Numerical Analysis
Roundoff Errors

Aleksandar Donev
Courant Institute, NYU[1]
donev@courant.nyu.edu

[1] Course MATH-UA.0252/MA-UY_4424, Spring 2021

Spring 2021

Outline

Representing Real Numbers

- Computers represent everything using bit strings, i.e., integers in base-2. Integers can thus be exactly represented. But not real numbers! This leads to **roundoff errors.**
- Assume we have N digits to represent real numbers on a computer that can represent integers using a given number system, say decimal for human purposes.
- **Fixed-point** representation of numbers

$$x = (-1)^s \cdot [a_{N-2}a_{N-3}\dots a_k \,.\, a_{k-1}\dots a_0]$$

has a problem with representing large or small numbers: 1.156 but 0.011.

Floating-Point Numbers

- Instead, it is better to use a **floating-point** representation

$$x = (-1)^s \cdot [0 \,.\, a_1 a_2 \ldots a_t] \cdot \beta^e = (-1)^s \cdot m \cdot \beta^{e-t},$$

  akin to the common scientific number representation: $0.1156 \cdot 10^1$ and $0.1156 \cdot 10^{-1}$.
- A floating-point number in base β is represented using one **sign bit** $s = 0$ or 1, a t-digit integer **mantissa** $0 \leq m = [a_1 a_2 \ldots a_t] \leq \beta^t - 1$, and an integer **exponent** $L \leq e \leq U$.
- Computers today use **binary numbers** (bits), $\beta = 2$.
- Also, for hardware reasons, numbers come in 32-bit and 64-bit packets (words), sometimes 128 bits also (powers of two).

The IEEE Standard for Floating-Point Arithmetic (IEEE 754)

The IEEE 754 (also IEC559) standard documents:

- **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, **not a number** like $\sqrt{-1}$ in real numbers).

IEEE Standard Representations

- **Normalized single precision** floating-point numbers (`single` in MATLAB, `float` in C/C++) use 32 bits = **4 bytes** to store sign + power + mantissa:

$$N_s + N_p + N_f = 1 + 8 + 23 = 32 \text{ bits}$$

- For example, $x = 2752 = 0.2752 \cdot 10^4$. Converting 2752 to the binary number system

$$x = 2^{11} + 2^9 + 2^7 + 2^6 = (101011000000)_2 = 2^{11} \cdot (1.01011)_2$$

is represented internally as the 32-bit string $[0 \quad | \quad 100,0101,0 \quad | \quad 010,1100,0000,0000,0000,0000]$ (details not important).

- **Double precision numbers** (default in MATLAB, `double` in C/C++) follow the same principle, but use 64 bits=**8 bytes** to give higher precision and range

$$N_s + N_p + N_f = 1 + 11 + 52 = 64 \text{ bits}$$

Outline

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

$$\frac{|\hat{x} - x|}{|x|} \leq u = 2^{-(N_f+1)} = \begin{cases} 2^{-24} \approx 6.0 \cdot 10^{-8} & \text{for single precision} \\ 2^{-53} \approx 1.1 \cdot 10^{-16} & \text{for double precision} \end{cases}$$

  The rule of thumb is that single precision gives 7-8 digits of precision and double 16 digits.
- There is a smallest and largest possible number due to the limited range for the exponent.

Important Floating-Point Constants

Important: MATLAB uses double precision by default (for good reasons!).
Use `x=single(value)` to get a single-precision number.

	MATLAB code	Single precision	Double precision
ϵ	eps, eps('single')	$2^{-23} \approx 1.2 \cdot 10^{-7}$	$2^{-52} \approx 2.2 \cdot 10^{-16}$
x_{max}	realmax	$2^{128} \approx 3.4 \cdot 10^{38}$	$2^{1024} \approx 1.8 \cdot 10^{308}$
x_{min}	realmin	$2^{-126} \approx 1.2 \cdot 10^{-38}$	$2^{-1022} \approx 2.2 \cdot 10^{-308}$

IEEE Arithmetic

- The IEEE standard specifies that the basic arithmetic operations (addition,subtraction,multiplication,division) ought to be performed using rounding to the nearest number of the *exact* result:

$$\hat{x} \circledcirc \hat{y} = \widehat{x \circ y}$$

- This guarantees that such operations are performed to within machine precision in relative error.
- Floating-point addition and multiplication are **not associative** but they are commutative.
- Operations with infinities follow sensible mathematical rules (e.g., $finite/inf = 0$).
- Any operation involving **not-a-number** or *NaN*'s gives a *NaN*.

