Humboldt University of Berlin

Einführung in das wissenschaftliche Rechnen

Introduction to Floating Point Arithmetic via Python

Christian Parpart & Kei Thoma

May 31, 2019

Contents

1 Introduction

In this document, we first present fundamental definitions and theoretical results about floating point arithmetic and the relative machine precision. We have also provided a Python program which verifies these theoretical results and offers the user to also verify our findings with different numbers.

2 Theoretical Results

2.1 Fundamental Definitions and Lemmas

Definition 1. Let z be an integer in the decimal system. To convert z to the *binary system*, we have

$$z := d_{n-2}d_{n-3}\dots d_1 d_2 = \sum_{i=0}^{n-2} d_i 2^i$$

where $d_i \in \{0, 1\}$ are digits. [1]

Lemma 1. Let $\beta \in \mathbb{N}, \beta \geq 2$ and $x \in \mathbb{R}$ with $x \neq 0$. Then there is one and only one representation for x in the form of

$$x = (-1)^\nu \beta^N \sum_{i=1}^{\infty} x_i \beta^{-i}$$

where $\nu \in \{0, 1\}$; $N \in \mathbb{Z}$; $x_1 = 1$ and $x_i \in \{0, 1, \dots, \beta - 1\}$; and for every $n \in \mathbb{N}$ exists an index $i \geq n$ with $x_i \neq \beta - 1$. [1]

Definition 2. x is a normalized t-digit long floating point number infty

$$x = (-1)^\nu 2^N \sum_{i=1}^{t} x_i 2^{-i} = (-1)^\nu 2^N \cdot (0.x_1 x_2 \dots x_t)_2$$

with $\nu \in 0, 1$; $N_{\min} \leq N \leq N_{\max}$; $N \in \mathbb{Z}$; $x_i \in \{0, 1\}$ for all $i = 2, \dots, t$ and $x_1 = 1$.

The number $m = \sum_{i=1}^{t} x_i 2^{-i} = (0.x_1 x_2 \dots x_t)_2$ is called mantissa of x and t is the mantissa length. [1]

We will see practical examples to convert decimal numbers to binary and back in section 5.

Remark 1. There are special values reserved on the computer. These are $+\infty$, $-\infty$ and NaN (not a number). [1]

Definition 3. Let t be the mantissa length. We define the rounding of x to a floating point as follows.
If $N_{\min} \leq N \leq N_{\max}$, then

$$\mathrm{rd}_t(x) := \begin{cases} (-1)^\nu 2^N \sum_{i=1}^{t} x_i 2^{-i} \text{ for } x_{t+1} = 0 \\ (-1)^\nu 2^N (\sum_{i=1}^{t} x_i 2^{-i} + 2^{-t}) \text{ for } x_{t+1} = 1 \end{cases}$$

If $N \leq N_{\min} - t$, then $\mathrm{rd}_t(x) := 0$.
If $N_{\min} - t < N \leq N_{\min}$, then

$$\mathrm{rd}_t(x) := \begin{cases} (-1)^\nu 2^{N_{\min}} \sum_{j=n+1}^{t} x_{j-n} 2^{-j} \text{ for } x_{t+1-n} = 0 \\ (-1)^\nu 2^{N_{\min}} (\sum_{j=n+1}^{t} x_{j-n} 2^{-i} + 2^{-t}) \text{ for } x_{t+1-n} = 1 \end{cases}$$

If $|x| > x_{\min}$, then we get an overflow and in most cases we continue with ∞. [1]

Lemma 2. For absolute and relative error between a real number and its floating point representation we have the margin [1]

$$e_{\text{abs}} = |\mathrm{rd}_t(x) - x| \leq 2^{N-t-1}$$
$$e_{\text{rel}} = \left| \frac{\mathrm{rd}_t(x) - x}{\mathrm{rd}_t(x)} \right| \leq 2^{-t}$$

Definition 4.

$$\tau := 2^{-t} \geq \max \left\{ \left| \frac{\mathrm{rd}_t(x) - x}{x} \right|, \left| \frac{\mathrm{rd}_t(x) - x}{\mathrm{rd}_t(x)} \right| \right\}$$

is called the relative machine precision. [1]

Theorem 1. The relative machine precision can be computed with the algorithm illustrated in figure 1.

