Caterin Duarte Reynosa
cduarter@ucsc.edu

CSE13S Fall 2020
Assignment 2: A Small Numerical Library
Write up

Purpose of the Assignment

The purpose of this assignment is to get a deeper understanding of how specific libraries' functions work. Specifically, our task for this assignment was to create a Sin, Cos, Tan, Exponent, and a Log function. The 'problem' we had to solve for this assignment was that we needed to implement all of the functions without using the math library, and by using Taylor's series and Newton's method for the Log function.

My Solution and Code for the Assignment

For my solution, I looked at the equation we were given to create a loop that mimicked the equations. I also looked at the lecture slides where some of our functions were already implemented and got inspired by them, but tried to replicate them in a way where I understand all the steps.

Sin:

For the Sin function, I first had a variable, num, that kept track of the numerator of my sums. Since Taylor's series starts with x, I set my numerator to be equal to x and my den (denominator) equals 1.0 to give me the first part of Taylor's series. After that, I also had a counter that kept track of my term for the current x number I was trying to calculate the current sum for. I set the term to num/den because the sin's Taylor series utilizes the odd terms. I had this also set similar to the term's value. To get an accurate approximation for my Sin function, I had a loop that broke when my term was larger than 1e-14 since it's a number that will give us a fairly accurate approximation of when the term approaches 0. In my loop, I created a factorial variable to keep track of the current x's factorial. I set it equal to 3 and increment it by 2 since I just want to calculate the odd terms. Since I only want the odd terms, I set my num variable to the old num amount times -x and times x to give the most current numerator. I multiplied by one -x because, in the Taylor series, the signs alternate, and a negative x would account for that. I did a similar thing for the denominator but simply subtracted one of my factorials by one in order to make sure I was calculating the odd terms. Then, I updated the term's value by having it set to = num/den again. Lately, I had the sum be equal to the old sum plus the term and returned whatever value that gave me.

Cos:

For the Cos function, I first had a variable, num, that kept track of the numerator of my sums. Since Taylor's series starts with 1, I set my numerator to be equal to 1 and my den (denominator) equals 1.0 to give me the first part of Taylor's series. After that, I also had a counter that kept track of my term for the current x number I was trying to calculate the current sum for. I set the term to num/den because the sin's Taylor series utilizes the even terms. I had this also set similar to the term's value. In order to get an accurate approximation for my Sin function, I had a loop that broke when my term was larger than 1e-14 since it's a number that will give us a fairly accurate approximation of when the term approaches 0. In my loop, I created

a factorial variable to keep track of the current x's factorial. I set it equal to 2 and increment it by 2 since I just want to calculate the even terms. Since I only want the even terms, I set my num variable to the old num amount times -x and times x to give the most current numerator. I multiplied by one -x because, in the Taylor series, the signs alternate, and a negative x would account for that. I did a similar thing for the denominator but simply subtracted one of my factorials by one in order to make sure I was calculating the even terms. Then, I updated the term's value by having it set to = num/den again. Lately, I had the sum be equal to the old sum plus the term and returned whatever value that gave me.
My output

Tan

Since the Taylor series of tan could be simplified to sin/cos, I simply returned what sin / cos would give me.

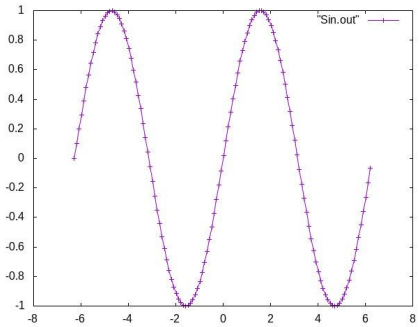
Exp

Similar to the Cos function, I set my numerator and denominator to 1. I also set my term to num/den since that's how I calculate what x/k in the Taylor series would give me. I then created a loop that would break when the term was > 1e-14. In that loop, I initialized a variable that kept track of my factorials and had that equal to 1 and incremented by 1 since the Taylor series for Exp required me to calculate all the terms not just evens/odds. I updated the value of my term by having it equal to num/den again and then multiplying it by the previous term. Lastly, I had my sum equals to the previous sum plus the new term and returned it.

