 \middle| 1 + \delta > 1 \right\}$$

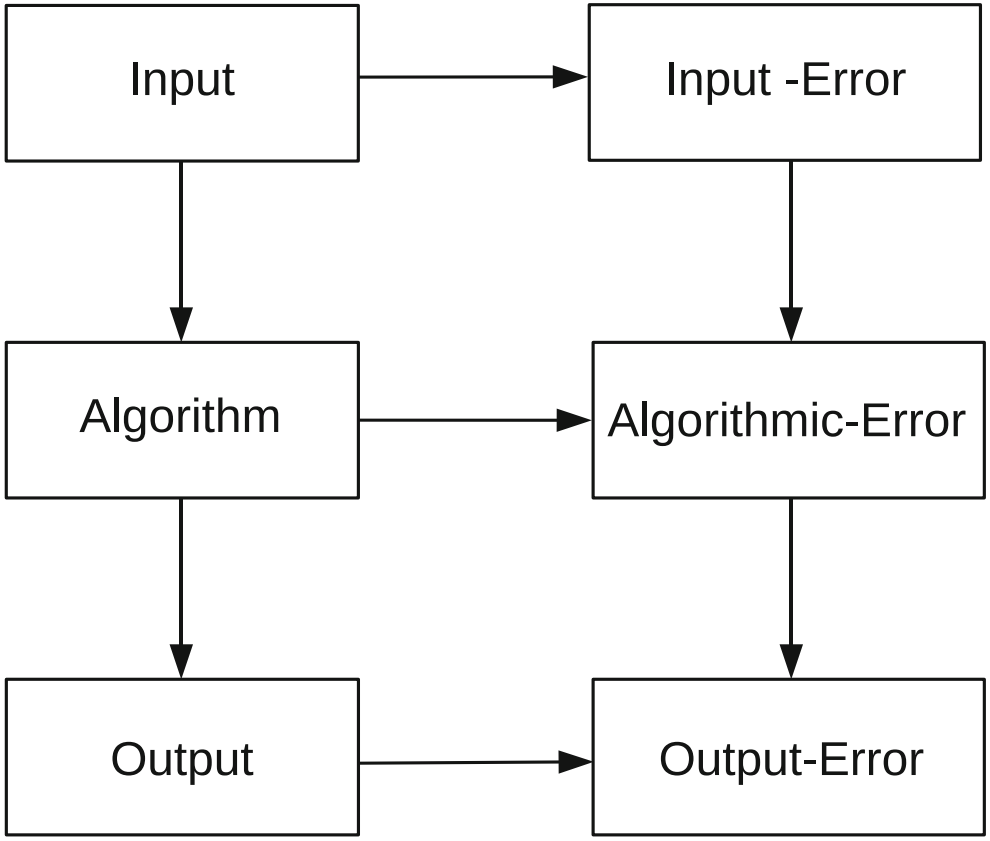
Algorithms

a sequence of logical and arithmetic operations (addition, subtraction, multiplication or division) [on floating point numbers], which allows to approximate the solution of the problem under consideration.

→ numerical errors will be unavoidable

Errors

Schematic classification of the errors occurring within a numerical procedure



roundoff errors and measurement errors contained in the input data, e.g. input as (round) decimal number → converted to floating point

roundoff errors during evaluation and of **methodological errors** due to mathematical approximations

roundoff errors

Main sources of errors

In any applied numerical computation, there are several key sources of error:

Modeling/Measurement errors

i. Inexactness of the mathematical model for the underlying physical phenomenon.

ii. Errors in measurements of parameters entering the model.

These do not concern us too much. The latter is the domain of the experimentalists.

Algorithmic errors

i. **Roundoff errors** in computer arithmetic (finite numerical precision imposed by the computer).

ii. **Discretization (methodological) errors**: continuous processes are replaced by discrete ones (e.g., summation to calculate an integral).

iii. "Termination" errors: infinite algorithms are terminated after a finite number of steps (e.g., iterations like $x_{n+1}=(x_n+a/x_n)/2 \rightarrow \text{sqrt}(a)$, $n \rightarrow \infty$).

