Phys 512 Assignment 1

Andrew Lewis

September 2022

Problem 1- Numerical Differentiation with Four Points

a)

First, we are allowing ourselves 4 function evaluations at f(x+h), f(x-h), f(x+2h) and f(x-2h), (i have replaced δ with h as it is easier to type). We will then taylor expand all of our equations to the 5th derivative (as we will motivate later).

$$f(x+h) = f(x) + hf'(x) + \frac{h^2 f''(x)}{2!} + \frac{h^3 f'''(x)}{3!} + \frac{h^4 f''''(x)}{4!} + \frac{h^5 f^{(5)}(x)}{5!} \tag{1}$$

$$f(x-h) = f(x) - hf'(x) + \frac{h^2 f''(x)}{2!} - \frac{h^3 f'''(x)}{3!} + \frac{h^4 f''''(x)}{4!} - \frac{h^5 f^{(5)}(x)}{5!} \tag{2}$$

$$f(x+2h) = f(x) + 2hf'(x) + \frac{4h^2 f''(x)}{2!} + \frac{8h^3 f'''(x)}{3!} + \frac{16h^4 f''''(x)}{4!} + \frac{32h^5 f^{(5)}(x)}{5!} \tag{3}$$

$$f(x-2h) = f(x) - 2hf'(x) + \frac{4h^2 f''(x)}{2!} - \frac{8h^3 f'''(x)}{3!} + \frac{16h^4 f''''(x)}{4!} - \frac{32h^5 f^{(5)}(x)}{5!} \tag{4}$$

Subtracting equation (2) from equation (1) gives:

$$f(x+h) - f(x-h) = 2hf'(x) + 2h^3 \frac{f'''(x)}{3!} + 2h^5 \frac{f^{(5)}(x)}{5!} \tag{5}$$

Similarly, subtracting equation (4) from equation (3) gives:

$$f(x+2h) - f(x-2h) = 4hf'(x) + 16h^3 \frac{f'''(x)}{3!} + 64h^5 \frac{f^{(5)}(x)}{5!} \tag{6}$$

next- multiplying equation (5) by 8 and subtracting (5) from (6)

$$f(x+2h) - f(x-2h) - 8f(x+h) + 8f(x-h) = -12hf'(x) + 48h^5 \frac{f^{(5)}(x)}{5!} \tag{7}$$

multiplying by -1 and dividing by 12 h yields the following result

$$\boxed{\frac{f(x-2h) - 8f(x-h) + 8f(x+h) - f(x+2h)}{12h} = f'(x) - 4h^4 \frac{f^{(5)}(x)}{5!}}$$

This will be our expression for the numerical derivative using 4 points. The $4h^4 \frac{f^{(5)}(x)}{5!}$ term I have included is simply the truncation error.

b)

Now that we have our operator for the derivative we should check what our h should be to minimize the error. The error will contain two terms.

The truncation error, given by $4h^4\frac{-f^{(5)}(x)}{5!}$ and the round off error, which according to numerical recipes we can write as the possible difference in the function evaluation

$$\frac{f(x)*\epsilon}{dx}$$

where ϵ is the machine precision/ the smallest number that can be represented by the computer. By replacing h with dx (once again for my own gratification) we can write the equation for the error as:

$$\text{error} = dx^4\frac{f^{(5)}(x)}{30} - \frac{f(x)\epsilon}{dx}$$

taking the derivative and setting to 0 (a classic minimization procedure)

$$0 = \frac{4}{30}dx^3 f^{(5)}(x) - \frac{f(x)\epsilon}{dx^2}$$

rearranging

$$\left(\frac{7.5f(x)\epsilon}{f^{(5)}(x)}\right)^{(1/5)} = dx$$

This will be our expression for the optimal δ or in my case, dx. We will test this by graphing the difference between the analytical derivative and our dx on a log log scale (thanks Jon!). We will be able to see if the estimate for optimal dx is good if it minimizes the error in between the numerical derivative and the analytical derivative that we knew ahead of time.
For e^x