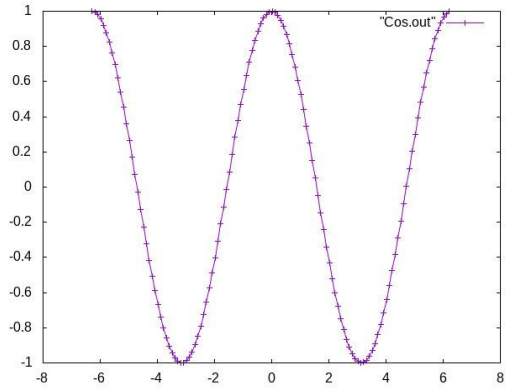
Log

For the log function, I used Newton's method to calculate an approximate answer. First I had a variable that kept track of the 'power' I was having my x be multiplied by. Since I already had a function that calculated the Exp, I simply stored what the value of the function call passing my power variable to the Exp function would give me. I had a loop that didn't break until (x-p) > 1e-14. I gave my power variable a new value by having it equal to the old value plus my x minus my stored function called output divided by my stored function call output. Lastly, I updated my function call variable by doing a function call on my new power value and returning my power value at the end.

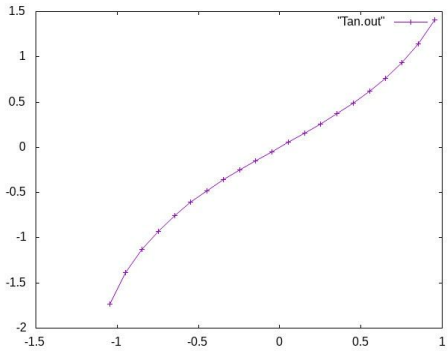
My output



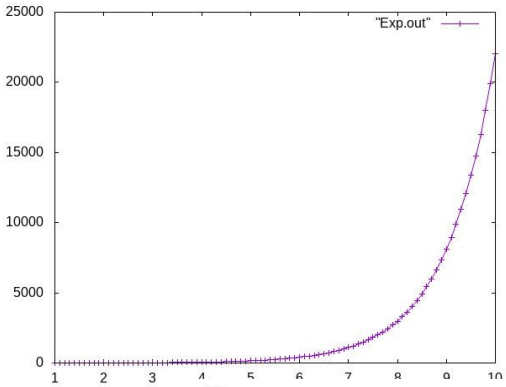
This graph shows what my Sin function looks like with various different x values. The x plane shows the different xs tasted in my mthalib-test.c file and the y plane shows what my implementation calculated the Sin(x) to be.



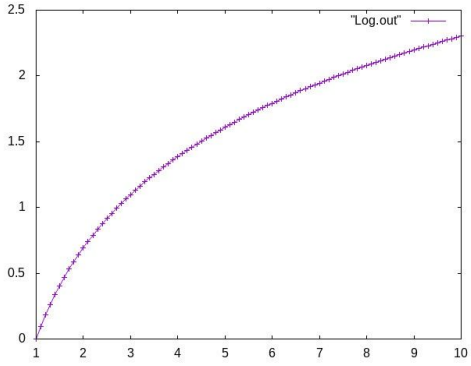
This graph shows what my Cos function looks like with various different x values. The x plane shows the different zxs tasted in my mthalib-test.c file and the y plane shows what my implementation calculated the Cos(x) to be.



This graph shows what my Tan function looks like with various different x values. The x plane shows the different xs tasted in my mthalib-test.c file and the y plane shows what my implementation calculated the Tan(x) to be.

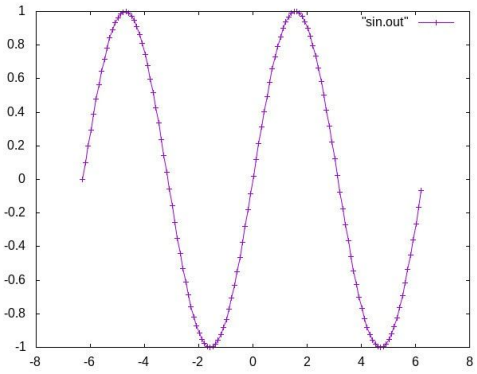


This graph shows what my Exp function looks like with various different x values. The x plane shows the different xs tasted in my mthalib-test.c file and the y plane shows what my implementation calculated the Exp(x) to be.

