Scientific Computing
Numerical Computing

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

[1]Course MATH-GA.2043 or CSCI-GA.2112, Fall 2015

September 3rd and 10th, 2015

Outline

Course Essentials

- Course webpage: `http://cims.nyu.edu/~donev/Teaching/SciComp-Fall2015/`
- Registered students: NYU Courses for announcements, submitting homeworks, grades, and sample solutions. **Make sure you have access**.
- Office hours: 4-5 pm Tuesdays, 4-5pm on Thursdays, or by appointment. Grader's office hours **TBD**.
- Main textbooks (optional but at least one recommended): see course homepage.
- Secondary textbook: Draft book by Jonathan Goodman, linked on webpage.
- Other optional readings linked on course page.
- Computing is an essential part: **MATLAB** forms a common platform and is chosen because of its linear algebra and plotting strengths (**Octave** is a possible but incomplete alternative).

To-do

- Assignment 0 is a **Questionnaire** with basic statistics about you: submit via email as **plain text** or **PDF**.
- If you have not done it already: **Review Linear Algebra** using Jonathan Goodman's notes or other sources.
- Get access to **MATLAB** asap (e.g., Courant Labs) and start playing.
- There will be regular homework assignments (60% of grade), mostly computational. Points from all assignments will be added together.
- Submit the solutions as a PDF (give LaTex/lyx a try!), via NYU Courses, or handwritten if appropriate, details provided next class. **First assignment posted next week and due in two weeks.**
- There will also be a **take-home final** (40%) due **Dec 17th**, which will be similar in spirit to the homeworks but more involved and will be **graded by me**.

Academic Integrity Policy

- If you use any external source, even Wikipedia, make sure you acknowledge it by **referencing all help**.
- It is encouraged to **discuss** with other students the mathematical aspects, algorithmic strategy, code design, techniques for debugging, and compare results.
- Copying of any portion of someone else's solution or allowing others to copy your solution is considered **cheating**.
- **Code sharing is not allowed**. You must type (or create from things you've typed using an editor, script, etc.) every character of code you use.
- Submitting an **individual and independent final** is crucial and **no collaboration** will be allowed for the final.
- Common bad justifications for copying:
  - We are too busy and the homework is very hard, so we cannot do it on our own.
  - We do not copy each other but rather "work together."
  - I just emailed Joe Doe my solution as a "reference."

Grading Standards

- **Points** will be **added** over all assignments (60%) and the take-home final (40%).
- **No makeup** points (solutions may be posted on NYU Courses).
- The actual grades will be rounded upward (e.g., for those that are close to a boundary), but not downward:
  - 92.5-max = A
  - 87.5-92.5 = A-
  - 80.0-87.5 = B+
  - 72.5-80.0 = B
  - 65.0-72.5 = B-
  - 57.5-65.0 = C+
  - 50.0-57.5 = C
  - 42.5-50.0 = C-
  - min-42.5 = F

Peculiarities of MATLAB

- MATLAB is an **interpreted language**, meaning that commands are interpreted and executed as encountered. MATLAB caches some stuff though...
- Many of MATLAB's **intrinsic routines** are however compiled and optimized and often based on well-known libraries (BLAS, LAPACK, FFTW, etc.).
- Variables in scripts/worspace are global and persist throughout an interactive session (use *whos* for info and *clear* to clear workspace).
- Every variable in MATLAB is, unless specifically arranged otherwise, a matrix, **double precision float** if numerical.
- Vectors (column or row) are also matrices for which one of the dimensions is 1.
- **Complex arithmetic** and complex matrices are used where necessary.

Matrices

```
>> format compact; format long
>> x=-1; % A scalar that is really a 1x1 matrix
>> whos('x')
  Name        Size                Bytes  Class     Attributes
  x           1x1                     8  double

>> y=sqrt(x) % Requires complex arithmetic
y =                   0 + 1.000000000000000i
>> whos('y')
  Name        Size                Bytes  Class     Attributes
  y           1x1                    16  double    complex

>> size(x)
ans =     1     1
>> x(1)
ans =    -1
>> x(1,1)
ans =    -1
>> x(3)=1;
>> x
x =    -1     0     1
```

Vectorization / Optimization

- MATLAB uses **dynamic memory management** (including garbage collection), and matrices are re-allocated as needed when new elements are added.
- It is however much better to **pre-allocate space** ahead of time using, for example, *zeros*.
- The **colon notation** is very important in accessing array sections, and x is different from $x(:)$.
- **Avoid for loops** unless necessary: Use array notation and intrinsic functions instead.
- To see how much CPU (computing) time a section of code took, use *tic* and *toc* (but beware of timing small sections of code).
- MATLAB has built-in **profiling tools** (*help profile*).

Pre-allocation (fibb.m)

```
format compact; format long
clear; % Clear all variables from memory

N=100000; % The number of iterations

% Try commenting this line out:
f=zeros(1,N); % Pre-allocate f

tic;
f(1)=1;
for i=2:N
    f(i)=f(i-1)+i;
end
elapsed=toc;

fprintf('The result is f(%d)=%g, computed in %g s\n', ...
        N, f(N), elapsed);
```

Vectorization (vect.m)

```
function vect(vectorize)
    N=1000000; % The number of elements
    x=linspace(0,1,N); % Grid of N equi-spaced points

    tic;
    if(vectorize) % Vectorized
        x=sqrt(x);
    else % Non-vectorized
        for i=1:N
            x(i)=sqrt(x(i));
        end
    end
    elapsed=toc;

    fprintf('CPU time for N=%d is %g s\n', N, elapsed);
end
```

MATLAB examples

```
>> fibb % Without pre-allocating
The result is f(100000)=5.00005e+09, computed in 6.53603 s

>> fibb % Pre-allocating
The result is f(100000)=5.00005e+09, computed in 0.000998 s

>> vect(0) % Non-vectorized
CPU time for N=1000000 is 0.074986 s

>> vect(1) % Vectorized -- don't trust the actual number
CPU time for N=1000000 is 0.002058 s
```

Vectorization / Optimization

- Recall that everything in MATLAB is a double-precision matrix, called **array**.
- Row vectors are just matrices with first dimension 1. Column vectors have row dimension 1. Scalars are 1×1 matrices.
- The syntax x' can be used to construct the **conjugate transpose** of a matrix.
- The **colon notation** can be used to select a subset of the elements of an array, called an **array section**.
- The default arithmetic operators, +, -, *, / and ^ are **matrix addition/subtraction/multiplication**, linear solver and matrix power.
- If you prepend a **dot before an operator** you get an **element-wise operator** which works for arrays of the same shape.

Pre-allocation (fibb.m)

```
>> x=[1 2 3; 4 5 6] % Construct a matrix
x =
     1     2     3
     4     5     6

>> size(x) % Shape of the matrix x
ans =
     2     3

>> y=x(:) % All elements of y
y =
     1     4     2     5     3     6

>> size(y)
ans =
     6     1

>> x(1,1:3)
ans =
     1     2     3

>> x(1:2:6)
ans =
     1     2     3
```

Pre-allocation (fibb.m)

```
>> sum(x)
ans =
     5     7     9

>> sum(x(:))
ans =
    21

>> z=1i; % Imaginary unit

>> y=x+z
y =
   1.0000 + 1.0000i   2.0000 + 1.0000i   3.0000 + 1.0000i
   4.0000 + 1.0000i   5.0000 + 1.0000i   6.0000 + 1.0000i

>> y'
ans =
   1.0000 - 1.0000i   4.0000 - 1.0000i
   2.0000 - 1.0000i   5.0000 - 1.0000i
   3.0000 - 1.0000i   6.0000 - 1.0000i
```

