PHYS 512 Problem set 1

Teophile Lemay, 281081252

1

Given a function f, evaluated at the points $x \pm \delta$ and $x \pm 2\delta$.

1.1 a)

The derivative at x can be calculated from the points around x using the central derivative formula (in the limit of small δ):

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

Performing a Taylor expansion of f at each point gives

$$f(x+\delta) \approx f(x) + f'(x)\delta + \frac{1}{2}f''(x)\delta^2 + \frac{1}{6}f'''(x)\delta^3 + \frac{1}{24}f''''(x)\delta^4 + \frac{1}{120}f'''''(x)\delta^5 + ...$$

$$f(x-\delta) \approx f(x) - f'(x)\delta + \frac{1}{2}f''(x)\delta^2 - \frac{1}{6}f'''(x)\delta^3 + \frac{1}{24}f''''(x)\delta^4 - \frac{1}{120}f'''''(x)\delta^5 + ...$$

$$f(x+2\delta) \approx f(x) + f'(x)2\delta + 2f''(x)\delta^2 + \frac{4}{3}f'''(x)\delta^3 + \frac{2}{3}f''''(x)\delta^4 + \frac{4}{15}f'''''(x)\delta^5 + ...$$

$$f(x-2\delta) \approx f(x) - f'(x)2\delta + 2f''(x)\delta^2 - \frac{4}{3}f'''(x)\delta^3 + \frac{2}{3}f''''(x)\delta^4 - \frac{4}{15}f'''''(x)\delta^5 + ...$$

For the $x \pm \delta$ case, subtracting $f(x-\delta)$ from $f(x+\delta)$ gives the central derivative:

$$f(x+\delta) - f(x-\delta) = 2f'(x)\delta + \frac{1}{3}f'''(x)\delta^3 + \frac{1}{60}f'''''(x)\delta^5 + ...$$

$$f'(x) = \frac{f(x+\delta) - f(x-\delta)}{2\delta} - \frac{1}{6}f'''(x)\delta^2 - \frac{1}{120}f'''''(x)\delta^4 + ...$$

Performing the same operation on the $x \pm 2\delta$ case:

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

The δ^2 term can be eliminated by subtracting four times the $x \pm \delta$ derivative from the $x \pm 2\delta$ derivative

$$f'(x) - 4f'(x) = \left(\frac{f(x+2\delta) - f(x-2\delta)}{4\delta} - \frac{2}{3}f'''(x)\delta^2 - \frac{2}{15}f'''''(x)\delta^4 - ...\right) - 4\left(\frac{f(x+\delta) - f(x-\delta)}{2\delta} - \frac{1}{6}f'''(x)\delta^2 - \frac{1}{120}f'''''(x)\delta^4 + ...\right)$$

$$-3f'(x) = \frac{8f(x-\delta) + f(x+2\delta) - 8f(x+\delta) - f(x-\delta)}{4\delta} - \frac{1}{10}f'''''(x)\delta^4$$

$$\boxed{f'(x) = \frac{8f(x+\delta) - 8f(x-\delta) + f(x-\delta) - f(x+2\delta)}{12\delta} + \frac{1}{30}f'''''(x)\delta^4 + ...}$$

1.2 b)

The error for this derivative is comes from rounding error and discarding higher order terms from the Taylor series. For rounding error ϵ, the error is approximately

$$\text{error } = \frac{|f(x)| \cdot \epsilon}{\delta} + \frac{1}{30} f''''''(x)\delta^4.$$

Minimizing with respect to δ:

$$0 = \frac{\mathrm{d}}{\mathrm{d}\delta}\left(\frac{|f(x)| \cdot \epsilon}{\delta} + \frac{1}{30} f''''''(x)\delta^4\right) = -\frac{|f(x)| \cdot \epsilon}{\delta^2} + \frac{2}{15} f''''''(x)\delta^3$$