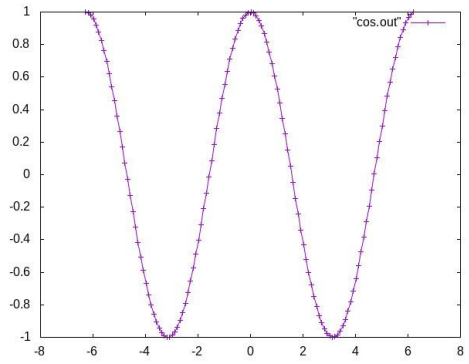


This graph shows what my Log function looks like with various different x values. The x plane shows the different xs tasted in my mthalib-test.c file and the y plane shows what my implementation calculated the Log(x) to be.

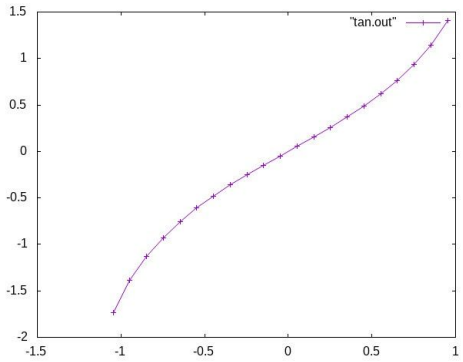
The library's output



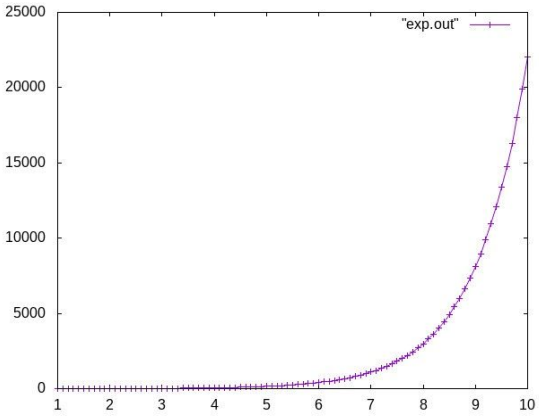
This graph shows what the sin function from the math library looks like with various different x values. The x plane shows the different xs tasted in the mthalib-test binary and the y plane shows what the library's implementation calculated the sin(x) to be



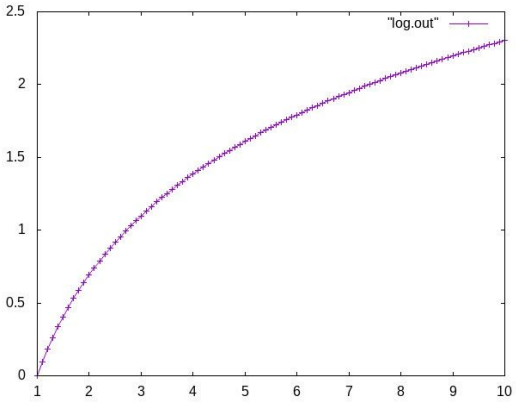
This graph shows what the cos function from the math library looks like with various different x values. The x plane shows the different xs tasted in the mthalib-test binary and the y plane shows what the library's implementation calculated the cos(x) to be



This graph shows what the tan function from the math library looks like with various different x values. The x plane shows the different xs tasted in the mthalib-test binary and the y plane shows what the library's implementation calculated the tan(x) to be

