Phys 512 PS2

Andrew Lewis

September 24, 2022

Problem 1

In this problem I used my numerical integrator from question 2 as well as the integrator from scipy.quad. The center of the sphere is at z= 0.

Unlike when the integral is computed analytically, here is singularity in the integral at z=r, which tends to break my integrator. This is due to the (z-r) term that appears in the integrand. However, scipy.integrate.quad does not care about the singularity.

See the plot below of the electric field integral. For the singularity in the integral, instead of having the value go to infinity i set it to -5.

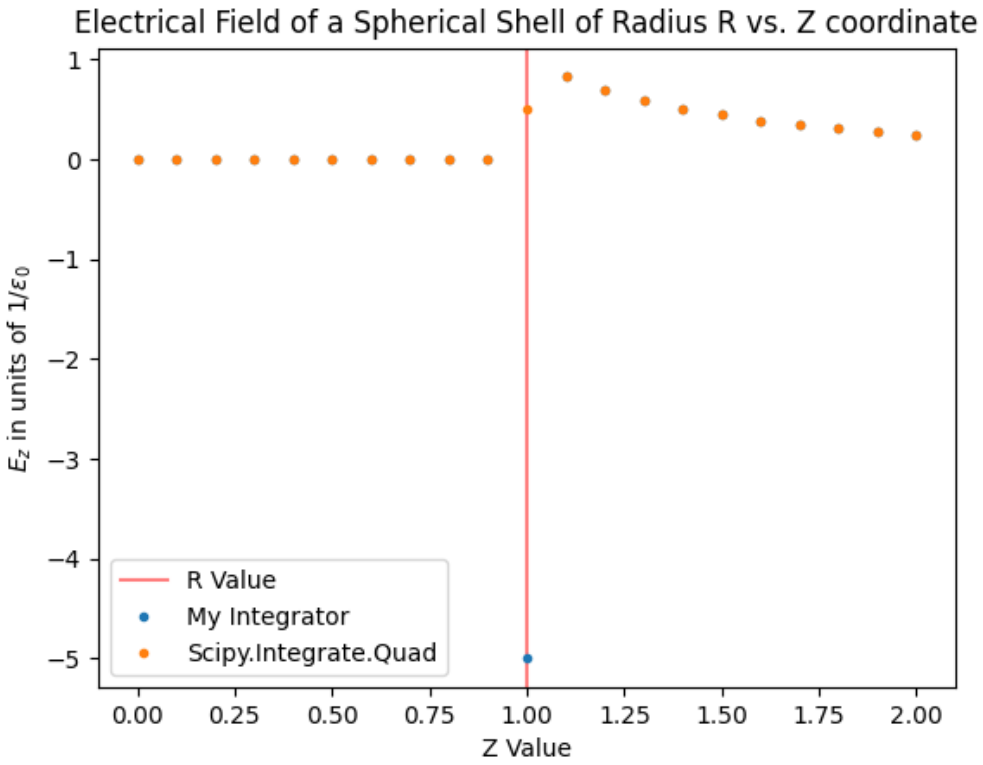

Figure 1: A comparison between the scipy integrator and my custom numerical integrator. My integrator's values are shown in blue and the scipy integrator is shown in orange. As can be seen in the figure, the only differences between the scipy integrator's values and my integrator are at z=r

The only difference in the values returned from the integrator are at Z=R, where there is a singularity in the integral. While my integrator breaks (with many errors) scipy does not care, and handles it like a champ.

In order to plot multiple z values I passed a lambda function into my integrator, very fun.

Problem 2

See the code for my adaptive integrator under PS2/PS-2. To implement the improved variable step size integrator I adapted the code from class. The important insight in order to minimize calls compared the naive approach is to see what values we can carry forward from previous calls,I will explain this in the following.

Take the following linspace:

$$np.linspace(1, 5, 5) = [1, 2, 3, 4, 5]$$

However, when we proceed to do the step size mid we can see that half of the points will be in the first interval and the other half will be in the other

$$mid = \frac{a + b}{2} = 3$$

Therefore our two new intervals will be

$$1, 3 \text{ and } 3, 5$$

Which will contain the point

$$np.linspace(1, 3, 5) = 1, 1.5, 2, 2.5, 3$$

and

$$np.linspace(3, 5, 5) = 3, 3.5, 4, 4.5, 5$$

So you see (hopefully) we can carry forward 3 of our x values forward to the next function call. That means we will only have to do 2 function evaluations at each recursive step! not bad.