Pre-allocation (fibb.m)

```
>> x*y
??? Error using ==> mtimes
Inner matrix dimensions must agree.

>> x.*y
ans =
   1.0000 + 1.0000i   4.0000 + 2.0000i   9.0000 + 3.0000i
  16.0000 + 4.0000i  25.0000 + 5.0000i  36.0000 + 6.0000i

>> x*y'
ans =
  14.0000 - 6.0000i  32.0000 - 6.0000i
  32.0000 -15.0000i  77.0000 -15.0000i

>> x'*y
ans =
  17.0000 + 5.0000i  22.0000 + 5.0000i  27.0000 + 5.0000i
  22.0000 + 7.0000i  29.0000 + 7.0000i  36.0000 + 7.0000i
  27.0000 + 9.0000i  36.0000 + 9.0000i  45.0000 + 9.0000i
```

Coding Guidelines

- Learn to reference the **MATLAB help**: Including reading the examples and "fine print" near the end, not just the simple usage. **Know what is under the hood!**
- **Indendation, comments, and variable naming** make a big difference! Code should be readable by others.
- Spending a few extra moments on the code will pay off when using it.
- Spend some time learning how to **plot in MATLAB**, and in particular, how to plot with different symbols, lines and colors using *plot, loglog, semilogx, semilogy.*
- Learn how to **annotate plots**: *xlim, ylim, axis, xlabel, title, legend.* The intrinsics *num2str* or *sprintf* can be used to create strings with embedded parameters.
- Finer controls over fonts, line widths, etc., are provided by the intrinsic function *set*...including using the LaTex interpreter to typeset mathematical notation in figures.

Conditioning of a Computational Problem

- A rather generic computational problem is to find a **solution** x that satisfies some condition $F(x, d) = 0$ for given **data** d.
- Scientific computing is concerned with devising an **algorithm** for computing x given d, **implementing** the algorithm in a code, and **computing** the actual answer for some relevant data.
- For example, consider the simple problem of computing $x = \sqrt{d}$ if you were not given an intrinsic function *sqrt*:

$$F(x, d) = x - \sqrt{d}.$$

- **Well-posed** problem: Unique solution that depends continuously on the data.
  If we perturb d by a little, the solution x gets perturbed by a small amount (can be made precise).
- Otherwise it is an intrinsically **ill-posed** problem and no numerical method can help with that.

Absolute and Relative Errors

- A numerical algorithm always computes an **approximate solution** $\hat{x}$ given some **approximate data** d instead of the (unknown) exact solution x.
- We define **absolute error** δx and **relative error** $\epsilon = \delta x / x$:

$$\hat{x} = x + \delta x, \quad \hat{x} = (1 + \epsilon)x$$

- The relative **conditioning number**

$$K = \sup_{\delta d \neq 0} \frac{\|\delta x\| \, / \, \|x\|}{\|\delta d\| \, / \, \|d\|}$$

is an important *intrinsic* property of a computational problem.
Here sup stands for supremum, (almost) the same as **maximum** over all perturbations of the data.

Conditioning Number

- If $K \sim 1$ the problem is **well-conditioned**. If the relative error in the data is small, we can compute an answer to a similar relative accuracy.
- An **ill-conditioned** problem is one that has a large condition number, $K \gg 1$.
- Note that ill-conditioining depends on the desired accuracy:
  K is "large" if a given **target solution accuracy** of the solution cannot be achieved for a given **input accuracy** of the data.
- We may still sometimes solve a problem that is ill-conditioned, if it is known that the possibly large error δx does not matter.
  But we ought to always be aware of it!

Conditioning Example

- Consider solving the equation, for some given d:

$$x^3 - 3x^2 + 3x - 1 = (x-1)^3 = d.$$

- The solution (assume real numbers) is

$$x = d^{1/3} + 1.$$

- If we now perturb $d \leftarrow d + \delta d$,

$$x + \delta x = (d + \delta d)^{1/3} + 1 \quad \Rightarrow \quad \delta x = (d + \delta d)^{1/3} - d^{1/3}$$

- If we know $d = 0$ to within $|\delta d| < 10^{-6}$, then we only know $x \approx 1$ to within an absolute error

$$|\delta x| < \left(10^{-6}\right)^{1/3} = 10^{-2}$$

with the same relative error, which is much worse than the error in d (ill-conditioned?).

- This may not be a problem if all we care about is that $(x-1)^3 \approx 0$, and do not really care about x itself!

Consistency, Stability and Convergence

Instead of solving $F(x, d) = 0$ directly, many numerical methods generate a sequence of solutions to

$$F_n(x_n, d_n) = 0, \text{ where } n = 0, 1, 2, \ldots$$

where for each n it is easier to obtain x_n given d.

- A numerical method is **consistent** if the approximation error vanishes as $F_n \to F$ (typically $n \to \infty$).
- A numerical method is **stable** if propagated errors decrease as the computation progresses (n increases).
- A numerical method is **convergent** if the numerical error can be made arbitrarily small by increasing the computational effort (larger n).
- Rather generally

$$\text{consistency} + \text{stability} \to \text{convergence}$$

Example: Consistency

- [*From Dahlquist & Bjorck*] Consider solving

$$F(x) = f(x) - x = 0,$$

where $f(x)$ is some non-linear function so that an exact solution is not known.

- A simple problem that is easy to solve is:

$$f(x_n) - x_{n+1} = 0 \quad \Rightarrow \quad x_{n+1} = f(x_n).$$

- This corresponds to choosing the sequence of approximations:

$$F_n\left(x_n,\, d_n \equiv x_{n-1}\right) = f(x_{n-1}) - x_n$$

- This method is consistent because if $d_n = x$ is the solution, $f(x) = x$, then

$$F_n\left(x_n,\, x\right) = x - x_n \quad \Rightarrow \quad x_n = x,$$

which means that the true solution x is a **fixed-point of the iteration**.

Example: Convergence

- For example, consider the calculation of square roots, $x = \sqrt{c}$.
- Warm up MATLAB programming: Try these calculations numerically.
- First, rewrite this as an equation:

$$f(x) = c/x = x$$

- The corresponding fixed-point method

$$x_{n+1} = f(x_n) = c/x_n$$

oscillates between x_0 and c/x_0 since $c/(c/x_0) = x_0$.
The error does not decrease and the method does not converge.

- But another choice yields an algorithm that converges (fast) for any initial guess x_0:

$$f(x) = \frac{1}{2}\left(\frac{c}{x} + x\right)$$

Example: Convergence

- Now consider the Babylonian method for square roots

$$x_{n+1} = \frac{1}{2}\left(\frac{c}{x_n} + x_n\right), \text{ based on choosing } f(x) = \frac{1}{2}\left(\frac{c}{x} + x\right).$$

- The relative error at iteration n is

$$\epsilon_n = \frac{x_n - \sqrt{c}}{\sqrt{c}} = \frac{x_n}{\sqrt{c}} - 1 \quad \Rightarrow \quad x_n = (1+\epsilon_n)\sqrt{c}.$$

- It can now be proven that the error will decrease at the next step, at least in half if $\epsilon_n > 1$, and quadratically if $\epsilon_n < 1$.

$$\epsilon_{n+1} = \frac{x_{n+1}}{\sqrt{c}} - 1 = \frac{1}{\sqrt{c}} \cdot \frac{1}{2}\left(\frac{c}{x_n} + x_n\right) - 1 = \frac{\epsilon_n^2}{2(1+\epsilon_n)}.$$

$$\text{For } n > 1 \text{ we have } \epsilon_n \geq 0 \quad \Rightarrow \quad \epsilon_{n+1} \leq \min\left\{\frac{\epsilon_n^2}{2}, \ \frac{\epsilon_n^2}{2\epsilon_n} = \frac{\epsilon_n}{2}\right\}$$

Beyond Convergence

- An algorithm will produce the correct answer if it is convergent, but...
- Not all convergent methods are equal. We can differentiate them further based on:

Accuracy How much computational work do you need to expand to get an answer to a desired relative error?
The Babylonian method is very good since the error rapidly decays and one can get relative error $\epsilon < 10^{-100}$ in no more than 8 iterations if a smart estimate is used for x_0 [see Wikipedia article].

