**Amanda Di Nitto**
**268609696**
**Homework 1**
**PHYS-512**
**Prof. Jon Seivers**
**Due: September 18th 2020**

**Problem 1A:**

In order to obtain the derivative, I started with the equation we saw in class for one of the ways we can take a simple numerical derivative;

$$f' = \frac{f(x+\delta)-f(x-\delta)}{2\delta}$$

I created two versions of the equation, one for $(x \pm \delta)$ and another for $(x \pm 2\delta)$. From there I subtracted the two equations to arrive at my estimated derivative equation. The reason for this is that by doing a Taylor expansion for *f* at each of the points, the second order terms will cancel out leaving only the lower terms (given by the assumption that we only went to the second order term for the Taylor expansion). This left the series of equations:

$$f(x+\delta) - f(x-\delta) = 2\delta f'$$
$$f(x+2\delta) - f(x-2\delta) = 4\delta f'$$

Therefore, my final subtraction was:

$$\frac{4\delta f'}{2\delta} - \frac{2\delta f'}{2\delta} = f'$$

**Problem 1B:**

Based on the fact that is should be single precision I decided to make my delta equal to $10^{-4}$. The single precision is due to there being no higher order terms to deal with in the derivative as they cancelled out and the equation was cut off after the 2nd order. This means the truncation error should not be a big problem leaving only the roundoff error to deal with. Testing this for the first function, $f = \exp(x)$, I received very close values, with a difference of only $1.56\ x\ 10^{-8}$. Testing with the second function, $f = \exp(0.01x)$, was not nearly as precise, with a difference of 0.00995 this time meaning I had a change of 5 orders of magnitude in precision. The reason for this could be due to the way the estimate was calculated as it did discount higher order Taylor terms and therefore lost some precision. Since the second derivative had a much smaller exponent and therefore a smaller numerical answer for the derivative, a discrepancy in numbers at the 3rd decimal point and on could have a large effect on the final answer's precision.

**Problem 2:**

For this problem I chose to use the cubic interpolation instead of a polynomial since the data had some small wiggles when focusing on the region at the beginning of the list (for temperature values up to 60). As with the example in class, the cubic interpolation would calculate to polynomial fit for each point, which can be slower but since the dataset we are using was not that large it seemed like the better option to be a bit more precise. Looking a polynomial fit for a group of 4 points at a time allowed for a better fit in the wavy section for temperatures around 20. The first code was the interpolation for both the full lakeshore text and a reduced version [figure 1]; the reduced one aimed to focus on the area where the curve is since it is a very noisy area and I wanted to be able to see how close the interpolation was in that section.