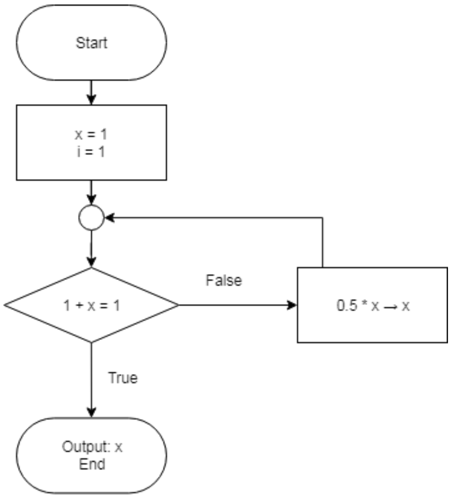


Figure 1: algorithm to find the relative machine precision

Proof. By definition $\tau = \frac{1}{2^{-t}}$. Consider $\mathrm{rd}_t(\frac{1}{2^{t+1}})$. This is by definition rounded to 0, but $\mathrm{rd}_t(\frac{1}{2^t})$ is not. Therefore, let x_p be the output of the algorithm. Then, $p = t$ and we have $x_p = \tau$. □

Theorem 2. Using a computer, the relative machine precision can be computed in the following manner

$$\tau = \left|\frac{7}{3} - \frac{3}{4} - 1\right|$$

Proof. We will first evaluate $\frac{7}{3} - \frac{4}{3}$. We have

$$\mathrm{rd}_t(\frac{7}{3}) = \mathrm{rd}_t((10.\overline{01})_2) = \mathrm{rd}_t((0.10\overline{01})_2 \times 2^2) \tag{1}$$

$$\mathrm{rd}_t(\frac{4}{3}) = \mathrm{rd}_t((1.\overline{01})_2) = \mathrm{rd}_t((0.1\overline{01})_2 \times 2^1) \tag{2}$$

The decimal places of the two numbers only differ in placing. Therefore, if we wound the two numbers above one will be rounded up and the other will be rounded down, and we have

$$\left|\mathrm{rd}_t(\frac{7}{3}) - \mathrm{rd}_t(\frac{4}{3})\right| = 2^2 \cdot \sum_{i=1}^{t} \frac{1}{2} - \frac{1}{4} + 0 + \cdots + 0 + \frac{1}{2^t} = 1 + \frac{1}{2^t}$$

If we subtract 1 from the last term, we get $\tau = \frac{1}{2^t}$ as desired. □

2.2 Examples

For all following examples, let the mantissa length be $t = 8$ and the exponent of the floating point arithmetic be bounded by $N_{\text{min}} = -5$ and $N_{\text{max}} = 8$.

Example 1. Given the context as defined above, the largest number that can be represented is $x_{\text{max}} = 255$. The calculation is fairly simple, choose the largest exponent possible and fill every digit of the mantissa with ones. In binary, this would be

$$x_{\text{max}} = (0.11111111)_2 \times 2^8 = (11111111)_2,$$

or in decimal

$$\begin{aligned} x_{\text{max}} &= (-1)^\nu \cdot 2^N \cdot \sum_{i=1}^{t} x_i \beta^{-i} \\ &= 2^8 \cdot \left(\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \frac{1}{32} + \frac{1}{64} + \frac{1}{128} + \frac{1}{256}\right) \\ &= 255. \end{aligned}$$

Example 2. To find the smallest possible normalized positive value in the defined floating point arithmetic, we proceed similary to the example 1. Set the exponent as small as possible and fill the mantissa with zeros but the first place. We have in binary

$$x_{\text{norm. min}} = (0.10000000)_2 \times 2^{-4}$$

which in decimal this translates to

$$x_{\text{norm. min}} = (-1)^\nu \cdot 2^N \cdot \sum_{i=1}^{t} x_i \beta^{-i} = 2^{-4} \cdot \frac{1}{2} = \frac{1}{32} = 0.03125$$

Example 3. If we do not require the value to be normalized, the smallest possible positive value is much smaller. To find $x_{\min}$, we again set N to -4 and fill the mantissa with 0 except for the last place. We have in binary representation

$$x_{\min} = (0.00000001)_2 \times 2^{-4}$$

and decimal would be

$$x_{\min} = (-1)^{\nu} \cdot 2^N \cdot \sum_{i=1}^{t} x_i \beta^{-i} = 2^{-4} \times \frac{1}{256} = \frac{1}{4096} = 0.000244140625$$

Example 4. We want to find the margin for the absolute and the relative error. To find the largest possible absolute error, set the exponent to the maximum value and consider two neighboring floating point numbers such as $(1.00000000)_2 \times 2^8$ and $(1.00000001)_2 \times 2^8 = 1$. Worst case scenario, the given number is right in the middle of these two numbers; therefore, the maximum absolute error is $(0.00000001)_2 \times 2^7 = \frac{1}{2}$. This result is also verified by the lemma 2. The same lemma gives us the boundaries for the relative error, 2^{-8}. To conclude, we have

$$0 \leq e_{\text{abs}} \leq \frac{1}{2}$$
$$0 \leq e_{\text{rel}} \leq \frac{1}{256}$$