Robustness Does the algorithm work (equally) well for all (reasonable) input data d? The Babylonian method converges for every positive c and x_0, and is thus robust.

Efficiency How fast does the **implementation** produce the answer? This depends on the algorithm, on the computer, the programming language, the programmer, etc. (more next class)

Computational Error

Numerical algorithms try to control or minimize, rather then eliminate, the various **computational errors**:

Approximation error due to replacing the computational problem with an easier-to-solve approximation. Also called **discretization error** for ODEs/PDEs.

Truncation error due to replacing limits and infinite sequences and sums by a finite number of steps. Closely related to approximation error.

Roundoff error due to finite representation of real numbers and arithmetic on the computer, $x \neq \hat{x}$.

Propagated error due to errors in the data from user input or previous calculations in iterative methods.

Statistical error in stochastic calculations such as Monte Carlo calculations.

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 various reasons, numbers come in 32-bit and 64-bit packets (words), sometimes 128 bits also.
  Note that this is different from whether the machine is 32-bit or 64-bit, which refers to memory address widths.

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).

IEEE Standard Representations

- **Normalized single precision IEEE** floating-point numbers (`single` in MATLAB, `float` in C/C++, `REAL` in Fortran) have the standardized *storage format* (sign+power+fraction)

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

and are interpreted as

$$x = (-1)^s \cdot 2^{p-127} \cdot (1.f)_2,$$

where the **sign** $s = 1$ for negative numbers, the power $1 \leq p \leq 254$ determines the **exponent**, and f is the **fractional part of the mantissa**.

IEEE representation example

[From J. Goodman's notes] Take the number $x = 2752 = 0.2752 \cdot 10^4$. Converting 2752 to the binary number system

$$x = 2^{11} + 2^9 + 2^7 + 2^6 = (101011000000)_2 = 2^{11} \cdot (1.01011)_2$$
$$= (-1)^0 2^{138-127} \cdot (1.01011)_2 = (-1)^0 2^{(10001010)_2 - 127} \cdot (1.01011)_2$$

On the computer:

$$x = [s \quad | \quad p \quad | \quad f]$$
$$= [0 \quad | \quad 100,0101,0 \quad | \quad 010,1100,0000,0000,0000,0000]$$
$$= (452c0000)_{16}$$

```
format hex;
>> a=single(2.752E3)
a =
    452c0000
```

IEEE formats contd.

- **Double precision IEEE numbers** (default in MATLAB, `double` in C/C++, `REAL(KIND(0.0d0))` in Fortran) follow the same principle, but use 64 bits to give higher precision and range

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

$$x = (-1)^s \cdot 2^{p-1023} \cdot (1.f)_2.$$

- Higher (extended) precision formats are not really standardized or widely implemented/used (e.g., quad=$1 + 15 + 112 = 128$ bits, `double double`, `long double`).
- There is also software-emulated **variable precision arithmetic** (e.g., Maple, MATLAB's symbolic toolbox, libraries).

IEEE non-normalized numbers

- The extremal exponent values have special meaning:

value	power p	fraction f
± 0	0	0
denormal (subnormal)	0	> 0
$\pm\infty(inf)$	255	$= 0$
Not a number (NaN)	255	> 0

- A denormal/subnormal number is one which is smaller than the smallest normalized number (i.e., the mantissa does not start with 1). For example, for single-precision IEEE

$$\tilde{x} = (-1)^s \cdot 2^{-126} \cdot (0.f)_2.$$

- Denormals are *not always supported* and may incur performance penalties (specialized hardware instructions).

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 (note denormals).

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	$2^{-126} \approx 1.2 \cdot 10^{-38}$	$2^{-1022} \approx 2.2 \cdot 10^{-308}$
$\tilde{x}_{max}$	realmin*(1-eps)	$2^{-126} \approx 1.2 \cdot 10^{-38}$	$2^{1024} \approx 1.8 \cdot 10^{308}$
$\tilde{x}_{min}$	realmin*eps	$2^{-149} \approx 1.4 \cdot 10^{-45}$	$2^{-1074} \approx 4.9 \cdot 10^{-324}$

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 (requires a guard digit for subtraction).
- 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 *NaN*'s gives a *NaN* (signaling or not), and comparisons are tricky (see homework).

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.
- Optimization, especially in compiled languages, can rearrange terms or perform operations using **unpredictable** alternate forms (e.g., wider internal registers).
  Using parenthesis helps , e.g. $(x + y) - z$ instead of $x + y - z$, but does not eliminate the problem.
- 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.
Note that if denormals are supported one gets **gradual underflow**, which helps but may cost more.

- Numerical software needs to be careful about avoiding exceptions where possible:
Mathematically equivalent expressions (forms) are not necessarily computationally-equivalent!

Avoiding Overflow / Underflow

- For example, computing $\sqrt{x^2+y^2}$ may lead to overflow in computing x^2+y^2 even though the result does not overflow.
- MATLAB's `hypot` function guards against this. For example (see Wikipedia "hypot"),

$$\sqrt{x^2+y^2} = |x|\sqrt{1+\left(\frac{y}{x}\right)^2} \text{ ensuring that } |x| > |y|$$

  works correctly!
- These kind of careful constructions may have higher computational cost (more CPU operations) or make roundoff errors worse.
- A more sophisticated alternative is to **trap floating exceptions** (e.g., *throw/catch* construct) when they happen and then use an alternative mathematical form, depending on what exception happened.

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 representations, recall that

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

- In general, the **absolute error** $\delta x = \hat{x} - x$ may have contributions from each of the different types of error (roundoff, truncation, propagated, statistical).
- Assume we have an **estimate or bound for the relative error**

$$\left| \frac{\delta x}{x} \right| \lessapprox \epsilon_x \ll 1,$$

based on some analysis, e.g., for roundoff error the IEEE standard determines $\epsilon_x = u$.

Propagation of Errors: Multiplication/Division

- *How does the relative error change (propagate) during numerical calculations?*
- 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.**

Loss of Digits

- 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!
- 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.
We can do the sum in **forward** or in **reverse order**.