The latter two are the true domain of numerical analysis and are a consequence of the fact that

- Most systems of equations are too complicated to solve explicitly
- Even with known analytic solution, directly obtaining the precise numerical values may be difficult

Roundoff Errors

- Example: store 2/3=0.6666666… in a floating point number: With e.g. 10 decimal digit accuracy this would be stored as 0.6666666667 > 2/3
  Or formally with Fl(x) being the floating point form of x

$$\mathrm{Fl}\left(\frac{2}{3}\right) = 0.6666666667 \quad \rightarrow \quad \mathrm{Fl}(\sqrt{3}) \cdot \mathrm{Fl}(\sqrt{3}) \neq \mathrm{Fl}(\sqrt{3} \cdot \sqrt{3}) = 3$$

- In binary, storing 0.1 is also problematic: $0.1_{10} = 0.000110011001100\ldots_2$

The difference between true value y and approximation $\bar{y} \equiv \mathrm{Fl}(y)$ can be characterized by absolute error:

$$\epsilon_a = |y - \bar{y}|$$

and relative error:

$$\epsilon_r = \left|\frac{y - \bar{y}}{y}\right| = \frac{\epsilon_a}{|y|}$$

	y	$\bar{y}$	ϵ_a	ϵ_r
(1)	0.1	0.09	0.01	0.1
(2)	1000.0	999.99	0.01	0.00001

Roundoff error example

Solve the quadratic equation with parameter b:

$$x^2 + 2bx - 1 = 0$$

Yielding:

$$x_\pm = -b \pm \sqrt{b^2+1}$$

Now we look at the solution for b>0 and x>0, i.e.

$$x = -b + \sqrt{b^2+1} \qquad (*)$$

For large b→∞:

$$\begin{aligned} x &= -b + \sqrt{b^2+1} \\ &= -b + b\sqrt{1+1/b^2} \\ &= b(\sqrt{1+1/b^2} - 1) \\ &\approx b\left(1 + \frac{1}{2b^2} - 1\right) \\ &= \frac{1}{2b}. \end{aligned}$$

For x=realmin≈2.2 × 10^{-308} , we get b≈1/(2 × realmin) ≈ 2 × 10^{307}

What would we get from (*)? **x=0 ,** when $b^2 \cong 1+b^2$ or $1+1/b^2 \cong 1$
This happens when $1/b^2 = \epsilon_{mach}/2$! Or $b = \sqrt{2/\epsilon_{mach}} \approx 10^8$

...

In the example this round-off error can be avoided by writing (*) as:

$$x = \frac{1}{b + \sqrt{b^2 + 1}}$$

This gives the correct limit $x \approx 1/(2b)$, when $b^2 \cong 1+b^2$

More examples, error propagation

Let x and y be two real numbers and x* and y* their floating point approximations.

Now, let the computer calculate x-y : First, x and y are replaced by x* and y*. The final result is then (x*-y*)*.

Example: x=301/2000≈.15050000 and y=301/2001≈.150424787
→ The exact result is x-y=301/4002000≈.00007521239
Let us assume we have decimal floating point numbers with 4-digit mantissa, i.e., x*=.1505 and y*=.1504 → d*=(x*-y*)*=0.0001 → $\varepsilon_r(x^*)=0$, $\varepsilon_r(y^*)=10^{-4}$, $\varepsilon_r(d^*)=0.25$

Example: coefficient error in polynomials of 10^{th} degree:

$$p(x) = (x-1)(x-2)(x-3)(x-4)(x-5)(x-6)(x-7)(x-8)(x-9)(x-10)$$

$$q(x) = p(x) + x^5$$

→ Coefficients of the x^5 terms: −902055 in p and −902054 in q
→ relative error of these coefficients 10^{-5}. However, the roots of q are:
1.0000027558, 1.99921, 3.02591, 3.82275,
5.24676 ± 0.751485 i , 7.57271 ± 1.11728 i , 9.75659 ± 0.368389 i.

Methodological Errors

Result from replacement of mathematical expressions by approximate, simpler ones.