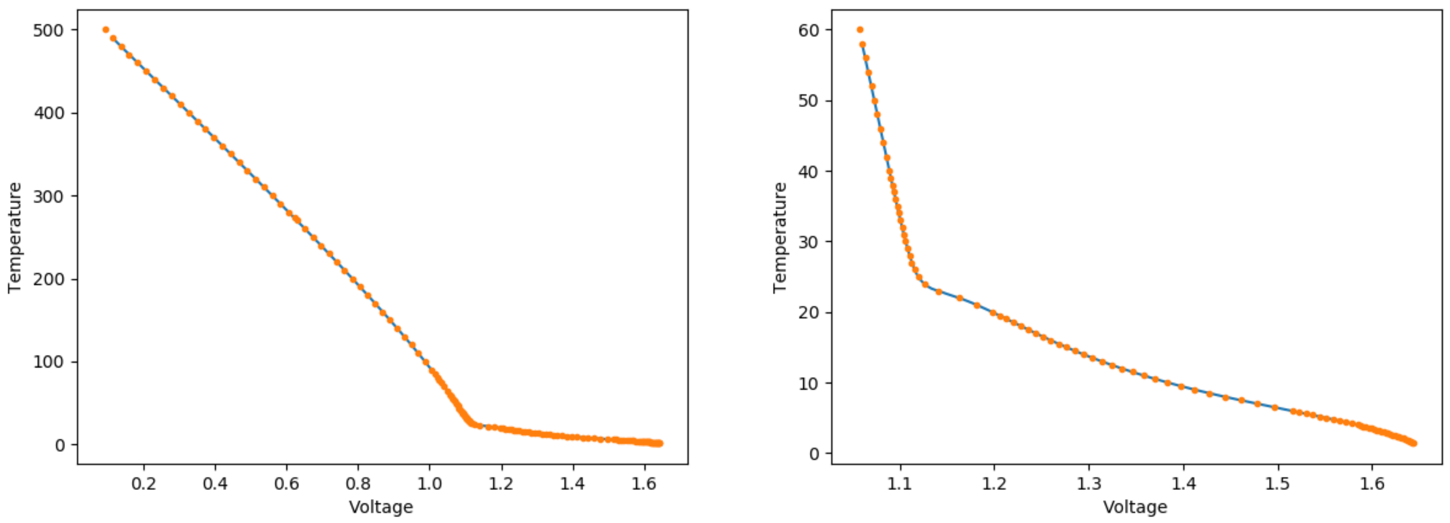

Figure 1: Interpolation of temperature for full and reduced lakeshore.txt

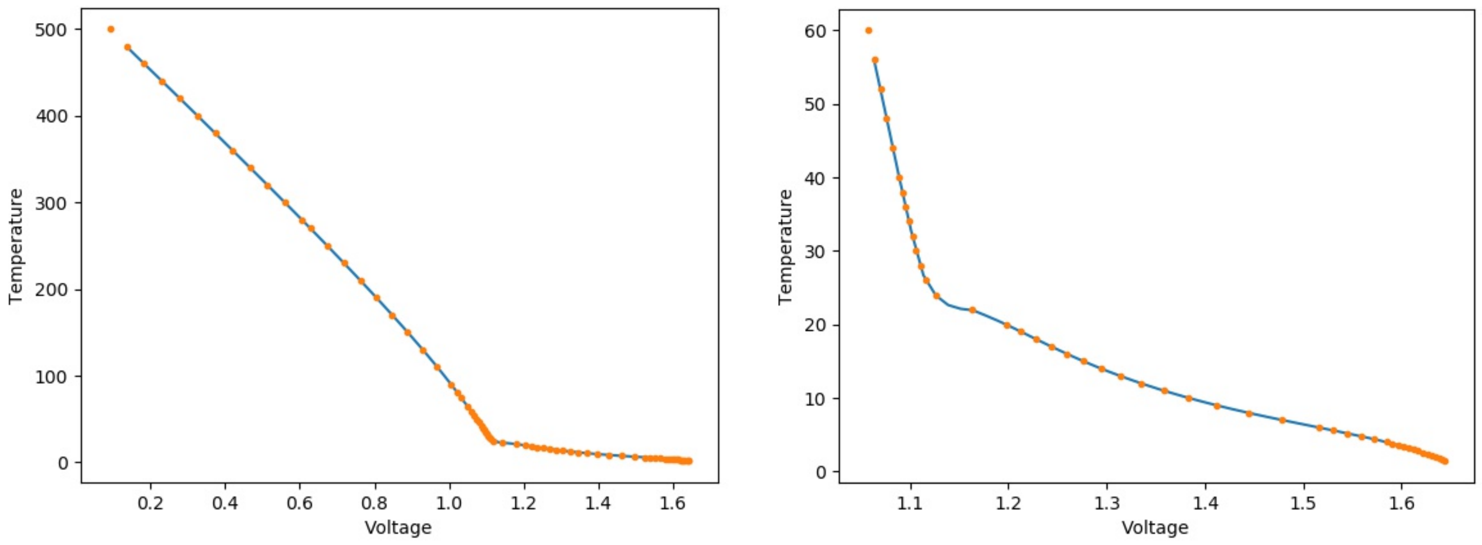

Figure 2: Error interpolation of temperature for full and reduced lakeshore.txt

The second code the approximate error was calculated [Figure 2]. This was done by splitting the array into even and odd points for both voltage and temperature and then running the same interpolation code on only the even array. Afterwards, the interpolated values for the temperature were compared to the actual values from the odd array. It was run again for both the full text and shortened one and the accuracy was much better on the shortened lakeshore text. The full text had a standard deviation of 72.99 whereas the shortened text had only 5.48. It is possible that by shortening the table I eliminated the area where there were more problems creating a cubic fit.