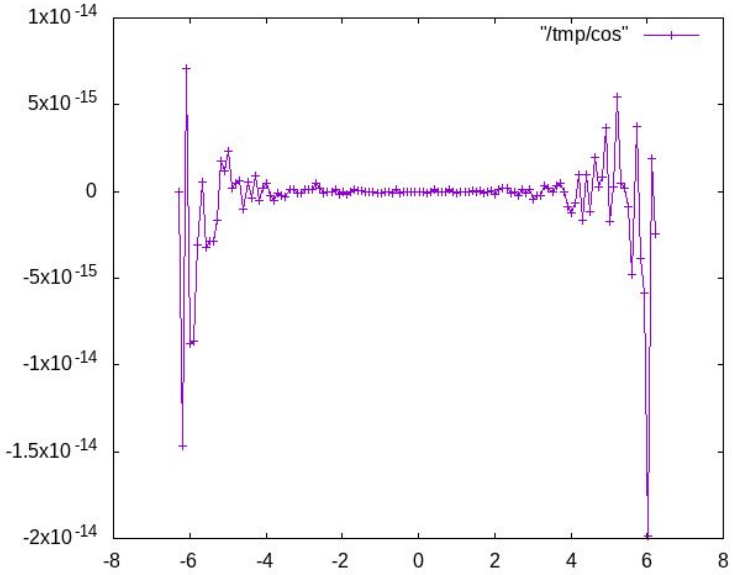


This graph shows what the exp function from the math library looks like with various different x values. The x plane shows the different xs tasted in the mthalib-test binary and the y plane shows what the library's implementation calculated the exp(x) to be

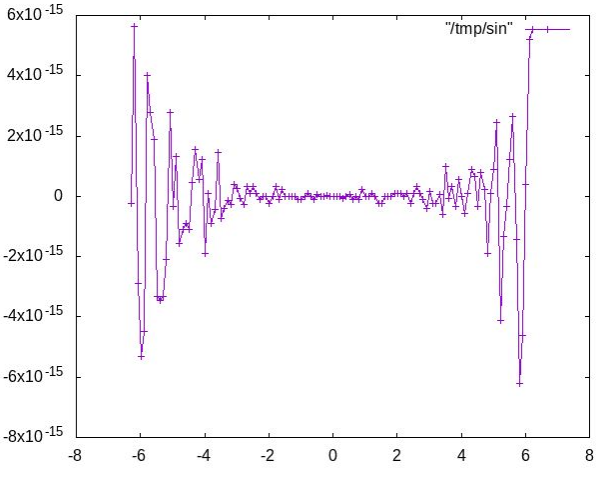


This graph shows what the log function from the math library looks like with various different x values. The x plane shows the different xs tasted in the mthalib-test binary and the y plane shows what the library's implementation calculated the log(x) to be

The difference between both outputs

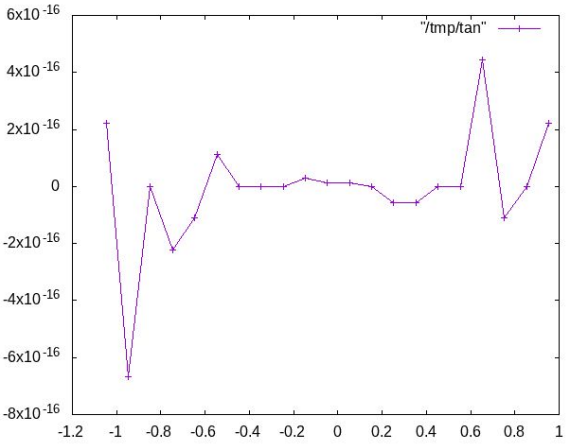


This graph shows the difference between my Cos function and the library's cos function. The x-axis represents the xs being tested and the y axis represents the difference between my implementation of Cos(x) and cos(x). As the graph shows, there's some difference between my output versus the library's output. Even though these outputs are very similar some floating points are different. Thus, these small differences are what cause a -0.00000… when I print out their differences. Since my function isn't as accurate as the library's functions, I don't get a difference of zero but instead get a difference with a small decimal. This graph represents the differences between the decimal points between both functions.

