**Computational Sciences and Informatics Course 690**

**Assignment 1**

**Jericho McLeod**

**Introduction:**

This assignment evaluates and compares several algorithms for calculating or estimating various functions in three stages. In the first stage, two algorithms for estimating powers of powers of Euler’s Number are compared in terms of accuracy and CPU time. Then, the Maclaurin series is approximated with an algorithm to determine the number of iterative steps necessary to achieve a target quantity of significant digits. Finally, the standard deviations of three samples are calculated using traditional methods and an algorithm which only iterates through the data set a single time, rather than twice to obtain a mean prior to calculating variance from the mean.

**Methods:**

The initial calculation for estimating negative powers of Euler’s Number uses the following equations:

and

In both cases, the series is equivalent to in order to simplify the code utilized to implement the calculation. The specified limit for objects to consider for the problem was 20, thus a description of the algorithm used to calculate the first formula would be as follows:

*While Iterations < 20,*

*Calculate ( X to the power of the number of iterations) / the factorial of the number of iterations,*

*Sum the values calculated for iterations divisible by two, and subtract the summation of values not divisible by two.*

And the second formula would be calculated using an algorithm as follows:

*While Iterations < 20,*

*Calculate ( X to the power of the number of iterations) / the factorial of the number of iterations,*

*Divide 1 by the sum the values calculated for iterations all iterations*

Next, the Maclaurin series for approximating the cosine of x was evaluated for number of terms necessary to reach 8 significant digits. This series uses the following equation:

A possible implementation of this algorithm could be described as follows:

*Loop until the resulting value is less than 0.000000005 (thus not changing the first 8 digits of the decimal)*

*Calculate ( X ^ Y ) / factorial of Y, where Y is the number of iterations in the loop*

*Sum the calculated values when the number of iterations is divisible by 4*

*Sum the calculated values where the number of iterations is not divisible by 4*

*Subtract the second summation from the first*

*Once 8 significant digits are reached, end the loop and report the count for iterations.*

Finally, the standard deviation algorithms for both the traditional method and the ‘one pass’ method were compared. The traditionally used formula for standard deviations is:

While the single pass algorithm uses a formula such as:

For k=1 … n, define two sums: A and Q:

Where Ak is the mean value up to the sample k.

Where the sample variance is:

This formula was implemented in python as follows:

*The current Mean is 0*

*The current S is 0*

*For each value in the range,*

*X is the current value*

*The “Prior Mean value” is set to the current Mean*

*The Mean is updated to “The Prior Mean + ( X – the Prior Mean ) divided by the count of iterations through the loop”*

*The S is updated to “The previously known S Value + (X – the Mean) times ( X – the “Prior Mean”)*

*For the last value in the range to be calculated, divide S by (the number of iterations -1)*

**Results:**

The algorithms for estimating roots of Euler’s Number yielded the following results:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Iterations | Algorithm 1 True Error | Algorithm 2 True Error | Algorithm 1 Relative Error | Algorithm 2 Relative Error |
| 1 | -0.993262053 | -0.993262053 | -147.4131591 | -147.4131591 |
| 2 | 4.006737947 | -0.15992872 | 594.6526363 | -23.73552651 |
| 3 | -8.493262053 | -0.047316107 | -1260.511852 | -7.022332923 |
| 4 | 12.34007128 | -0.018685782 | 1831.428962 | -2.773215909 |
| 5 | -13.70159539 | -0.00855842 | -2033.497056 | -1.270182166 |
| 6 | 12.34007128 | -0.004200977 | 1831.428962 | -0.623480318 |
| 7 | -9.361317609 | -0.002102375 | -1389.342719 | -0.312020069 |
| 8 | 6.139674455 | -0.001036951 | 911.2084816 | -0.153897201 |
| 9 | -3.548445585 | -0.000492336 | -526.6360191 | -0.073069181 |
| 10 | 1.833843326 | -0.000221506 | 272.1664813 | -0.032874385 |
| 11 | -0.857301129 | -9.36E-05 | -127.2347689 | -0.013885433 |
| 12 | 0.36594635 | -3.69E-05 | 54.31125393 | -0.005482991 |
| 13 | -0.143740099 | -1.36E-05 | -21.33292224 | -0.002022935 |
| 14 | 0.052293151 | -4.71E-06 | 7.76099167 | -0.000698477 |
| 15 | -0.017718724 | -1.52E-06 | -2.62969187 | -0.000226305 |
| 16 | 0.005618567 | -4.65E-07 | 0.83386931 | -6.90E-05 |
| 17 | -0.001674336 | -1.34E-07 | -0.248493559 | -1.99E-05 |
| 18 | 0.000470635 | -3.65E-08 | 0.069848462 | -5.42E-06 |
| 19 | -0.00012519 | -9.44E-09 | -0.018579877 | -1.40E-06 |
| 20 | 3.16E-05 | -2.33E-09 | 0.004690738 | -3.45E-07 |