Of course, it would be easier to come up with an expression for how many calls save. If the naive approach requires 5 calls at each step we can write the number of calls as:

$$\text{Number of calls: } = 5n$$

Where n is the number of recursions. For our improved method, we have to think about it a bit more. The initial call takes 5, but then all subsequent calls take only 2. Therefore we can write the number of function evaluations as:

$$\text{Number of calls in improved function: } = 5 + (2(n-1)) = 3 + 2n$$

Therefore, as n-> ∞ we will only use 2/5ths of the function evaluations.

We can test this expression in our code. We will try a few functions, $e^x, sin(x), x^7$ with a starting error tolerance of $1e-8$

For e^x The naive function required 475 calls while the improved method required 193 calls

For $cos(x)$ the naive function required 315 calls while the improved method required 129 calls.

For x^7 both functions required 5 calls.

All of these values agree with the relationship we defined earlier.

Problem 3

I wrote a function that model the base 2 of x from 0.5 to 1. I achieved an accuracy in the region better than 10^-6 by using 8 terms (but I probably could've gotten away with 7), where the final coefficient was $-2.67468148e-07$. To get this fit I used 10001 x and y values, and re scaled the x axis using:

$$newX = 4x - 3$$

Which is only valid on the interval 0.5 to 1.

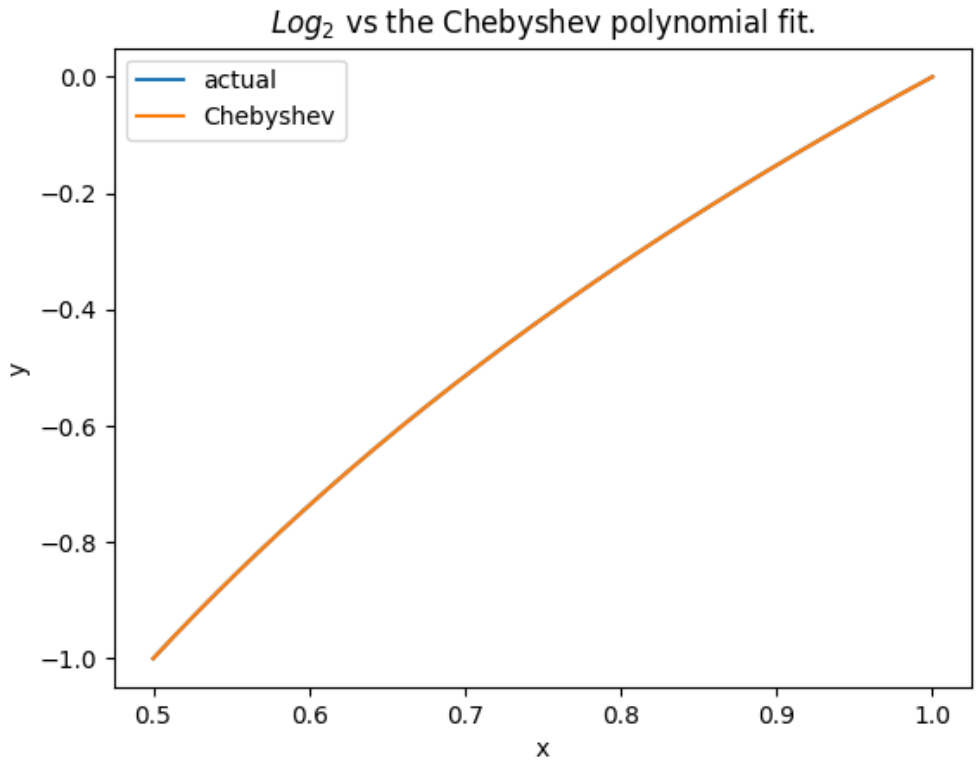

Figure 2: Comparison between Log_2 and the Chebyshev fit. As is evident, without further investigation it is impossible to distinguish the two. The maximum error will be bounded by the sum of the remaining terms starting at $10e - 7$