Example 5. Let $z_1 = 67.0$. We want to find the normalized binary form of this integer with ten decimal place accuracy. According to lemma 1, we have

$$\begin{aligned}
67.0 \div 2 &= 33.0 + 1 \\
33.0 \div 2 &= 16.0 + 1 \\
16.0 \div 2 &= 8.0 + 0 \\
8.0 \div 2 &= 4.0 + 0 \\
4.0 \div 2 &= 2.0 + 0 \\
2.0 \div 2 &= 1.0 + 0 \\
1.0 \div 2 &= 0.0 + 1.
\end{aligned}$$

Reading the reminders on the left from bottom to top yields $z_1 = 67.0 = (1000011)_2$. To normalize this number, we move the decimal point seven digits to the left. Since z_1 only has seven digits, we do not need to cut off any digits. We have

$$z_1 = 67.0 = (0.1000011)_2 \times 2^7$$

If one wants to check the validity of the conversion from decimal to binary above, we can check the solution by applying the formula from the other way.

$$(-1)^{\nu} \cdot 2^N \cdot \sum_{i=1}^{t} x_i \beta^{-i} = 2^7 \cdot \left(\frac{1}{2} + \frac{1}{64} + \frac{1}{128} \right) = 128 \cdot \frac{67}{128} = 67$$

Now, let's consider the floating point number of 67.0. $N = 7$ is between $N_{\min} = -5$ and $N_{\max} = 8$, also 67.0 has 7 digits in binary form; therefore, there is no rounding to do which means that 67.0 can be represented with the given floating point arithmetic without loss of precision.

$$\mathrm{rd}_8(z_1) = (0.1000011)_2 \times 2^7$$

Since there is no loss of precision, one can easily conclude that the absolute and relative error of 67.0 and $\mathrm{rd}_8(67.0)$ is zero.

Example 6. Let $z_2 = 287.0$. To find the normalized binary form with ten decimal place accuracy, we have

$$\begin{aligned}
287.0 \div 2 &= 143.0 + 1\\
143.0 \div 2 &= 71.0 + 1\\
71.0 \div 2 &= 35.0 + 1\\
35.0 \div 2 &= 17.0 + 1\\
17.0 \div 2 &= 8.0 + 1\\
8.0 \div 2 &= 4.0 + 0\\
4.0 \div 2 &= 2.0 + 0\\
2.0 \div 2 &= 1.0 + 0\\
1.0 \div 2 &= 0.0 + 1,
\end{aligned}$$

therefore, $z_2 = 287.0 = (100011111)_2$. Again, there is no need to round any digits. Its normalized binary form is

$$z_2 = 287.0 = (0.100011111)_2 \times 2^9$$

In this example, we have an exponent $N = 9$ which is greater than $N_{\max} = 8$. This means that with the given floating point arithmetic, we have an overflow and 287.0 cannot be sensibly rounded to a floating point number (instead, computing with the given arithmetic, z_2 is the same as $+\infty$). In example 1, we showed that $x_{\max} = 255$ which is another reason for a overflow. According to IEEE 754 standard, both absolute and relative error are also infinity for 287.0.

Example 7. For a non-integer example, let $z_3 = 10.625$. To find the binary form of this number, we first separate $z_3 = 10.0 + 0.625$ and apply the algorithm of 1 on each summand. For 10.0 we have

$$\begin{aligned}
10.0 \div 2 &= 5.0 + 0\\
5.0 \div 2 &= 2.0 + 1\\
2.0 \div 2 &= 1.0 + 0\\
1.0 \div 2 &= 0.0 + 1
\end{aligned}$$

and for 0.625 we will multiply it with 2 until we get 0

$$\begin{aligned} 0.625 \times 2 &= 0.25 + 1 \\ 0.25 \times 2 &= 0.5 + 0 \\ 0.5 \times 2 &= 0.0 + 1 \end{aligned}$$

Combining both results together, we get $z_3 = (1010.101)_2$. To normalize, we move the decimal place four digits to the left and we have

$$z_3 = 10.625 = (0.1010101 \times 2^4)_2.$$

Again we see that the exponent is between $N_{\min} = -5$ and $N_{\min} = 8$. The mantissa is also short enough; therefore, z_3 is already a floating point number and we have

$$\mathrm{rd}_8(z_3) = (0.1010101)_2 \times 2^4.$$

Needless to say, the absolute and relative errors are both zero.

