

Assignment 1

By Christian Mauffette Denis

For PHYS-512

Question 1

a)

We want to compute the derivative using the given 4 points. We know that

$$f(x + \delta) = f(x) + f'(x)\delta + \frac{f''(x)}{2}\delta^2 + \frac{f'''(x)}{6}\delta^3 + \frac{f^{(4)}(x)}{24}\delta^4 + \frac{f^{(5)}(x)}{120}\delta^5 + \dots$$

$$f(x - \delta) = f(x) - f'(x)\delta + \frac{f''(x)}{2}\delta^2 - \frac{f'''(x)}{6}\delta^3 + \frac{f^{(4)}(x)}{24}\delta^4 - \frac{f^{(5)}(x)}{120}\delta^5 + \dots$$

$$f(x + 2\delta) = f(x) + 2f'(x)\delta + \frac{4f''(x)}{2}\delta^2 + \frac{8f'''(x)}{6}\delta^3 + \frac{16f^{(4)}(x)}{24}\delta^4 + \frac{32f^{(5)}(x)}{120}\delta^5 + \dots$$

$$f(x - 2\delta) = f(x) - 2f'(x)\delta + \frac{4f''(x)}{2}\delta^2 - \frac{8f'''(x)}{6}\delta^3 + \frac{16f^{(4)}(x)}{24}\delta^4 - \frac{32f^{(5)}(x)}{120}\delta^5 + \dots$$

If we subtract the first two expressions we get:

$$\begin{aligned} f(x + \delta) - f(x - \delta) &= 2f'(x)\delta + \frac{2f'''(x)}{6}\delta^3 + \frac{2f^{(5)}(x)}{120}\delta^5 + \dots \\ &= 2f'(x)\delta + \frac{f'''(x)}{3}\delta^3 + \frac{f^{(5)}(x)}{60}\delta^5 + \dots \end{aligned}$$

Now we subtract the two last expressions:

$$f(x + 2\delta) - f(x - 2\delta) = 4f'(x)\delta + \frac{8f'''(x)}{3}\delta^3 + \frac{8f^{(5)}(x)}{15}\delta^5 + \dots$$

This means we can get rid of the 3rd order corrections by doing:

$$\frac{2}{3} \left((f(x + \delta) - f(x - \delta)) - \frac{1}{8}(f(x + 2\delta) - f(x - 2\delta)) \right) = f'(x)\delta - \frac{1}{30}f^{(5)}(x)\delta^5 + O(x^6)$$

Hence the error term is

$$\frac{1}{30}f^{(5)}(x)\delta^4$$

And the estimate will be

$$\frac{2}{3\delta} \left((f(x+\delta) - f(x-\delta)) - \frac{1}{8}(f(x+2\delta) - f(x-2\delta)) \right)$$

b)

If the roundoff error is ϵ , then the error is bounded by the truncation error and the machine error:

$$\frac{2}{3\delta} \left(\epsilon - (-\epsilon) - \frac{1}{8}((-\epsilon) - \epsilon) \right) + \frac{1}{30}f^{(5)}(x)\delta^4 = \text{error}$$

$$\frac{2}{3\delta} \left(2\epsilon - \frac{\epsilon}{4} \right) + \frac{1}{30}f^{(5)}(x)\delta^4 = \text{error}$$

$$\frac{7\epsilon}{6\delta} + \frac{1}{30}f^{(5)}(x)\delta^4 = \text{error}$$

Taking the derivative with respect to δ and setting it equal to 0 to minimize it we have:

$$\frac{d}{d\delta} \left(\frac{7\epsilon}{6\delta} + \frac{1}{30}f^{(5)}(x)\delta^4 \right) = 0$$

$$\frac{2\delta^3 k}{15} - \frac{7\epsilon}{6\delta^2} = 0$$

$$\Rightarrow \delta \approx \left(\frac{35\epsilon}{4f^{(5)}(x)} \right)^{1/5}$$