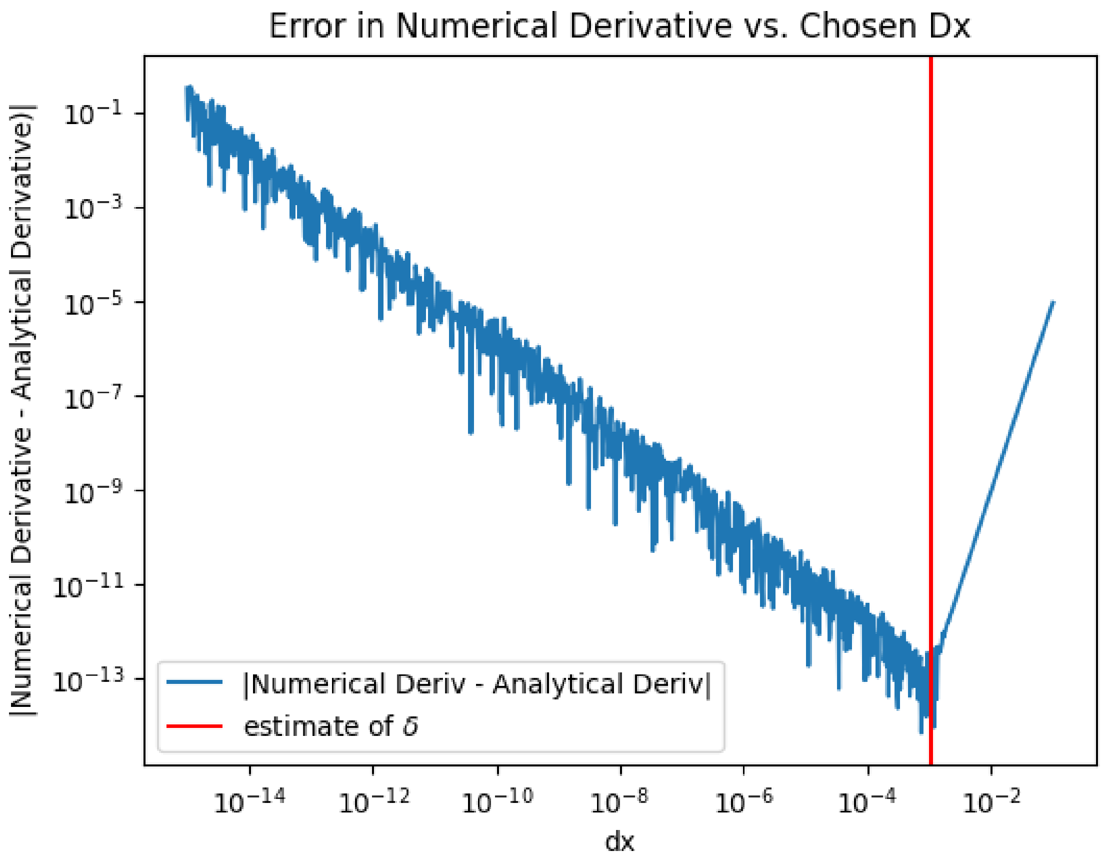

Figure 1: Error in the numerical derivative for e^x. The blue curve is the absolute value between the numerical derivative and the analytical derivative . The error on the left side of the optimal dx is dominated by round off error while the error on the right hand side is dominated by truncation error. The estimate of the optimal dx is shown in red.

As is apparent by figure 1, our estimate for optimal dx, shown in red, is at the minimum of the error in between the numerical and analytical derivative.
Now for $e^{0.01x}$

