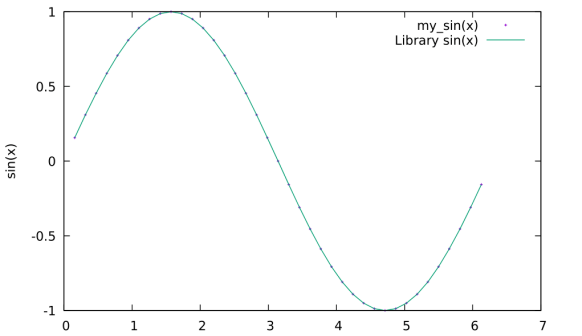
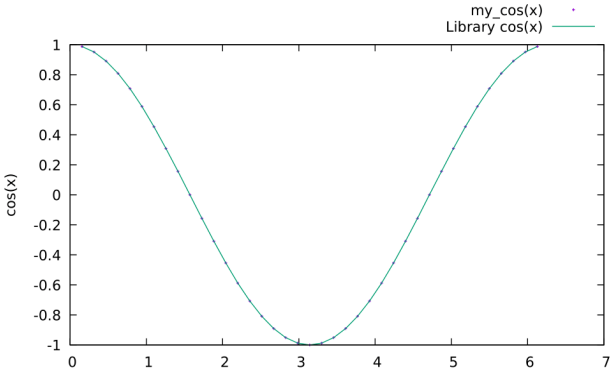
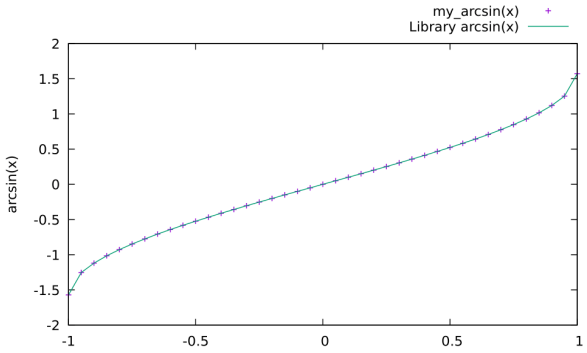
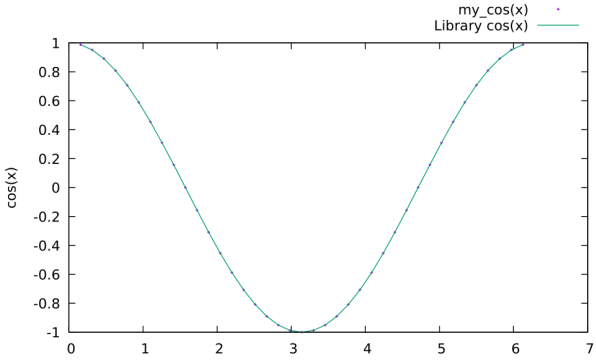
NOTE: My tests are including the end values because the AutoGrader was set up to test for the endpoints as well. This is an intentional decision as I know the lab doc specifies to not use endpoints for certain functions. I also find including the endpoints to feel more complete when approximating values.

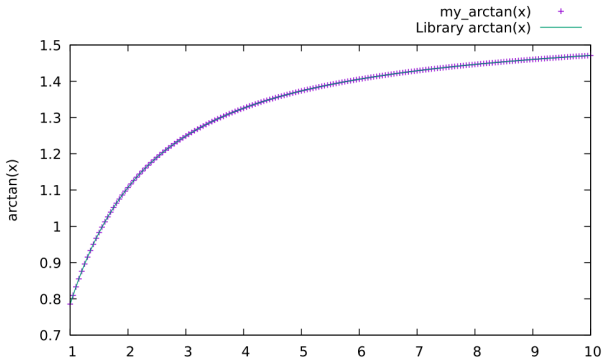
Writeup

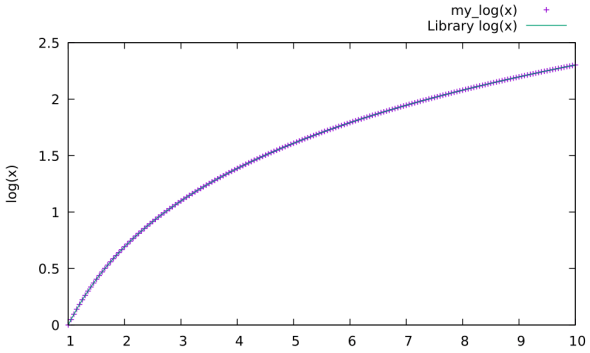












Overall, my approximations were fairly accurate to the library's calculations. For graphs sin, cos, arcsin, and arccos, I noticed that my approximation was extremely accurate to the library.

