Scientific Computing
Numerical Computing

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

[1]Course MATH-GA.2043 or CSCI-GA.2112, Spring 2012

January 26th, 2012

Outline

1. Logistics
2. Conditioning
3. Roundoff Errors
   - Floating-Point Computations
4. Some Computing Notes

Course Essentials

- Course webpage: `http://cims.nyu.edu/~donev/Teaching/SciComp-Spring2012/`
- Registered students: Blackboard page for announcements, submitting homeworks, grades, and sample solutions. **Sign up for Blackboard ASAP**.
- Office hours: 4 - 6 pm Tuesdays, or by appointment. Grader's office hours **TBD**.
- Main textbook: **Scientific Computing with MATLAB and Octave**, Alfio M. Quarteroni & Fausto Saleri, Springer, **any** *edition*.
- 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.
- If you have not done it already: **Review Linear Algebra** using Jonathan Goodman's notes.
- Get access to **MATLAB** asap (e.g., Courant Labs) and start playing.
- There will be regular homework assignments (70% of grade), mostly computational. Points from all assignments will be added together.
- Submit the solutions as a PDF (give LaTex/lyx a try!), via BlackBoard, 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** (30%) due May 10th, which will be similar in spirit to the homeworks but more involved.

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-conditioning 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!

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

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

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.

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.
- **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.

Conclusions/Summary

- We will continue our discussion of errors, including propagated and truncation errors, **next class**...
- Specific examples illustrating the concepts will also be given next class, and of course in the homework.