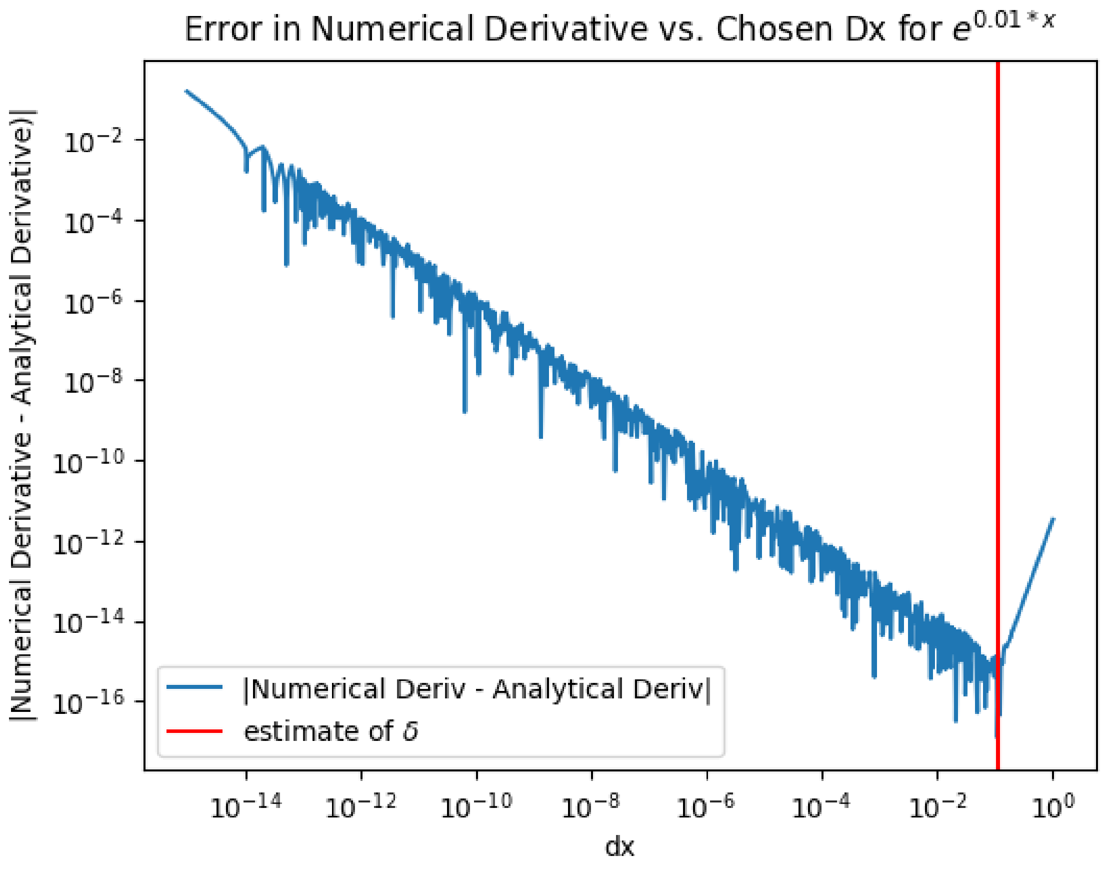

Figure 2: Error in the numerical derivative for $e^0.01x$. The blue curve is the absolute value between the numerical derivative and the analytical derivative . The error on the left side of the optimal dx is dominated by round off error while the error on the right hand side is dominated by truncation error. The estimate of the optimal dx is shown in red.

As is apparent here, the estimate of optimal δ is a good choice for minimizing the error in the numerical derivative, regardless of the function as long as the fifth analytical derivative is known.

$\boxed{\text{Therefore, our estimate of } \delta \text{ minimizes the error in the derivative compared to a naive approach like dx =.001.}}$

Problem 2- nDiff

Please see attached code in the GitHub repo entitled PS1-2.py

How did i write my numerical differentiator? Instead of trying to do a third derivative evaluation numerically, requiring many calls to estimate dx, I simply decreased an initial value of dx until the error stopped getting smaller. By dividing the dx by $10^{(1/9)}$ at each step and comparing the difference in the previous numerical differentiation, we can approximately get to the point where the difference in numerical derivatives, given a dx, are at their smallest- implying an optimal dx.

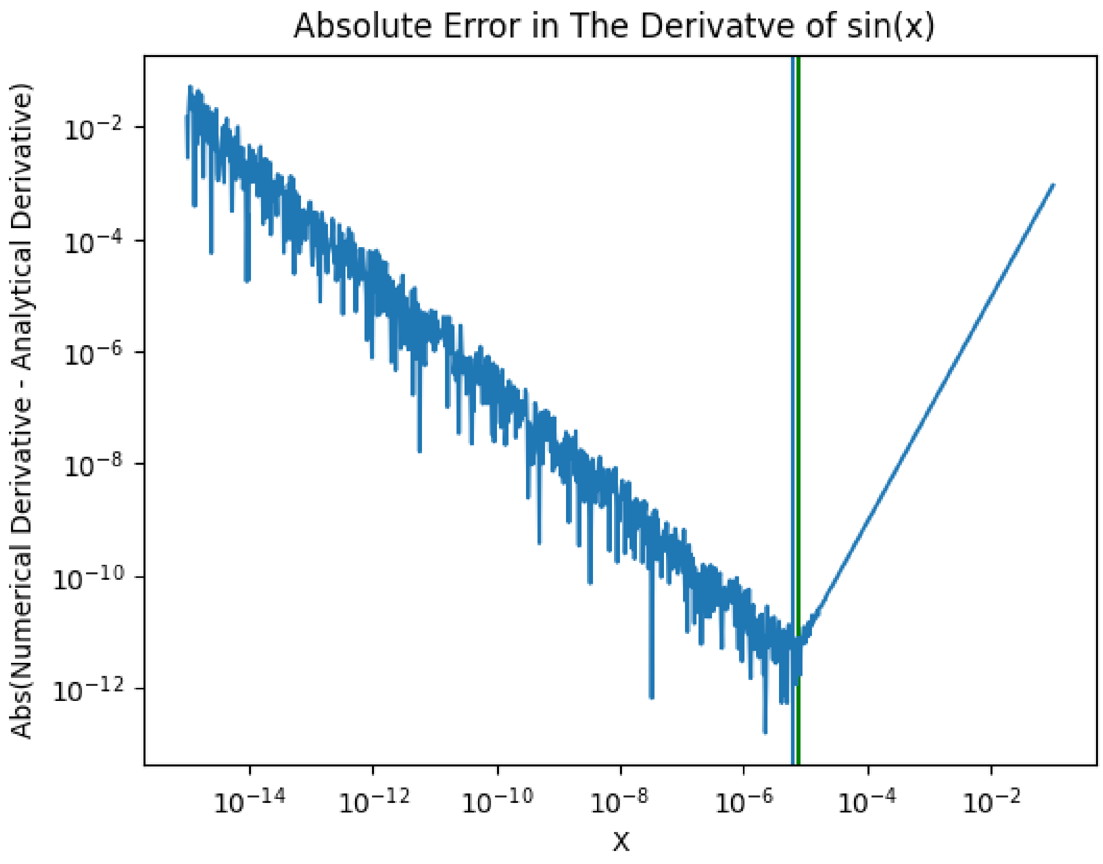

Figure 3: Comparison of the dx given by the expression given knowledge of the derivative of the function and its derivative (Blue) and the derivative computed using my error minimization method (Green). As one can see, the two estimates of dx are almost identical.