Floating-Point in Practice

- Most scientific software **uses double precision** to avoid range and accuracy issues with single precision (better be safe then sorry). Single precision may offer speed/memory/vectorization advantages however (e.g. GPU computing).
- **Do not compare floating point numbers** (especially for loop termination), or more generally, do not rely on logic from pure mathematics.
- **Using parenthesis helps control order of operations**, e.g. $(x + y) - z$ instead of $x + y - z$.
- Library functions such as sin and ln will typically be computed almost to full machine accuracy, but do not rely on that for special/complex functions.

Floating-Point Exceptions

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

  Divide-by-zero if the result is $\pm\infty$, e.g., $1/0$.

  Invalid if the result is a *NaN*, e.g., taking $\sqrt{-1}$ (but not MATLAB uses complex numbers!).

  Overflow if the result is too large to be represented, e.g., adding two numbers, each on the order of *realmax*.

  Underflow if the result is too small to be represented, e.g., dividing a number close to *realmin* by a large number.

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

  Mathematically equivalent expressions (forms) are not necessarily computationally-equivalent!

Outline

Propagation of Errors

- Assume that we are calculating something with numbers that are not exact, e.g., a rounded floating-point number $\hat{x}$ versus the exact real number x.
- For IEEE floating-point numbers, recall that we are guaranteed a **relative error due to roundoff**

$$\frac{|\hat{x}-x|}{|x|} \leq u = \begin{cases} 6.0 \cdot 10^{-8} & \text{for single precision} \\ 1.1 \cdot 10^{-16} & \text{for double precision} \end{cases}$$

- *How does the relative error change (propagate) during numerical calculations?*
- In general, the **absolute error** $\delta x = \hat{x} - x$ may have contributions from different **sources of error** (roundoff, mathematical approximations of limits, truncating infinite iterations or sums, etc.).

Propagation of Errors: Multiplication/Division

- For **multiplication and division**, the bounds for the **relative** error in the operands are added to give an estimate of the relative error in the result:

$$\epsilon_{xy} = \left| \frac{(x + \delta x)(y + \delta y) - xy}{xy} \right| = \left| \frac{\delta x}{x} + \frac{\delta y}{y} + \frac{\delta x}{x} \frac{\delta y}{y} \right| \lessapprox \epsilon_x + \epsilon_y .$$

- This means that multiplication and division are **safe**, since operating on accurate input gives an output with similar accuracy.

Addition/Subtraction

- For **addition and subtraction**, however, the bounds on the **absolute** errors add to give an estimate of the absolute error in the result:

$$|\delta(x+y)| = |(x+\delta x)+(y+\delta y)-(x+y)| = |\delta x+\delta y| < |\delta x|+|\delta y|.$$

- This is much more **dangerous** since the relative error is not controlled, leading to so-called **catastrophic cancellation.**
- Adding or subtracting two numbers of **widely-differing magnitude** leads to loss of accuracy due to roundoff error.
- If you do arithmetic with only 5 digits of accuracy, and you calculate

$$1.0010 + 0.00013000 = 1.0011,$$

only registers one of the digits of the small number!

Outline

Loss of Digits

- This type of roundoff error can accumulate when adding many terms, such as calculating infinite sums.
- As an example, consider computing the **harmonic sum** numerically:

$$H(N) = \sum_{i=1}^{N} \frac{1}{i} = \Psi(N+1) + \gamma,$$

where the digamma special function Ψ is *psi* in MATLAB.

- For large N, $\Psi(N+1) \approx \ln(N)$.
- We can do the sum in **forward** or in **reverse order** (in single or double precision).