**Problem 3:**

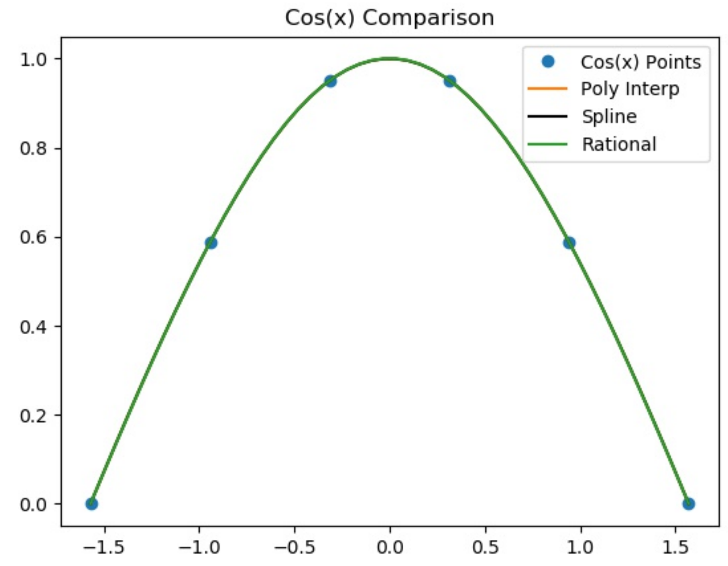

Figure 3: Cos(x) interpolation comparisons

I decide to use six points for my interpolation of cosine [Figure 3]. All three had the same level of accuracy, the only time a difference was noticeable was when the number of x values for the polynomial interpolation was decreased (i.e. the x value in my code). Changing up the values for m and n in the rational fit did not changed anything either. Overall the different interpolators worked to the same level of accuracy for the cosine function. There was, however a difference for the Lorentzian. For the Polynomial and Spline interpolations there was not a significant difference [Figure 4], same as earlier but there difference does occur on the rational interpolation was introduced. The expected error for the rational function was supposed to be small as a Lorentz function itself is a rational function. Error would be around $10^{-8}$ since our affected term is in the denominator and only varies between -1 and 1, meaning we need double precision. For n=4 and m=5 the rational interpolation blew up, meaning the shape of the function was not terribly different but the value for the peak was much higher than the actual value [Figure 5a].

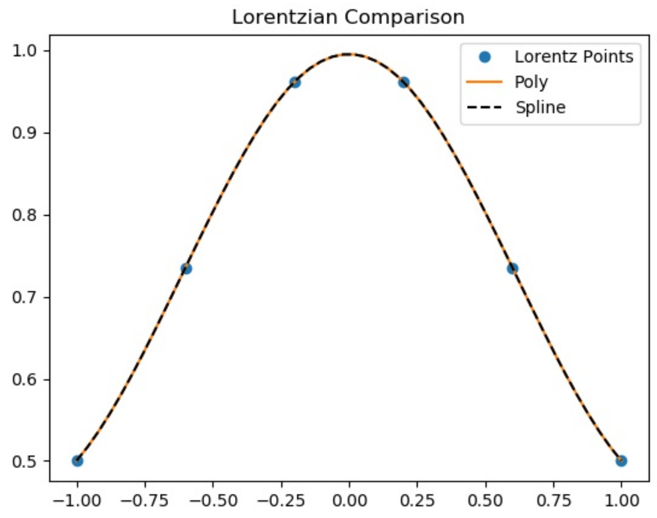

Figure 4: Lorentz function for polynomial and spline interpolation only.

