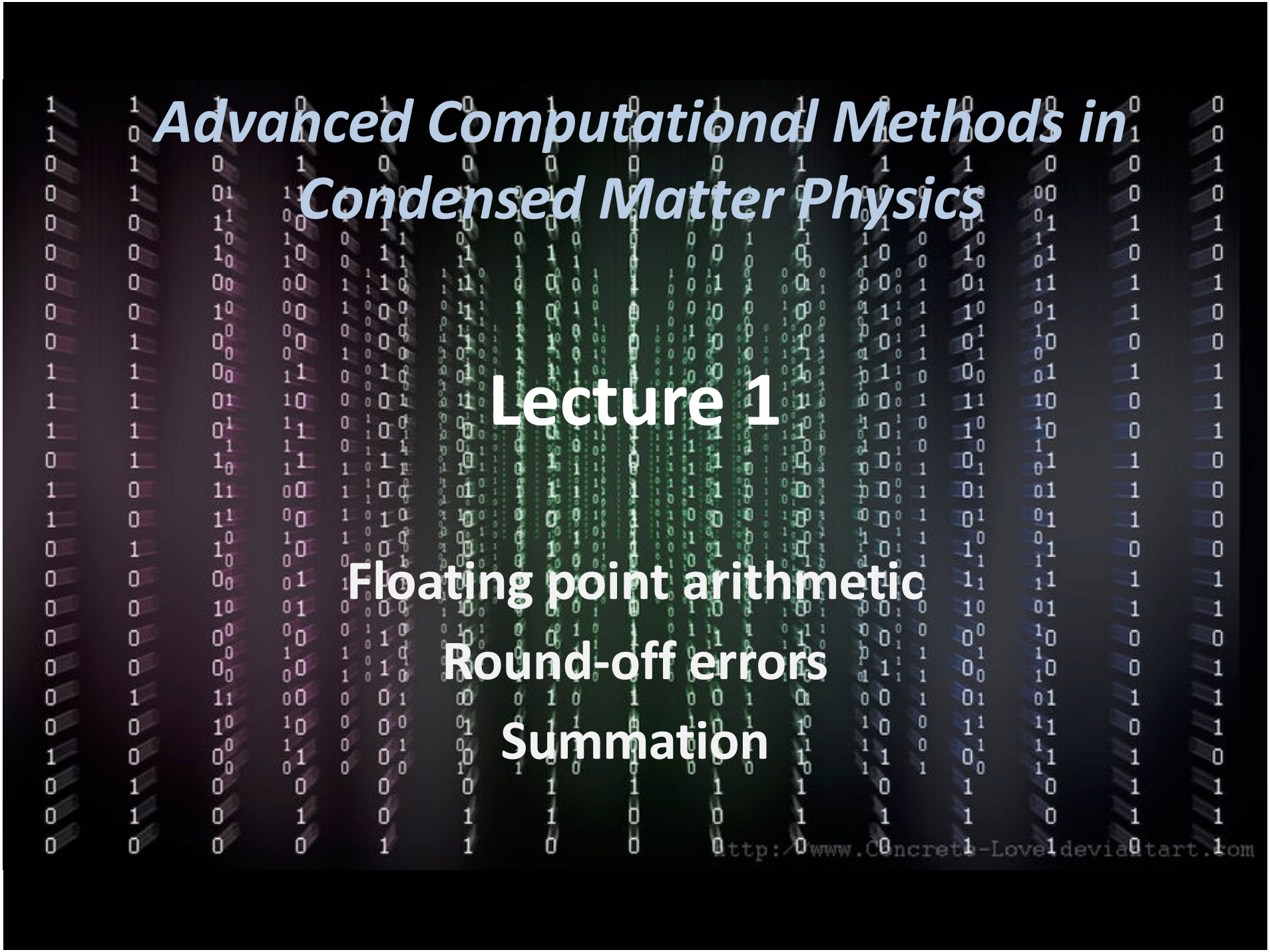
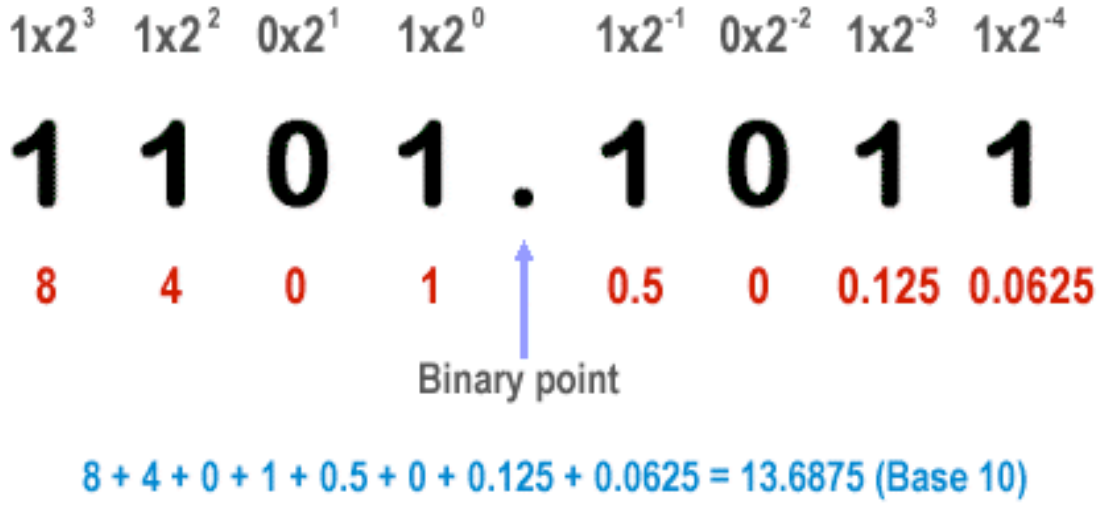