Example 8. Perhaps a more interesting example is needed. Let $z_4 = 1.01$. As we did in 7, we will separate z_4 in two parts; however, we immediately see that 1 is 1 in both decimal and binary system. We will therefore consider 0.01.

$$\begin{aligned} 0.01 \times 2 &= 0.02 + 0 \\ 0.02 \times 2 &= 0.04 + 0 \\ 0.04 \times 2 &= 0.08 + 0 \\ 0.08 \times 2 &= 0.16 + 0 \\ 0.16 \times 2 &= 0.32 + 0 \\ 0.32 \times 2 &= 0.64 + 0 \\ 1.28 \times 2 &= 0.28 + 1 \\ 0.28 \times 2 &= 0.56 + 0 \\ 0.56 \times 2 &= 0.12 + 1 \end{aligned}$$

We could go on, but since we only need to find the normalized binary form with respect to ten decimal places. We have

$$z_4 = 1.01 \approx (1.000000101)_2 \times 2^0$$

and in normalized form

$$z_4 = 1.01 \approx (0.1000000101)_2 \times 2^1$$

Using the formula from lemma 1, we have the floating point number

$$\mathrm{rd}_8(z_4) = (-1)^\nu \cdot 2^N \cdot \sum_{i=1}^{t} x_i \beta^{-i} = 2 \cdot \left(\frac{1}{2} + \frac{1}{256}\right) = \frac{129}{128} = 1.0078125.$$

For the absolute and relative error we have

$$\begin{aligned}
e_{\text{abs}} &= |rd_8(1.01) - 1.01| \\
&= 0.0021875 \\
e_{\text{rel}} &= \left|\frac{rd_8(1.01) - 1.01}{rd_8(1.01)}\right| \\
&= \frac{7}{3225} \approx 0.00217
\end{aligned}$$

Example 9. As we already fell into the rabit hole of numbers which have endlessly long binary forms, let's continue with $z_5 = 0.0002$. For this example, we must stay diligent and iterate many times over the algorithm.

$$\begin{aligned}
0.0002 \times 2 &= 0.0004 + 0 \\
0.0004 \times 2 &= 0.0008 + 0 \\
0.0008 \times 2 &= 0.0016 + 0 \\
0.0016 \times 2 &= 0.0032 + 0 \\
0.0032 \times 2 &= 0.0064 + 0 \\
0.0064 \times 2 &= 0.0128 + 0 \\
0.0128 \times 2 &= 0.0256 + 0 \\
0.0256 \times 2 &= 0.0512 + 0 \\
0.0512 \times 2 &= 0.1024 + 0 \\
0.1024 \times 2 &= 0.2048 + 0 \\
0.2048 \times 2 &= 0.4096 + 0 \\
0.4096 \times 2 &= 0.8192 + 0 \\
0.8192 \times 2 &= 0.6384 + 1
\end{aligned}$$

We got our first 1! Now we only have to find a maximum of 10 more digits.

$$\begin{aligned}
0.6384 \times 2 &= 0.2768 + 1 \\
0.2768 \times 2 &= 0.5536 + 0 \\
0.5536 \times 2 &= 0.1072 + 1 \\
0.1072 \times 2 &= 0.2144 + 0 \\
0.2144 \times 2 &= 0.4288 + 0 \\
0.4288 \times 2 &= 0.8576 + 0 \\
0.8576 \times 2 &= 0.7152 + 1 \\
0.7152 \times 2 &= 0.4304 + 1 \\
0.4304 \times 2 &= 0.8608 + 0 \\
0.8608 \times 2 &= 0.7216 + 1
\end{aligned}$$

Therefore, we have $z_5 = 0.0002 \approx (0.00000000000011010001101)_2$ and normalized we have

$$z_5 = 0.0002 \approx (0.1101000110)_2 \times 2^{-12}.$$

Following the formula from 3 we have with $n := N_{\min} - N = -5 + 12 = 7$

$$\begin{aligned} \mathrm{rd}_8(z_5) &= (-1)^\nu \cdot 2^{N_{\min}} \cdot \left(\sum_{j=n+1}^{t} x_{j-n} 2^{-j} + 2^{-t} \right) \\ &= 2^{-5} \cdot \left(\sum_{j=8}^{8} x_{j-7} \cdot 2^{-j} + 2^{-8} \right) \\ &= \frac{1}{4096} \approx 0.000244140625 \end{aligned}$$

since $x_{t+1-7} = x_2 = 1$. We have for the absolute and the relative error

$$\begin{aligned} e_{\text{abs}} &= \left| 0.0002 - \frac{1}{4096} \right| = \frac{113}{2560000} \approx 4.4140 \times 10^{-5} \\ e_{\text{rel}} &= \frac{\left|0.0002 - \frac{1}{4096}\right|}{\frac{1}{4096}} = \frac{113}{625} = 0.1808 \end{aligned}$$