So, assuming machine precision is about 10^{-16} , then we have about

$$\delta \approx 10^{-16/5} \approx 10^{-3.2}$$

We can now try this with code by evaluating the given functions with our derivative. First we create the derivative taking function.

```

# take the derivative with a given step size
def deriv(func, x0, delta):
    pt1 = func(x0 + delta) - func(x0 - delta)
    pt2 = 1/8*(func(x0 + 2*delta) - func(x0 - 2*delta))
    return 2/(3*delta) * (pt1 - pt2)

```

With this function, we can pick a `x0` and then scan (logarithmically) through different `delta` values. For the function e^x whose derivative is evaluated at $x = 0$, this produces the following figure:

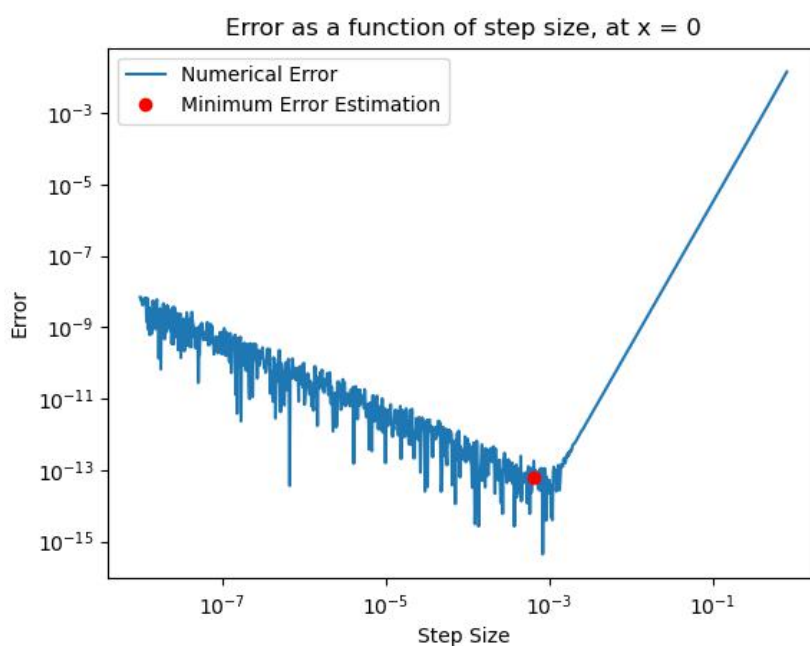


Figure 1: q1_error_plot1

We can see on it the estimated optimal error is indeed in the 10^{-3} ballpark. Also, it was assumed that the fifth derivative is roughly of order 1, which is exactly right for our specific function, since $\frac{d^5}{dx^5}e^x = e^x$.

Now we can try the same procedure, but for the $e^{0.01x}$ function. Again, we scan the different deltas, and produce the following plot:

For this plot the order of magnitude of the 5th derivative was in fact significant since $\frac{d^5}{dx^5}e^x = (0.01)^5 e^x$. Hence, the optimal step size had to be divided by 0.01 and after doing so, we see that it does indeed fall roughly on the minimum of the curve for the error.