E.g. pendulum period:

$$\tau = 4\sqrt{\frac{\ell}{g}}\int_0^{\frac{\pi}{2}} \frac{\mathrm{d}\alpha}{\sqrt{1-k^2\sin^2(\alpha)}} \qquad k = \sin(\theta_0/2)$$

$$= 4\sqrt{\frac{\ell}{g}}K_1(k)\ .$$

Truncate K_1 series at some N:

$$K_1(k) = \frac{\pi}{2}\sum_{n=0}^{\infty}\left[\frac{(2n)!}{2^{2n}(n!)^2}\right]^2 k^{2n}$$

$$= \frac{\pi}{2}\sum_{n=0}^{N}\left[\frac{(2n)!}{2^{2n}(n!)^2}\right]^2 k^{2n} + R_N(k)$$

R_N: **truncation error**

Or for derivatives (Chapter 2):

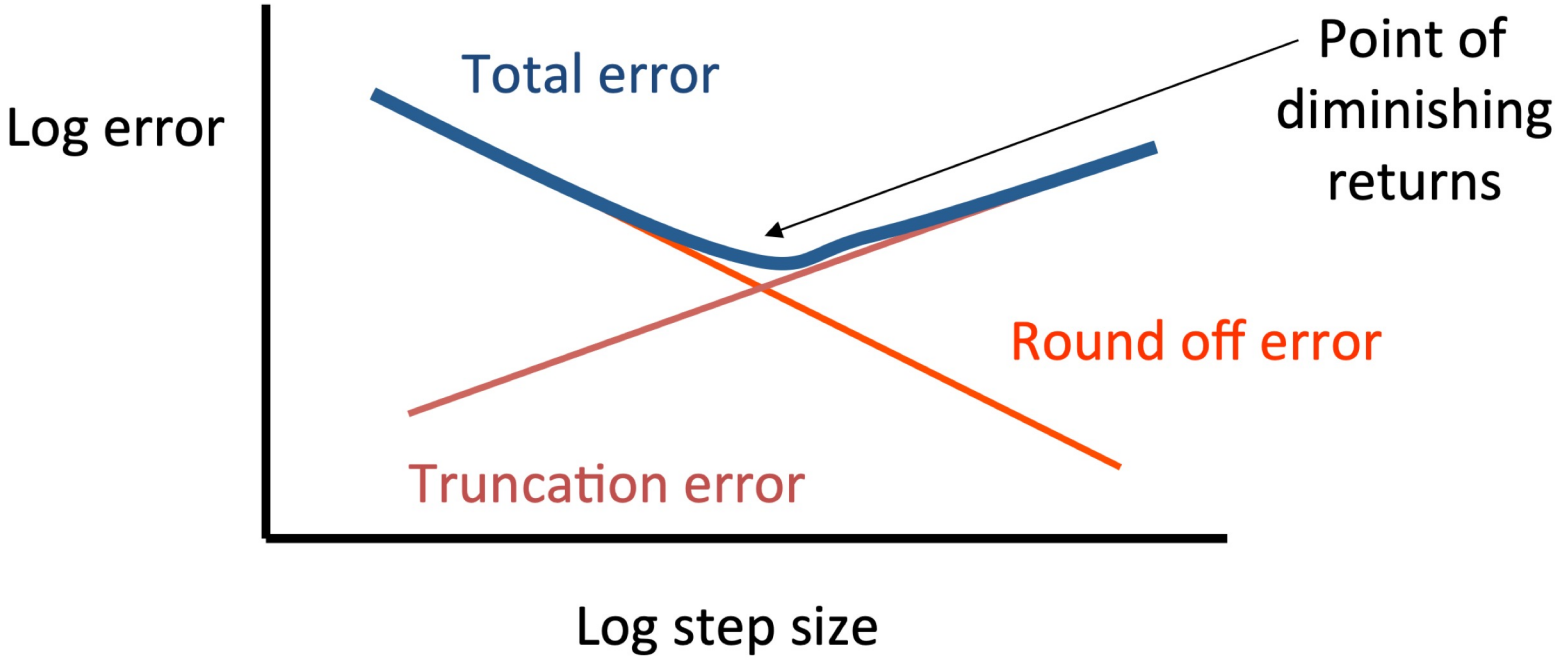
$$\left.\frac{\mathrm{d}}{\mathrm{d}x}f(x)\right|_{x=x_0} = \lim_{h\to 0}\frac{f(x_0+h)-f(x_0)}{h} \approx \frac{f(x_0+h)-f(x_0)}{h}$$

For some finite h → finite difference (relative error can be minimal for some finite! h)

Error trade-off

$$f'(x_0) = \left.\frac{\mathrm{d}}{\mathrm{d}x} f(x)\right|_{x=x_0} \approx \frac{f(x_0+h) - f(x_0)}{h}$$

- Using a smaller step size reduces truncation error.
- However, it increases the round-off error.
- Trade off/diminishing returns occurs: Always think and test!



