Projektpraktikum I: Series 2 Documentation

Tom Lambert, Yuuma Odaka-Falush

13. November 2017

Contents

1 Introduction to underlying Theory **2**

2 Usage **3**

3 Implementation **4**
3.1 summe.py 4
3.1.1 main() 4
3.1.2 AddendGenerator 4
3.2 Sum.py 4
3.3 Console.py 4

4 Experimental Results **5**
4.1 Harmonic Series 5
4.2 Exponential Taylor Series 5
4.2.1 Data Types 6
4.2.2 Different Algorithms 6
4.2.3 Summation Types 8

5 Conclusion **9**

6 Sources **10**

1 Introduction to underlying Theory

In Series 2 of Projektpraktikum I, we tested various algorithms tasked with computing partial sums of series such as the harmonic series and the Taylor series which approximates e^x. The harmonic series is defined as:

$$\sum_{n=1}^{\infty} \frac{1}{n} = \infty$$

We know that the harmonic series diverges, however since computers only have limited space any attempted calculation will conclude that it converges, since there always exists a point where the fractions get too small to process. As data for large harmonic sums exists, the harmonic series is a good test for any program or data type dealing with rational numbers.

The Taylor Series which approximates e^x is defined as:

$$\sum_{i=0}^{\infty} \frac{x^i}{i!} = e^x$$

Taylor series can be used to approximate a wide variety of functions. Due to the division by a factorial, approximations carried out on computers face similar issues as the harmonic series since factorials grow faster than any polynomials. There is also the problem that the differences between two summands may be several orders of magnitude in size, potentially resulting in major errors.

In the next chapters we will explain how we implemented a program to find harmonic sums and approximate e^x using Taylor series, and what kind of results our experiments turned out.

2 Usage

To run the program, simply open `summe.py` using a file explorer or terminal. You can choose which summation to carry out, as well as choose the data types you want to work with. It is also possible to choose your own inputs instead of using the default values.

3 Implementation

Our program was written in Python 2.7.

The following is a list of modules we wrote for this task.

3.1 summe.py

3.1.1 main()

The `main()` method contains the environment where a user can enter custom values and select options.

3.1.2 AddendGenerator

The class `AddendGenerator` contains the methods which create individual addends in the harmonic series and in the Taylor series.

3.2 Sum.py

`Sum.py` contains our summation algorithms to add up large sums. They include the options of sorting by size or sign if desired.

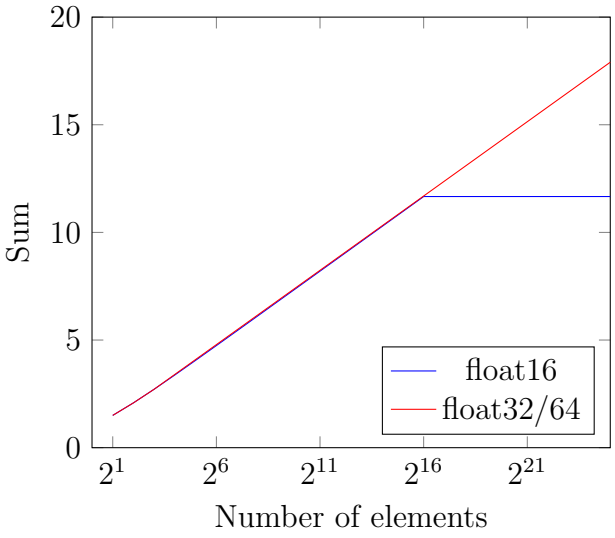
3.3 Console.py

The `Console.py` module provided us with many useful tools for the user interface in `main()` to give it structure and accept various inputs.

4 Experimental Results

4.1 Harmonic Series

We ran tests with three different data types: `float16`, `float32` and `float64`. Below is a chart of the sums computed for the harmonic series.



Float32 and float64 produce almost exactly the same results within our data set. Float16 had very similar results until around 2^{16}, where it leveled off completely. This is consistent with what we would expect, and given sufficient computing resources we would probably see float32 and float64 level off at approximately 2^{32} and 2^{64} respectively.

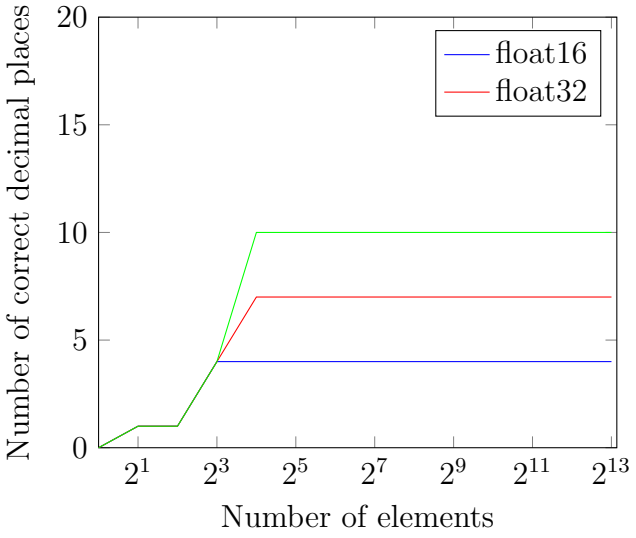
Up to approximately 2^{25} addends, the computation can be achieved in a reasonable (up to $\approx$ 120 seconds) amount of time. Beyond that, more powerful or efficient computing facilities are needed.

