SimMethEx1

Tobias Jaeger, Gregor Neumann, Sophia Milanov

October 27, 2016

1 Machine Epsilon

(a) float

```
    #include <iostream>

int main(){

    float a = 1;

    float e = 1;

    float h = 1.1;

    while ((a + e) > a){
        std::cout<< (e) << std::endl;
        e = e/h;
    }
```

(b) double

```
    double b = 1;

    double eb = 1;

    double hb = 1.1;

    while ((b + eb) > b){
        std::cout<< (eb) << std::endl;
        eb = eb/hb;
    }
```

data type	precision	$1.0 + \epsilon$
float	5.7053e-08	1.000000063
double	1.05299e-16	1.00000000000000022
long double	5.14054e-20	1.000000000000000000011

Table 1: Results

(c) long double

```
    long double c = 1;

    long double ec = 1;

    long double hc = 1.1;

    while ((c + ec) > c){
        std::cout<< (ec) << std::endl;
        ec = ec/hc;
    }



    return 0;
}
```

2 Pitfalls of floating point arithmetic

```
double a = 1.0e17;
double b = -1.0e17;
double c = 1.0;
double x = (a + b) + c;
double y = a + (b + c);


cout << "x = " << x << ", " << "y = " << y << endl;
```

Result in the console: x = 1, y = 0
The precision for $1.0e17$ is lower then for smaller numbers. In this case the precision is too small to take the 1.0 in to account.

3 Pitfalls of floating point representation

(a)

```
float xx = 0.01;
double yy = xx;
double zz = 0.01;

 cout << "x = " << xx << ", " << "y = " << yy << ", z = " << zz << endl;

int i = xx * 10000;
int j = yy * 10000;
int k = zz * 10000;

printf("%d %d %d\n", i, j, k);
```

Result in the console: 100, 99, 1000

The dissenting value for (int j) results from the convert of the (float xx) into a (double yy). One might think a conversion from a lower to a higher precision number will not worsen the accuarcy of the value, however it turns out to be differnet.

- float $a = 0.1 \quad \rightarrow$ 0 01111000 01000111101011100001010
- double $b = 0.1 \rightarrow$ 0 01111111000 0100011110101110000101000111101011100001010001111011
- double $c = a \quad \rightarrow$ 0 01111111000 0100011110101110000101000000000000000000000000000000

By comparing b and c one notice that the better representation in b got additionally more ones in the mantissa.

(b)

float $x = 0.01$, so it has a 32 bit representation (1 bit sign S, 8 bit exponent E, 23 bit mantissa M).

$x = 0 \quad 01111000 \quad 01000111101011100001010$

$\rightarrow S = 0, \quad E = 120, \quad M = 2348810$

$$
\begin{aligned}
x &= (-1)^S 2^{E-b}\left(1+\frac{M}{2^p}\right) && (1)\\
&= 2^{-7}\left(1+\frac{2348810}{2^{23}}\right) && (2)\\
&= \frac{2^{23}+2348810}{2^{30}} = \frac{10737418}{1073741824} = \frac{5368709}{536870912} && (3)
\end{aligned}
$$

where p = 23 and b = 127

4 Packing of numbers

Lösungsvorschlag 1

Single precision (float):
1.0: $1.00000000000000000000000 * 2^0$
2.0: $1.00000000000000000000000 * 2^1$
$\Rightarrow 2^{23} - 1$ numbers are in the interval of 1.0 and 2.0
256.0: $1.00000000000000000000000 * 2^8$
255.0: $1.11111111000000000000000 * 2^7$
$\Rightarrow 2^{23-8} - 1 = 2^{15} - 1 = 32767$ numbers are in the interval of 255.0 and 256.0
Double precision:
$\Rightarrow 2^{52} - 1$ numbers are in the interval of 1.0 and 2.0
$\Rightarrow 2^{52-8} - 1 = 2^{44} - 1$ numbers are in the interval of 255.0 and 256.0

Lösungsvorschlag 2

```
int main(){

    float x = 1;

    float e = 1;

    float h = 1.1;

    while ((x + e) > x){
        e = e/h;

    }
    std::cout<< (e) << std::endl;


    float y = 255;

    float e = 1;

    float h = 1.1;

    while ((y + e) > y){
        e = e/h;
    }
    std::cout<< (e) << std::endl;
```

This code returns the minimal resolution for float x = 1 and float y = 255, which is

- $res(x) \sim 10^{-8}$
- $res(y) \sim 10^{-6}$

Therefor the intervall between 1 &2 contains $\sim 10^8$ and the intevall between 255 & 256 $\sim 10^6$ numbers.

For double precision the code is equivalent and returns:

- $res(x) \sim 10^{-16}$
- $res(y) \sim 10^{-14}$

Therefor the intervall between 1 &2 contains $\sim 10^16$ and the intevall between 255 & 256 $\sim 10^14$ numbers.