x	my_sin	Library	error
0.0000	0.000000000	0.000000000	0.000000000000
0.1571	0.156434465	0.156434465	0.000000000000
0.3142	0.309016994	0.309016994	0.000000000000
0.4712	0.453990500	0.453990500	0.000000000000
0.6283	0.587785252	0.587785252	0.000000000000
0.7854	0.707106781	0.707106781	0.000000000000
0.9425	0.809016994	0.809016994	0.000000000000
1.0996	0.891006524	0.891006524	0.000000000000

x	my_cos	Library	error
0.0000	1.000000000	1.000000000	0.000000000000
0.1571	0.987688341	0.987688341	0.000000000000
0.3142	0.951056516	0.951056516	0.000000000000
0.4712	0.891006524	0.891006524	0.000000000000
0.6283	0.809016994	0.809016994	0.000000000000
0.7854	0.707106781	0.707106781	0.000000000000
0.9425	0.587785252	0.587785252	0.000000000000
1.0996	0.453990500	0.453990500	0.000000000000

x	my_arcsin	Library	error
-1.0000	-1.570796327	-1.570796327	0.000000000000
-0.9500	-1.253235898	-1.253235898	0.000000000000
-0.9000	-1.119769515	-1.119769515	0.000000000000
-0.8500	-1.015985294	-1.015985294	0.000000000000
-0.8000	-0.927295218	-0.927295218	0.000000000000
-0.7500	-0.848062079	-0.848062079	0.000000000000
-0.7000	-0.775397497	-0.775397497	0.000000000000
-0.6500	-0.707584437	-0.707584437	0.000000000000
-0.6000	-0.643501109	-0.643501109	0.000000000000

x	my_arccos	Library	error
-1.0000	3.141592654	3.141592654	0.000000000000
-0.9500	2.824032224	2.824032224	0.000000000000
-0.9000	2.690565842	2.690565842	0.000000000000
-0.8500	2.586781621	2.586781621	0.000000000000
-0.8000	2.498091545	2.498091545	0.000000000000
-0.7500	2.418858406	2.418858406	0.000000000000
-0.7000	2.346193823	2.346193823	0.000000000000
-0.6500	2.278380764	2.278380764	0.000000000000
-0.6000	2.214297436	2.214297436	0.000000000000

In fact, my approximations are accurate to the libraries to over 12 decimal places!

By looking at the implementation of these functions online, I found that these functions were implemented in a very similar way which may be why my approximations were so close. The biggest difference I found was that these functions would have multiple ways of approximating the values from (0.0, 0.5) and (0.5, 1). This is most likely because, for certain values of sin, cos, arcsin, and arccos, there are faster methods to calculate the value.

I noticed that the approximations I used to create these functions were also used inside the library functions but only for certain ranges of inputs.

I also noticed that the library also included special cases for x = -1 and x = 1 for arcsin. My method of approximation gets very strange as it approaches the endpoints as in order to get the approximation, you need to take the derivative. However, at the end point of the line, the derivative is undefined. This is why the approximation acts so

strange when given the exact endpoint values but is still fine when given value extremely close to 1 or -1.

One of the biggest differences I found was in the sin and cos implementation. I approximated to around 33 terms of the Taylor series where the library only went to 14. This is telling me that you are probably not gaining much accuracy past 14 terms.

In addition, limiting the terms required to calculate the answers means that the function will be faster.

Another large difference was the implementation of atan() and log. Both of these functions did not use newtons approximation to calculate their values. I suspect that this is because this method is quite slow when compared to the libraries method.

In fact, neither of these functions even uses a while loop and the atan() function does not even include a for a loop!

Unlike sin, cos, asin, and acos, the library functions were not only faster than my implementation but more accurate! Since I had to specify the EPSILON of accuracy and set it to 10e-10, that meant that I would only be accurate up to 10 decimal places. This lead to errors between my implementation and the library's function past 10 decimal places.

Take the first few terms of my implementation compared to the libraries implementation

x	my_arctan	Library	error
1.0000	0.785398163	0.785398163	0.000000000033
1.0500	0.809783573	0.809783573	0.000000000032
1.1000	0.832981267	0.832981267	0.000000000001
1.1500	0.855052737	0.855052737	0.000000000024
1.2000	0.876058051	0.876058051	0.000000000041

x	my_log	Library	error
1.0000	0.000000000	0.000000000	0.000000000000
1.0500	0.048790164	0.048790164	0.000000000000
1.1000	0.095310180	0.095310180	0.000000000000
1.1500	0.139761942	0.139761942	0.000000000003
1.2000	0.182321557	0.182321557	0.000000000001

As shown above, my implementations are only accurate for 10 decimal places, I believe that the library is using a much better method of approximation which allows it to be more accurate. This difference creates the error between my implementation and the libraries.

Despite this inaccuracy, 10 decimal places is still extremely accurate. Looking at the graphs at the top of the page, it is clear that my approximations (the points) fall exactly on the line given by the library version of each function. For general use, 10 decimal places of accuracy will be more than enough as it is practically indistinguishable when plotted on graphs and used for most calculations.