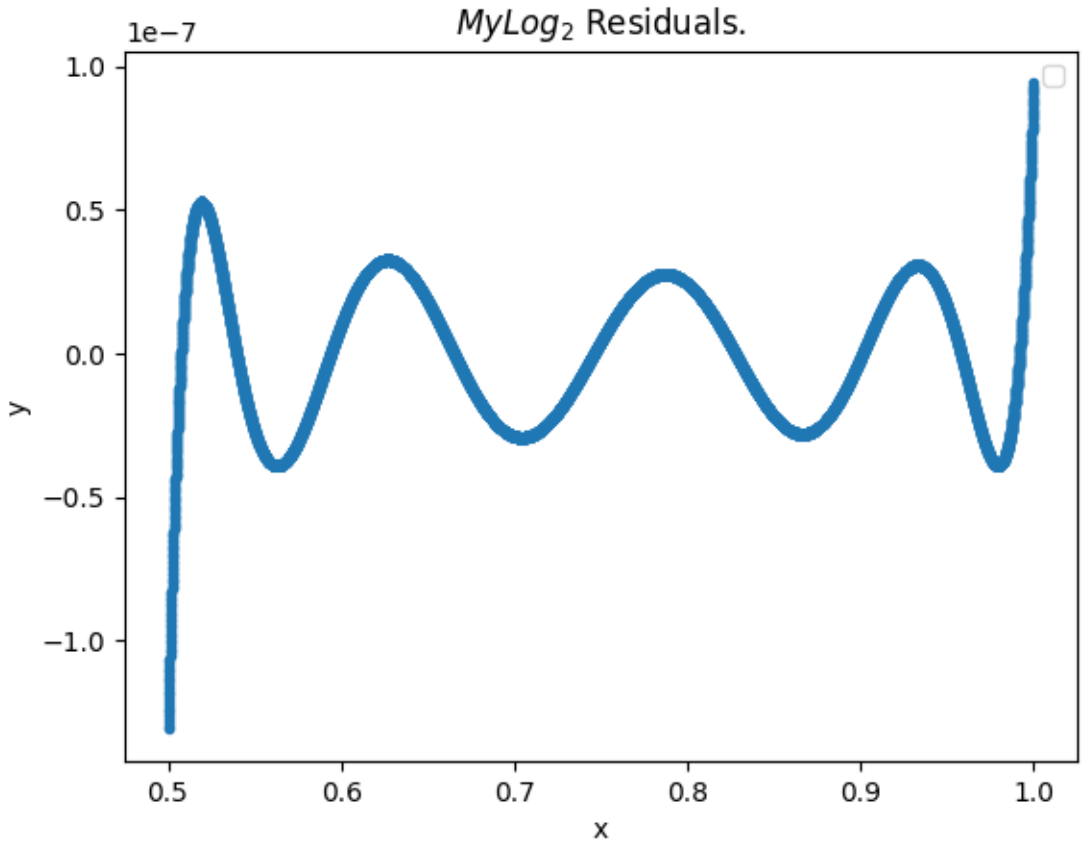

Figure 3: Residuals between the Chebyshev fit for the function and the actual log values. Despite the periodic nature of the residuals the error is quite small as expected, generally in the 10^{-7} range.

Now that I had a good approximation, with a maximum error less than $10e-6$. I created my natural log calculator in myLog2.