Hier `https://www.h-schmidt.net/FloatConverter/IEEE754.html` ist so ein IEEE 754 converter. Dort wird die Zahl als flaot auch immer nur auf ca die oben angegebene Genauigkeit angegeben und mit entsprechend mehr Kommastellen als double, daher denke ich das Programm rechnet richtig...

5 Summing a long list of numbers

Results from Python:

Sum beginning to end: -1.27636535808e+89
Sum end to beginning: 1.60798539786e-190
Sorted sum most neg to most pos: nan
Sorted sum (absolute values): nan

was machen wir bei der 4?, Erklärung bei der 3 a!

Exercise_1.5

October 26, 2016

In [1]:
```python
import numpy as np
```

In [2]:
```python
f = open('numbers.dat', 'rb') # opens numbers file
data = np.fromfile(f, dtype = float) # read binary data out of file
data128 = np.fromfile(f, dtype = np.float128) # now with long double data type
```

In [3]:
```python
len(data) #make sure it's a million numbers
```

Out[3]: 1000000

In [4]:
```python
### Part a, sum from beginning to end ###
sumup = 0.
for i in range(len(data)):
    sumup =+ data[i]
print('Sum (a) = ', sumup)
```

```
Sum (a) =  -1.27636535808e+89
```

In [5]:
```python
### Part b, reversed direction of summing ###
sumdown = 0.
for i in reversed(range(len(data))):
    sumdown =+ data[i]
print('Sum (b) = ', sumdown)
```

```
Sum (b) =  1.60798539786e-190
```

In [6]:
```python
### Part c, sort by magnitude (does that mean most negative or the smallest absolute value?) an

# Sort data
datasort = np.sort(data) # sort from most negative to most positive
datasortabs = np.sort(np.abs(data)) # sort from smallest absolute to highest absolute

# Check data types
print('Datasort type: ',datasort.dtype, '\nDatasort absolute data type: ', datasortabs.dtype)

# Sum
sumsortabs = 0.
sumsort = 0.
for i in range(len(datasort)):
    sumsort =+ datasort[i]
    sumsortabs =+ datasortabs[i]
print('Sum (c) =', sumsort, '\nSum (c) absolute =', sumsortabs)
```

```
Datasort type:  float64
Datasort absolute data type:  float64
Sum (c) = nan
Sum (c) absolute = nan
```

```
In [7]: print(np.sum(np.isnan(data)))
```

```
526
```

```
/home/sophia/anaconda3/lib/python3.4/site-packages/IPython/kernel/__main__.py:1: RuntimeWarning: invalid
  if __name__ == '__main__':
```

There are 526 nan values. In both sorted arrays the nan values are at the end of the array. Operations with these values will again give nan results. Therefore both sorted sums are nan. The nan values could be below the double precision.

```
In [8]: print(np.min(data[np.isnan(data) ==False]), np.max(data[np.isnan(data) == False]))
```

```
-1.78701556856e+308 1.79005606756e+308
```

```
/home/sophia/anaconda3/lib/python3.4/site-packages/IPython/kernel/__main__.py:1: RuntimeWarning: invalid
  if __name__ == '__main__':
```

The smallest and biggest numbers which are not nan are near to the smallest representable numbers in double precision. A higher precision could avoid the nan values.

```
In [9]: ### Part d, as c only with long double data ###

        # Sort data
        datasort128 = np.sort(data128) # sort from most negative to most positive
        datasortabs128 = np.sort(np.abs(data128)) # sort from smallest absolute to highest absolute

        # Check data types
        print('Datasort type: ',datasort128.dtype, '\nDatasort absolute data type: ', datasortabs128.dty

        # Sum
        sumsortabs128 = 0.
        sumsort128 = 0.
        for i in range(len(datasort128)):
            sumsort128 =+ datasort128[i]
            sumsortabs128 =+ datasortabs128[i]
        print('Sum (d) =', sumsort128, '\nSum (d) absolute =', sumsortabs128)
```

```
Datasort type:  float128
Datasort absolute data type:  float128
Sum (d) = 0.0
Sum (d) absolute = 0.0
```

```
In [10]: print('Sum beginning to end: ', sumup, '\nSum end to beginning: ', \
               sumdown, '\nSorted sum most neg to most pos: ',sumsort, '\nSorted sum (absolute values):
              '\nSorted sum long double: ', sumsort128, '\nSorted sum long double (abs values): ', sums
```

```
Sum beginning to end:  -1.27636535808e+89
Sum end to beginning:  1.60798539786e-190
Sorted sum most neg to most pos:  nan
Sorted sum (absolute values):  nan
Sorted sum long double:  0.0
Sorted sum long double (abs values):  0.0
```