Assignment 2 Writeup

Shirin Rokni

October 8, 2022

1 Introduction

For this write-up, I will be discussing the results of of the **mathlib-test.c** file. I will be analyzing the differences between my implementation and the standard library's. This will be paired with graphs of how close my approximation is to the returned value by the library function.

2 My Implementation vs Standard Library's Implementation

2.1 Sin

In **my implementation**of **sin**, I followed the patterns and equations created by the Taylor Series. I would declare a few variables to act as representations for the numerator and denominator (num, denom). Then, I would declare some variables (de_var) that would iterate throughout the loop and act as the increasing factorial. Then the while loop would run where all of these variables would change and increment until the changes between the new value and the old values become less than epsilon. The **standard library implementation** of **sin** is much more complex and implements several different algorithms. The code is much longer and is ready to output a value that is much longer than just 16 decimal places. Thus, my implementation follows a trigonometric function while the standard library is computing a much more accurate value from various algorithms and computations.

2.2 Cos

In **my implementation** of **cos**, I followed the patterns and equations created by the Taylor Series. Similar to sin, I would declare a few variables to act as the numerator and denominator, and a few others to increment and act as a factorial tracker. The while loop would initiate and keep on iterating until the previous value added onto the total value is so small that it becomes less than epsilon. The **standard library implementation** of **cos** is much more complex, and follows it's own algorithms. It does not follow any pre-existing calculus equations like the taylor series. The code is much longer and in-depth with the intent to return the most accurate value of cosine possible. Thus, my implementation follows a pre-existing equation while the standard library relies on its own abilities to return the most accurate cosine.

2.3 Arcsin

In **my implementation** of **arcsin**, I followed the Newtons formula. Similar to the previous two functions, I would create variables, create indices, and compare the previous to the current variable in the while loop to ensure that the returned value was being incremented so inconsequentially that it no longer

made a significant difference. Similarly to the last few functions, the **standard library implementation** of **arcsin** doesn't follow a pre-existing equation. It follows its own logic to create the most accurate value of arcsin without sacrificing any values in the math with double floats. The code is much more complex and in-depth. Thus, my implementation follows an equation and stops when the changes become inconsequential, but the standard library follows its own logic and tries to obtain the most accurate value.

2.4 Arccos

In **my implementation** of **arccos**, I followed an equation that relied on the existence and accuracy of arcsin. I would simply do some computations on the pre-existing arcsin function and return it. The **standard library implementation** of **arccos** most definitely does not have one line of code referencing a pre-existing function. It relies on its own ability to create an accurate arccos value. Thus, my implementation is a rather quick way to obtain arccos values but it could also sacrifice some accuracy in the math with double floats, while the standard library implementation doesn't cut such corners and fully implements its own way to obtain such values.

2.5 Arctan

In **my implementation** of **arctan**, I followed an equation that relied on the existence and accuracy of both arcsin and arccos. Similar to the arccos function, I would do some computations on both arcsin and arccos to then return the output. The math is very simple and very open for double float rounding errors. The **standard library implementation** of **arctan** very much does not follow this pattern. It creates and follows its own logic and is very lengthy. Thus, my implementation is a quick way of obtaining arctan as long as arcsin and arccos are both accurate, while the standard library implementation takes no such shortcuts and follows its own logic.

2.6 Log

In **my implementation** of **log**, I followed an equation using the Newton-Raphson Method using normalization. I essentially followed the equation and normalized it for when the input is greater than eÎ because I would generate errors otherwise. The **standard library implementation** of **log** does not follow such an equation and follows its own logic with various algorithms and complex functions that do much more than normalize a few variables and divide a few double floats. Thus, my implementation follows a pre-existing trigonometric function and suffers from double float rounding errors while the standard library follows a more intricate and in-depth logic.

3 Why there are differences + Graphs

Clearly, my implementation does a lot of **math with various double float variables**, so there is a lot of room for rounding error. This is a function-wide issue that my all of my implementations due to the nature of doing calculations with double floats such as division and multiplication. There is no way to accurately compute so many double floats within such a tight boundary of 16 decimal places without sacrificing some accuracy. I will go more in depth with the reasons behind differences between my functions and those from the standard library.

