

CMSC 117 Problem Set 1

Clyde Julius Ceniedo,* and Elaine Pajarillo

Department of Mathematics and Computer Science, University of the Philippines Baguio

*Corresponding author: cpceniedo@up.edu.ph

1 Introduction

Measuring errors is a significant process in computation. Results from measurements that are entirely correct are incredibly unusual. Knowing any inaccuracies that occur allows us to compute confidently and helps identify potential solutions. This, as well as knowing the ways of accurately manipulating different numerical systems of numbers are the important concepts of this paper.

Here, we will discuss the solution and findings for the given problems involving concepts on fundamental numerical methods. The two problems given to us were as follows:

1. Write Python functions `abserr` and `relerr` that compute the absolute and relative errors of a given data, respectively. Use the functions to tabulate the absolute and relative errors obtained by comparing the stored value `np.exp(x)` and the computed value using the approximate Maclaurin expansion

$$e^x = \sum_{k=0}^n \frac{x^k}{k!} \quad (1)$$

for $x = [2, 4, 6, 9, 10, 12, 15, 18, 20, 25]$ and $n = 10, 20, 50, 75, 100, 125$.

2. Formulate an algorithm that writes the octal representation of a positive real number into its decimal representation with a specified number of digits. Discuss your algorithm.

2 Writing Python Functions for Error Computation

2.1 Functions

Here, the `factorial`, `macLaurin`, `abserr`, and `relerr` functions were formulated.

1. `factorial`

```
1  import numpy as np
2
3  #function for factorial used in macLaurin series
4  def factorial(k):
5      facValue = 1
6      for k in range(1,k + 1):
7          facValue *= k
8      return facValue
```

Figure 1: Screenshot of the `factorial` python code

Here, we have defined a function named `factorial` that takes one argument `k` which represents the number that will be calculated. It initializes `facValue` to 1 since the factorial of 0 and 1 is 1. Then, it uses a `for` loop to run through 1 to `k` that multiplies its value to `facValue` and assigns it to `facValue`. Lastly, it returns the value of the factorial.

2. `macLaurin`

```

10  #function for macLaurin series of e^x
11  def macLaurin(x, n):
12      macValue = 0
13      for k in range(0, n+1):
14          macValue += x**k / factorial(k)
15      return macValue

```

Figure 2: Screenshot of the `macLaurin` python code

Here, we defined a function named `macLaurin` that calculates the computed value of the given data using the approximate Maclaurin Expansion that takes two arguments, `x` and `n`. The code initializes `macValue` to 0. Then, the `for` loop iterates from 0 to `n`, where `n` is the number of terms for the Maclaurin expansion. Now, for each iteration, it calculates `x` raised to the power of `k` and divides it by calling the function `factorial(k)` - which then accumulatively adds to the `macValue` variable. Finally, it returns the computed value of the Maclaurin series expansion.

3. `abserr`

```

17  #the absolute error function
18  def abserr(x1, x2):
19      return np.linalg.norm(np.array(storArray) - np.array(compArray))

```

Figure 3: Screenshot of the `abserr` python code

This is the first function needed to compute the absolute error. It takes two arguments, `storArray` and `compArray`. The `np.array(storArray)` converts the global variables from the array `storArray[]` into NumPy variables. The same method is applied to `np.array(compArray)` stored to `compArray[]`. The `np.array()` function relies on the NumPy library to perform the calculation - which ensures that mathematical operations can be performed element-wise on arrays. Now, we have `np.array(storArray) - np.array(compArray)` that calculates the element-wise difference between the two arrays. It subtracts each corresponding element of `compArray` from the corresponding element of `compArray`. Then, the function `np.linalg.norm(...)` computes the Euclidian norm of the resulting array - which effectively calculates the absolute error. The `np.linalg.norm` takes the square root of the sum of the squares of its components. We based the formula for this function on the formula below

$$\|x - \tilde{x}\| := \sum_{k=0}^{n-1} (|x_k - \tilde{x}|^2)^{\frac{1}{2}} \quad (2)$$

and the concept that the notation $\|x\|$ is the Euclidean norm (or the vector distance) that can be traced back to the Pythagorean theorem. Finally, the function returns the computed value of the absolute error.

