Numerical Methods I
Numerical Computing

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

[1] Course G63.2010.001 / G22.2420-001, Fall 2010

September 9th, 2010

Outline

1. Logistics
2. Sources of Error
3. IEEE Floating-Point Numbers
4. Floating-Point Computations
    - Floating-Point Arithmetic

Course Essentials

- Course webpage: `http://cims.nyu.edu/~donev/Teaching/NMI-Fall2010`
- Registered students: Blackboard page for announcements, grades, and sample solutions. **Sign up for Blackboard ASAP**.
- Office hours: 3 - 5 pm Tuesdays **but open to discussion**, or by appointment.
- Main textbook: **Numerical Mathematics** by Alfio Quarteroni, Riccardo Sacco & Fausto Saleri, Springer, **any** *edition*.
- Secondary textbook: Scientific Computing with MATLAB and Octave, Alfio M. Quarteroni & Fausto Saleri, Springer, **any** *edition*.
- Other optional readings linked on course page.
- Computing is an essential part: MATLAB and preferably compiled languages. Get access to them asap (e.g., Courant Labs).

Assignment 0: Questionnaire

Please log into Blackboard (email me for access if not registered or there is a problem) and submit the following information (also under Assignments on Blackboard and course webpage):

1. Name, degree, and class, any prior degree(s) or professional experience.
2. List all programming languages/environments that you have used, when and why, and your level of experience (just starting, beginner, intermediate, advanced, wizzard).
3. Why did you choose this course instead of Scientific Computing (spring)? Have you taken or plan to take any other course in applied mathematics or computing (e.g., Numerical Methods II)?
4. Was the first lecture at a reasonable level/pace for your background?
5. What are your future plans/hopes for activities in the field of applied and computational mathematics? Is there a specific area or application you are interested in (e.g., theoretical numerical analysis, finance, computational genomics)?

Agenda

- If you have not done it already: Review Linear Algebra through Chapter I of the textbook. Start playing with MATLAB.
- There will be regular homework assignments, usually computational, but with lots of **freedom**. Submit the solutions **on time** (preferably early), preferably as a PDF (give LaTex/lyx a try!), via email or BlackBoard, or handwritten. *Always submit codes electronically*. **First assignment posted and due in two weeks.**
- Very important to the grade is your final research project: choose topic early on! Writeup and presentation due at the end of the semester.
- **Final presentations**: Officially scheduled for 5pm Dec. 23rd (!?!). Email me if you want an alternate earlier date or time (12/20-12/23).
- Please ask questions! Note that I am not a MATLAB expert (I am a Fortran fan).

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.
- **Well-posed** problem: Unique solution that depends continuously on the data. Otherwise it is an intrinsically **ill-posed** problem and no numerical method will work.
- Absolute error δx and relative error ϵ

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

- 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. If $K \sim 1$ the problem is **well-conditioned**. An **ill-conditioned** problem is one that has a large condition number, i.e., one for which a given target relative accuracy of the solution cannot be computed for a given accuracy of the data.

Computational Error

- A **numerical method** must use a finite representation for numbers and thus cannot possibly produce an exact answer for all problems, e.g, 3.14159 but never π.
- Instead, we want to control the **computational errors** (other terms/meanings are used in the literature!):

Approximation error due to replacing the computational problem with an easier-to-solve approximation $\hat{F}_n(\hat{x}_n, \hat{d}_n) = 0$. Also called **discretization error**.

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

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.

Consistency, Stability and Convergence

Many methods generate a sequence of solutions to

$$\hat{F}_n(\hat{x}_n, \hat{d}_n) = 0,$$

where for each n there is an **algorithm** that produces $\hat{x}_n$ given $\hat{d}_n$.

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

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

- Not less important are: **accuracy**, reliability/**robustness**, and **efficiency**.

A Priori Error Analysis

- It is great when the computational error in a given numerical result can be bounded or estimated and the absolute or relative error reported along with the result.
- **A priori analysis** gives guaranteed error bounds but it may involve quantities that are difficult to compute (e.g., matrix inverse, condition number).
- **A posteriori analysis** tries to estimate the error from quantities that are actually computed.
- Take the example

$$\text{Solve the linear system } \mathbf{A}\mathbf{x}=\mathbf{b}$$

where the matrix **A** is considered free of errors, but **b** is some input data that has some error.