4.2 Exponential Taylor Series

We carried out tests for for a variety of parameters: three different data types (`float16`, `float32`, `float64`), three ways of adding up the individual addends (No order, by size, separated by sign), two different approximation series and number of addends, for four different values of x (-20,-1, 1, 20). Conducting a mathematically thorough analysis of all of the data is outside the scope of this documentation, but all of the data can be viewed by starting the program and selecting the default inputs.

4.2.1 Data Types

Below is the number of decimal places our Taylor approximation correctly calculates for x = 1.



Ultimately, we can see that each doubling of float size yields around three additional decimal spaces of accuracy. It makes sense that the approximations do not gain accuracy after 32 addends, since $1/32! \approx 1/2.631^{35}$ is already too small for float64 to process.

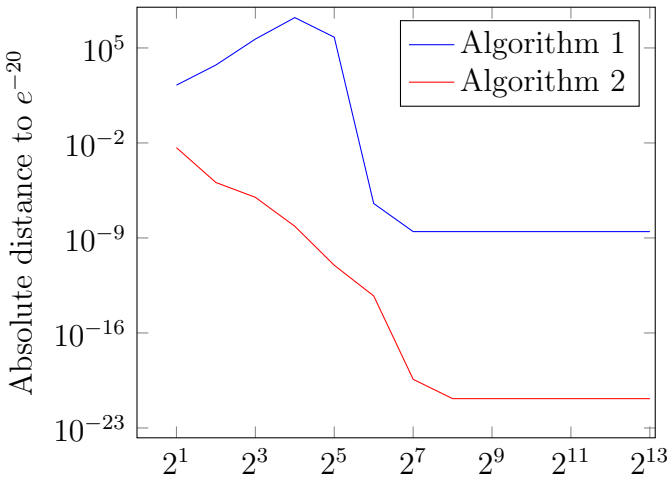
4.2.2 Different Algorithms

There are other ways of approximating e^x; for example, we know that $e^x = \frac{1}{e^{-x}}$, thus we can also try the series:

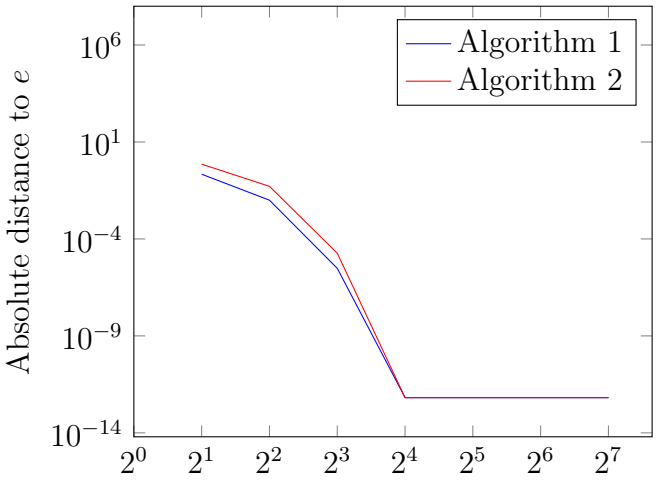
$$\left(\sum_{i=0}^{\infty} \frac{(-1)^i x^i}{i!}\right)^{-1}.$$

Obviously it also converges on e^x, but when we implemented the second series in our algorithm, the results were different. We will call the original approximation Algorithm 1, and the one shown above as Algorithm 2.

Deviations of the approximations for x = -20



Deviations of the approximations for x = 1



We can see that Algorithm 2 is several orders of magnitude more accurate than Algorithm 1 for (as an absolute value) larger values of x such as -20. On the other hand, x=1 ultimately results in identical approximations for both algorithms. The results above highlight the fact that it is important to take care when choosing an algorithm to approximate.

4.2.3 Summation Types

All of the analyses above were for summation by indices, i.e. simply adding up in the order in which the addends are generated. We also tested two other ways of summing: first sorting by size before adding, and sorting into positive and negative values before adding. In most cases the final results stayed the same, but there were some variations for x = 20 with Algorithm 1, as well as some complete aberrations for x = 20 with Algorithm 2. While adding by indices was fairly accurate for x = 20, the other summations resulted in negative values. It is unclear why this happens, especially when we expected an ordered summation to be more accurate due to fewer inaccuracies for adding up pairs of addends. We conclude that in this case, summation by indices turned out to be accurate enough, but it is still worth trying out other summations for such approximations.

5 Conclusion

Our experimental results have shown that even an ordinary computer can carry out fairly accurate calculations with fractions larger than $1\frac{1}{10^{20}}$. This is sufficient for most macro-scale applications; for example, orbital calculations and long-range space missions only need 15 decimal places of π to be precise enough.

6 Sources

Sources of harmonic numbers and multiples of e:

- `https://www.math.utah.edu/~carlson/teaching/calculus/harmonic.html`
- Wolfram Mathematica 10.3 `HarmonicNumber[]` function
- `https://en.wikipedia.org/wiki/E_(mathematical_constant)`
- `http://keisan.casio.com/calculator`