Cancellation Error

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

% Single-precision version:
function nhsum=harmonicSP(N)
    nhsumSP=single(0.0);
    for i=1:N % Or, for i=N:-1:1
        nhsumSP=nhsumSP+single(1.0)/single(i);
    end
    nhsum=double(nhsumSP);
end
```

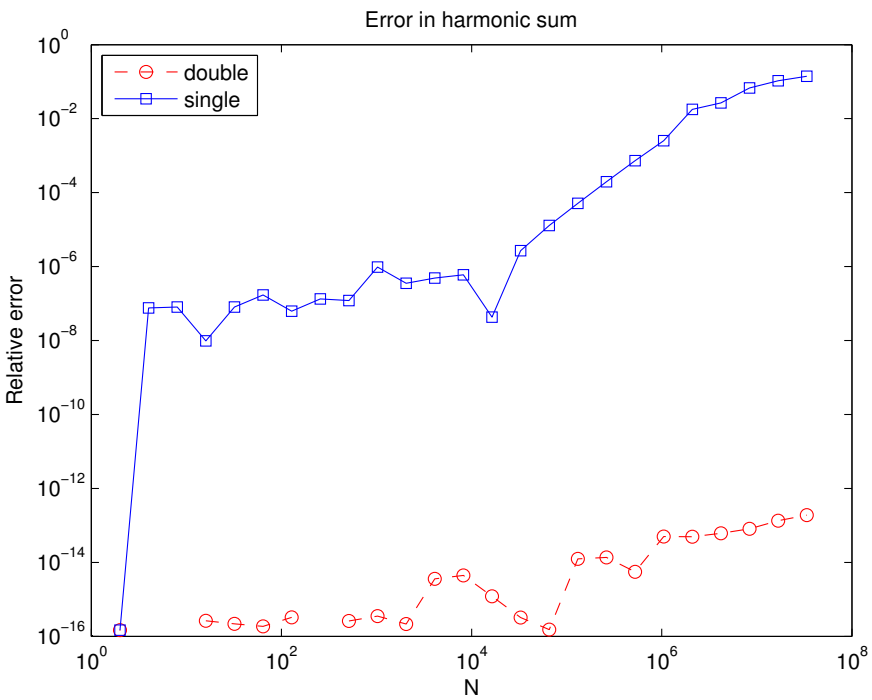
contd.

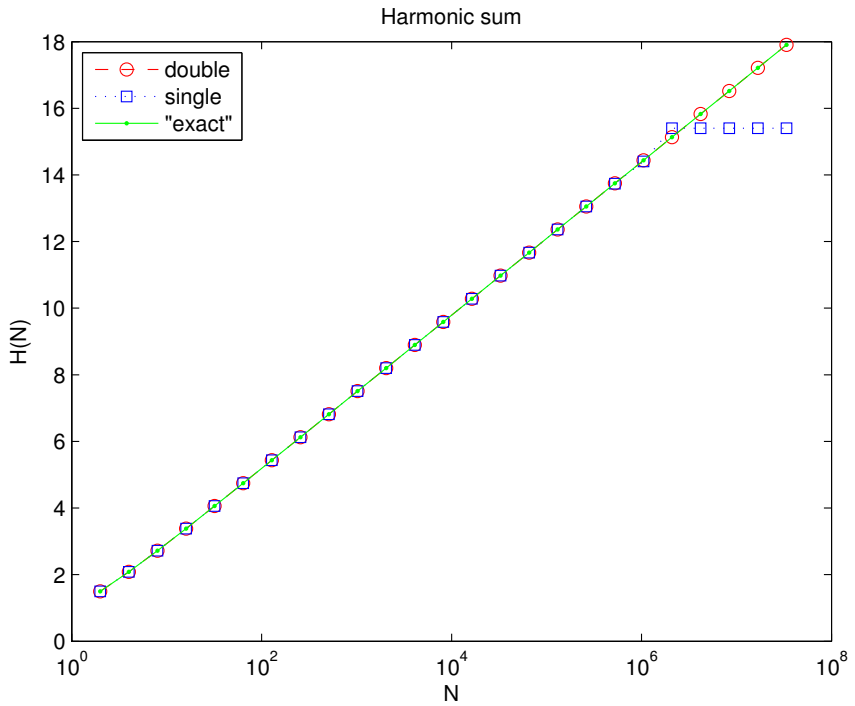
```
clear all; format compact; format long e

npts=25;
Ns=zeros(1,npts); hsum=zeros(1,npts);
relerr=zeros(1,npts); relerrSP=zeros(1,npts);
nhsum=zeros(1,npts); nhsumSP=zeros(1,npts);
for i=1:npts
    Ns(i)=2^i;
    nhsum(i)=harmonic(Ns(i));
    nhsumSP(i)=harmonicSP(Ns(i));
    hsum(i)=(psi(Ns(i)+1)-psi(1)); % Theoretical result
    relerr(i)=abs(nhsum(i)-hsum(i))/hsum(i);
    relerrSP(i)=abs(nhsumSP(i)-hsum(i))/hsum(i);
end
```

contd.

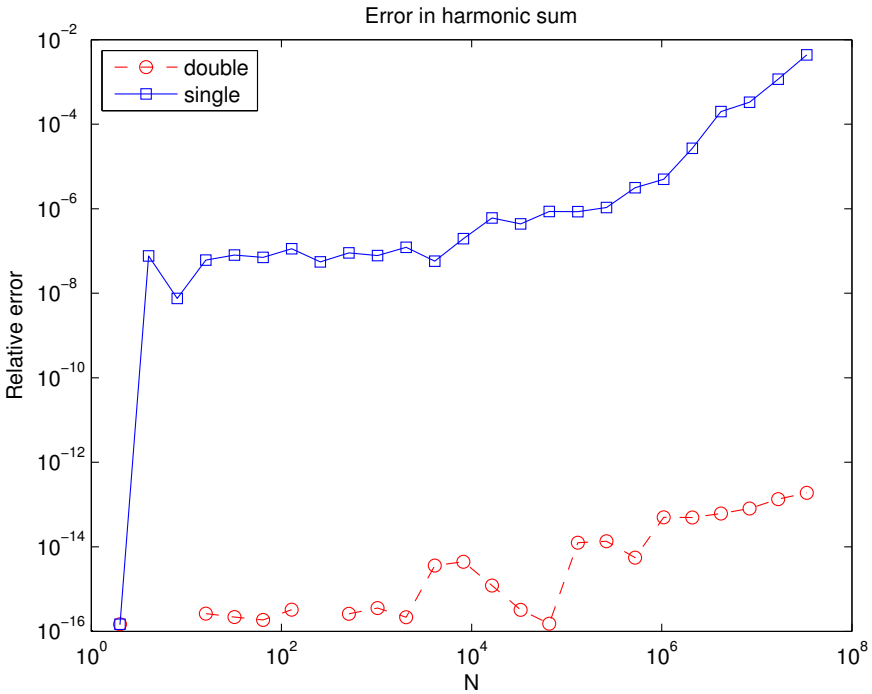
```
figure(1);
loglog(Ns,relerr,'ro--', Ns,relerrSP,'bs-');
title('Error in harmonic sum');
xlabel('N'); ylabel('Relative error');
legend('double','single', 'Location','NorthWest');

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

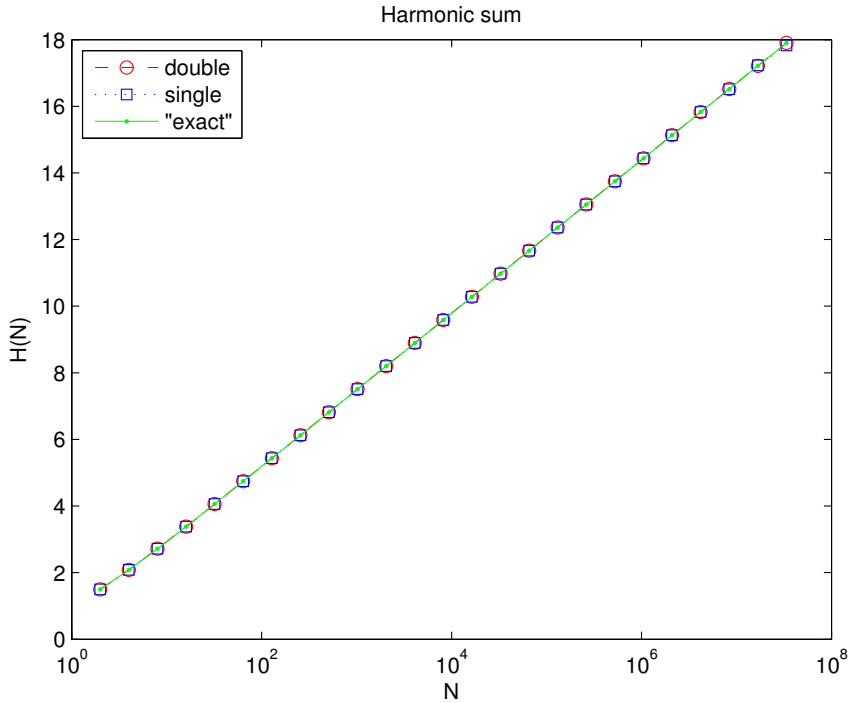
Results: Forward summation





Results: Backward summation





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 $1 - c$ in floating-point arithmetic adds the absolute errors,

$$\mathrm{fl}(1 - c) - (1 - c) \approx |1| \cdot u + |c| \cdot u \approx u,$$

so the absolute and relative errors are on the order of the roundoff unit u for small c.

example contd.