Advanced Computational Methods in Condensed Matter Physics
Lecture 1
Floating point arithmetic
Round-off errors
Summation

Binary representation

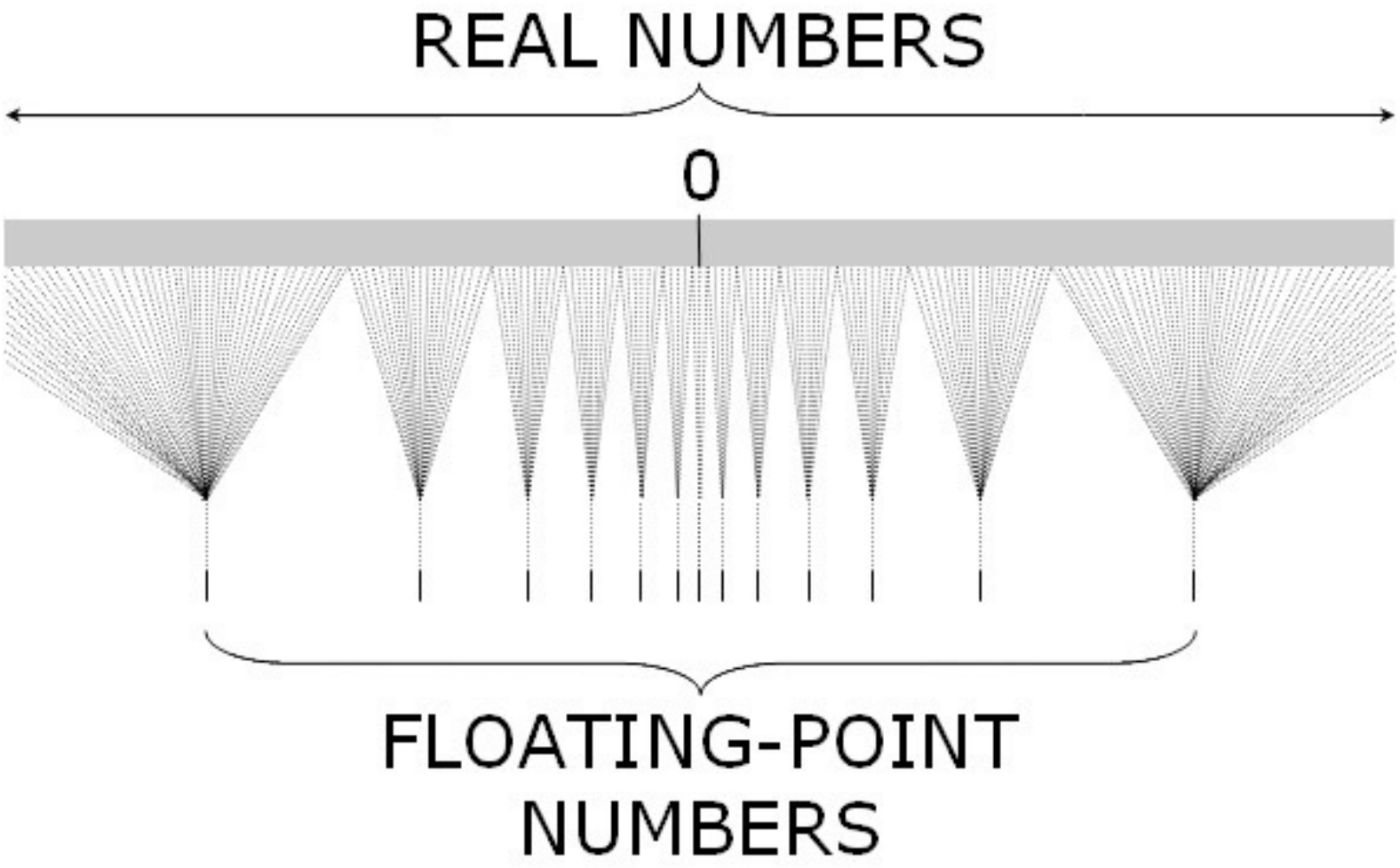
$$x = \pm(\alpha_n 2^n + \alpha_{n-1}2^{n-1} + \cdots + \alpha_0 2^0 + \alpha_{-1}2^{-1} + \alpha_{-2}2^{-2} + \cdots)$$

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}


REAL NUMBERS
0
FLOATING-POINT
NUMBERS

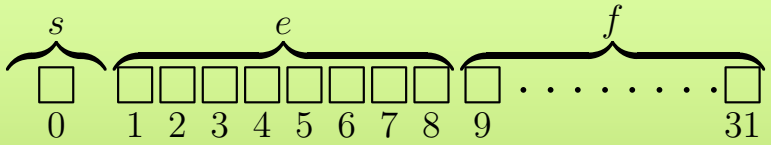
Floating point numbers

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

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

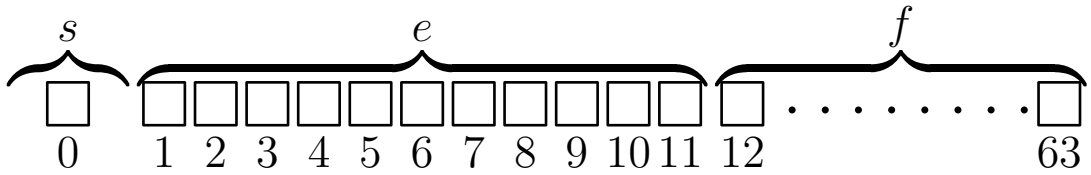
Allow 1.f $\rightarrow$ 0.f (in software)

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.

Sources of errors

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

Modeling 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.

Numerical errors

i. Round-off errors in computer arithmetic (finite numerical precision imposed by the computer).

ii. Discretization 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 $x_{n+1}=(x_n+a/x_n)/2 \rightarrow \mathrm{sqrt}(a)$, $n \rightarrow \infty$).

