Lecture 2 Errors

December 9, 2024

This week's lecture topic: numerical errors (from textbook, chapter 4)

In lab, we'll also do some Linux command-line basics

1 Numerical errors

Aside from errors in programming or discretizing the physical model, there are 2 types of errors in computations: 1. Rounding errors: errors in how the computer stores or manipulates numbers. 2. Approximation errors (sometimes called truncation errors): errors in approximations to various functions or methods.

Reason: computers are machines, with inherent limitations.

1.1 Integers and floats

How does a computer represent a number?

- Variables `a`, `b` and `c` below are all equal to 1000.
- Yet, `a` is treated very differently by da Python than `b` and `c`:

[15]:
```
a = 1000
b = 1000.
c = 1e3
print('The type of a is', type(a))
print('The type of a is', type(b))
print('The type of a is', type(c))
```

```
The type of a is <class 'int'>
The type of a is <class 'float'>
The type of a is <class 'float'>
```

1.2 Error #1: rounding errors

1.2.1 General principle

- Integer numbers: python does not limit the number of digits stored. Can cause problems!

[13]:
```
from numpy import iinfo, int32, int64
#iinfo(int32).max
print(2**1000)
```

```
1071508607186267320948425049060001810561404811705533607443750388370351051124936l
2249319837881569585812759467291755314682518714528569231404359845775746985748039З
45677748242309854210746050623711418779541821530464749835819412673987675591655439
46077062914571196477686542167660429831652624386837205668069376
```

- Non-integer numbers with more than 16 significant figures: **rounding** after 16 figures.
- Each mathematical operation (FLOP, Floating-point Operation) introduces errors in the 16th digit. You can't assume

  `1.3 + 2.4 = 3.7,`

  it might be

  `3.6999999999999999`

  even though the 2 numbers you are adding only have 2 significant figures.
- Know this, accept it, work with it. You are better at adding 1.3 and 2.4 than your computer, but you are much slower. This is why we use computers.

There are also limitations as to the largest number representable in Python, the closest to zero, etc. If you want to know what they are on your machine, you can run the code below:

```
[14]: from numpy import finfo, float64, float32
      # float64 contains double-precision floats, float32 is single-precision
      print("attributes you can access in finfo(float64) ", dir(finfo(float64)))
```

```
attributes you can access in finfo(float64)  ['__class__', '__delattr__',
'__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__',
'__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', '__weakref__', '_finfo_cache', '_init', '_machar',
'_str_eps', '_str_epsneg', '_str_max', '_str_resolution',
'_str_smallest_normal', '_str_smallest_subnormal', '_str_tiny', 'bits', 'dtype',
'eps', 'epsneg', 'iexp', 'machar', 'machep', 'max', 'maxexp', 'min', 'minexp',
'negep', 'nexp', 'nmant', 'precision', 'resolution', 'smallest_normal',
'smallest_subnormal', 'tiny']
```

```
[28]: print( "maximum numbers in 64 bit and 32 bit precision: ",
            finfo(float64).max, finfo(float32).max)
```

```
maximum numbers in 64 bit and 32 bit precision:  1.7976931348623157e+308
3.4028235e+38
```

```
[29]: print( "minimum numbers in 64 bit and 32 bit precision: ",
            finfo(float64).min, finfo(float32).min)
```

```
minimum numbers in 64 bit and 32 bit precision:  -1.7976931348623157e+308
-3.4028235e+38
```

[30]:
```
print( "epsilon for 64 bit and 32 bit: ",
      finfo(float64).eps, finfo(float32).eps)
```

```
epsilon for 64 bit and 32 bit:  2.220446049250313e-16 1.1920929e-07
```

[15]:
```
print( "Should be epsilon for this machine if it's 64 bit:",
      float64(1)+finfo(float64).eps-float64(1))
```

```
Should be epsilon for this machine if it's 64 bit: 2.220446049250313e-16
```

This tells us that the error made on a given evaluation (here `float64(1)`) is always the same for a given software and machine.

It may vary across software and hardware configurations.

[16]:
```
print( "Should be zero, and it is",
      float64(1)+finfo(float64).eps/2.0-float64(1))
```

```
Should be zero, and it is 0.0
```

Here, adding 1/2-of the roundoff error to one didn't register, it was too small of Python to "notice".

1.2.2 Simple examples

7/3 - 4/3 - 1 = 0 , right? Well...

[17]:
```
7./3. - 4./3 - 1.
```

[17]:
```
2.220446049250313e-16
```

And how about this one: under what circumstances is the following possible?

$$(x + y) + z \neq x + (y + z)$$

[7]:
```
x = 1.e16
y = -1.e16
z = 1.

print("(x+y)+z = {}".format((x+y)+z))
print("x+(y+z) = {}".format(x+(y+z)))
```

```
(x+y)+z = 1.0
x+(y+z) = 0.0
```

1.2.3 Error constant

- Newman defines $\sigma = C|x|$.
  - x = number you want to represent.
  - σ = standard deviation of error
  - C = fractional error for a single floating point number

- For 64 bit float: $C = O(10^{-16}) \sim$ machine precision ϵ_M.
- We cannot know a number better than this on the computer (otherwise it wouldn't be a limit on the precision).
- This fractional error is different on different computers but should not depend on x.

Note: **Fractional error** of a calculated value is not the same as **relative error** of a calculated value compared to a true value. Relative error of calculated value x compared to true value y is $(x-y)/y$.

1.2.4 Propagation of errors

Errors propagate statistically like they do in experimental physics.

These errors in differences become **very important** when taking numerical derivatives:

$$\frac{df}{dt} \approx \frac{f_{i+1} - f_i}{\Delta t} = \text{danger zone}$$

as we will see later.

1.2.5 One important rule

Never, ever. Never ever ever. Do something like this:

[43]:
```python
if 7./3. - 4./3. - 1. == 0:
    print('7/3 - 4/3 - 1 == 0')
else:
    print('7/3 - 4/3 - 1 != 0')
```

```
7/3 - 4/3 - 1 != 0
```

Instead:

[44]:
```python
delta = 1e-15
if abs(7./3. - 4./3. - 1.) < delta:
    print('7/3 - 4/3 - 1 == 0 (or close enough anyway...)')
else:
    print('7/3 - 4/3 - 1 != 0')
```

```
7/3 - 4/3 - 1 == 0 (or close enough anyway…)
```

1.3 Error #2: Approximation errors

- errors introduced in functions due to approximations
- very important to consider for **integration** & **differentiation** algorithms
- we approximate these operations and there is an error in that approximation
- Most integration/differentiation algorithms are somehow based on Taylor series expansions, so you can usually figure out the approximation error by looking at the terms in the Taylor expansion that you ignore.

This should become clearer once we illustrate it with numerical integrations. We'll also do a simple illustration in the informal lab exercises.

```
[ ]:
```