3.1 Sin

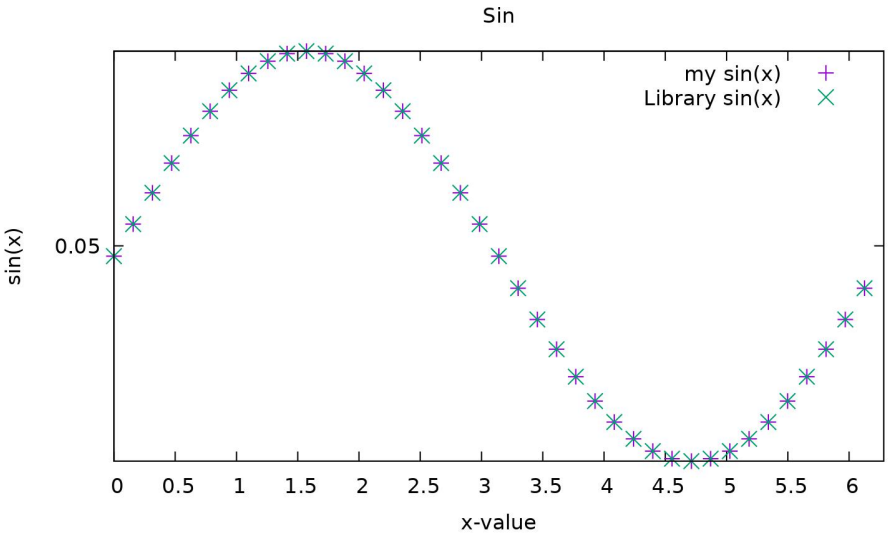

Figure 1: my sin function and the standard library's sin function starting from x in [0,...,2*π]

As the graph above shows, there is **minimal differences** between my sin function and the one implemented by the standard library, but there are very small differences that occur beyond the epsilon boundary. These could come from how my sin function does a lot of arithmetic with **double float variables**, so rounding errors are bound to happen which make the final return value less accurate.

3.2 Cos

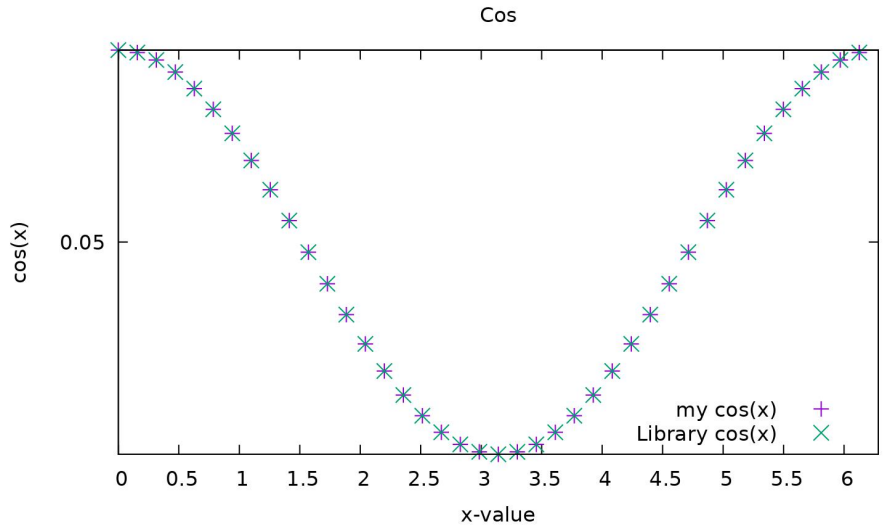

Figure 2: my cos function and the standard library's cos function starting from x in [0,...,2*π]

As the graph above shows, there is **minimal differences** between my cos function and the one implemented by the standard library, but there are very small differences that occur beyond the epsilon boundary. These could come from how my cos function does a lot of arithmetic with **double float variables**, so rounding errors are bound to happen which make the final return value less accurate.

