Lecture 3

Introduction to Numerical Fundamentals

Numbers, Errors

IKC-MH.55 *Scientific Computing with Python* at October 27, 2023

Dr. Cem Özdoğan
Engineering Sciences Department
İzmir Kâtip Çelebi University

Contents

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 problem through equations; **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 approximation
  - this can usually be as accurate as needed (level of accuracy)
- Numerical Methods require repetitive arithmetic operations $\Rightarrow$ a computer to carry out
- Also, a human would make so many mistakes

Some disasters attributable to bad numerical computing

Have you been paying attention in your numerical analysis or scientific computation courses? 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***.

Floating-Point Arithmetic I

- 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).

Table: Floating$\rightarrow$ 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
- What about the sign of the exponent? Rather than use one of the bits for the sign of the exponent, exponents are biased.

Floating-Point Arithmetic II

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:** 5.87747E-39 (!)
- For **double** and **extended** precision the bias values are 1023 and 16383, respectively.
- $\frac{0}{0}, 0 * \infty, \sqrt{-1} \Longrightarrow NaN$: Undefined.

Floating-Point Arithmetic III

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

Precision	Length	Sign	Mantissa	Exponent	Range
	Number of bits in				
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.

week3_HandsOn.py

```
import sys
print(sys.float_info)
# sys.float_info(max=1.7976931348623157e+308, max_exp=1024, max_10_exp=308, min=2.2250738585072014e-308, min_exp=-1021,
# min_10_exp=-307, dig=15, mant_dig=53, epsilon=2.220446049250313e-16, radix=2, rounds=1)
print(sys.float_info.max)
# 1.7976931348623157e+308
print("%10.6e " % 2**128)
# 3.402824e+38
print("%10.6e " % 2**1023)
# 8.988466e+307f
```

Floating-Point Arithmetic IV

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!

- *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.

```
import sys
print(sys.float_info.epsilon)
# 2.220446049250313e-16
eps=sys.float_info.epsilon
print(1+eps*0.5)
# 1.0
print(1+eps*0.5)
# 1.0
print((1+eps*0.5)+eps*0.5)
# 1.0
print(1+eps*0.6)
# 1.0000000000000002
```

Computer Number Representation I

Say we have six bit representation (not single, double) (see the Fig.)

- 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

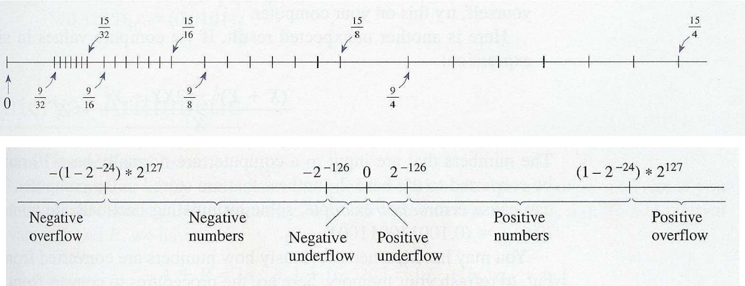

Figure: Upper: number line in the hypothetical system, Lower: IEEE standard.

- 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 lower Fig.)

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 **Round-off Error**
- ii **Truncation Error**
- iii **Propagated Error**

Kinds of Errors in Numerical Procedures II

- **Round-off Error**:

```
#!/usr/bin/python3
x=(4/3)*3; print(x)
# 4.0
a=4/3; print(a) # store double precision approx of 4/3
# 1.3333333333333333
b=a-1; print(b) # remove most significant digit
# 0.33333333333333326
c=1-3*b; print(c) # 3*b=1 in exact math
# 2.220446049250313e-16 # should be 0!!
```

```
from math import *
# import numpy as np
# kfirst=1.0; klast=360.0; kincrement=0.1
# for j in np.arange(kfirst, klast + kincrement, kincrement):
for j in range(1,360): # In degrees (1-360) as int. increment
    jj=j*(2*pi/360) # Conversion to radian
    a=cos(jj) # Return the cosine of jj (measured in radians)
    b=sin(jj) # Return the cosine of jj (measured in radians)
    z=a-(a/b)*b # Expected as being 0 !!
    print(j,jj,z)
352 6.14355896702004 0.0
353 6.161012259539984 1.1102230246251565e-16
354 6.178465552059927 1.1102230246251565e-16
355 6.19591884457987 1.1102230246251565e-16
356 6.213372137099813 0.0
357 6.230825429619756 0.0
358 6.2482787221397 0.0
359 6.265732014659643 0.0
```

Kinds of Errors in Numerical Procedures III

```
summation=1.0
for i in range(10000):  # Adding 0.00001 to 1.0 as 10000 times
    summation=summation+0.00001
print('summation = ', summation)
# summation =  1.1000000000006551 # Expected result is just 1.1 !!
print("summation = %f" % summation)
# summation = 1.100000 # Now expected result??
```

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.

