Lecture 2

Preliminaries

Analysis vs Numerical Analysis

Ceng375 *Numerical Computations* at October 7, 2010

Dr. Cem Özdoğan
Computer Engineering Department
Çankaya University

Contents

Introduction
Analysis vs Numerical Analysis
An Illustrative Example
Some disasters attributable to bad numerical computing
Kinds of Errors in Numerical Procedures
Absolute vs Relative Error & Convergence
Floating-Point Arithmetic
Round-off Error vs Truncation Error
Well-posed and well-conditioned problems
Forward and Backward Error Analysis
Computer Number Representation

How a computer can be used

1. to solve problems that may not be solvable by hand
2. to solve problems (that you may have solved before) in a different way

- Many of these simplified examples can be solved analytically (by hand)

$$x^3 - x^2 - 3x + 3 = 0, \textit{with solution } \sqrt{3}$$

- But most of the examples can not be simplified and can not be solved analytically
- mathematical relationships $\Longrightarrow$ simulate some real word situations

Five Basic Operations

- In mathematics, solve a <u>problem</u> through <u>equations</u>; **algebra, calculus, differential equations (DE), Partial DE,** . . .
- In numerical analysis; four operations (add, subtract, multiply, division) and Comparison.
  - These operations are exactly those that computers can do

$$\int_0^{\pi} \sqrt{1 + cos^2 x} dx$$

    - length of one arch of the curve y-sinx; no solution with "a substitution' or "integration by parts"
    - numerical analysis can compute the length of this curve by standardised methods that apply to essentially any integrand
- Another difference between a numerical results and analytical answer is that the former is always an <u>approximation</u>
  - this can usually be <u>as accurate as needed</u> (level of accuracy)
- Numerical Methods require repetitive arithmetic operations $\Rightarrow$ a computer to carry out
- Also, a human would make so many mistakes

Illustrative Example I

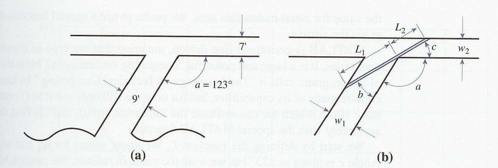

Figure: An illustrating example: The ladder in the mine.

What is the longest ladder ($L_1 + L_2$)? (see the Fig. 1)

$$L_1 = \frac{w_1}{Sinb}, \quad L_2 = \frac{w_2}{Sinc}, \quad b = \pi - a - c$$

$$L = L_1 + L_2 = \frac{w_1}{sin(\pi - a - c)} + \frac{w_2}{sinc}$$

The maximum length of the ladder $\Rightarrow \frac{dL}{dc}\rfloor_{c=C} = 0 \Rightarrow$ calculus way

Illustrative Example II

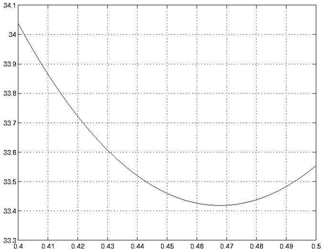

Figure: An illustrating example: The ladder in the mine. Solution with MATLAB

MATLAB way is as the following: (see the Fig. 2)

```
a=123*2*pi*/360
L=inline('9/sin(pi-2.1468-c)+7/sin(c)')
fplot(L,[0.4,0.5]); grid on
fminbnd(L,0.4,0.5)
L(0.4677)
fminbnd(L,0.4,0.5,optimset('Display','iter'))
```

Some disasters attributable to bad numerical computing

Have you been paying attention in your numerical analysis or scientific computation courses? If not, it could be a costly mistake. Here are some real life examples of what can happen when numerical algorithms are not correctly applied.

- The Patriot Missile failure, in Dharan, Saudi Arabia, on February 25, 1991 which resulted in 28 deaths, is ultimately attributable to *poor handling of rounding errors*.
- The explosion of the Ariane 5 rocket just after lift-off on its maiden voyage off French Guiana, on June 4, 1996, was ultimately the *consequence of a simple overflow*.
- The sinking of the Sleipner A offshore platform in Gandsfjorden near Stavanger, Norway, on August 23, 1991, resulted in a loss of nearly one billion dollars. It was found to be the *result of inaccurate finite element analysis*.

Kinds of Errors in Numerical Procedures I

Computers use only a fixed number of digits to represent a number.

- As a result, the numerical values stored in a computer are said to have *finite precision*.
- Limiting precision has the desirable effects of increasing the speed of numerical calculations and reducing memory required to store numbers.
- But, what are the undesirable effects?