4. relerr

```

21 #the relative error function
22 def relerr(x1, x2):
23     return (np.linalg.norm(np.array(storArray) - np.array(compArray)))/np.linalg.norm(storArray)
24 
```

Figure 4: Screenshot of `relerr` code

This is the second function needed to compute the relative error. It also takes two arguments, `storArray` and `compArray`. The return value of this function differs from the absolute error function because the value of `np.linalg.norm(...)` is divided by the norm of `storArray`, or essentially the notation of $\|x\|$. We based the formula for this function on the formula below,

$$\frac{\|x - \tilde{x}\|}{\|x\|} \quad (3)$$

2.2 Data and Execution

This part describes how the program is executed.

```

25 x = [2, 4, 6, 9, 10, 12, 15, 18, 20, 25]
26 n = [10, 20, 50, 75, 100, 125]
27
28 #for loop to execute the whole program
29 print("\n \tAbsolute Error \t Relative Error")
30 for k in n:
31     storArray = []
32     compArray = []
33     for i in x:
34         storArray.append(np.exp(i))
35         compArray.append(macLaurin(i,k))
36     print(f"{k}\t{abserr(storArray,compArray):.15e} \t {relerr(storArray, compArray):.15e}")

```

Figure 5: Screenshot of the final part of the python code

This final part of the code is where the data and calling of functions are located and executed. We have the 10 values initialized for `x` and 6 values initialized for `n` - which are both used to calculate and print the absolute error and relative error for the Maclaurin series approximation of the exponential function, e^x , for different values of `x` and different numbers of terms, `n`, in the Maclaurin series.

The outer `for` loop iterates over the values of `n` and declares a new array `storArray[]` and `compArray[]` that stands for the stored value and the computed value respectively of each element in `x`. For every iteration, the new computations overwrite the initial values stored in the two arrays.

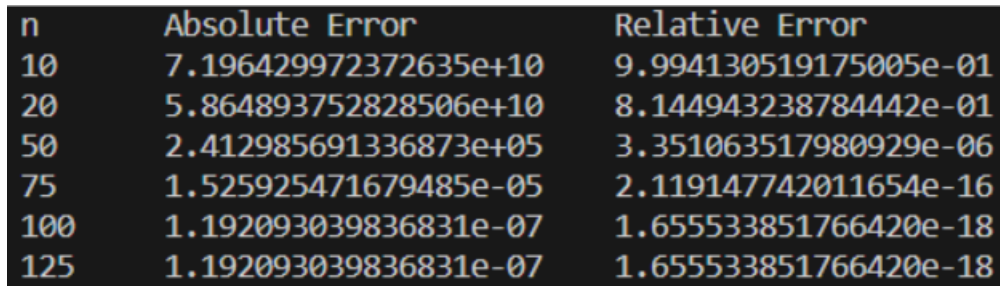
The inner loop iterates over the values of `x`, for every iteration, the stored value of every value in `x` is computed using the function `np.exp(x)` which calculates the exponential value for every element of `x` and appends it/stores it to the `storArray[]`. The same goes for the values for `compArray[]`, which calculates the Maclaurin expansion

value for every element of `x` and uses the number of terms, `n`, needed for the expansion.

Lastly, the function prints the computed absolute error and relative error of the given data `n` times (the number of elements of `n`) by calling the functions `abserr` and `relerr` defined before. The results are printed in a tabular format with columns `n`, `Absolute error`, and `Relative error`. The loop iterates over different values of `n` to show how the errors change as we use more terms in the Maclaurin series to approximate e^x . The format specifier `:.15e` is used to format the error values in scientific notation with 15 decimal places for clarity.

2.3 Results and Discussion

This part discusses the findings and observations seen through the execution of the program. When the program is run, the table with values below is shown on the terminal of the VS Code.



n	Absolute Error	Relative Error
10	7.196429972372635e+10	9.994130519175005e-01
20	5.864893752828506e+10	8.144943238784442e-01
50	2.412985691336873e+05	3.351063517980929e-06
75	1.525925471679485e-05	2.119147742011654e-16
100	1.192093039836831e-07	1.655533851766420e-18
125	1.192093039836831e-07	1.655533851766420e-18

Figure 6: Screenshot of the table shown on the terminal after running the program

n	Absolute Error	Relative Error
10	7.196429972372635e+10	9.994130519175005e-01
20	5.864893752828506e+10	8.144943238784442e-01
50	2.412985691336873e+05	3.351063517980929e-06
75	1.525925471679485e-05	2.119147742011654e-16
100	1.192093039836831e-07	1.655533851766420e-18
125	1.192093039836831e-07	1.655533851766420e-18

Table 1: Program result

This output shows how the accuracy of the Maclaurin series approximation improves as the number of terms `n` increases, with the absolute and relative errors decreasing significantly. Eventually, for larger values of `n`, the errors become very close to zero, indicating a very accurate approximation to e^x . Also, the data has shown that the computed relative error for every value of `n` is significantly lesser than the computed absolute error.

The computed relative error gives the quantitative analysis of the deviation from the true value. Since it is significantly lower, we can say that the relative error is more accurate than the computed absolute error. Though both computations were used to calculate errors, the only difference between these two methods is the element where these methods are compared. The absolute error finds the difference between the actual value and the measured value of a quantity, while the relative error finds the ratio of the absolute error of a measurement and the actual value of the quantity.