- Assuming that the numerical *sqrt* function computes the root to within roundoff, i.e., to within relative accuracy of u.
- Taking the square root does not change the relative error by more than a factor of 2:

$$\sqrt{x+\delta x} = \sqrt{x}\left(1+\frac{\delta x}{x}\right)^{1/2} \approx \sqrt{x}\left(1+\frac{\delta x}{2x}\right).$$

- For quick analysis, we will simply ignore constant factors such as 2, and estimate that $\sqrt{1-c}$ has an absolute and relative error of order u.
- The absolute errors again get added for the subtraction $1-\sqrt{1-c}$, leading to the estimate of the relative error

$$\left|\frac{\delta x}{x}\right| \approx \epsilon_x = \frac{u}{x}.$$

Avoiding Cancellation

- For small c the solution is

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

so the relative error can become much larger than u when c is close to u,

$$\epsilon_x \approx \frac{u}{c}.$$

- 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: (In)Stability

[From Dahlquist & Bjorck] Consider error propagation in evaluating

$$y_n = \int_0^1 \frac{x^n}{x+5} dx$$

based on the identity

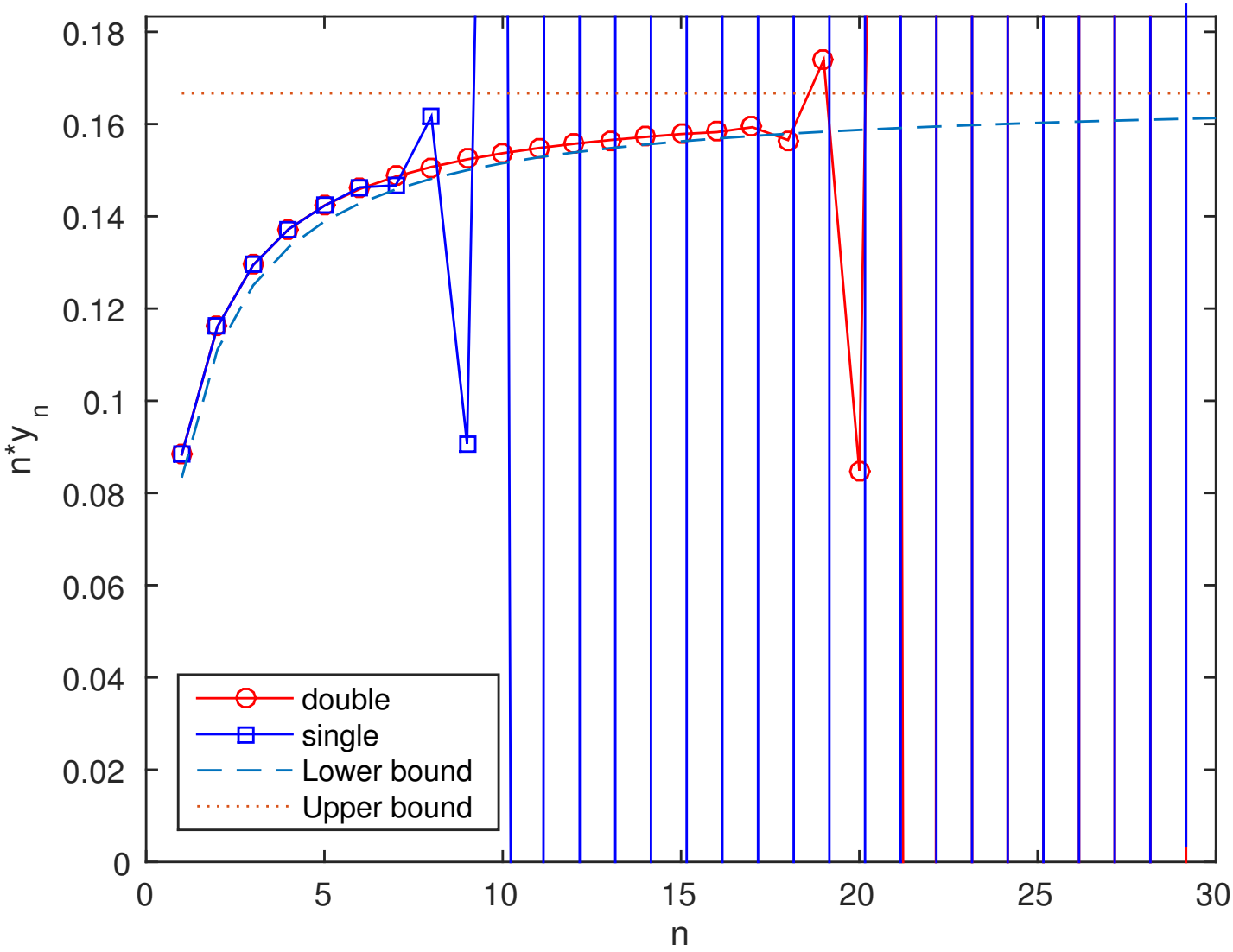
$$y_n + 5y_{n-1} = n^{-1}.$$

- Forward iteration $y_n = n^{-1} - 5y_{n-1}$, starting from $y_0 = \ln(1.2)$, enlarges the error in y_{n-1} by 5 times, and is thus unstable.
- Backward iteration $y_{n-1} = (5n)^{-1} - y_n/5$ reduces the error by 5 times and is thus stable. But we need a starting guess?
- Since $y_n < y_{n-1}$,

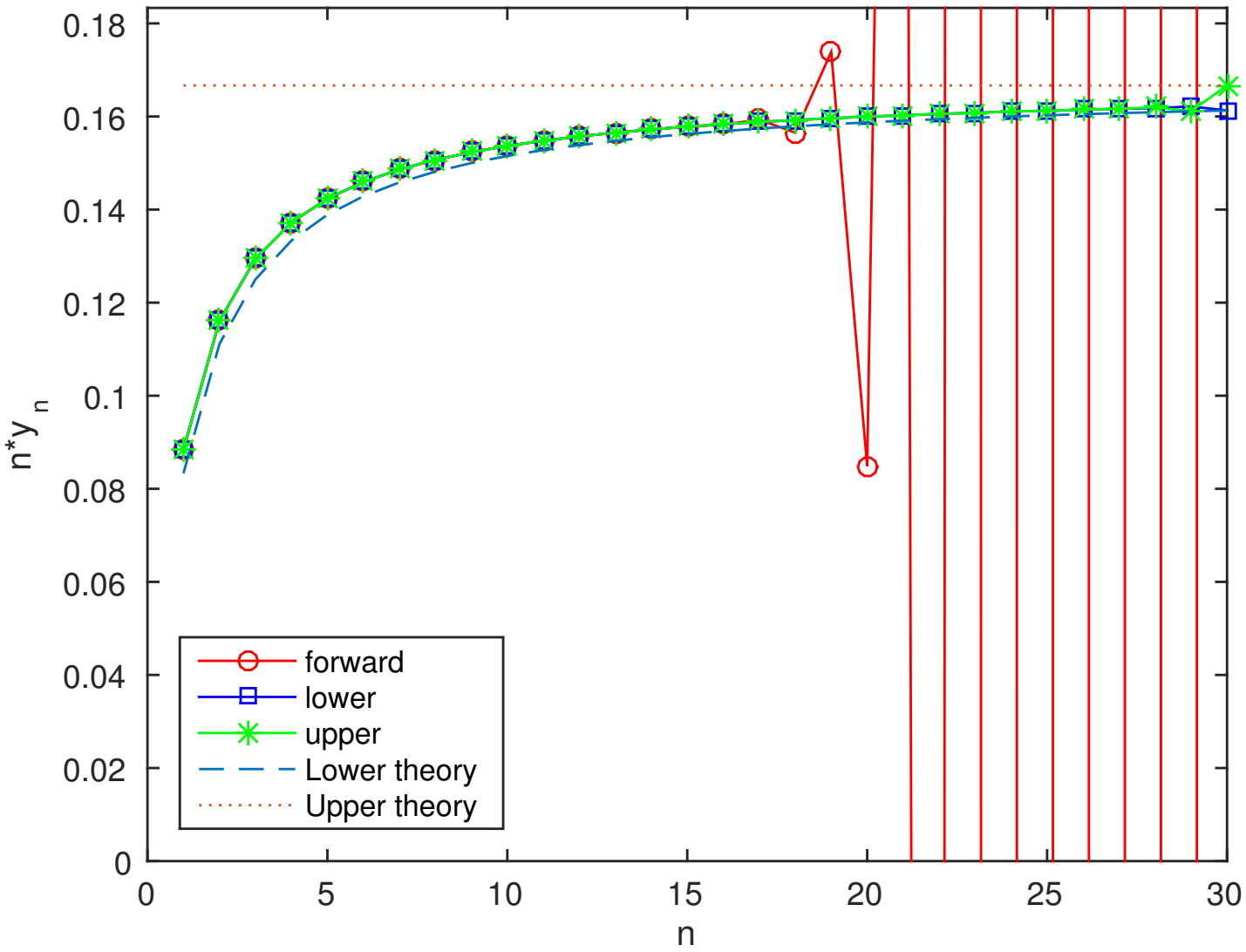
$$6y_n < y_n + 5y_{n-1} = n^{-1} < 6y_{n-1}$$