3.3 Arcsin

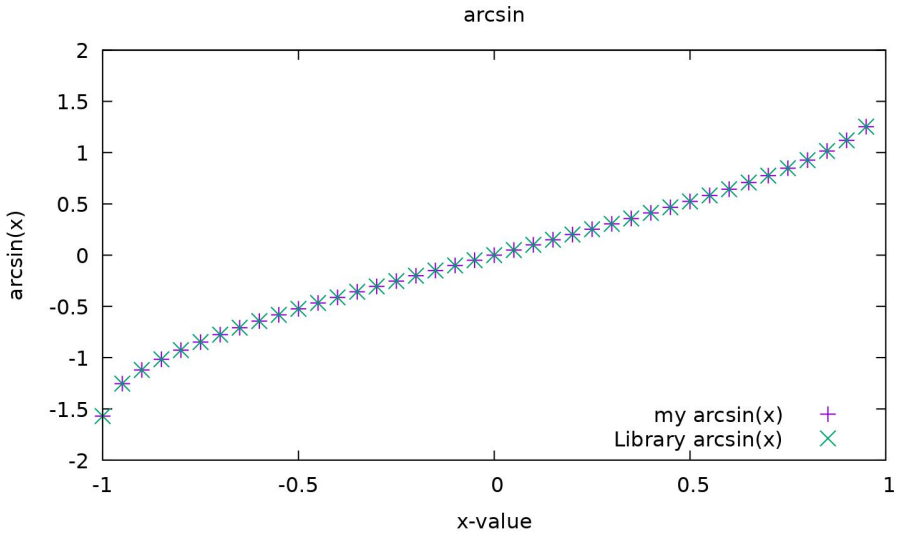

Figure 3: my arcsin function and the standard library's arcsin function starting from x in [-1,...,1]

As the graph above shows, there is **minimal differences** between my arcsin function and the one implemented by the standard library, but there are very small differences that occur beyond the epsilon boundary. These could come from how my arcsin function does a lot of arithmetic with **double float variables**, so rounding errors are bound to happen which make the final return value less accurate. There is another notable difference is a larger difference value when x = -1. The graph doesn't show this very well since the difference is larger than epsilon yet still very small. This occurs because at x = 1, the graph becomes more vertical which means that polynomial functions like the taylor series need to use a large number of terms and loops to get to a more accurate return value. Thus, getting accurate approximations with such polynomial equations cannot output an exact value

3.4 Arccos

As the graph below shows, there is **minimal differences** between my arccos function and the one implemented by the standard library, but there are very small differences that occur beyond the epsilon boundary. The way the arccos function was implemented was that it uses the arcsin function and does some computations on it. This does even more **double float arithmetic** which leads to more **double float rounding errors**. The final result is bound to be even less accurate since it returns outputs based on other functions and their small errors. There is another notable difference is a larger difference value when x = -1. The graph doesn't show this very well since the difference is larger than epsilon yet still very small. This occurs because at x = 1, the graph becomes more vertical which means that polynomial

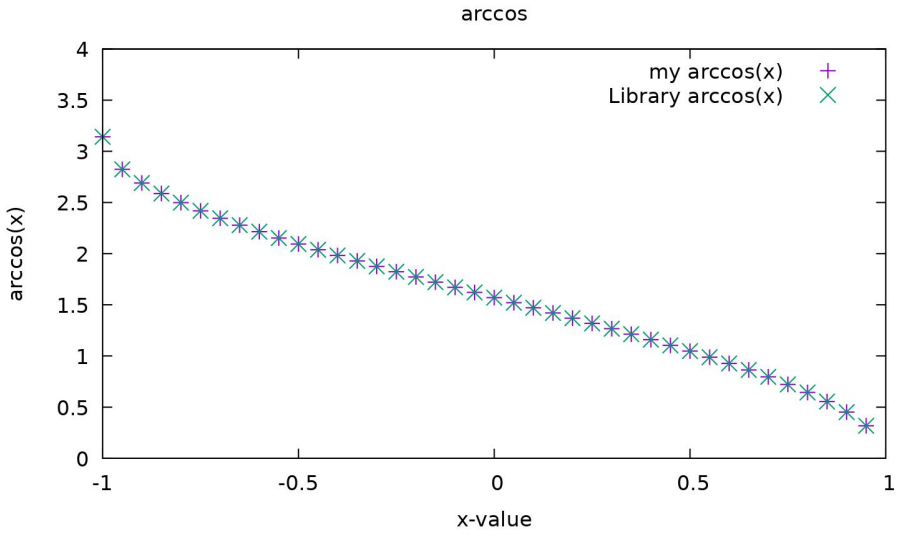

Figure 4: my arccos function and the standard library's arcsin function starting from x in [-1,...,1]

functions like the taylor series need to use a large number of terms and loops to get to a more accurate return value. Thus, getting accurate approximations with such polynomial equations cannot output an exact value