Kinds of Errors:

i **Error in Original Data**

ii **Blunders** (an embarrassing mistake): Sometimes a test run with known results is worthwhile, but is no guarantee of freedom from foolish error.

Kinds of Errors in Numerical Procedures II

Kinds of Errors:

iii **Truncation Error**: i.e., approximate e^x by the cubic power

$$P_3(x) = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!}; \qquad e^x = P_3(x) + \sum_{n=4}^{\infty} \frac{x^n}{n!}$$

- Approximating e^x with the cubic gives an inexact answer. The error is due to truncating the series,
- When to cut series expansion $\Longrightarrow$ be satisfied with an approximation to the exact analytical answer.
- Unlike roundoff, which is controlled by the hardware and the computer language being used, truncation error is under control of the programmer or user.
- Truncation error can be reduced by selecting more accurate discrete approximations. But, it can not be eliminated entirely.

Evaluating the Series for sin(x)

$$sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots$$

Kinds of Errors in Numerical Procedures III

- *Cont:* Evaluating the series for sin(x) (**Example m-file:** sinser.m)
- An efficient implementation of the series uses recursion to avoid overflow in the evaluation of individual terms. If T_k is the kth term ($k = 1, 3, 5, \ldots$) then

$$T_k = \frac{x^2}{k(k-1)} T_{k-2}$$

$>> sinser(pi/6)$

- Study the effect of the parameters *tol* and *nmax* by changing their values (Default values are 5e-9 and 15, respectively).

Kinds of Errors:

iv **Propagated Error**:
- more subtle (difficult to analyse)
- by propagated we mean an error in the succeeding steps of a process due to an occurrence of an earlier error
- of critical importance
- stable numerical methods; errors made at early points die out as the method continues
- unstable numerical method; does not die out

Kinds of Errors in Numerical Procedures IV

Kinds of Errors:

- **Round-off Error**:

```
>> format long e  %display all available digits
>> x=(4/3)*3
x =      4
>> a=4/3            %store double precision approx of 4/3
a =      1.333333333333333e+00
>> b=a-1            %remove most significant digit
b =      3.333333333333333e-01
>> c=1-3*b          %3*b=1 in exact math
c =      2.220446049250313e-16  %should be 0!!
```

To see the effects of roundoff in a simple calculation, one need only to force the computer to store the intermediate results.

- All computing devices represents numbers, except for integers and some fractions, with some imprecision
- Floating-point numbers of fixed word length; the true values are usually not expressed exactly by such representations
- If the number are rounded when stored as floating-point numbers, the round-off error is less than if the trailing digits were simply chopped off

Kinds of Errors in Numerical Procedures V

```
>> x=tan(pi/6)
x =      5.773502691896257e-01
>> y=sin(pi/6)/cos(pi/6)
y =      5.773502691896256e-01
>> if x==y
fprintf('x and y are equal \n');
else
fprintf('x and y are not equal : x-y =%e \n',x-y);
end
x and y are not equal : x-y =1.110223e-16
```

- The test is true only if x and y are exactly equal in bit pattern.
- Although x and y are equal in exact arithmetic, their values differ by a small, but nonzero, amount.
- When working with floating-point values the question “are x and y equal?” is replaced by “are x and y close?” or, equivalently, “is $x - y$ small enough?”

Absolute vs Relative Error & Convergence I

- Accuracy (how close to the true value) $\longrightarrow$ great importance,
- *absolute error* $=$ $|$*true value* $-$ *approximate error*$|$
  A given size of error is usually more serious when the magnitude of the true value is small,
- *relative error* $= \frac{\textit{absolute error}}{|\textit{true value}|}$

Convergence of Iterative Sequences:

- Iteration is a common component of numerical algorithms. In the most abstract form, an iteration generates a sequence of scalar values $x_k, k = 1, 2, 3, \ldots$. The sequence converges to a limit ξ if

$$|x_k - \xi| < \delta, \textit{ for all } k > N$$

  where δ is a small number called the convergence tolerance. We say that the sequence has converged to within the tolerance δ after N iterations.

Absolute vs Relative Error & Convergence II

- Iterative computation of the square root (**Example m-file:** testSqrt.m, newtsqrtBlank.m)
- The goal of this example is to explore the use of different expressions to replace the ”NOT_CONVERGED“ string in the *while* statement (see newtsqrtBlank.m, then save as newtsqrt.m). Some suggestions are given as:
  $r \sim= rold$
  $(r - rold) > delta$
  $abs(r - rold) > delta$
  $abs((r - rold)/rold) > delta$
- Study each case ($>>$ *testSqrt*), and which one should be used?