and thus $0 < y_n < \frac{1}{6n} < y_{n-1} < \frac{1}{6(n-1)}$ so for large n we have tight bounds on y_{n-1} and the error should decrease as we go backward.

Forward Iteration



Forward vs Backward Iteration



Roundoff error propagation: forward

Since the absolute error increases by a factor of 5 every iteration, after n iterations roundoff errors will grow (propagate) by a factor of 5^n. The relative error in $y_n \approx (6n)^{-1}$ can thus be estimated to be $(6n) \cdot 5^n \cdot u$, where u is the unit roundoff. Calculating this shows that the relative error becomes on the order of unity for the n's observed in practice:

1. The relative error in y_8 for single precision is about $6 \cdot 8 \cdot 5^8 \cdot 2^{-24} \approx 1$.
2. The relative error in y_{19} for double precision is about $6 \cdot 19 \cdot 5^{19} \cdot 2^{-53} \approx 0.25$.

For double precision and the forward iteration we get five digits for y_{14}, four digits for y_{15} and only 3 digits for y_{16}.
This is consistent with known values $y_{14} = 0.0112291866264753$, $y_{15} = 0.01052073353428904$ and $y_{16} = 0.009896332328554 84$.

Roundoff error propagation: backward

For the backward iteration we get essentially 16 digits correctly, for example, for y_{15} the difference between the upper and lower bounds is $6 \cdot 10^{-15}$.
To explain this, observe that the initial error for at $n = 30$ is the uncertainty between the upper and lower bounds,

$$\frac{1}{6(n-1)} - \frac{1}{6n} = 1.792114695340500 \cdot 10^{-4}$$

which after 15 steps going backward to $n = 15$ gets reduced by a factor of $5^{15} \approx 3.05 \cdot 10^{10}$ down to $6 \cdot 10^{-15}$.

Local Truncation Error

- To recap: **Approximation error** comes about when we replace a mathematical problem with some easier to solve **approximation**.
- This error is separate and **in addition to** from any numerical algorithm or computation used to actually solve the approximation itself, such as **roundoff or propagated error**.
- Truncation error is a common type of approximation error that comes from replacing **infinitesimally** small quantities with finite **step sizes** and truncating **infinite** sequences/sums with finite ones.
- This is the most important type of error in methods for numerical interpolation, integration, solving differential equations, and others.

Taylor Series

- Analysis of local truncation error is almost always based on using Taylor series to approximate a function around a given point x:

$$f(x+h) = \sum_{n=0}^{\infty} \frac{h^n}{n!} f^{(n)}(x) = f(x) + hf'(x) + \frac{h^2}{2} f''(x) + \ldots,$$

  where we will call h the **step size**.
- This **converges** for finite h only for **analytic functions** (smooth, differentiable functions).
- We cannot do an infinite sum numerically, so we **truncate** the sum:

$$f(x+h) \approx F_p(x,h) = \sum_{n=0}^{p} \frac{h^n}{n!} f^{(n)}(x).$$

- What is the truncation error in this approximation?
  [Note: This kind of error estimate is one of the most commonly used in numerical analysis.]

Taylor Remainder Theorem

- The **remainder theorem** of calculus provides a formula for the error (for sufficiently smooth functions):

$$f(x+h) - F_p(x,h) = \frac{h^{p+1}}{(p+1)!} f^{(p+1)}(\xi),$$

where $x \leq \xi \leq x+h$.

- In general we do not know what the value of ξ is, so we need to estimate it. We want to know what happens for **small step size** h.
- If $f^{(p+1)}(x)$ does not vary much inside the interval $[x, x+h]$, that is, $f(x)$ is sufficiently smooth and h is sufficiently small, then we can approximate $\xi \approx x$.
- This simply means that we **estimate the truncation error with the first neglected term**:

$$f(x+h) - F_p(x,h) \approx \frac{h^{p+1}}{(p+1)!} f^{(p+1)}(x).$$

The Big O notation

- It is justified more rigorously by looking at an **asymptotic expansion for small** h :
$$|f(x+h) - F_p(x,h)| = O(h^{p+1}).$$
- Here the **big** O **notation** means that for small h the error is of smaller magnitude than $\left|h^{p+1}\right|$.
- A function $g(x) = O(G(x))$ if $|g(x)| \leq C\,|G(x)|$ whenever $x < x_0$ for some finite constant $C > 0$.
- Usually, when we write $g(x) = O(G(x))$ we mean that $g(x)$ is of the same order of magnitude as $G(x)$ for small x,
$$|g(x)| \approx C\,|G(x)|\,.$$
- For the truncated Taylor series $C = \frac{f^{(p+1)}(x)}{(p+1)!}$.

Example: Numerical Differentiation

[20 points] Numerical Differentiation

The derivative of a function $f(x)$ at a point x_0 can be calculated using finite differences, for example the first-order *one-sided difference*

$$f'(x = x_0) \approx \frac{f(x_0 + h) - f(x_0)}{h}$$

or the second-order *centered difference*

$$f'(x = x_0) \approx \frac{f(x_0 + h) - f(x_0 - h)}{2h},$$

where h is sufficiently small so that the approximation is good.

example contd.

1. [10pts] Consider a simple function such as $f(x) = \cos(x)$ and $x_0 = \pi/4$ and calculate the above finite differences for several h on a logarithmic scale (say $h = 2^{-m}$ for $m = 1, 2, \cdots$) and compare to the known derivative. For what h can you get the most accurate answer?
2. [10pts] Obtain an estimate of the *truncation error* in the one-sided and the centered difference formulas by performing a Taylor series expansion of $f(x_0 + h)$ around x_0. Also estimate what the *roundoff error* is due to cancellation of digits in the differencing. At some h, the combined error should be smallest (optimal, which usually happens when the errors are approximately equal in magnitude). Estimate this h and compare to the numerical observations.