Upon changing it from np.linalg.inv to np.linalg.pinv the rational interpolation fits much better and does exactly what was expected initially. It is even more accurate than the other interpolations as it can continue the function beyond the points provided, whereas the other interpolators cannot, and they do not go past the first and last points. This change is due to the fact that for a singular matrix the inverse cannot be calculated, but the np.lianlg.pinv command comes up with a pseudo inverse matrix. This is visible if we play around with the n and m values. For example, at n=2 and m=3, no matter which command we use the results are identical; this is because the inverse matrix exists therefore both commands will have the same inverse matrix output. For a value such as n=4 and m=5 [Figure 5a] the matrix is singular so the inverse cannot be calculated. In that case, pinv gives an output matrix which is based on what the function should look like if the inverse matrix was calculable [Figure 5b].

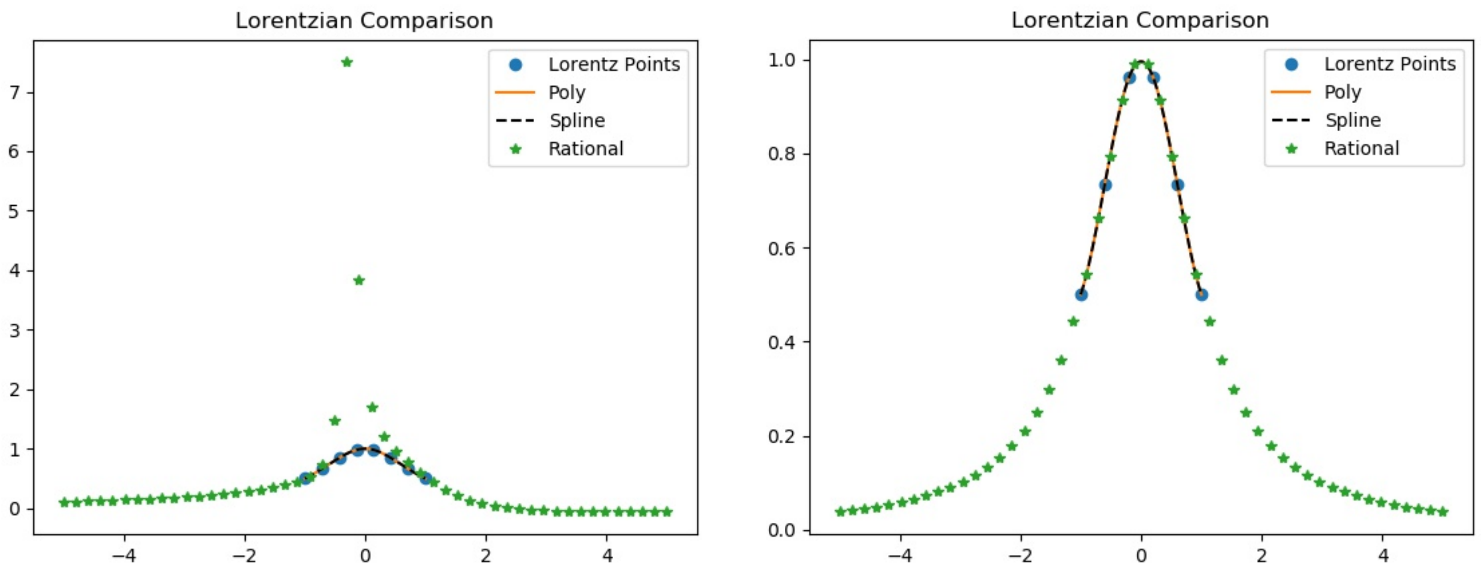

Figure 5a: Interpolation comparison with rational fit using np.lialg.inv at n=4, m=5.
Figure 5b: Comparison using np.linalg.pinv at n=4, m=5.