A priori Analysis

- In **forward error analysis** one tries to estimate the error bounds on the result in each operation in the algorithm in order to bound the error in the result
$$\|\delta \mathbf{x}\| \text{ given } \|\delta \mathbf{b}\|$$
It is often **too pessimistic** and hard to calculate: $\delta \mathbf{x} = \mathbf{A}^{-1}(\delta \mathbf{b})$.
- In **backward error analysis** one calculates, for a given output, how much one would need to perturb the input in order for the answer to be exact.
$$\|\delta \mathbf{b}\| \text{ given } \hat{\mathbf{x}} \approx \mathbf{x}$$
It is often much **tighter and easier** to perform than forward analysis: $\delta \mathbf{b} = \mathbf{r} = \mathbf{A}\hat{\mathbf{x}} - \mathbf{b}$.
- Note that if $\mathbf{b}$ is only known/measured/represented with accuracy smaller than $\|\mathbf{r}\|$ then $\hat{\mathbf{x}}$ is a perfectly good solution.
- **A posteriori analysis** tries to estimate $\|\delta \mathbf{x}\|$ given $\|\mathbf{r}\|$.

Example: Convergence

[*From Dahlquist & Bjorck*] Consider solving

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

by using a fixed-point iteration

$$x_{n+1} = f(x_n), \text{ i.e., } F_{n+1} = f(x_n) - x_{n+1}$$

along with some initial guess x_0. This is (strongly) consistent with the mathematical problem since $F_{n+1}(x) = 0$.

- Consider the calculation of square roots, $x = \sqrt{c}$.
- First, take the numerical method $x_{n+1} = f(x_n) = c/x_n$. It is obvious this 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.
- On the other hand, the Babylonian method for square roots

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

  is also consistent but it also converges (quadratically) for any non-zero initial guess (see Wikipedia article)!

Example: Stability

[From Dahlquist & Bjorck, also **homework**] 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.

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

Computers represent everything using bit strings, i.e., integers in base-2. Integers can thus be exactly represented. But not real numbers!
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).

Floating Point Representation

- 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

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

  has a problem with representing large or small numbers: 1.156 but 0.011.
- 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** $L \leq e \leq U$.

IEEE Standard Representations

- 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.
- **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

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

On the computer:

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

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

IEEE formats contd.

- Double precision 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 in implementing **gradual underflow** arithmetic.

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 $\hat{x} = \text{fl}(x)$.
- 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).

Practical advice about IEEE arithmetic

- 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).
- Optimization, especially in compiled languages, can rearrange terms or perform operations using **unpredictable** alternate forms. **Using parenthesis helps** , e.g. $(x + y) - z$ instead of $x + y - z$, but does not eliminate the problem.
- Intermediate results of calculations do not have to be stored in IEEE formats (e.g., Intel chips may use 80-bits internally), which helps with accuracy but leads to unpredictable results.
- **Do not compare floating point numbers** (especially for loop termination), or more generally, do not rely on logic from pure mathematics.
- Library functions such as sin and ln will typically be computed almost to full machine accuracy, but do not rely on that.

Floating-Point Exceptions

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

Divide-by-zero	if the result is $\pm\infty$
Invalid	if the result is a NaN
Overflow	if the result is too large to be represented
Underflow	if the result is too small to be represented

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

Propagation of Errors

- 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. This is good!
- For addition and subtraction, the bounds on the **absolute** errors add to give an estimate of the absolute error in the result.
  This is much more dangerous since the relative error is not controlled!
- Adding two numbers of widely-differing magnitude leads to loss of accuracy due to roundoff error. This can become a problem when adding many terms, such as infinite series.
- 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**.

Growth of Truncation 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
```

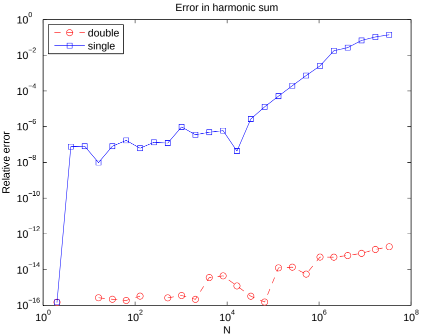
contd.

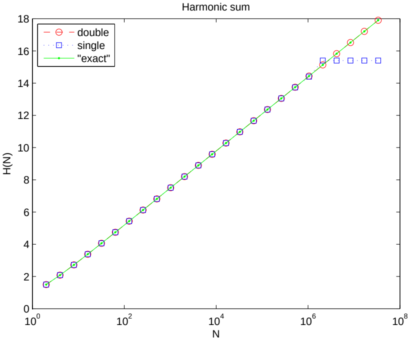
```
npts=25;
Ns=zeros(1,npts); hsum=zeros(1,npts);
relerr=zeros(1,npts); relerrSP=zeros(1,npts);
nhsum=zeros(1,npts); nhsumSP=zeros(1,npts);
for i=1:npts
    Ns(i)=2^i;
    nhsum(i)=harmonic(Ns(i));
    nhsumSP(i)=harmonicSP(Ns(i));
    hsum(i)=(psi(Ns(i)+1)-psi(1)); % Theoretical result
    relerr(i)=abs(nhsum(i)-hsum(i))/hsum(i);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

contd.