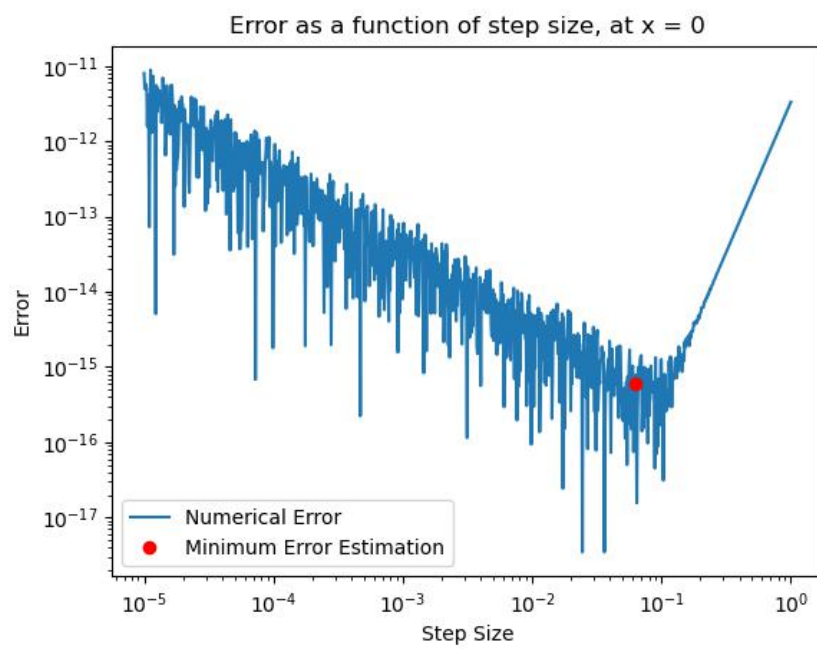


Figure 2: q1_error_plot2

Question 2

We now code a derivative taking function. We used the centered derivative for that. We simply use the formula

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

However, we must pick the appropriate Δx . If we look at the Taylor expansion of the previous expression, we have

$$f(x + \delta) = f(x) + f'(x)\delta + \frac{f''(x)}{2}\delta^2 + \frac{f'''(x)}{6}\delta^3 + \frac{f^{(4)}(x)}{24}\delta^4 + \frac{f^{(5)}(x)}{120}\delta^5 + \dots$$

$$f(x - \delta) = f(x) - f'(x)\delta + \frac{f''(x)}{2}\delta^2 - \frac{f'''(x)}{6}\delta^3 + \frac{f^{(4)}(x)}{24}\delta^4 - \frac{f^{(5)}(x)}{120}\delta^5 + \dots$$

$$\begin{aligned} \Rightarrow \frac{f(x + \Delta x) - f(x - \Delta x)}{2\Delta x} &= \frac{1}{2\Delta x} \left(2f'(x)\Delta x + \frac{f'''(x)}{3}\Delta x^3 \right) + \dots \\ &= f'(x) + \frac{f'''(x)}{6}\Delta x^2 + \dots \end{aligned}$$

Hence, the error is

$$\text{error} \approx \frac{\epsilon}{\Delta x} + \frac{f'''(x)}{6}\Delta x^2$$

If we minimize it with respect to Δx , we have

$$\Delta x \approx \left(\frac{3\epsilon}{f'''(x)} \right)^{1/3}$$

We will code a function that will find the 3rd derivative with a delta value that is not optimal and then use that derivative to find the first derivative, but this time with a delta that is quite optimal. I am assuming that the error will be relatively small in the 3rd derivative, hence, it should not be too much of a problem since it's only used to roughly find the optimal value for the derivative.

We still need to find some delta to use that is not too far-fetched for the three consecutive derivatives. In class we have seen that to minimize the error for such a derivative prescription (central), we must use $\Delta x \approx 10^{-5}$, assuming second derivatives are not too crazy and that our machine ϵ is $\approx 10^{-16}$.

```

def ndiff(fun, x, full = False):
    '''Function to take a derivate.
    Optional: can output the estimated error on the result.'''

    ini_step = 10**-5 # Initial step size (for third derivative)

    # Anonymous function to take derivatives
    diff_op=lambda func, x: (func(x+ini_step)-func(x-ini_step))/(2*ini_step)

    deriv_1 = lambda x: diff_op(fun, x) # Calculating the first derivative
    deriv_2 = lambda x: diff_op(deriv_1, x) # Calculating the second derivative
    deriv_3 = lambda x: diff_op(deriv_1, x) # Calculating the third derivative

    # Optimal step size
    third_deriv = deriv_3(x)
    print(third_deriv)
    opt_step = (abs(3*np.finfo(float).eps/third_deriv))**(1/3)

    # Finding the derivative
    deriv = (fun(x + opt_step) - fun(x - opt_step))/(2*opt_step)

    # Conditional system for optional argument
    if full:
        # Returns derivative and estimated error
        est_err=np.finfo(float).eps/opt_step+third_deriv*opt_step**2/6
        return np.array([deriv, est_err])

    elif not full:
        # Returns only derivative
        return deriv