MATLAB code (1)

```
format compact; format long
clear all;

% Roundoff unit / machine precision:
u=eps('double')/2;

x0 = pi/4;
exact_ans = -sin(x0);
h = logspace(-15,0,100);

%calculate one-sided difference:
f_prime1 = (cos(x0+h)-cos(x0))./h;
%calculate centered difference
f_prime2 = (cos(x0+h)-cos(x0-h))./h/2;

%calculate relative errors:
err1 = abs((f_prime1 - exact_ans)/exact_ans);
err2 = abs((f_prime2 - exact_ans)/exact_ans);
```

MATLAB code (2)

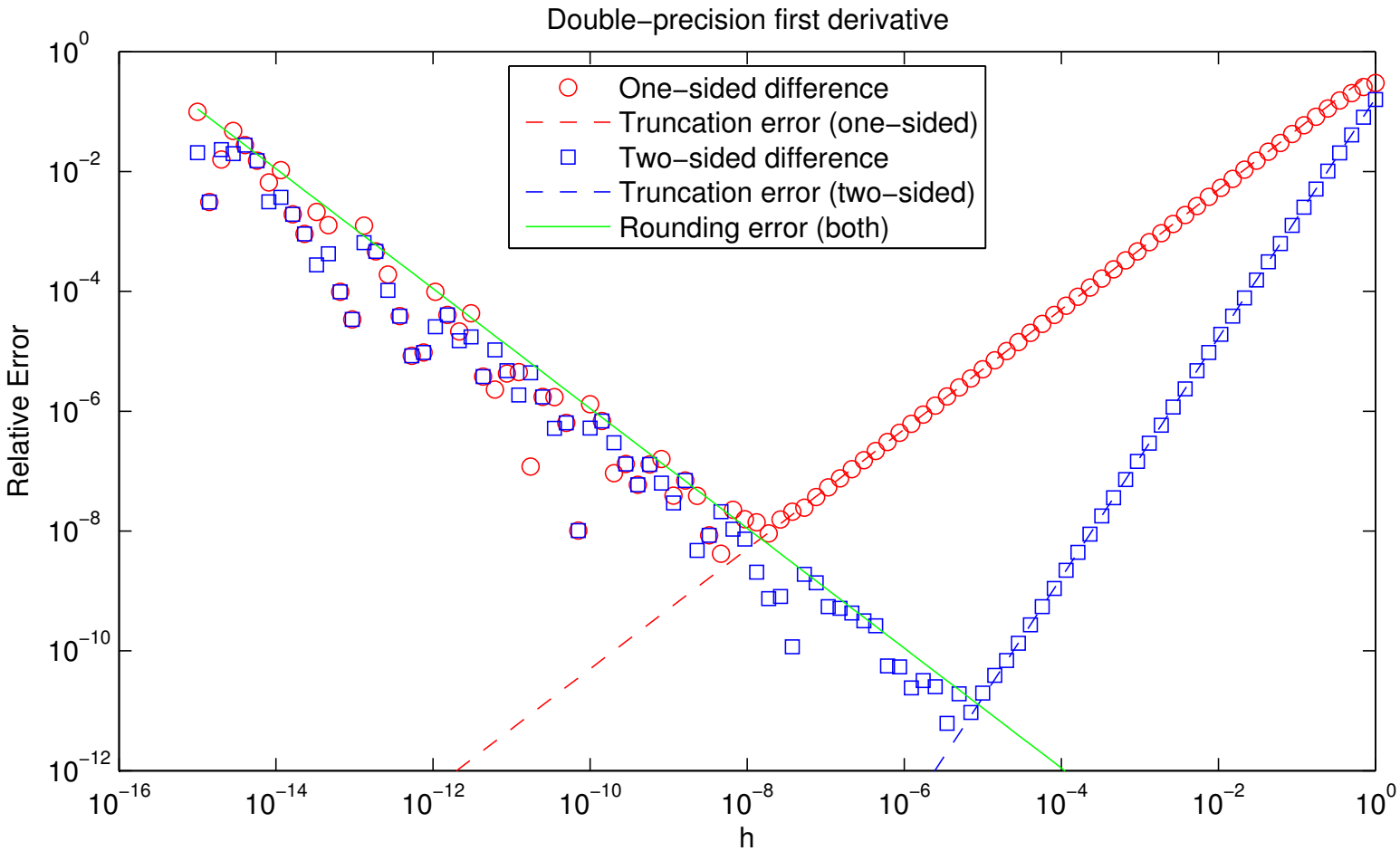
```
%calculate estimated errors
trunc1 = h/2;
trunc2 = h.^2/6;
round = u./h;

% Plot and label the results:
figure(1); clf;
loglog(h,err1,'or', h,trunc1,'--r');
hold on;
loglog(h,err2,'sb', h,trunc2,'--b');
loglog(h,round,'-g');

legend('One-sided difference','Truncation (one-sided)',...
       'Two-sided difference','Truncation (two-sided)',...
       'Rounding error (both)',     'Location','North');
axis([1E-16,1, 1E-12,1]);
xlabel('h'); ylabel('Relative Error')
title('Double-precision first derivative')
```

MATLAB code (2)

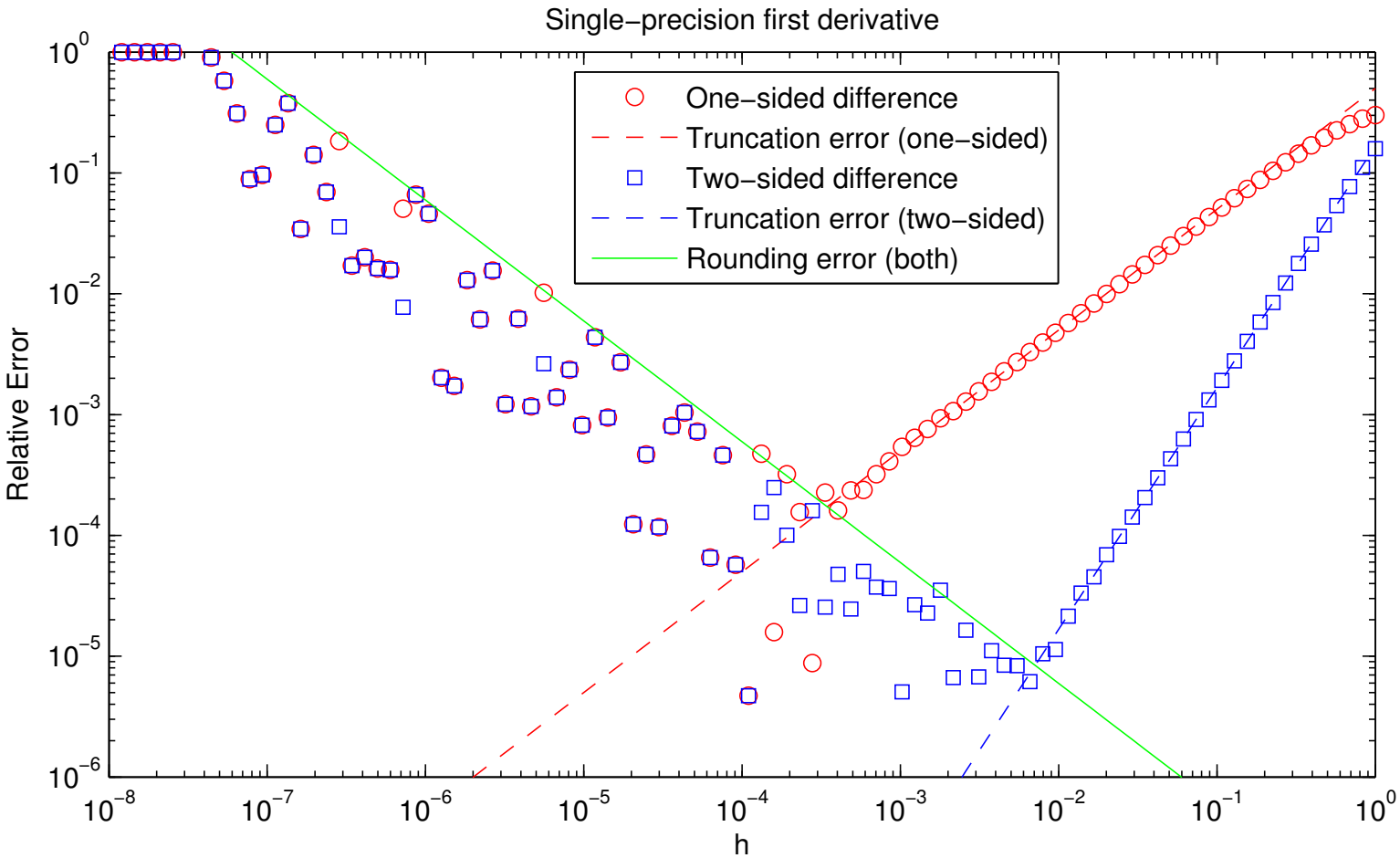
For single, precision, we just replace the first few lines of the code:

```
u=eps('single')/2;
...
x0 = single(pi/4);
exact_ans = single(-sqrt(2)/2);
...
h = single(logspace(-8,0,N));
...
axis([1E-8,1, 1E-6,1]);
```