Example 10. For the more mathematically minded, we have last but not least $z_6 = \frac{1}{3}$.

$$\begin{aligned} \frac{1}{3} \times 2 &= \frac{2}{3} + 0 \\ \frac{2}{3} \times 2 &= \frac{1}{3} + 1 \end{aligned}$$

We already see a patern here; further calculations are not needed. We simply have

$$z_6 = \frac{1}{3} \approx (0.1010101010)_2 \times 2^{-1})$$

To find the floating point we have

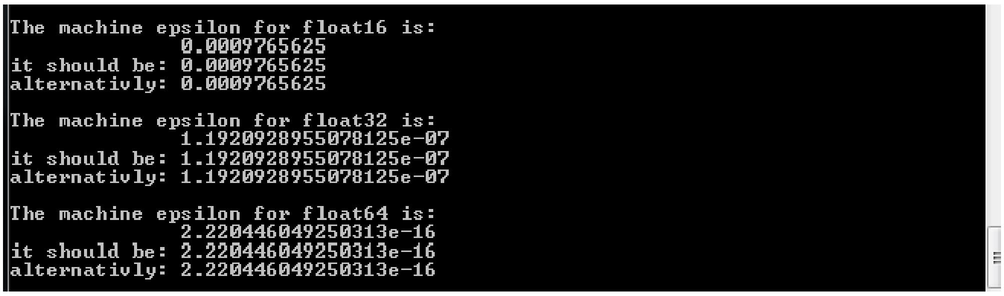
$$\begin{aligned} \mathrm{rd}_8(z_6) =& (-1)^\nu \cdot 2^N \cdot \sum_{i=1}^{t} x_i 2^{-i} + 2^t \\ =& 2^{-1} \cdot \left(\frac{1}{2} + \frac{1}{8} + \frac{1}{32} + \frac{1}{128} + \frac{1}{256} \right) = \frac{171}{512} = 0.333984375, \end{aligned}$$

and for its errors

$$\begin{aligned} e_{\text{abs}} &= \left| \frac{1}{3} - \frac{171}{512} \right| = \frac{1}{1536} \approx 0.0006510416 \\ e_{\text{rel}} &= \frac{\left|\frac{1}{3} - \frac{171}{512}\right|}{\frac{171}{512}} = \frac{1}{513} \approx 0.00194931. \end{aligned}$$

3 User Manual of ab4.py

The Python command line interface application ab4.py computes the solution of both sides of $\frac{1}{x} - \frac{1}{x+1} = \frac{1}{x(x+1)}$ depending on the value for x and the desired mantissa length of the floating point arithmetic.
After starting, ab4.py first prints out the machine precision for np.float16, np.float32 and np.float64 (see figure 4).



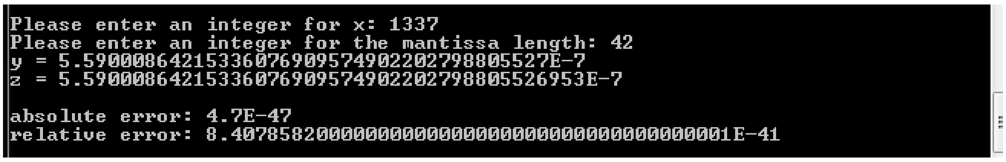
```
The machine epsilon for float16 is:
               0.0009765625
it should be: 0.0009765625
alternativly: 0.0009765625

The machine epsilon for float32 is:
               1.1920928955078125e-07
it should be: 1.1920928955078125e-07
alternativly: 1.1920928955078125e-07

The machine epsilon for float64 is:
               2.220446049250313e-16
it should be: 2.220446049250313e-16
alternativly: 2.220446049250313e-16
```

Figure 2: command line output of ab4.py regarding different floating types

Then, the user may input the values for x and the mantissa length. ab4.py will print the solution of both sides of the equation and the absolute and the relative error. Two corresponding plot for both errors with varied mantissa length (from 1 to 28) are also drawn. See figure 3 for an example input and output.



```
Please enter an integer for x: 1337
Please enter an integer for the mantissa length: 42
y = 5.590008642153360769095749022027988055527E-7
z = 5.590008642153360769095749022027988055526953E-7

absolute error: 4.7E-47
relative error: 8.4078582000000000000000000000000000000001E-41
```

Figure 3: example input with x = 1337 and mantissa length = 42 and its output

The user may exit the application by typing "exit" or leaving the line empty when asked to enter the value for x.