```

Question 3

To answer this question we create the following function

```

def lakeshore(V, data):
    '''Function to interpolate the data with a cubic spline'''
    temperatures = np.array([i[0] for i in data[:,:-1]]) # temperatures array (from raw data)
    voltages = np.array([i[1] for i in data[:,:-1]]) # voltages array (from raw data)
    approx_vol_size = abs(voltages[2]-voltages[1])

    # First we find the interpolation
    cs = sci.interpolate.CubicSpline(voltages, temperatures) # spline
    inter_val = cs(V)

```

```

# Now we roughly estimate the error
lin_spline = sci.interpolate.interp1d(voltages, temperatures) # Linear interpolation
err_range=np.array(np.linspace(V-approx_vol_size, V+approx_vol_size,1000))

approx_error=np.std(abs(lin_spline(err_range)-cs(err_range)),axis=0)

return inter_val, approx_error

```

This function takes as input the value(s) we want to evaluate at an interpolated value and the raw data to create the interpolation. The function starts by extracting arrays for the temperature and the voltages from the raw data. From these x and y set of data, we are able to create the interpolation. To find the error on the obtained values, we create another interpolation, but this time linear. We look at the difference between the linear and the cubic splined interpolations to obtain a rough estimate for the error. This difference should give us roughly an order of magnitude estimate of the error. The rationale behind it is that the cubic spline can be “curvier” than the actual data, while the linear fit is often not curvy enough, hence the true value is hypothesized to fall within the two fit. This is where this subtraction came from.

All this is down with arrays, so the function can work with both single value or arrays for V . It will return two elements, the first is either an array or single value for the interpolated value and the second is the rough estimate for the error (also as array or value).

The interpolation can be observed in the following plot:

Question 4

$\cos(x)$

We create three different interpolations for samples of the $\cos(x)$ function.

The first is a polynomial fit of degree $n - 1$ where n is the number of samples used. `scipy`’s `polyfit` function was used for that.

The second interpolation scheme used was a cubic spine. Similarly to the polynomial fit, we used `scipy`’s `polyfit`’s `interpolate.CubicSpline` function.

For the third interpolation scheme (the rational function). We proceeded the same way that was taught in class. We used a polynomial as a numerator and another polynomial as denominator. Each coefficient can be controlled separately. We used `scipy`’s `linalg` package to solve the linear algebra problem that is finding the coefficients of the polynomials.

For the \cos function we obtained the following fit and residuals:

The errors for these are:

- The error for the polynomial interpolation is 0.00073

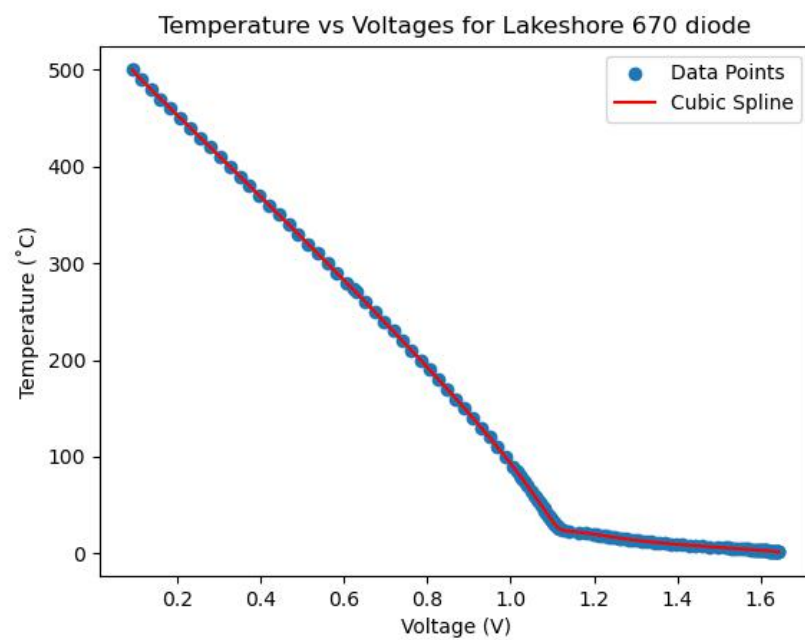


Figure 3: q3_interp_plot

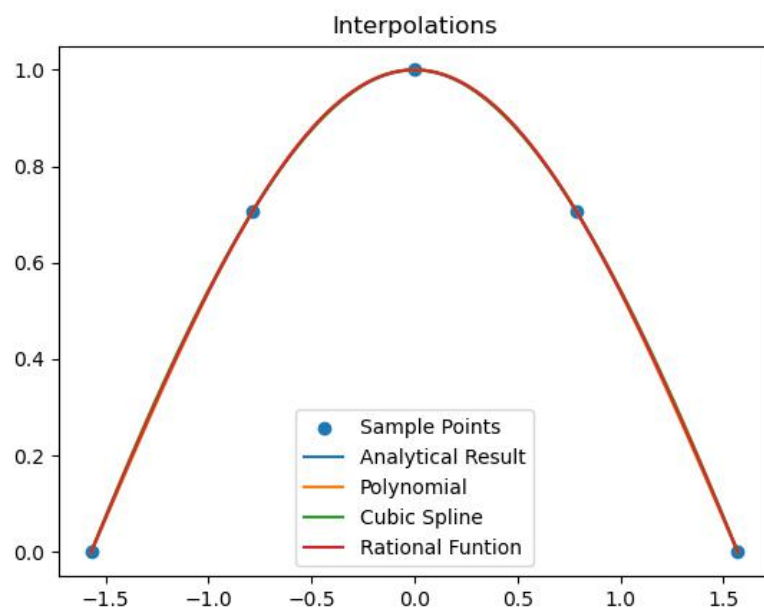


Figure 4: q4_interp1

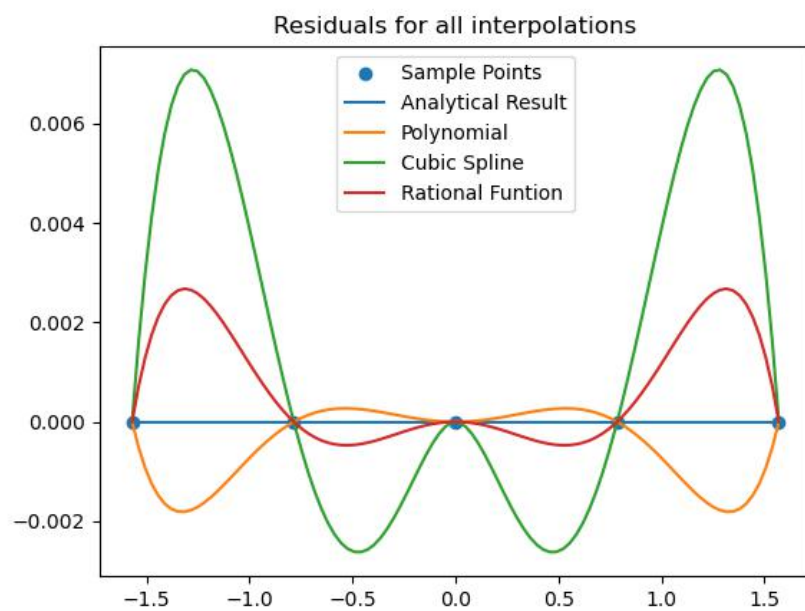


Figure 5: q4_res1

- The error for the cubic spline interpolation is 0.0034
- The error for the rational function interpolation is 0.0011

We see that interpolation with the least error is the polynomial fit. A possible explanation is that the cos function is very well approximated by the first terms of a Taylor series. The rational function does not make as good of a job but close. This is most likely because of the distribution of the degrees of freedom between the numerator and denominator. The degrees of freedom in the denominator are possibly not as useful as the ones in the numerator. The cubic spline does the least good job at approximating it.