Cancellation Error

```
% Calculating the harmonic sum for a given integer N:
function nhsum=harmonic(N)
    nhsum=0.0;
    for i=1:N % Or, for i=N:-1:1
        nhsum=nhsum+1.0/i;
    end
end
```

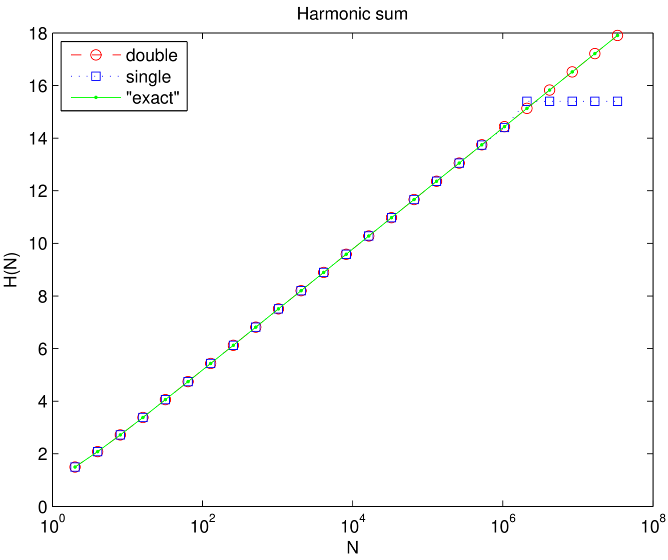
contd.

```
clear all; format compact; format long e

npts=25;
Ns=zeros(1,npts);
hsum=zeros(1,npts);
relerr=zeros(1,npts);
nhsum=zeros(1,npts);
Euler_gamma = psi(1) % Actual value of Euler constant
for i=1:npts
    Ns(i)=2^i;
    nhsum(i)=harmonic(Ns(i));
    hsum(i)=(psi(Ns(i)+1)-psi(1)); % Theoretical result
    relerr(i)=abs(nhsum(i)-hsum(i))/hsum(i);
    gamma = nhsum(i)-ln(Ns(i))
end
```

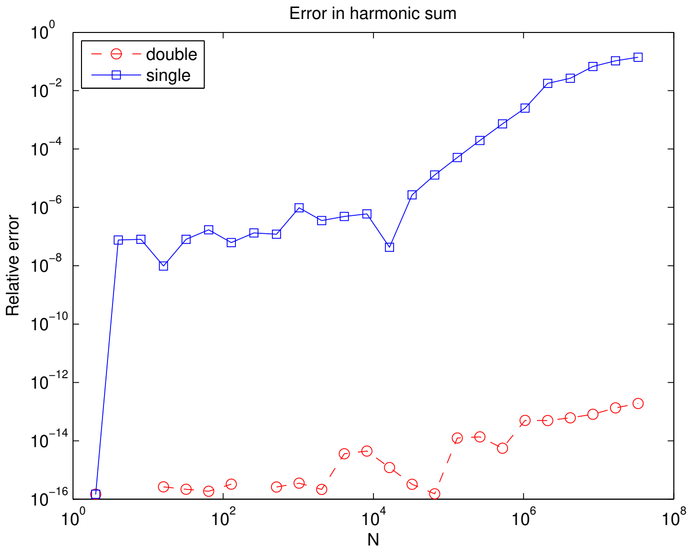
contd.

```
figure(1);
loglog(Ns,relerr,'ro--');
title('Error in harmonic sum');
xlabel('N'); ylabel('Relative error');

figure(2);
semilogx(Ns,nhsum,'ro--', Ns,hsum,'g.-');
title('Harmonic sum');
xlabel('N'); ylabel('H(N)');
legend('double','"exact"', 'Location','NorthWest');
```