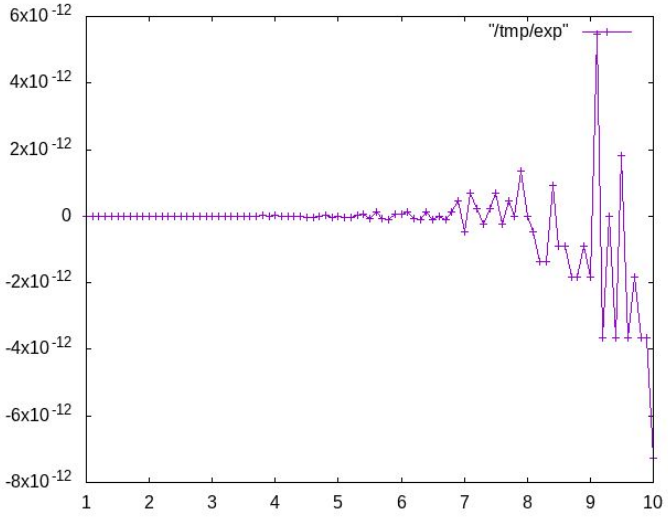


This graph shows the difference between my Sin function and the library's sin function. The x-axis represents the xs being tested and the y axis represents the difference between my implementation of Sin(x) and sin(x). As the graph shows, there's some difference between my output versus the library's output. Even though these outputs

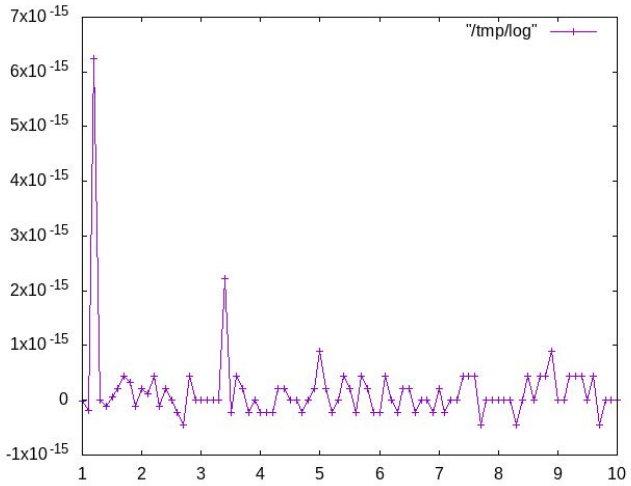
are very similar some floating points are different. Thus, these small differences are what cause a -0.00000… when I print out their differences. Since my function isn't as accurate as the library's functions, I don't get a difference of zero but instead get a difference with a small decimal. This graph represents the differences between the decimal points between both functions.



This graph shows the difference between my Tan function and the library's tan function. The x-axis represents the xs being tested and the y axis represents the difference between my implementation of Tan(x) and tan(x). As the graph shows, there's some difference between my output versus the library's output. Even though these outputs are very similar some floating points are different. Thus, these small differences are what cause a -0.00000… when I print out their differences. Since my function isn't as accurate as the library's functions, I don't get a difference of zero but instead get a difference with a small decimal. This graph represents the differences between the decimal points between both functions.



This graph shows the difference between my Exp function and the library's Exp function. The x-axis represents the xs being tested and the y axis represents the difference between my implementation of Exp(x) and exp(x). As the graph shows, there's only a little bit of difference between my output versus the library's output. Even though these outputs are very similar some floating points are different. Thus, these small differences are what cause a -0.00000… when I print out their differences. Since my function isn't as accurate as the library's functions, I don't get a difference of zero but instead get a difference with a small decimal.



This graph shows the difference between my Log function and the library's log function. The x-axis represents the xs being tested and the y axis represents the difference between my implementation of Log(x) and log(x). As the graph shows, there's some difference between my output versus the library's output. Even though these outputs are very similar some

floating points are different. Thus, these small differences are what cause a -0.00000… when I print out their differences. Since my function isn't as accurate as the library's functions, I don't get a difference of zero but instead get a difference with a small decimal. This graph represents the differences between the decimal points between both functions.

What I learned from the lab

During this lab, I learned the importance of floating point numbers and what a huge difference they make. I also learned how to implement some of the math library's functions without utilizing any other functions. I also learned how even though we can try to get as close and approximate to a number as we want, sometimes it doesn't work out because decimal points can be tricky