4 Documentation of tools4.py and ab4.py

4.1 tools4.py Library

4.1.1 Imports

`tools4.py` requires three libraries namely `decimal`, `numpy` and `matplotlib.pyplot`. `decimal` is used to control the mantissa length for a given floating point number; we import `numpy` for its float data types; and finally, we use `matplotlib.pyplot` to draw the plot for the absolute and relative error.

4.1.2 class Equation

This is more of an auxiliary class to store the given equation

$$\frac{1}{x} - \frac{1}{x+1} = \frac{1}{x(x+1)},$$

and the two formulas which returns the absolute and the relative error. He can also draw graphs for both of the errors.

4.1.2.1 Attributes

- precision_ (int): the precision set for the whole Equation object; every term inside of an Equation object adheres to this precision; note that this attribute should be private and must not be changed unless 'change_precision(self, _precision)' is called
- equation_context_ (Context): this is the Context object from the decimal library with its 'prec' attribute to 'precision_' (see above); again this attribute should be private

4.1.2.2 __init__(self, _precision=28)

Arguments

1. _precision (int): the precision for the decimal.Decimal object; must not be zero or negative; is directly stored under 'precision_'; the default value is 28, the same as the default value in the decimal library

Returns

- nothing

Raises

- ValueError: if zero or negative values are passed as '_precision'

Description She constructs an equation object with respect to the desired precision.

4.1.2.3 change_precision(self, _precision)

Arguments

1. _precision (int): the precision for the decimal.Decimal object; must not be zero or negative; is directly stored under 'precision_'

Returns

- nothing

Raises

- ValueError: if zero or negative values are passed as '_precision'

Description Since merely changing the 'precision_' attribute from the outside won't do anything, this method allows the user to change the precision for a given object by correctly changing the 'prec' attribute of the Context object

4.1.2.4 left_side(self, _x)

Arguments

1. _x (int): the value for x; 0 and -1 are not allowed and this function will naturally raise a ZeroDivisionError

Returns

- (Decimal): the solution for the left side of the equation

Description She represents the left side of the equation

$$\frac{1}{x} - \frac{1}{x+1}.$$

4.1.2.5 right_side(self, _x)

Arguments

1. _x (int): the value for x; 0 and -1 are not allowed and this function will naturally raise a ZeroDivisionError

Returns

- (Decimal): the solution for the right side of the equation

Description She represents the left side of the equation

$$\frac{1}{x(x+1)}.$$

4.1.2.6 absolute_error(self, _x)

Arguments

1. _x (int): the value for x for the equation

Returns

- (Decimal): the absolute difference between both side of the equation

Description This methods computes the absolute difference between 'left_side(self, _x)' and 'right_side(self, _x)'.

4.1.2.7 relative_error(self, _x)

Arguments

1. _x (int): the value for x for the equation

Returns

- (Decimal): the relative difference between both side of the equation

Description This methods computes the relative difference between 'left_side(self, _x)' and

4.1.2.8 draw_absolute_error(self, _x)

Arguments

1. _x (int): the fixed x for which the graph is drawn

Returns

- nothing

Description She draws a two dimensional graph of the absolute error of the equation for a fixed x depending on the mantissa length.

4.1.2.9 draw_relative_error(self, _x)

Arguments

1. _x (int): the fixed x for which the graph is drawn

Returns

- nothing

Description She draws a two dimensional graph of the relative error of the equation for a fixed x depending on the mantissa length.

4.1.3 Free Functions

4.1.3.1 explore_machine_epsilon(float_type)

Arguments

1. float_type (class): the class for the float type we want to inspect e.g. np.float32;

Returns

- epsilon (float_type): the machine precision; its data type corresponds to the data type passed as the argument

Description This little algorithm tries to find the machine precision of the given float type, such as np.float16, iterativly. See section **??** for the validity of this algorithm.

4.1.4 main()

She is our main-function. Use the switch, 'test_switch', to test various capabilities of this module. Here, we use as an alternative means to find the machine precision the following formula

$$\tau = \frac{7}{3} - \frac{4}{3} - 1$$

for the validity of this formula see section **??**.