Kinds of Errors in Numerical Procedures IV

```python
import numpy as np
x=np.tan(np.pi/6);  print(x)
# 0.5773502691896257
y=np.sin(np.pi/6)/np.cos(np.pi/6); print(y)
# 0.5773502691896256
if x==y:
    print("x and y are equal ")
else:
    print("x and y are not equal : x−y=%e " % (x−y))
# 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?"

Kinds of Errors in Numerical Procedures IV

ii **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.

Kinds of Errors in Numerical Procedures V

Evaluating the Series for sin(x) (**Example py-file:** sinser.py)

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

- An efficient implementation of the series uses recursion to avoid overflow in the evaluation of individual terms. If T_k is the k^{th} term ($k = 1, 3, 5, \ldots$) then

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

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

Kinds of Errors in Numerical Procedures VI

```python
import numpy as np
def sinser(x,tol,n):
    term = x
    ssum = term #  Initialize series
    print("Series approximation to sin(%f) \n  k       term          ssum" % (x*360/(2*np.pi)))
    print("  1   %11.3e   %20.16f  " % (term,ssum))
    for k in range(3, 2*n-1, 2):
        term = -term * x*x/(k*(k-1)) #  Next term in the series
        ssum = ssum + term
        print("%3d   %11.3e   %30.26f  " % (k,term,ssum))
        if abs(term/ssum)<tol:
            break  # True at convergence
    print("Truncation error after %d terms is %g " %  ((k+1)/2,abs(ssum-np.sin(x))))
sinser(np.pi/6,5e-9,10)
print("sin(%f)=%f with numpy library " % (np.pi/6*360/(2*np.pi),np.sin(np.pi/6)))
import math
print("sin(%f)=%f with math libarary" % (math.pi/6*360/(2*math.pi),math.sin(math.pi/6)))
```

```
Series approximation to sin(30.000000)
  k       term          ssum
  1    5.236e-01    0.5235987755982988
  3   -2.392e-02    0.49967417939436375995398976
  5    3.280e-04    0.50000213258879244726529123
  7   -2.141e-06    0.49999999186902321923753334
  9    8.151e-09    0.50000000002027988887931542
 11   -2.032e-11    0.49999999999996430632975830
Truncation error after 6 terms is 3.56382e-14
sin(30.000000)=0.500000 with numpy library
sin(30.000000)=0.500000 with math libarary
```

Figure: Output of sinser.py

Kinds of Errors in Numerical Procedures VII

Derivative of Sine function. Truncation & Round-off Errors (**Example py-file:** trunroun.py)

n	Step Length	Derivative	Error
1	0.10000000000000	0.4973637525	0.0429385533
2	0.01000000000000	0.5360859810	0.0042163249
3	0.00100000000000	0.5398814804	0.0004208255
4	0.00010000000000	0.5402602314	0.0000420744
5	0.00001000000000	0.5402980985	0.0000042074
6	0.00000100000000	0.5403018851	0.0000004207
7	0.00000010000000	0.5403022640	0.0000000418
8	0.00000001000000	0.5403023029	0.0000000030
9	0.00000000100000	0.5403023584	-0.0000000525
10	0.00000000010000	0.5403022474	0.0000000585
11	0.00000000001000	0.5403011372	0.0000011687
12	0.00000000000100	0.5403455461	-0.0000432402
13	0.00000000000010	0.5395683900	0.0007339159
14	0.00000000000001	0.5440092821	-0.0037069762

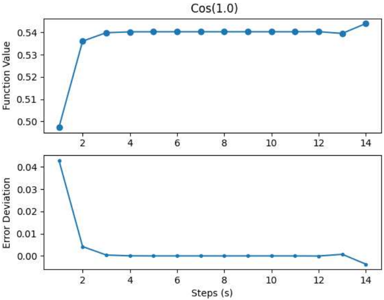

Figure: Output and Plot of trunroun.py

Kinds of Errors in Numerical Procedures VIII

iii **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

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

```python
def newtsqrt(x,delta,maxit):
    r = x/2;  rold = x # Initialize , make sure convergence test fails on first try
    it = 0
    while (r!=rold) and (it<maxit): # Convergence test
    # while ((r−rold) > delta) and (it<maxit): # Convergence test
    # while (abs(r−rold) > delta) and (it<maxit): # Convergence test
    # while (abs((r−rold)/rold) > delta) and (it<maxit): # Convergence test
        rold = r # Save old value for next convergence test
        r = 0.5*(rold + x/rold) # Update the guess
        it = it + 1
    return r
# Test the newtsqrt function for a range of inputs
xtest = [4, 0.04, 4e−4, 4e−6, 4e−8, 4e−10, 4e−12] # arguments to test
print(" Absolute Convergence Criterion")
print("    x          sqrt(x)   newtsqrt(x)    error       relerr")
import math
for x in  xtest: # repeat for each elementin xtest
    r = math.sqrt(x)
    rn = newtsqrt(x,5e−9,25)
    err = abs(rn − r)
    relerr = err/r
    print("%10.3e  %10.3e  %10.3e  %10.3e  %10.3e" % (x,r,rn,err,relerr))
```

```
Absolute Convergence Criterion
   x         sqrt(x)   newtsqrt(x)    error        relerr
4.000e+00   2.000e+00   2.000e+00   0.000e+00   0.000e+00
4.000e-02   2.000e-01   2.000e-01   0.000e+00   0.000e+00
4.000e-04   2.000e-02   2.000e-02   0.000e+00   0.000e+00
4.000e-06   2.000e-03   2.000e-03   0.000e+00   0.000e+00
4.000e-08   2.000e-04   2.000e-04   2.711e-20   1.355e-16
4.000e-10   2.000e-05   2.000e-05   3.388e-21   1.694e-16
4.000e-12   2.000e-06   2.000e-06   0.000e+00   0.000e+00
```

Figure: Output of newtsqrt.py

Newton's method to compute the square root of a number.

Example py-file: newtsqrt.py