As can be seen in the above figure the two estimates in dx are almost identical. Additionally, when comparing the estimated error to the actual function error, the orders of magnitude are identical, giving a rough estimate of the error.

Problem 3 - Lakeshore Diodes

For the code, please see the github repo PS1-3.

I will motivate the thinking behind the interpolation method i chose and the error in the returned value. I tried cubic spline, polynomial, and rational function interpolation on the dataset. Ultimately, I determined that the rational function returned the least error on interpolation through observing the residuals of each function. For coding the rational function I stole Jon's code from class. For the order of the function i chose n = 6 and m =5. This produced a fairly close fit to the data as can be seen in figures 4 and 5.

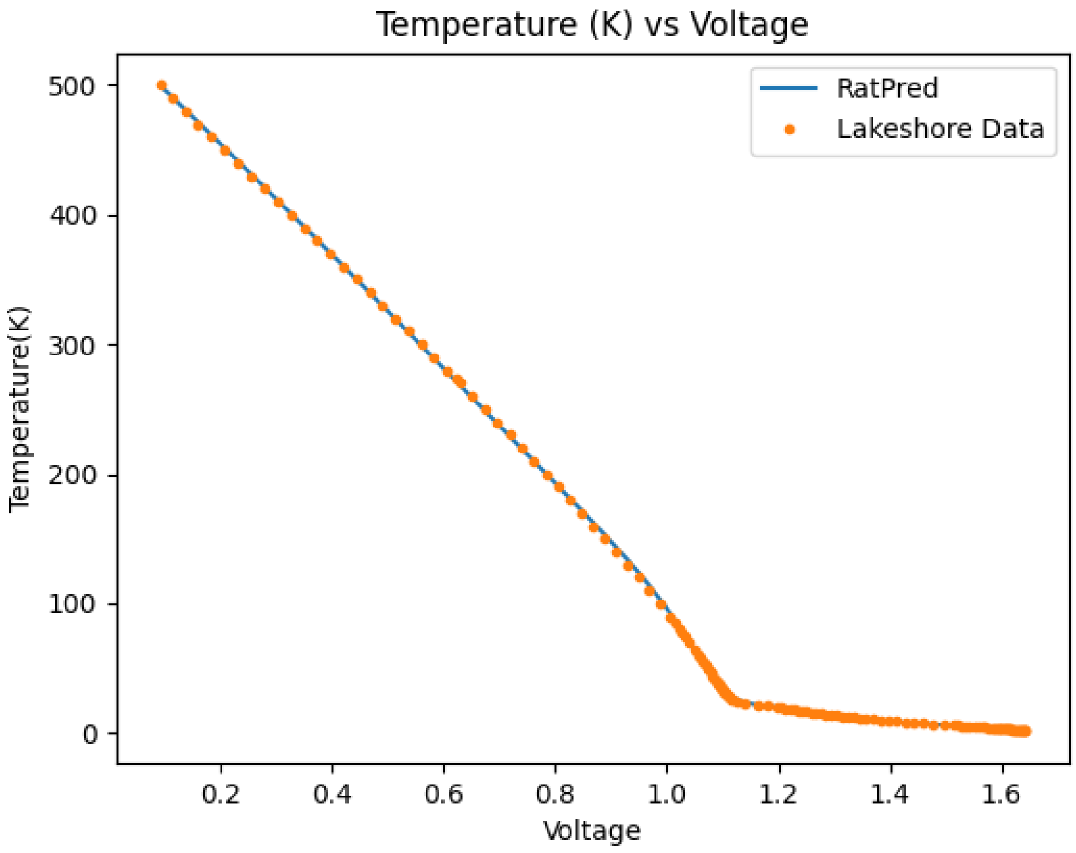

Figure 4: Plot of the rational function fit and raw data

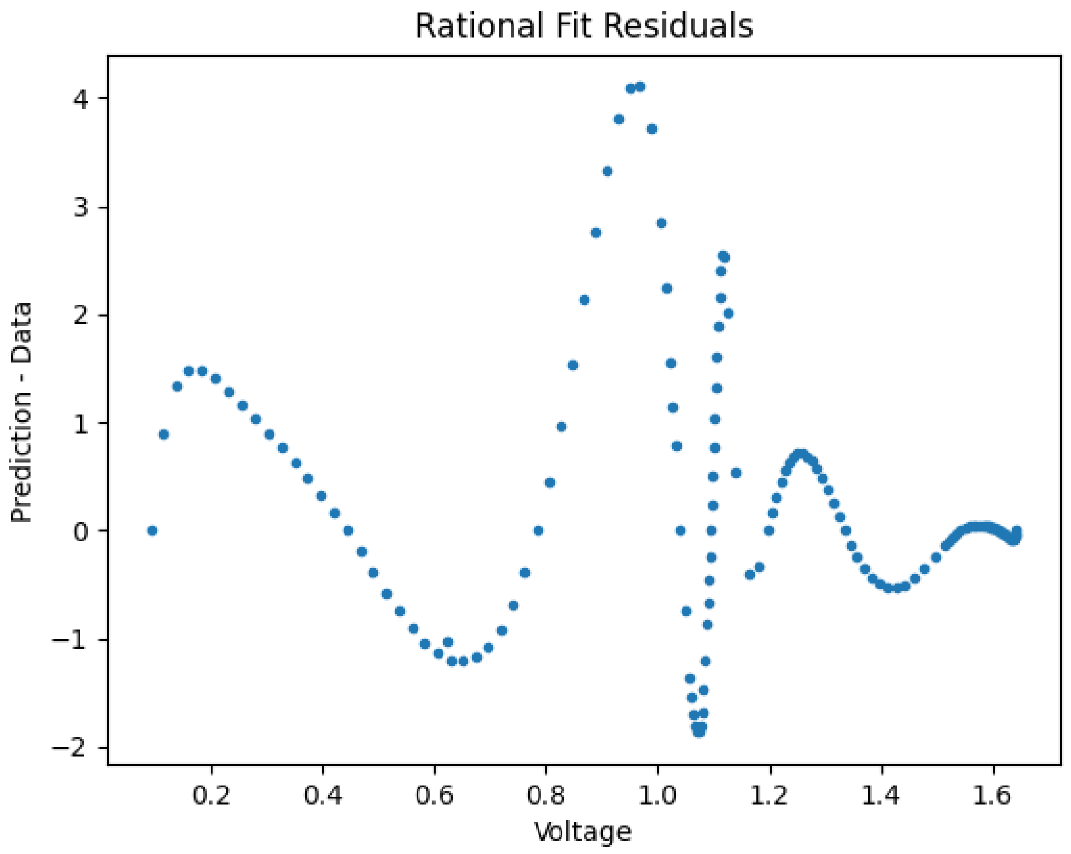

Figure 5: Plot of the rational function residuals. There is some periodicity to the residuals suggesting that the rational function doesn't perfectly fit the data. However, given that the temperatures are in 100s of kelvin, and the maximum error is around 4, the average error should be around 2-3%, not bad!

Now that we have an interpolation of the lakeshore diode data, we can write code to estimate the temperature given a voltage and return the estimate with error. The estimate is given by evaluating the rational function at a given voltage. The error is given by taking the two neighboring data points, taking the absolute error between the data point and the interpolation, and adding them. If the voltage extrapolates the data set, the error is simply given by multiplying the error in the final data point by 2. While this is by no means an exhaustive definition of the error it gives a good idea of the error if the voltage interpolates the lakeshore data and keeps all of the lakeshore data within the error bars.

In the code you will see that it supports the voltage being an number or an array.

Problem 4 Interpolation of Functions

First we will compare the accuracy of the polynomial, cubic spline and rational function interpolation on the cosine function. To make the comparison fair, the interpolations used the same 6 points. The polynomial was of degree 3. The rational function used arguments of n=4 and m=3.