3.5 Arctan

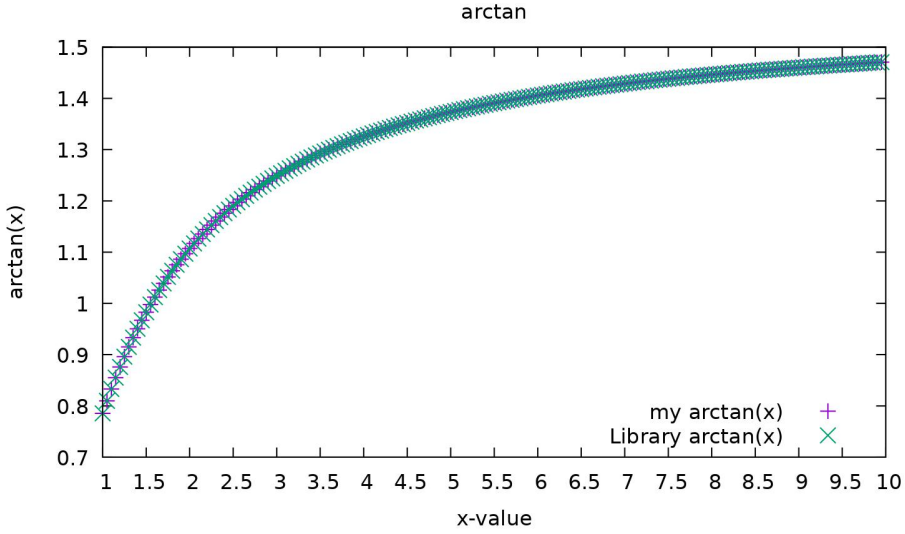

Figure 5: my arctan function and the standard library's arctan function starting from x in [1,...,10]

As the graph above shows, there is **minimal differences** between my arctan function and the one implemented by the standard library, but there are very small differences that occur beyond the epsilon

boundary. The way the arctan function was implemented was that it uses the arcsin and arccos functions and does some computations on them. This does even more **double float arithmetic** which leads to more **double float rounding errors**. The final result is bound to be even less accurate since it returns outputs based on other functions and their small errors.

3.6 Log

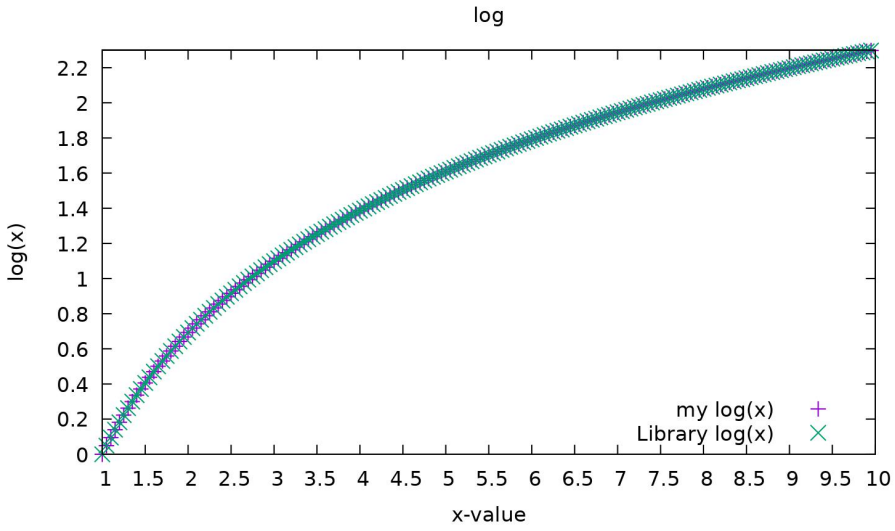

Figure 6: my log function and the standard library's log function starting from x in [1,...,10]

As the graph above shows, there is **minimal differences** between my log function and the one implemented by the standard library, but there are very small differences that occur beyond the epsilon boundary. These could come from how my log function does a lot of arithmetic with **double float variables**, so rounding errors are bound to happen which make the final return value less accurate.