Floating-Point Arithmetic:

- Performing an arithmetic operation $\Rightarrow$ no exact answers unless only integers or exact powers of 2 are involved,
- Floating-point (real numbers)$\rightarrow$ not integers,
- Resembles scientific notation,
- IEEE standard $\rightarrow$ storing floating-point numbers (see the Table 1).

Floating-Point Arithmetic I

Table: Floating→ Normalised.

floating	normalised (shifting the decimal point)
13.524	$.13524 * 10^2$ ($.13524E2$)
-0.0442	$-.442E-1$

- the sign $\pm$
- the fraction part (called the *mantissa*)
- the exponent part

There are three levels of precision (see the Fig. 3)

		Number of bits in			
Precision	Length	Sign	Mantissa	Exponent	Range
Single	32	1	23(+1)	8	$10^{\pm 38}$
Double	64	1	52(+1)	11	$10^{\pm 308}$
Extended	80	1	64	15	$10^{\pm 4931}$

Figure: Level of precision.

Floating-Point Arithmetic II

- What about the sign of the exponent? Rather than use one of the bits for the sign of the exponent, exponents are biased.
- For **single** precision (we have 8 bits reserved for the exponent):
  - 2^8=256
  - 0$\longrightarrow$00000000 = 0
  - 255$\longrightarrow$11111111=255
  - 0 (255)$\Longrightarrow$ -127 (128). An exponent of -127 (128) stored as 0 (255).
  - So biased$\longrightarrow$ $2^{128} = 3.40282E + 38$, mantissa gets 1 as maximum
  - **Largest:** 3.40282E+38; **Smallest:** 2.93873E-39
  - For **double** and **extended** precision the bias values are 1023 and 16383, respectively.
  - $\frac{0}{0}, 0 * \infty, \sqrt{-1} \Longrightarrow NaN$: Undefined.

```
>> realmin
ans =   2.2251e-308
>> realmax
ans =   1.7977e+308
>> format long e
>> 10*realmax
>> realmin/10
>> realmin/1e16
```

Floating-Point Arithmetic II

When a calculation results in a value smaller than **realmin**, there are two types of outcomes.

1. If the result is slightly smaller than **realmin**, the number is stored as a denormal (they have fewer significant digits than normal floating point numbers).
2. Otherwise, It is stored as 0.

- Interval Halving to Oblivion (the state of being disregarded or forgotten) (**Example m-file:** halfDiff.m)

  $x_1 = \ldots, x_2 = \ldots$
  for k=1,2,...
  $\delta = (x_1 - x_2)/2$
  if $\delta = 0$, stop
  $x_2 = x_1 + \delta$
  end

- As the floating-point numbers become closer in value, the computation of their difference relies on digits with decreasing significance.
- When the difference is smaller than the least significant digit in their mantissa, the value of δ becomes zero.

Round-off Error vs Truncation Error I

EPS: short for epsilon⟶used for represent the smallest machine value that can be added to 1.0 that gives a result distinguishable from 1.0. In MATLAB:

» eps

ans=2.2204E-016

- $eps \longrightarrow \varepsilon \Longrightarrow (1+\varepsilon)+\varepsilon = 1$ *but* $1+(\varepsilon+\varepsilon) > 1$
- Two numbers that are very close together on the *real* number line can not be distinguished on the *floating-point* number line if their difference is less than the least significant bit of their mantissas.

Round-off Error vs Truncation Error:

- Round-off occurs, even when the procedure is exact, due to the imperfect precision of the computer,
- Analytically $\frac{df}{dx} \Rightarrow f'(x) = lim_{h\to 0}\frac{f(x+h)-f(x)}{(x+h)-x}$: Procedure
- Approximate value for $f'(x)$ with a small value for $\boldsymbol{h}$,
- $h \longrightarrow$ *smaller*, the result is closer to the true value⟶truncation error is reduced,

Round-off Error vs Truncation Error II

- But at some point (depending of the precision of the computer) round-off errors will dominate⟶less exact⟹***There is a point where the computational error is least.***
- Roundoff and Truncation errors in the series for e^x (**Example m-file:** expSeriesPlot.m)
- Let T_k be the kth term in the series and S_k be the value of the sum after k terms:

$$T_k = \frac{x^k}{k!}, S_k = 1 + \sum_{j=1}^{k} T_k$$

If the sum on the right-hand side is truncated after k terms, the absolute error in the series approximation is

$$E_{abs,k} = |S_k - e^x|$$

» expSeriesPlot(-10,5e-12,60)

- as k increases, $E_{abs,k}$ decreases, due to a decrease in the truncation error.