4.2 ab4.py CLI

This module is equipped with capabilities to take the user's input which is then passed to the functions implemented in tools4.py. The main purpose of ab4.py is to test the equation

$$\frac{1}{x} - \frac{1}{x+1} = \frac{1}{x(x+1)}$$

controlling for x and the mantissa length.

4.2.1 class _EndOfInput

Auxiliary (exception) class for easy out of 'get_uint(prompt_text)'

4.2.2 get_uint(prompt_text)

Arguments

1. prompt_text (string): the string displayed to the user

Returns

- num (int): non-zero positive integer input by the user

Raises

- _EndOfInput: this is raised when the user wants to exit the application; used to exit the application in 'main()'

Description

Auxiliary function. She forces the user to enter a positive integer.

4.2.3 get_x_and_mantissa_length()

Arguments

1. nothing

Returns

- (int, int): a a pair of two ints input by the user

Description

Auxiliary function which returns valid user input using 'get_uint(prompt_text)'

4.2.4 precision_calculation(_x, _significand)

Arguments

1. _x (int): an unsigned int for which the equation is evaluated
2. _significand (int): an unsigned int for the mantissa length; this is used to calculate the absolute and the relative error, but is not used to draw the plot since there the mantissa length is the variable

Returns

- nothing

Description

She calculates both errors for a given x and mantissa length ('_significand'). Also draws the plot with the mantissa length as the variable.

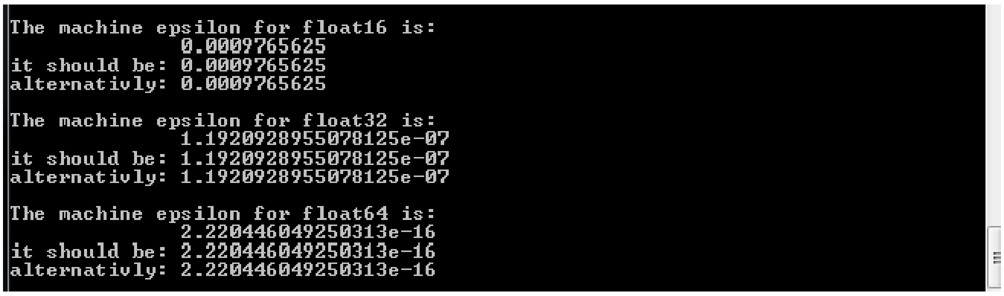
4.2.5 main()

The main function. She takes user's input for x and the length of the mantissa to calculate the absolute and the relative error. A corresponding graph is also drawn.

5 Experiments

5.1 Relative Machine Precision

ab4.py gives us the following table (see figure 4) for the relative machine precision of np.float16, np.float32 and np.float64.



```
The machine epsilon for float16 is:
              0.0009765625
it should be: 0.0009765625
alternativly: 0.0009765625

The machine epsilon for float32 is:
              1.1920928955078125e-07
it should be: 1.1920928955078125e-07
alternativly: 1.1920928955078125e-07

The machine epsilon for float64 is:
              2.220446049250313e-16
it should be: 2.220446049250313e-16
alternativly: 2.220446049250313e-16
```

Figure 4: command line output of ab4.py regarding different floating types

np.float16, np.float32 and np.float 64 have 10 bit, 23 bit, 52 bit mantissa respectively [2]. According to the definition of relative machine precision, we have

$$\begin{aligned} \tau_{16} &= 2^{-10} = 0.0009765625 \\ \tau_{32} &= 2^{-23} = 1.1920928955078125 \times 10^{-7} \\ \tau_{64} &= 2^{-52} \approx 2.220446049250313 \times 10^{-16} \end{aligned}$$

verifying our theory.
An interesting observation is that with each bit, the floating number becomes more precise, however this precision growth is not linear since $\tau = \frac{1}{2^t}$. Considering that each new bit is less effective than the last in terms of improving the precision, there should be a practical limit for the hard ware.

5.2 Rounding Error and Absorbtion

We will consider the plots of the absolute and the relative errors for values

$$\begin{aligned} x_1 &= 7733 && \text{see figure 5} \\ x_2 &= 100,000,000,011 && \text{see figure 6} \\ x_3 &= 2,222,222,222,222,222,222,222 && \text{(22 times 2; see figure 7).} \end{aligned}$$

As one can see on every plot, the plot of the absolute and relative error becomes 0 (or at least very close to) if the mantissa length is longer than number of digit of x. This means that if one wants to compute something in the floating algorithm, they should choose the mantissa to be as long as (preferably longer) than the maximum number of digits in their calculations.