Stability

An algorithm, equation or, even more general, a problem is referred to as unstable or ill- conditioned if small changes in the input cause a large change in the output.

Example 1:

$$x + y = 2.0,$$
$$x + 1.01y = 2.01$$

Solution: x=1.0, y=1.0

Let us suppose we make a small error in the rhs of the second equation:

$$x + y = 2.0,$$
$$x + 1.01y = 2.02$$

Solution: **x=0.0, y=2.0**

I.e. a 0.05% input error resulted in a 100% wrong result!

Furthermore, if the y-coefficient 1.01 in the original equation would have a 1% error and be 1.0, the equation **would be unsolvable** altogether!

Example 2:
$$\begin{cases} \ddot{y} - 10\dot{y} - 11y = 0\,, \\ y(0) = 1, \qquad \dot{y}(0) = -1 \end{cases}$$

General solution: $y = A\exp(-x) + B\exp(11x)$

With initial conditions: $y = \exp(-x)$

Adding a small input error in initial conditions: $y(0) = 1 + \delta$ and $\dot{y}(0) = -1 + \epsilon$

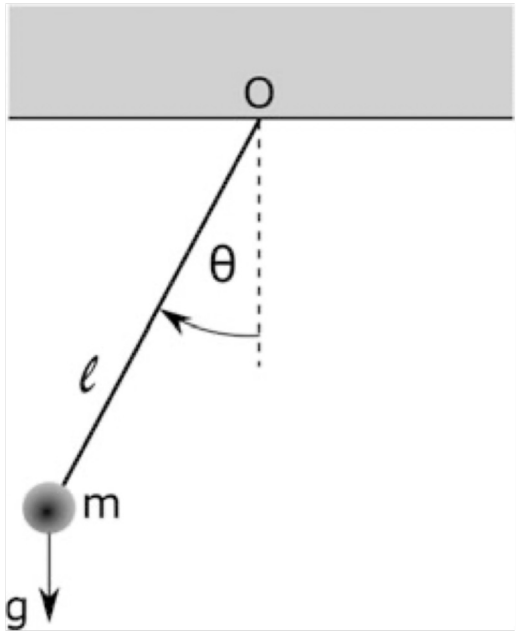
Yields:
$$\bar{y} = \left(1 + \frac{11\delta}{12} - \frac{\epsilon}{12}\right)\exp(-x) + \left(\frac{\delta}{12} + \frac{\epsilon}{12}\right)\exp(11x)$$

i.e. the relative error is
$$\epsilon_r = \left|\frac{y - \bar{y}}{y}\right| = \left(\frac{11\delta}{12} - \frac{\epsilon}{12}\right) + \left(\frac{\delta}{12} + \frac{\epsilon}{12}\right)\exp(12x)$$

→ problem is ill-conditioned: for large values of *x* the second term overrules the first one

See Book for another example for induced instability and the relation to Chaos Theory!

Pendulum Demo with Python



$$\ddot{\theta} + \frac{g}{\ell}\sin(\theta) = 0 \qquad \begin{cases} \theta(0) = \theta_0 \,, \\ \dot{\theta}(0) = 0 \,. \end{cases}$$

1. step, since 2nd order ODE:

$$\begin{aligned} \nu &\equiv \dot{\theta} \\ \dot{\nu} &= -\omega^2 \sin\theta \end{aligned} \qquad \omega = \sqrt{g/\ell}$$

2. step, discretization of time in steps h_t, i.e. $t_n = nh_t$

$$\frac{\theta(t_{n+1}) - \theta(t_n)}{h_t} \approx \nu(t_n)$$

$$\frac{\nu(t_{n+1}) - \nu(t_n)}{h_t} \approx -\omega^2 \sin\theta(t_n)$$

Euler scheme

3. step, implement

$$\begin{aligned} \theta(t_{n+1}) &\approx \theta(t_n) + h_t \nu(t_n) \\ \nu(t_{n+1}) &\approx \nu(t_n) - h_t \omega^2 \sin\theta(t_n) \end{aligned}$$