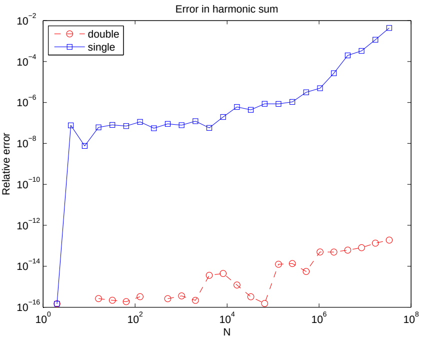
```
figure(1);
loglog(Ns,relerr,'ro--', Ns,relerrSP,'bs-');
title('Error␣in␣harmonic␣sum');
xlabel('N'); ylabel('Relative␣error');
legend('double','single', 'Location','NorthWest');

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');
```

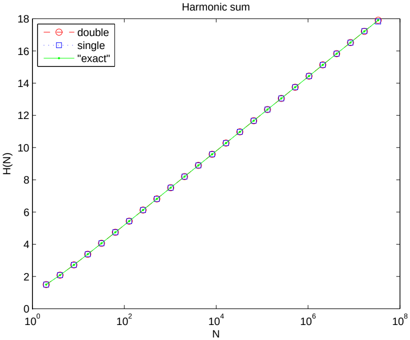
Results: Forward summation





Results: Backward summation





Numerical Cancellation

- If x and y are close to each other, $x - y$ can have reduced accuracy due to **cancellation** of digits.
  Note: If gradual underflow is not supported $x - y$ can be zero even if x and y are not exactly equal.
- **Benign cancellation**: subtracting two **exactly-known** IEEE numbers with the use of a guard digit results in a relative error of no more than an ulp. The result is **precise**.
- **Catastrophic cancellation** occurs when subtracting two nearly equal **inexact** numbers and leads to loss of accuracy and a large relative error in the result.
  For example, $1.1234 - 1.1223 = 0.0011$ which only has 2 significant digits instead of 4. The result is not **accurate**.

Cancellation Example

```
>> format long % or format hex
>> x=pi
x =    3.141592653589793
x =    400921fb54442d18 % Note 8=1000
>> y=x+eps(x)
y =    3.141592653589794
y =    400921fb54442d19 % Note 9=1001
>> z=x*(1+eps)
z =    3.141592653589794
z =    400921fb54442d1a % Note a=1010
>> w=x+x*eps(x)
w =    3.141592653589794
w =    400921fb54442d1b % Note b=1011
```

Cancellation Example

```
>> y-x
ans =      4.440892098500626e-16
ans =   3cc0000000000000
>> z-x % Benign cancellation (result is precise)
ans =      8.881784197001252e-16
ans =   3cd0000000000000
>> w-x % Benign (?) (result is not accurate)
ans =      1.332267629550188e-15
ans =   3cd8000000000000
>> 2^(-51)
ans =      4.440892098500626e-16
ans =   3cc0000000000000

>> x*eps % This is the actual result we are after!
ans =      1.395147399203453e-15
ans =   3cc921fb54442d18
```

Avoiding Cancellation

- Rewriting in mathematically-equivalent but numerically-preferred form is the first try, e.g., instead of

  $$\sqrt{x+\delta}-\sqrt{x} \text{ use } \frac{\delta}{\sqrt{x+\delta}+\sqrt{x}},$$

  or instead of $x^2 - y^2$ use $(x-y)(x+y)$ to avoid catastrophic cancellation instead of just benign cancellation in x and y.
  But what about the **extra cost**?
- Sometimes one can use Taylor series or other approximation to get an approximate but stable result, e.g.,

  $$\sqrt{x+\delta}-\sqrt{x} \approx \frac{\delta}{2\sqrt{x}} \text{ for } \delta \ll x.$$

- See homework for some examples.

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 (attempts?) 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!