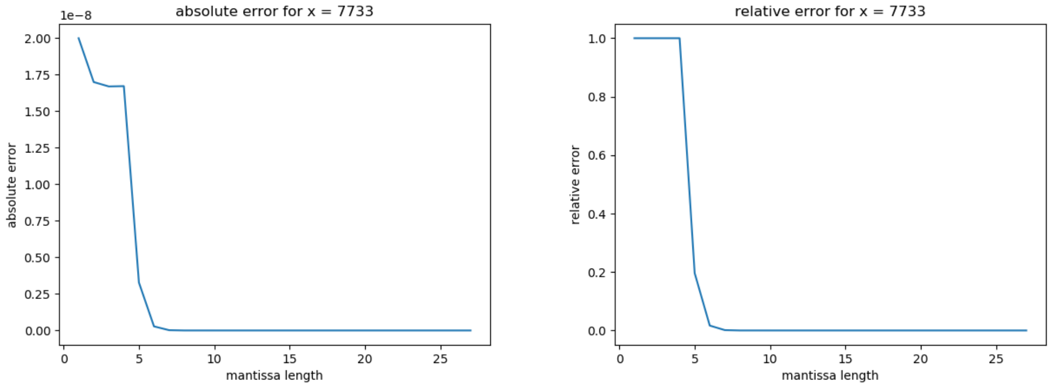


Figure 5: absolute and relative error plot for x = 7733

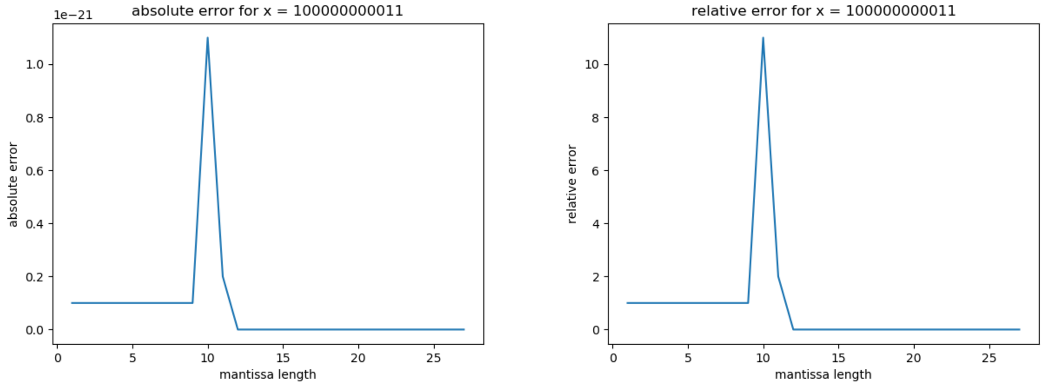


Figure 6: absolute and relative error plot for x = 100000000011

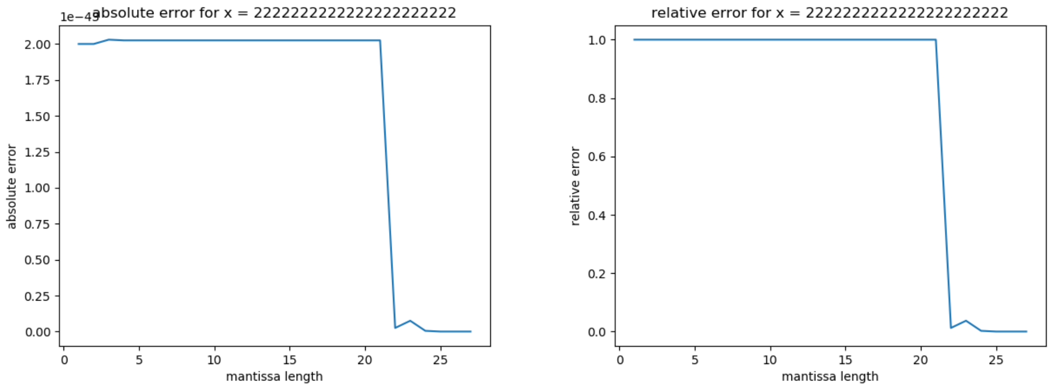


Figure 7: absolute and relative error plot for x = 2,222,222,222,222,222,222,222

6 Bibliography

References

[1] Dr. rer. nat. Hella Rabus. *Vorlesung Einfuehrung in das wissenschaftliche Rechnen.* lecture notes, Humboldt-University of Berlin, 2019.

[2] The Scipy community. *Data types* NumPy v1.13 Manual, `https://docs.scipy.org/doc/numpy-1.13.0/user/basics.types.html`, Accessed: 2019-05-31