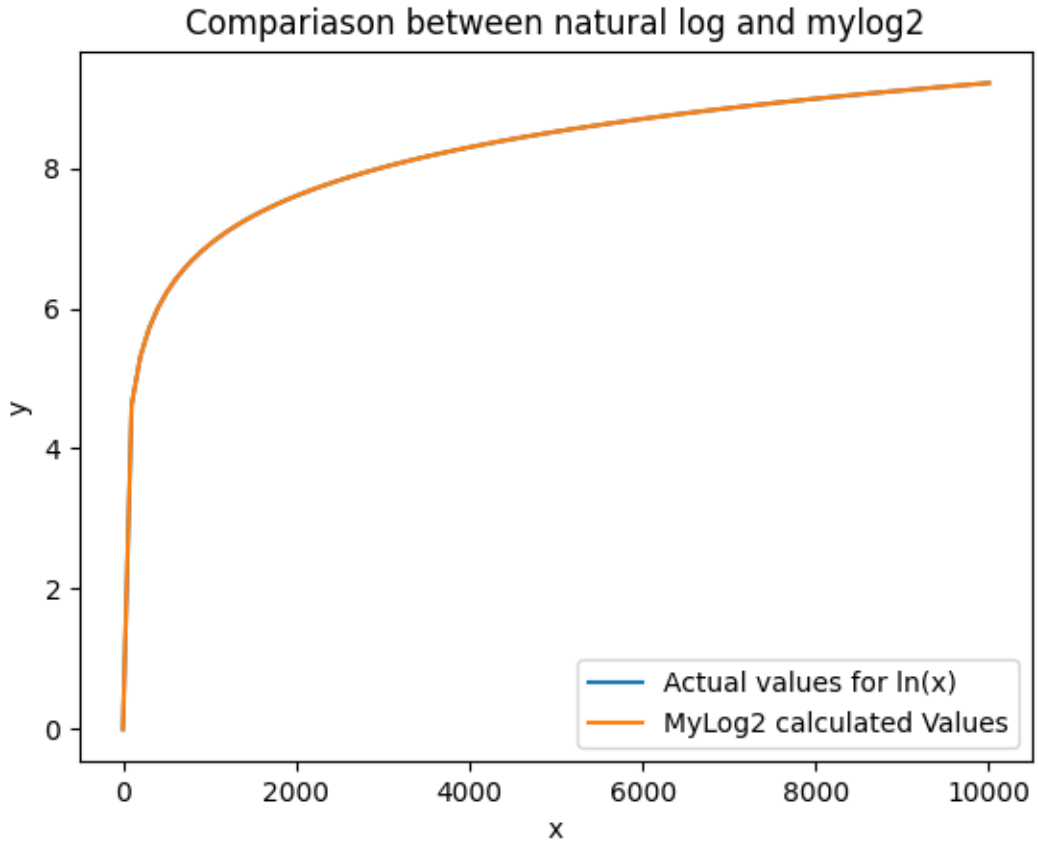

Figure 4: Figure showing the difference between the natural logarithm calculated by numpy and the myLog2 function, at first glance they appear to be identical

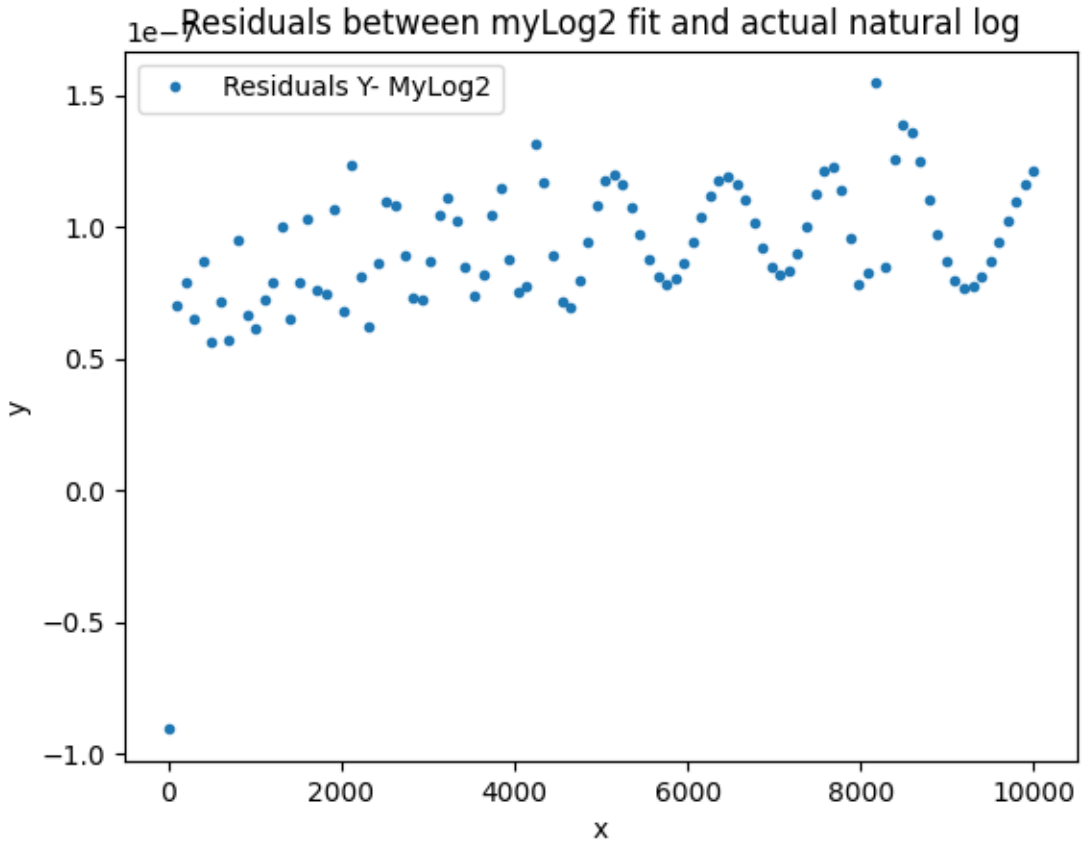

Figure 5: Residuals of the actual value of ln(x) compared to the myLog2 function. Since we made our error less that 10^-7 as you can see the associated errors in our natural logarithm are also in the regime of 10^-7

As can be seen in the above two figure the function myLog2 does a very good job of calculating the natural logarithm for a positive x. The error is bounded by the error in our log base 2 calculator which we determine from the chebyshev polynomial fit. The error is then carried forward into our natural logarithm calculator which shows the residuals to be in the regime of 10^{-7} which as desired is less than 10^{-6}.

I then did the exercise with the legendre polynomial as well. The maximum error for both of the methods is extremely close as well as the root mean square error. Additionally they have same orders of magnitude on the maximum error for both RMS and log2. This is because legendre and chebyshev polynomials form an orthogonal basis for a degree n polynomial. Since we used the same degree in both they both (most likely) form the exact same polynomial for our fit.