Round-off Error vs Truncation Error III

- Eventually, roundoff prevents any change in S_k. As $T_{k+1} \rightarrow 0$, the statement

$$ssum = ssum + term$$

  produces no change in *ssum*.
- For $x = -10$ this occurs at $k \sim 48$. At this point, the truncation error, $|S_k - e^x|$ is not zero.
- Rather, $|T_{k+1}/S_k| < \varepsilon_m$. This is an example of the independence of truncation error and roundoff error.
- For $k < 48$, the error in evaluating the series is controlled by truncation error.
- For $k > 48$, roundoff error prevents any reduction in truncation error.

Well-posed and well-conditioned problems

The accuracy depends not only on the computer's accuracy

- A problem is well-posed if a solution; exists, unique, depends on varying parameters
  - A nonlinear problem⟶linear problem
  - infinite⟶large but finite
  - complicated⟶simplified
- A well-conditioned problem is not sensitive to changes in the values of the parameters (small changes to input do not cause to large changes in the output)
- Modelling and simulation; the model may be not a really good one
- if the problem is well-conditioned, the model still gives useful results in spite of small inaccuracies in the parameters

Forward and Backward Error Analysis

- $y = f(x)$
- $y_{calc} = f(x_{calc}) : \ computed\ value$
- $E_{fwd} = y_{calc} - y_{exact}$ where y_{exact} is the value we would get if the computational error were absent
- $E_{backwd} = x_{calc} - x, \ y_{calc} = f(x_{calc})$
- Example: $y = x^2$, $x = 2.37$ used only two digits
- $y_{calc} = 5.6\ while\ y_{exact} = 5.6169$
- $E_{fwd} = -0.0169$, relative error $\rightarrow$0.3%
- $\sqrt{5.6} = 2.3664 \Rightarrow E_{backw} = -0.0036$, relative error $\rightarrow$0.15%

Computer Number Representation I

Examples of Computer Numbers: Say we have six bit representation (not single, double) (see the Fig. 4)

- 1 *bit* $\rightarrow$ *sign*
- $3(+1)$ *bits* $\rightarrow$ *mantissa*
- 2 *bits* $\rightarrow$ *exponent*

Sign	Mantissa	Exponent	Value
0	(1)001	00	$9/16 * 2^{-1} = +9/32$
0	(1)111	11	$15/16 * 2^{2} = +15/4$

Figure: Computer numbers with six bit representation.

- For positive range $\frac{9}{32} \longleftrightarrow \frac{15}{4}$
- For negative range $\frac{-15}{4} \longleftrightarrow \frac{-9}{32}$; even discontinuity at point zero since it is not in the ranges.

Computer Number Representation II

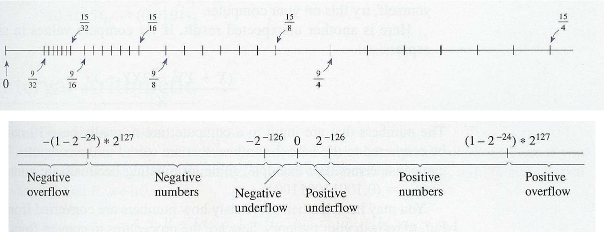

Figure: Upper: number line in the hypothetical system, Lower: IEEE standard.

Computer Number Representation III

- Very simple computer arithmetic system $\Rightarrow$ the gaps between stored values are very apparent.
- Many values can not be stored exactly. i.e., 0.601, it will be stored as if it were 0.6250 because it is closer to $\frac{10}{16}$, an error of 4%
- In IEEE system, gaps are much smaller but they are still present. (see the Fig. 5)

Anomalies with Floating-Point Arithmetic:
For some combinations of values, these statements are not true

- $(x+y)+z = x+(y+z)$
  $(x*y)*z = x*(y*z)$
  $x*(y+z) = (x*y)+(x*z)$
- adding 0.0001 one thousand times should equal 1.0 exactly but this is not true with single precision
- $z = \frac{(x+y)^2 - 2xy - y^2}{x^2}$, problem with single precision