Lorentzian

Just like for the cos function, we interpolate the Lorentzian function, $\frac{1}{1+x^2}$. The result for 5 sample points can be seen:

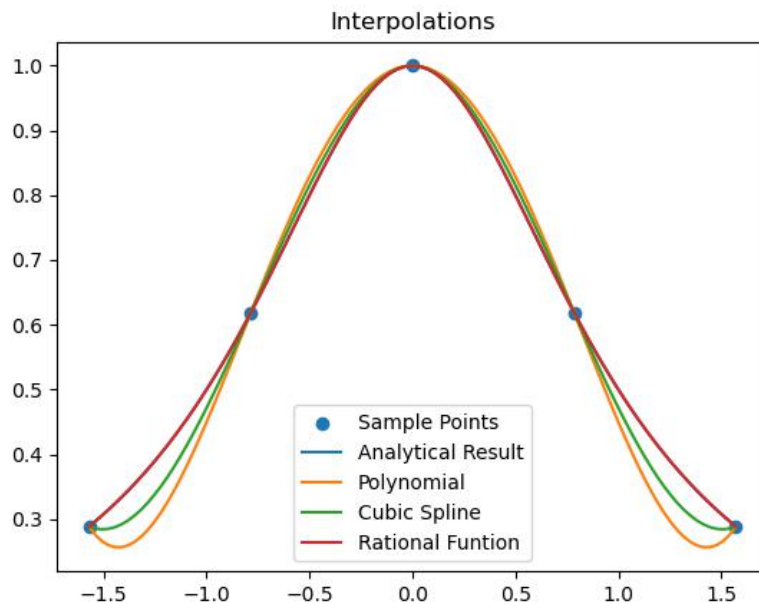


Figure 6: q4_interp2

- The error for the polynomial interpolation is 0.043
- The error for the cubic spline interpolation is 0.023
- The error for the rational function interpolation is 7.99e-17

In this case the cubic spline does a better job than the polynomial fit. The rational function produces an error that is essentially 0, being non-zero due

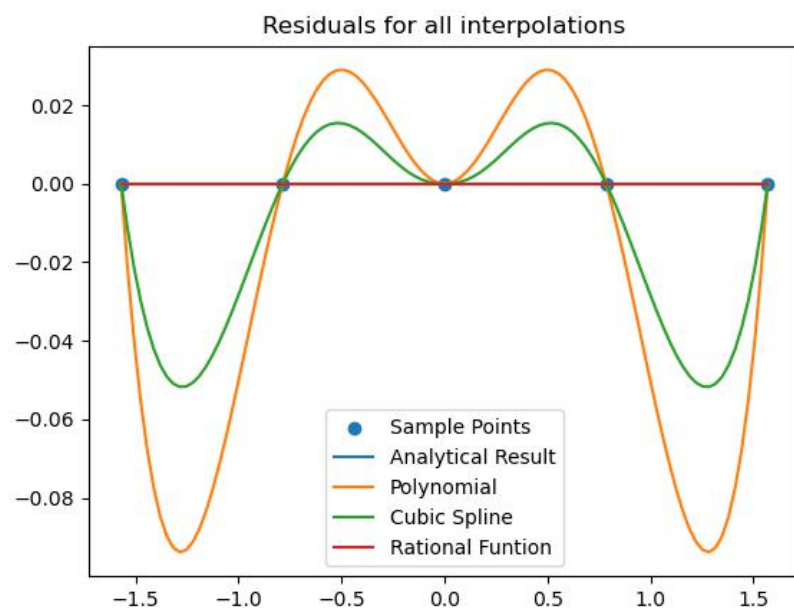


Figure 7: q4_resi2

to the machine doing linear algebra. This makes sense since the Lorentzian is essentially a rational function.