The second algorithm converges on an ideal solution most quickly than the first, as evidenced by the true error being measured in hundred-thousandths by the 11th iteration, compared to the first algorithm’s true error at this point still being approximately -0.86. Similarly, the second algorithm’s relative error was measured in hundred-thousandths by the 16th iteration, while the first algorithm’s relative error was still measure in tenths. The second algorithm retained performance of the same order of magnitude throughout the subsequent iterations in both cases.

The Maclaurin series algorithm was able to converge on a solution with 8 significant digits that would not be altered by subsequent iterations after only 7 iterations. The solution obtained for 0.3pi was 0.5877852522924731, which was then compared to the solution obtained using Python’s Math library, 0.5877852522974596. The obtained solution matches the ideal solution to 11 significant digits, highlighting a shortcoming in the algorithm used. Additional experimentation showed that a solution containing 8 significant digits that were identical to the ideal solution could be obtained in one less iteration, for a total of seven, but seeking subsequent iteration values less than 5x10^-7 rather than 5x10^-9, but there is uncertainty as to whether this would be true of all possible calculations or merely this one in particular.

Finally, the comparison of standard deviation algorithms yielded interesting if convoluted results. Using the previously described algorithms, the deviations and times obtained were:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | P1 | P2 | | P3 | |
| Single Pass | 6697.896 | 9232.206 | 12416.56 | |
| Traditional | 6697.896 | 9232.206 | 12416.56 | |
| Single Pass Time | 2.763623 | 2.767632 | 2.862349 | |
| Traditional Time | 0.061839 | 0.059808 | 0.060869 | |

However, the times obtained are not necessarily comparable. The traditional algorithm makes two passes over the data but makes two complete calculations. The single pass algorithm passes over the data only once, but in doing so recalculates the standard deviation and mean at each value. In attempting to compare the two I first considered calculating standard deviations at each step of the traditional algorithm, as this would mirror the time taken to use the algorithm in a real-world setting. However, this could potentially take an estimated 3.6 hours per column of data for a total of 10.8 hours. I attempted to approach this from the opposite direction and count only the time the single pass algorithm took to calculate the mean and standard deviation for the final value, but the results obtained was consistently 0.0 seconds.

Despite not obtaining a comparable value, the single pass algorithm is much more computationally efficient for large datasets. Given the upper bound of the traditional method of 10.8 hours, and the minimum observed time of approximately 20 minutes to calculate standard deviations for each new obtained value, the appeal of an algorithm that achieves this in approximately 10 seconds for the same data set is clear, especially when one considers that the results obtained were identical between the two algorithms.

**Appendix 1**

The Python code used for the first assigned problem is as follows:

import math

x = 5

def errors(n):

target = 0.006737947

true\_error = target-n

relative\_error = true\_error/target

return(true\_error,relative\_error)

def e\_power\_1(x):

print("For the first given function:")

n = 0

count = 0

while count < 20:

q = (x\*\*count)/math.factorial(count)

if count%2 == 0:

n+=q

else:

n-=q

true\_error,rel\_error = errors(n)

print("On the addition of term",count+1,"the true error is",true\_error)

print("and the approximate relative error is",rel\_error)

print()

count+=1

return(n)

def e\_power\_2(x):

print()

print("For the second given function:")

n = 0

count = 0

# n+=1

while count < 20:

q = (x\*\*count)/math.factorial(count)

n+=q

temp\_n = 1/n

true\_error,rel\_error=errors(temp\_n)

print("On the addition of term",count+1,"the true error is",true\_error)

print("and the approximate relative error is",rel\_error)

print()

count+=1

n=1/n

return(n)

epower=e\_power\_1(x)

epower2=e\_power\_2(x)

print("First Function:",epower)

print("Second Function:",epower2)

print("Actual Answer:",.006737947)

#testing python's built in math.e functions

e = math.e

eval\_e = e \*\* -5

print("Python's math library gives the following solution:",eval\_e)