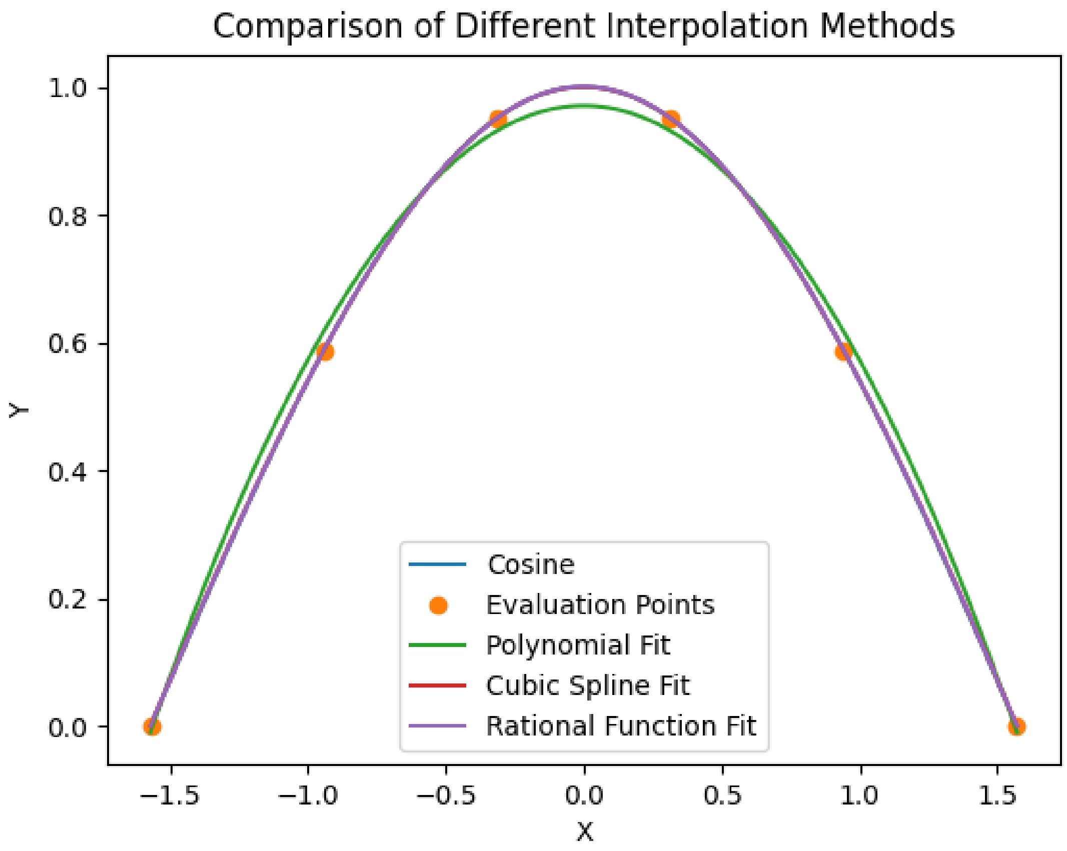

Figure 6: Plot of the different methods of interpolations. The evaluation points are shown by dots in orange while the different interpolation functions are shown in different colors. All 3 methods appear to do a decent job at interpolating the function, but we'll see what the residuals say.

Upon first glance, it appears that all 3 methods do a good job of interpolating the cosine function between $-\pi/2$ and $\pi/2$. But to get a better idea, we will look at the residuals. The residuals are given by:

$$f(x) - y(x) \tag{8}$$

Where f(x) is the interpolation function and y is the real value at x.

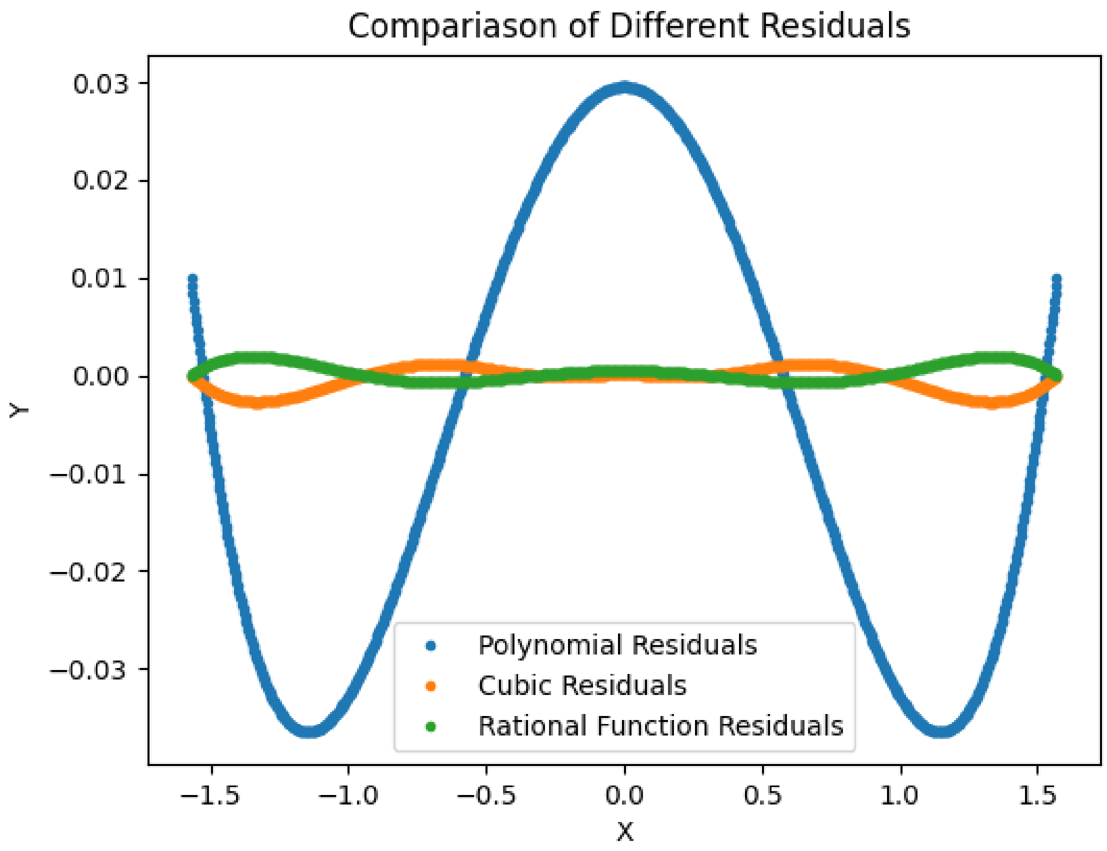

Figure 7: Residuals of the three interpolation methods. All 3 interpolation methods oscillate around the function, however the polynomial function oscillates the most by a few orders of magnitude!.

As can be seen from the figure, the cubic spline and the rational function do a similar job of interpolating the cosine function where the polynomial does a much worst job making it a poor option to interpolate a cosine.

Now we will try interpolating a lorentzian.