If we increase the order, we have observed that the rational fit starts becoming way off. This is with the `np.linalg.inv` function, 10 sample points, $n = 4$, and $n = 5$:

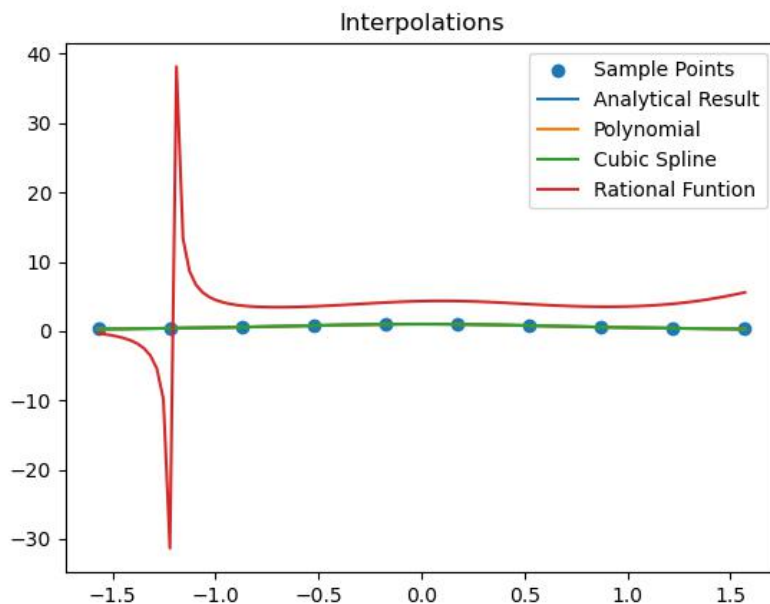


Figure 8: q4_interp3

This is very wrong... but when we switch the `np.linalg.inv` function to the `np.linalg.pinv` function we obtain virtually no error again (for the rational function):

Now, I'll be honest, I am not 100% sure of the reason why this happens, but here is an attempt at an explanation:

I know that the difference between `pinv` and `inv` is that `pinv` finds the “generalized inverse” or “pseudo-inverse” of the input matrix as opposed to the regular inverse. From what I’ve understood by reading things online, the pseudo-inverse “takes care” of infinities caused by machine precision. If I had to guess, I would think that these infinities arise because we are trying to overfit a function that is already a rational (the Lorentzian) to another rational that is of much higher degree. I think that this means there are more than one coefficient configurations that can work i.e there are free variables, thus making the input matrix non-invertible. I think this is also why we chose to set the constant term in