3 Algorithm for Octal to Decimal Representation

3.1 Algorithm

This algorithm will get an input string r_8 of numbers in octal representation and the required number of digits d , process it, and output its decimal representation *decimalrep* that follows the required number of digits d .

Algorithm 1 Octal to Decimal with d number of digits

Input: Octal representation of positive real number r_8 , specified number of digits d
Output: Decimal representation *decimalrep* with specified number of digits d

- 1: Initialize empty string *decimal_rep*, *decimal_int*, *decimal_frac*, *input_len* and the integer *point_pos* = -1
- 2: Convert the input r_8 to a string then scan the number of digits of r_8 , store its value in *input_len*. Scan again the input,
- 3: **if** *input_len* > d , **then**
- 4: get only up to the d number of digits as input, and chop off the excess.
- 5: **else**
- 6: put trailing zeros to complete the required number of digits by finding the difference between d and *input_len*, then add zeros to the input based on the result of the difference.
- 7: **end if**
- 8: Scan for the position of the octal point by converting the input to string, then scan again where the index of '.' is located.
- 9: **if** an octal point exists, **then**
- 10: store the value of the index to *point_pos*. Then, split r_8 into its integer and fractional parts based on the index value in *point_pos*.
- 11: **else**
- 12: assume that it is at the end of the point.
- 13: **end if**
- 14: Calculate: $decimal_int = digit_1 * 8^0 + digit_2 * 8^1 + digit_3 * 8^2 + \dots$
- 15: $decimal_frac = digit_p * 8^{-1} + digit_{p-1} * 8^{-2} + digit_{p-2} * 8^{-3} + \dots$
- 16: $decimalrep = decimal_int + "." + decimal_frac$
- 17: Scan *decimalrep*, get only up to d number of digits, chop off the excess
- 18: Return *decimalrep*

3.2 Discussion

The algorithm aims to convert the octal representation of a positive real number into its decimal representation equivalent with a specified number of digits. The algorithm performs the following steps:

1. Initialization:

Initializes four empty strings: *decimalrep* which will be the output and will store the final decimal representation, *decimal_int* will store the computed decimal representation of the integer part of the input, the *decimal_frac* which will store the computed decimal representation of the fractional part of the input, *input_len* which will store the initial number of digits of the input r_8 , and the string *point_pos* = -1 which will output the position of the octal point in the input r_8 .

2. Processing of Input:

Given the specified number of digits d , the algorithm will scan the input r_8 and

get the total number of digits to be stored in *input_len*. The algorithm will again perform a scan on the input r_8 , if *input_len* is greater than the specified number of digits d the algorithm will only get the first d number of digits, disregarding any excess digits. If *input_len* is less than the number of digits d , the algorithm will put trailing zeros to complete the required number of digits by finding the difference between d and *input_len*, to determine the number of zeros needed to be added to complete the required number of digits d .

3. Separating Integer and Fractional Parts of the Input:

Identifies the position of the octal point in the input r_8 . If the octal point is no point found, the algorithm will assume that it is at the end of the input. Splits r_8 into its integer and fractional parts.

4. Conversion:

For the conversion of input r_8 into its decimal representation, the algorithm performs three operations. The first is the conversion of the integer part *decimal_int* of the input, by getting the summation of the products of each digit with 8 raised to the corresponding position of the digit based on its index starting with index 0, starting from the rightmost digit of the integer part of r_8 . Second is the conversion of the fractional part *decimal_frac* of the input, by getting the summation of the products of each digit with 8 raised to the negative of the corresponding position of the digit starting with 1, going from the first digit right of the octal point. The last step for the conversion is the concatenation of *decimal_int* and *decimal_frac*, with a period separating them and storing it in *decimalrep* to be outputted.

5. Processing of Output:

Before returning *decimalrep* as output, the algorithm scans *decimalrep* and trims it into get only the first d number of digits, disregarding any excess digits.

6. Return Output:

Return *decimalrep* as the final output that represents the octal number converted into its decimal representation with the specified number of digits.

In general, this algorithm systematically converts the octal representation of a positive real number into its decimal form with a specified number of digits, while considering both its integer and fractional parts as well as the factor of having the number of digits less than the required d number of digits.

4 Conclusion

In this paper, we tackled two computational challenges in connection to fundamental concepts on numerical methods. We created Python functions, `abserr` and `relerr` that calculated the absolute and relative errors of numerical data and applied these functions to scrutinize the accuracy of Maclaurin expansion approximations for the exponential function. We also created an algorithm that converts octal representations of positive real numbers into their decimal counterparts, with the limitation on the number of digits. This paper showcases the importance of accuracy in error computation as well as the importance of knowing how to manipulate the different numerical representations of numbers which can help further the understanding of the different aspects of scientific calculations.