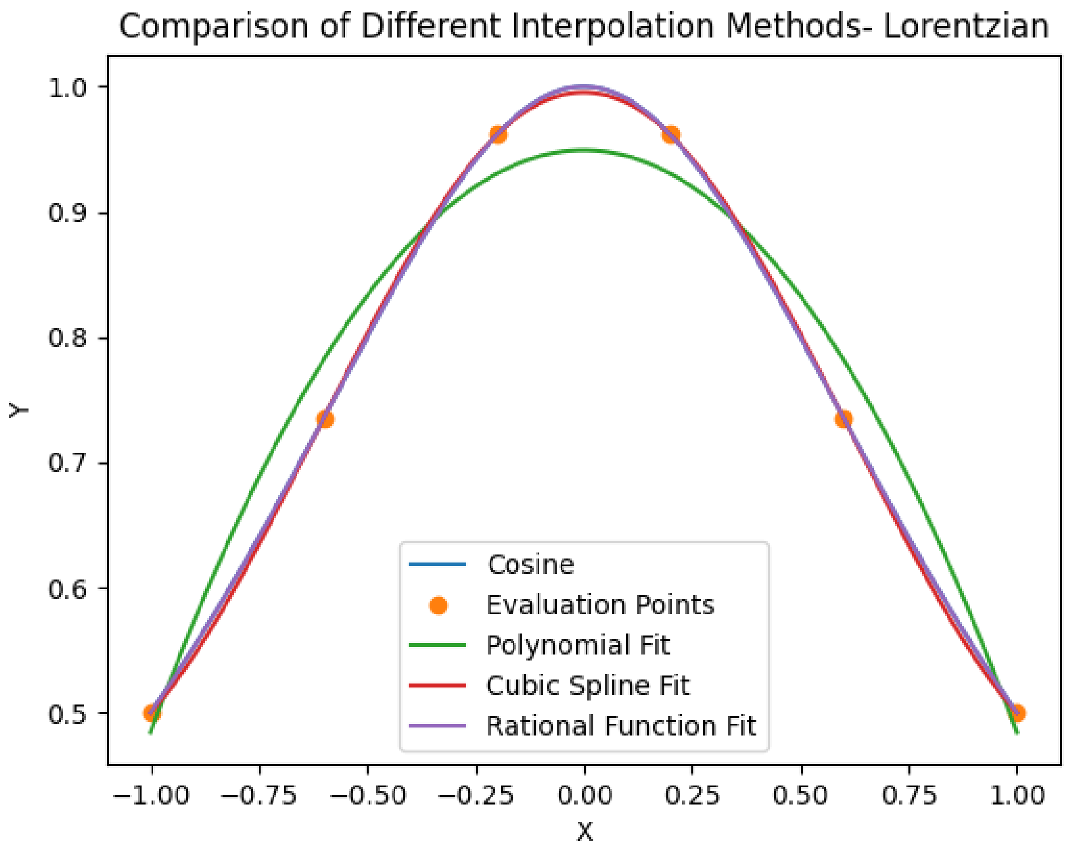

Figure 8: Interpolation of a Lorentzian with three different methods. The rational function does the best job for reasons we will discuss later.