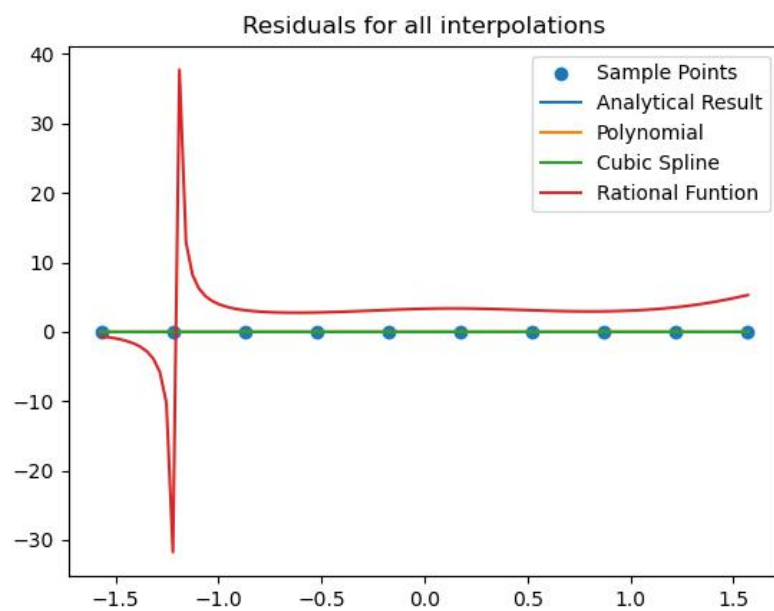


Figure 9: q4_resi3

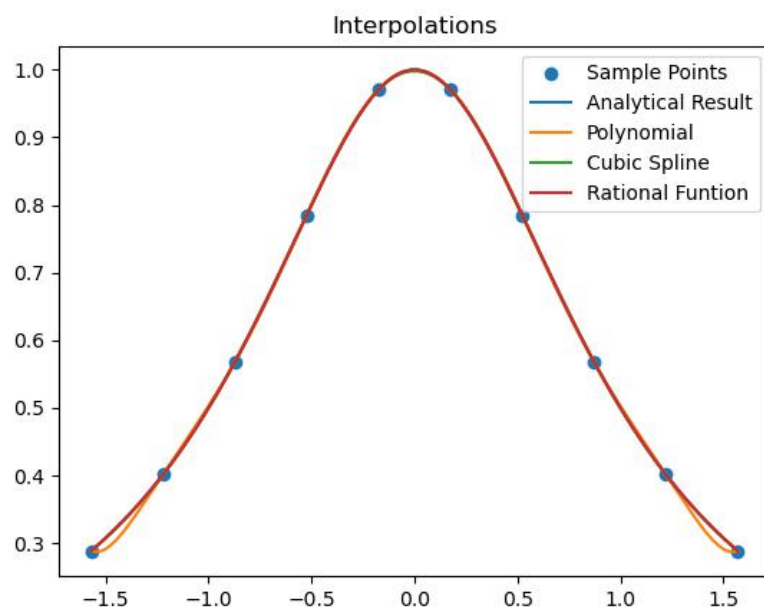


Figure 10: q4_interp4

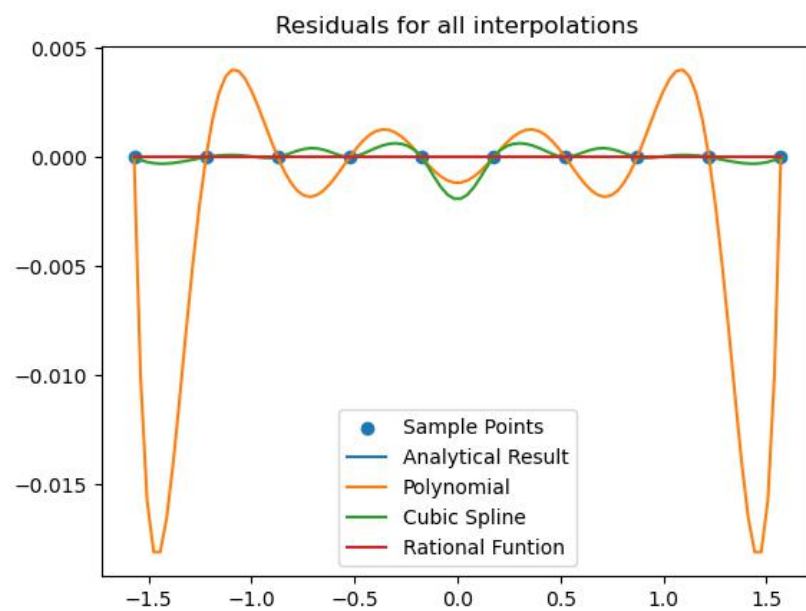


Figure 11: q4_resi4

the denominator to 1: we don't lose any generality by doing that since we are losing a degree of freedom when taking the fraction (e.g. $\frac{1}{2} = \frac{2}{4}$). If we didn't remove the factor for the 1, we'd be having a degeneracy in the coefficient.