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

Round-off 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 $b = \mathrm{sqrt}(2/\epsilon_{mach}) \approx 10^8$

...

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

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

This gives is 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 → (x*-*y*)*=0.001

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.

Summation

- We have seen that summation of close numbers, but opposite signs can lead to **loss of significance** or *Subtractive Cancelation*
- Adding numbers with exponents different by more than the length of the mantissa results in round-off errors.

→ This becomes worse when adding many numbers, which happens often in realistic simulations.

Example: (1.0+*10^{-16})*=1.0 (even in double precision), as it should be
Consider now the sum:

$$S_n = \sum_{i=1}^{n} x_i$$

and calculate it as: S_n=((((x_1+x_2)*+x_3)*+x_4+…)*
For x_1=1.0 and all other x_i=10^{-16} gives even for n=10^{16} still S_n=1.0

If the order of summation would be reversed,
we would get (more) accurately S_n≈2.0

floating point addition is *not associative*

How to improve accuracy?

If the behavior of a sequence x_i is know a priory one can change the order of summation:

- For a monotonically decaying positive/negative sequence, one can start with the last, smallest term

e.g. approximation of the geometric series (q<1): $\sum_{n=0}^{\infty} q^n = \frac{1}{1-q}$ **(?, demo)**

- If the sequence elements are saved in memory, sorting can improve accuracy

Otherwise, summing *N* numbers in sequence has a worst-case error that grows proportional to *N* !

The root mean square error grows as for random x_i like sqrt(N) (i.e. round-off errors form a random walk).

Solution: running compensation

In general our sequence is non-monotonic and cannot be sorted.
In that case we need a better summation algorithm, which keeps track of error!

One option: The **Kahan summation algorithm**

Kahan summation algorithm

```
double sum_kahan(double *f,int N)
{
   double sum = f[0];
   double c = 0.0,y,t;
   int i;

   for (i=1;i<N;i++) {
      y = f[i] - c;

      t = sum + y;

      c = (t - sum) - y;
      sum = t;
      }

   return sum;
}
```

sum is initialized with first term
c: the running compensation

The compensation is subtracted from next element
Typically: *sum* is big, *y* small, so low-order digits of *y* are lost.
(t - sum) recovers the high-order part of *y*; subtracting *y* recovers -(low part of *y*) → c

(demo)

Kahan or not?

Unfortunately, Kahan's algorithm requires **four times the arithmetic** and has a latency of four times a simple summation

However, Kahan's algorithm achieves error growth of *O(1)* for summing N numbers, which is only slightly worse than *O(log(N))* error growth, achieved by ***pairwise summation***: one recursively divides the set of numbers into two halves, sums each half, and then adds the two sums.

Pairwise summation has the advantage of requiring the **same number of arithmetic operations as the naive summation** and can be calculated in parallel.

Warning: Be aware that an aggressively optimizing compiler could destroy the main purpose of Kahan summation!

Runtime, Big-O-notation

Informal usage in Computer Science *(and not in the mathematical sense):*

An algorithm can be said to exhibit a growth rate on the order of a mathematical function if beyond a certain input size n, the function f(n) times a positive constant provides an upper bound or limit for the run-time of that algorithm.
In other words, for a given input size n greater than some n0 and a constant c, the running time of that algorithm will never be larger than $c \times f(n)$. This concept is frequently expressed using Big-O-notation. For example, since the run-time of insertion sort grows quadratically as its input size increases, insertion sort can be said to be of order $O(n^2)$.

Big-O-notation is a convenient way to express the worst-case scenario for a given algorithm, although it can also be used to express the average-case — for example, the worst-case scenario for quicksort is $O(n^2)$, but the average-case run-time is $O(n \log n)$.

See also ***Computational complexity theory***

Next lecture(s):

Basic numeric algorithms

- Linear algebra, mostly linear systems
- Numerical integration
- Root finding