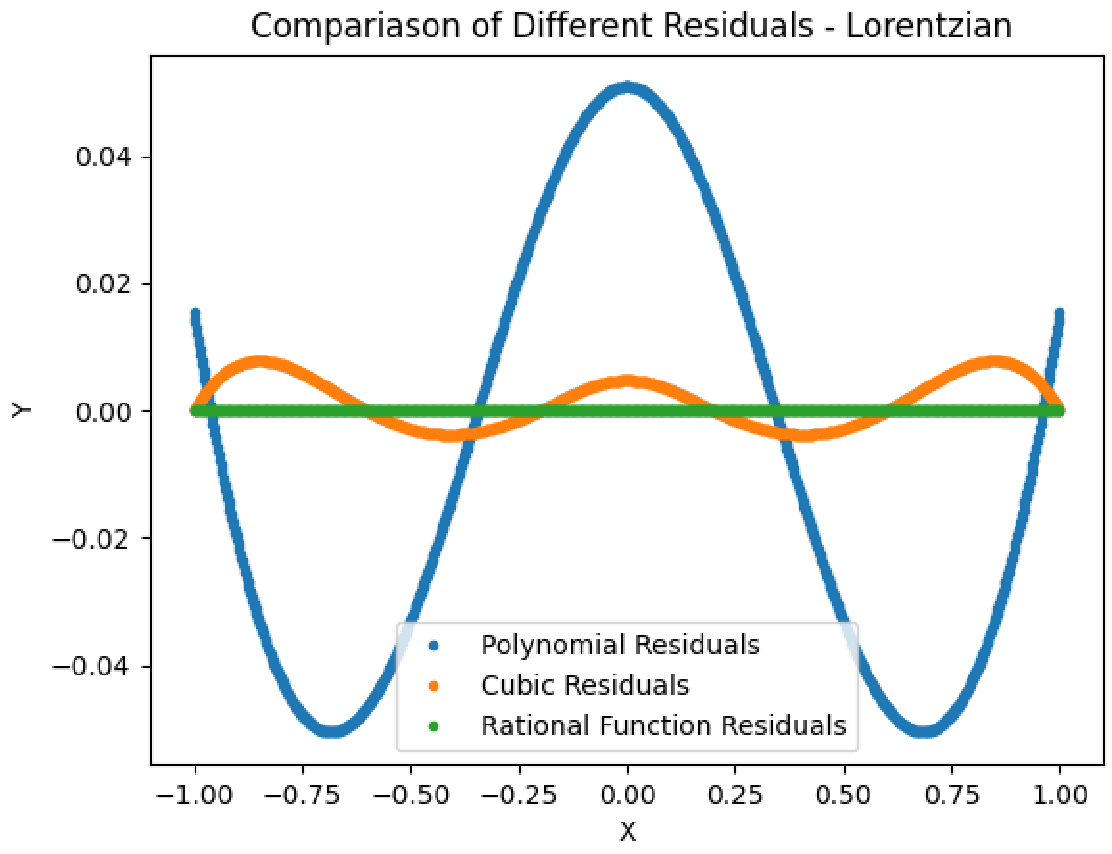

Figure 9: The residuals for the three different methods of interpolating the Lorentzian, as expected the rational function has no error, for reasons we will discuss later

After looking at the residuals we see some (expected) error in the polynomial and cubic spline fits, however we see no error in the rational function fit! why is this? This is because the Lorentzian is a ratio of two polynomials, exactly like the rational function. Therefore the rational function can perfectly interpolate the Lorentzian, now lets move to higher powers of the rational function. My expectation is that nothing should change as it should still be a ratio of two polynomials, but lets see what happens.

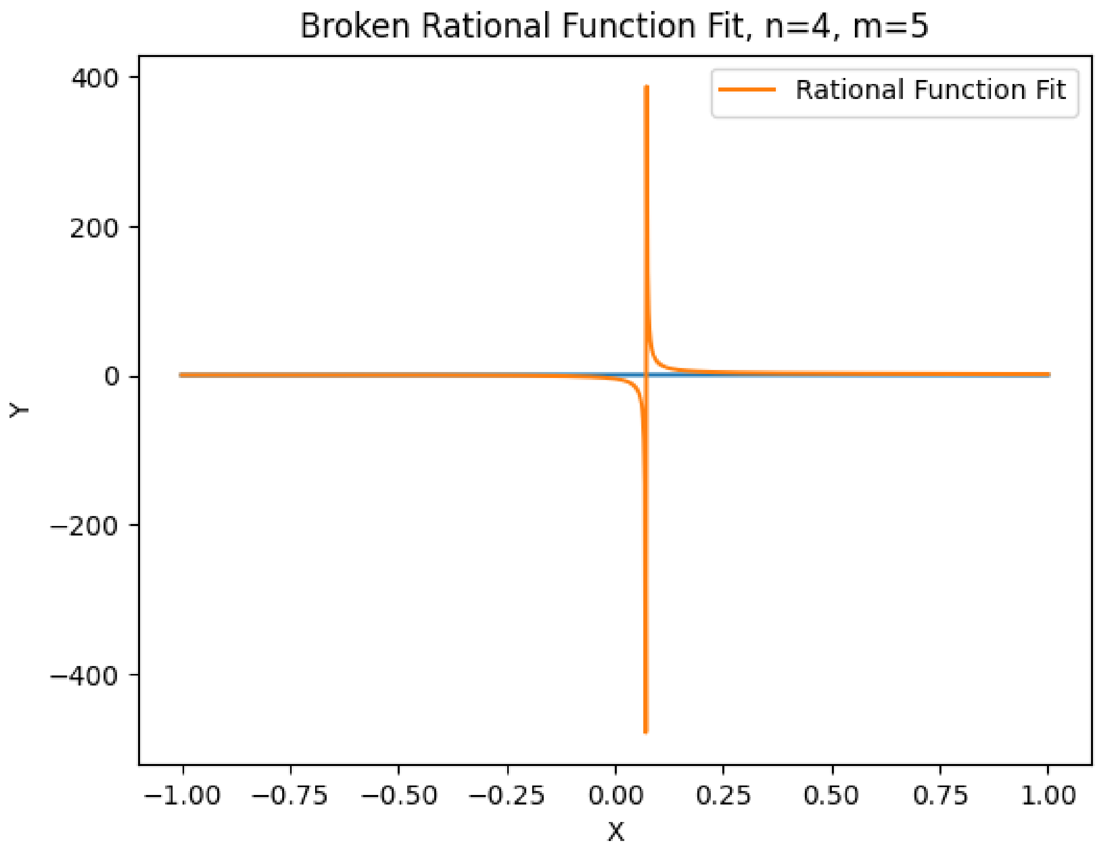

Figure 10: That doesn't look like a Lorentzian, uh oh.... what happened!?!

As we can see here, something disastrous has happened when we try to go to higher orders of n and m, but why has this occurred? Well the reason this has broken has to do with the np.inv function. If the matrix we are trying to invert is singular it will not has an inverse matrix therefore breaking the rational polynomial fit. Additionally, as the values get closer and closer to zero we don't actually reach zero. Therefore, when we divide a number by this very small number it shoots off to infinity, instead of being undefined. If we switch to pinv we can fix it as it will replace these very small numbers with zero fixing the interpolation.