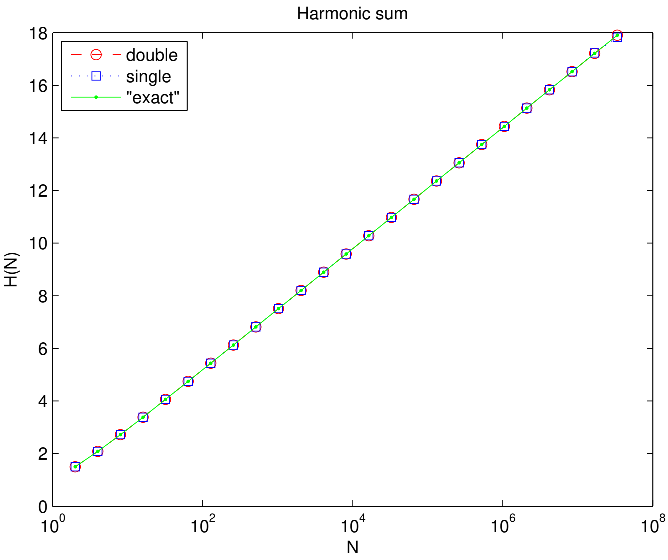
Results: Forward summation



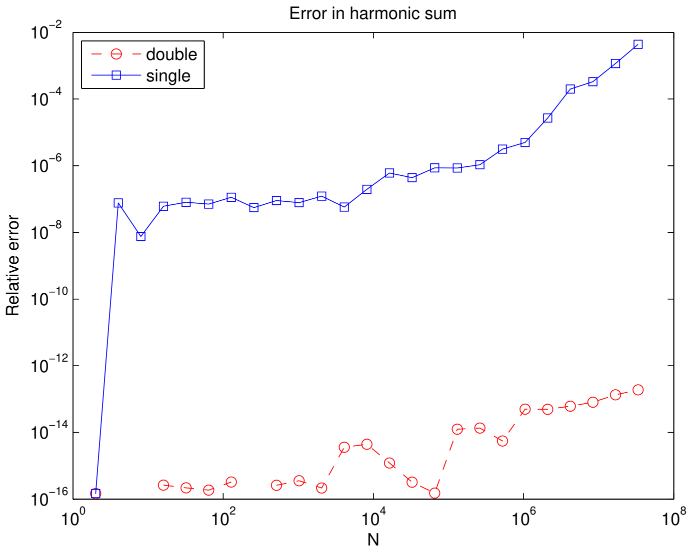
Forward summation error



Results: Backward summation



Backward summation error



Explanation of results

- The numerical forward sum will stop increasing when

$$\frac{1}{N} \approx u \cdot \ln N$$

- Solving this *nonlinear equation* (try it!) for single precision gives $N \approx 6.245 \cdot 10^5 \sim 10^6$, which is about what we see.
- For double precision, we get $N \approx 1.4 \cdot 10^{14}$ (let's check).
- **Backward** summation harder to explain but clearly **much better**, though **not perfect**.

Outline

Numerical Cancellation

- If x and y are close to each other, $x - y$ can have reduced accuracy due to **catastrophic cancellation**.
  For example, using 5 significant digits we get
  $$1.1234 - 1.1223 = 0.0011,$$
  which only has 2 significant digits!
- If gradual underflow is not supported $x - y$ can be zero even if x and y are not exactly equal.
- Consider, for example, computing the smaller root of the quadratic equation
  $$x^2 - 2x + c = 0$$
  for $|c| \ll 1$, and focus on propagation/accumulation of **roundoff error**.

Cancellation example

- Let's first try the obvious formula

$$x = 1 - \sqrt{1 - c}.$$

- Note that if $|c| \leq u$ the subtraction $1 - c$ will give 1 and thus $x = 0$. How about

$$u \ll |c| \ll 1?$$

- The calculation of $1 - c \approx 1$ in double-precision arithmetic will ignore/loose all digits in c after the 16th.
- For example, if $c = 10^{-9}$, we will only keep about $16 - 9 = 7$ digits, loosing $16 - 7 = 9$ digits of accuracy!

Avoiding Cancellation

- For small c the solution is

$$x = 1 - \sqrt{1-c} \approx \frac{c}{2},$$

  but we already lost all digits in c after the 16th, so we have made an *absolute error* of order u.
- Just using the Taylor series result, $x \approx \frac{c}{2}$, already provides a good approximation for small c. Here we can do better!
- Rewriting in **mathematically-equivalent but numerically-preferred form** is the first try, e.g., instead of

$$1 - \sqrt{1-c} \text{ use } \frac{c}{1+\sqrt{1-c}},$$

  which does not suffer any problem as c becomes smaller, even smaller than roundoff!

Example/practice (maybe worksheet)

There are many methods to compute many digits of π, and lots of them suffer from numerical accuracy problems. Here is one of them due to Archimedes: Start with $t_0 = 1/\sqrt{3}$ and then iterate

$$t_{i+1} = \frac{\sqrt{1 + t_i^2} - 1}{t_i} \tag{1}$$

and for large i you can get a good approximation $6 \cdot 2^i \cdot t_i \to \pi$.

1. Do this calculation with Matlab, and report how many digits of accuracy you get and after how many iterations (Note: MATLAB has a built-in constant *pi*), accompanied with some plots of the convergence. Can you explain what you see?
2. Find a way to rewrite the iteration (1) so that you avoid roundoff errors. Repeat the calculation and report how many digits of π you get then.

Another example in worksheet 1 (numerical differentiation).