Double Precision



Single Precision



Truncation Error Estimate

Note that for $f(x) = \sin(x)$ and $x_0 = \pi/4$ we have that

$$\left|f'(x_0)\right| = \left|f'(x_0)\right| = \left|f''(x_0)\right| = 2^{-1/2} \approx 0.71 \sim 1.$$

For the truncation error, one can use the next-order term in the Taylor series expansion, for example, for the one-sided difference:

$$f_1 = \frac{f(x_0+h) - f(x_0)}{h} = \frac{h\, f'(x_0) + h^2\, f''(\xi)/2}{h} = f'(x_0) - f''(\xi)\frac{h}{2},$$

and the magnitude of the relative error can be estimated as

$$|\epsilon_t| = \frac{|f_1 - f'(x_0)|}{|f'(x_0)|} \approx \frac{|f''(x_0)|}{|f'(x_0)|}\frac{h}{2} = \frac{h}{2}$$

and the relative error is of the same order.

Roundoff Error

Let's try to get a rough estimate of the roundoff error in computing

$$f'(x = x_0) \approx \frac{f(x_0 + h) - f(x_0)}{h}.$$

Since for division we add the relative errors, we need to estimate the relative error in the numerator and in the denominator, and then add them up.

Denominator: The only error in the denominator comes from rounding to the nearest floating point number, so the relative error in the denominator is on the order of u.

Numerator

The numerator is computed with absolute error of about u, where u is the machine precision or roundoff unit ($\sim 10^{-16}$ for double precision).
The actual value of the numerator is close to $h f'(x = x_0)$, so the magnitude of the relative error in the numerator is on the order of

$$\epsilon_r \approx \left| \frac{u}{h f'(x = x_0)} \right| \approx \frac{u}{h},$$

since $|f'(x = x_0)| \sim 1$.
We see that due to the cancellation of digits in subtracting nearly identical numbers, we can get a very large relative error when $h \sim u$.
For small $h \ll 1$, the relative truncation error of the whole calculation is thus dominated by the relative error in the numerator.

Total Error

The magnitude of the overall relative error is approximately the sum of the truncation and roundoff errors,

$$\epsilon \approx \epsilon_t + \epsilon_r = \frac{h}{2} + \frac{u}{h}.$$

The minimum error is achieved for

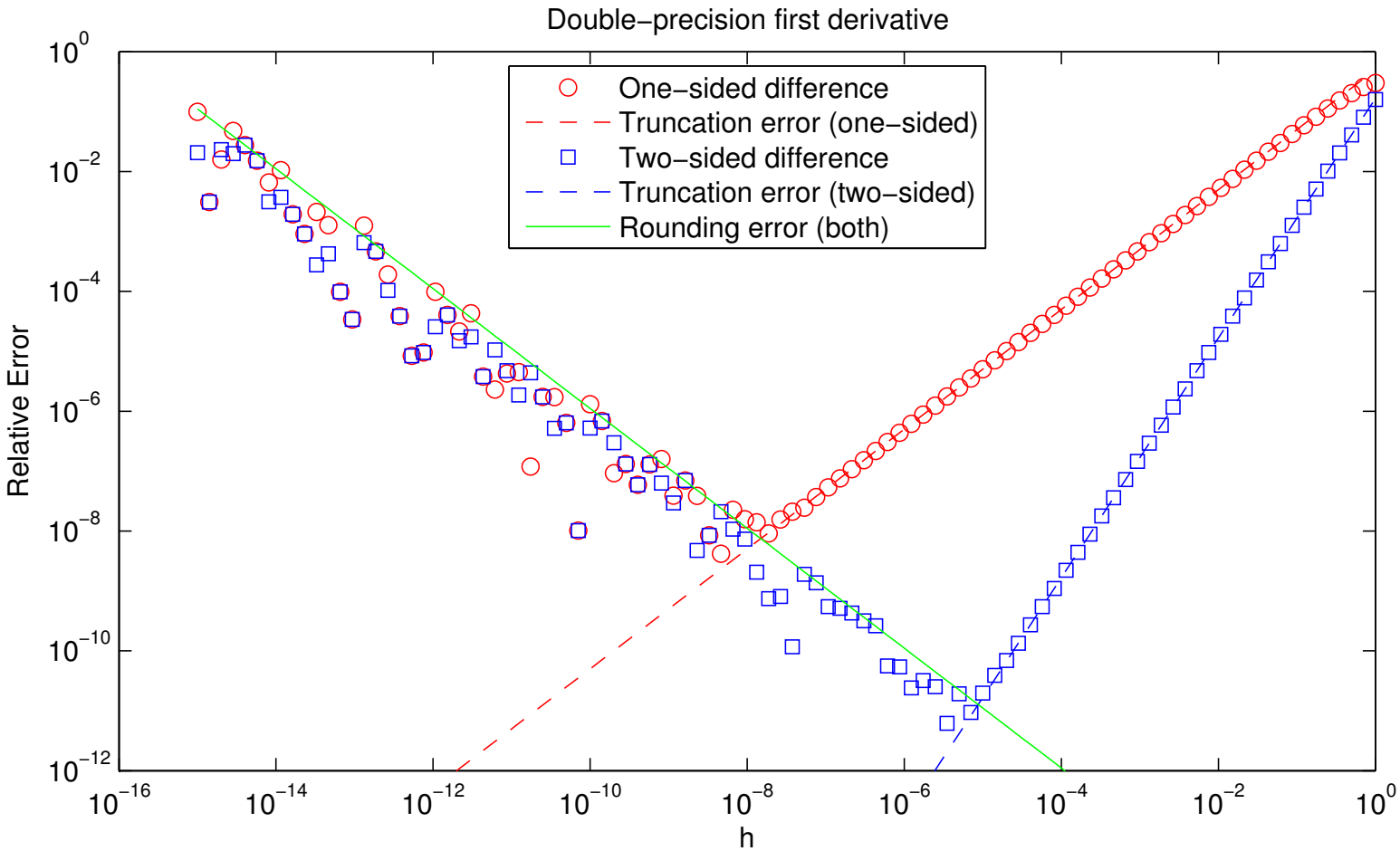
$$h \approx h_{opt} = (2u)^{1/2} \approx (2 \cdot 10^{-16})^{1/2} \approx 2 \cdot 10^{-8},$$

and the actual value of the smallest possible relative error is

$$\epsilon_{opt} = \frac{h_{opt}}{2} + \frac{u}{h_{opt}} = \sqrt{2u} \sim 10^{-8},$$

for double precision. Just replace $u \approx 6 \cdot 10^{-8}$ for single precision.

Double Precision



Conclusions/Summary

- No numerical method can compensate for an **ill-conditioned problem**. But not every numerical method will be a good one for a **well-conditioned problem**.
- A numerical method needs to control the various **computational errors** (**approximation, truncation, roundoff, propagated, statistical**) while balancing computational cost.
- A numerical method must be **consistent** and **stable** in order to **converge** to the correct answer.
- The **IEEE standard** standardizes the **single** and **double precision floating-point formats**, their **arithmetic**, and **exceptions**. It is widely implemented but almost never in its entirety.
- Numerical overflow, underflow and cancellation need to be carefully considered and may be avoided.
  Mathematically-equivalent forms are not numerically-equivalent!