Problem 4:

This integral can be set up using Gauss' Law. Being that it is a shell of charge the integral will need to be split up into 2 sections: $R < z$ and $R \geq z$. The initial set up will lead to the following integral:

$$E_z = \frac{R^2\sigma}{2\epsilon_0}\int_0^{\pi} \frac{sin\theta(z - Rcos\theta)}{(R^2 + z^2 - 2Rzcos\theta)^{3/2}} d\theta$$

For the code, the constant in front of the integral was ignored since it will not have an affect on the function and its relationship to z other than scaling it. A value of 1 was chosen for the radius and z was scaled from 0 to 0.99 for the inside of the shell, and 1.1 to 2.0 for outside the shell. As the charge is entirely on the surface of the shell, Gauss' law dictates that the electric field inside the shell should be zero then. There should be a discontinuity at $R = z$, and finally the electric field will decrease quadratically as the z value increases. When running the integration routine, as expected, the routine did not run at *z=1.0* and gave back a runtime error. On the other hand Scipy's quad was able to integrate it and did not run into any errors. One of the interesting things that did occur was in the results obtained for *z* outside the shell. The largest value for the electric field should be at the point where the radius and *z* are equal (i.e. the discontinuity) but the first few values after the shell (*z=1.1, 1.2* and *1.3*) gave results bigger than that of the result at the shell. This could be due to the discontinuity because the values were identical whether it was run via quad or the routine. The trend was correct but it was interesting that there would be an initial increase. Another interesting point is the fact that the precision for the routine when running for inside the shell was not nearly as good as quad. The result is supposed to be essentially zero (and therefor results that are extremely small work). The quad command gave values with orders of magnitude near -17, meaning all the results were essentially zero but the routine gave values of -7 instead. This is not nearly as small and therefore is not essentially the same thing as zero. This is not apparent in the graph [Figure 6], since any value that small could not be displayed in the same graph as the other values (meaning the outside the shell results). The star point on the graph represents the discontinuity calculated by quad. The black lines are the values calculated by the integral routine and the coloured points are those calculated by quad.

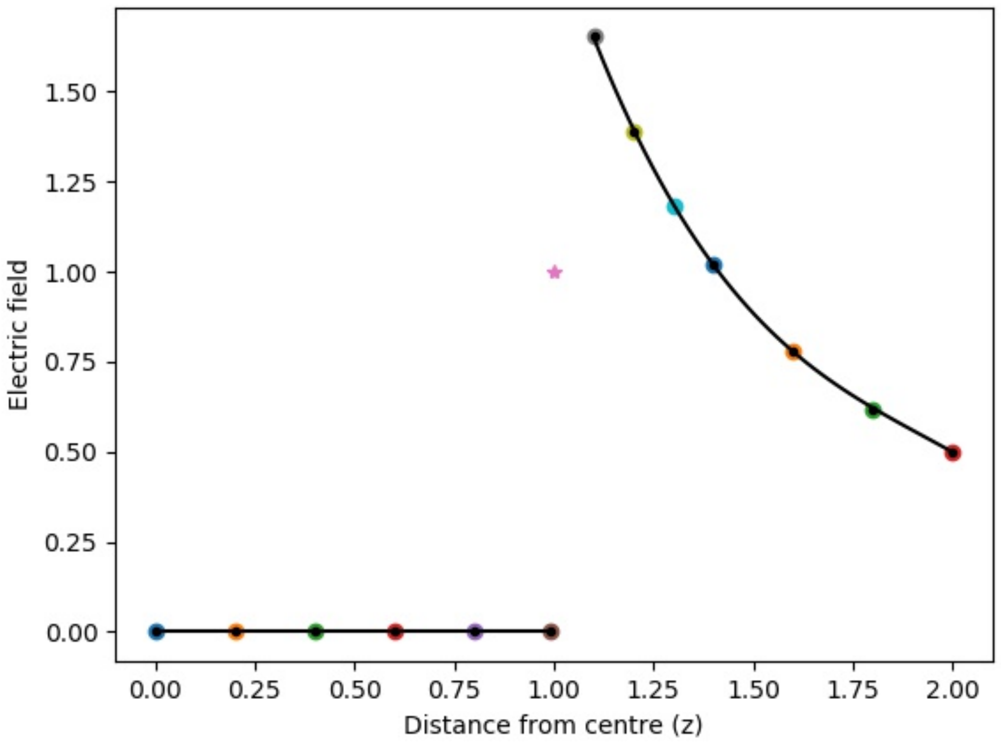

Figure 6: Comparison between integral routine and scipy.quad calculations for the electric field.