$$\frac{|f(x)| \cdot \epsilon}{\delta^2} = \frac{2}{15} f''''''(x)\delta^3$$

$$\delta^5 = \frac{15}{2}\frac{|f(x)| \cdot \epsilon}{f''''''(x)}$$

So the ideal value of δ should be

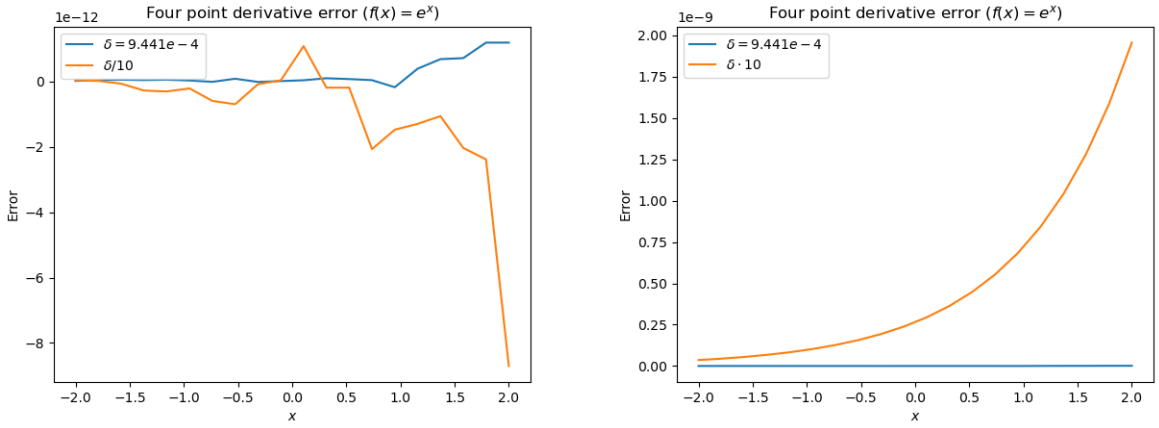
$$\boxed{\delta = \left(\frac{15}{2}\frac{|f(x)| \cdot \epsilon}{f''''''(x)}\right)^{1/5}}.$$

For $f = e^x$, the ideal step should be approximately

$$\delta \approx \left(\frac{15}{2}\frac{e^x}{e^x} \cdot 10^{-16}\right)^{1/5} = \left(\frac{15}{2} \cdot 10^{-16}\right)^{1/5} = 9.441 \cdot 10^{-4}$$

As shown in figure 1, the calculated value of δ performs better than steps sizes of an order of magnitude higher and lower.

Figure 1: Step size error comparison $f(x) = e^x$

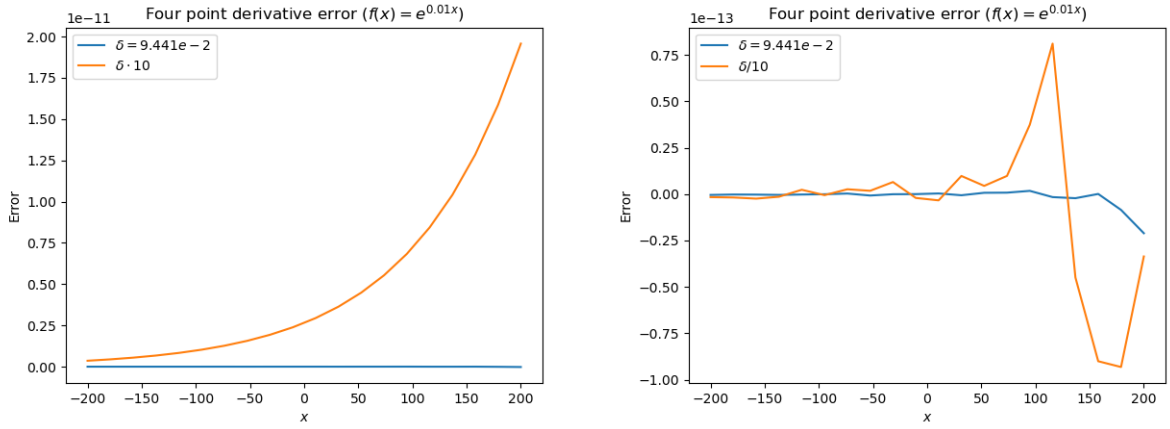


For $f(x) = e^{0.01x}$, the ideal step size should be approximately

$$\delta \approx \left(\frac{15}{2}\frac{e^{0.01x}}{10^{-10}e^{0.01x}} \cdot 10^{-16}\right)^{1/5} = \left(\frac{15}{2} \cdot 10^{-6}\right)^{1/5} = 9.441 \cdot 10^{-2}$$

As shown in Figure 2, this is also approximately the correct step size as it outperforms steps sizes of an order of magnitude larger and smaller.

Figure 2: Step size error comparison $f(x) = e^{0.01x}$



2

For my function, I estimate the ideal derivative step size dx by trying different values of dx and comparing the difference in derivatives calculated for close values of dx. Near the best value of dx the error is minimized, so other values around the best value will also have relatively small error compared to dx far from the ideal value. Therefore, I calculate the optimal dx by choosing the value around which there is the least change in the numerical derivative. My complete numerical differentiation function is described below.

My numerical differentiator function pseudocode:
Finding optimal step size dx:

- To get around the proper order of magnitude, evaluate the derivative for $dx \in \{10^{-16}, 10^{-15}, 10^{-14}, ..., 10^{-1}, 1\}$.
- Since I expect less variation near the optimal dx, evaluate differences of derivatives for adjacent dx values and choose dx corresponding to smallest difference.
- Refine the optimal dx guess by repeating the previous steps above with dx from the interval centered around the best guess $\{dx \cdot 10^-1, ...dx, ...dx \cdot 10\}$. I take the best dx at this point as good enough to proceed.
  - For array inputs of x , the derivative is evaluated at all points of x with all values of dx. Differences between derivatives for adjacent dx values are calculated, then summed along each value of x to get an array of the differences for each dx. The best dx is chosen in the same manner, from the dx corresponding to the minimum difference.

Evaluating the derivative:

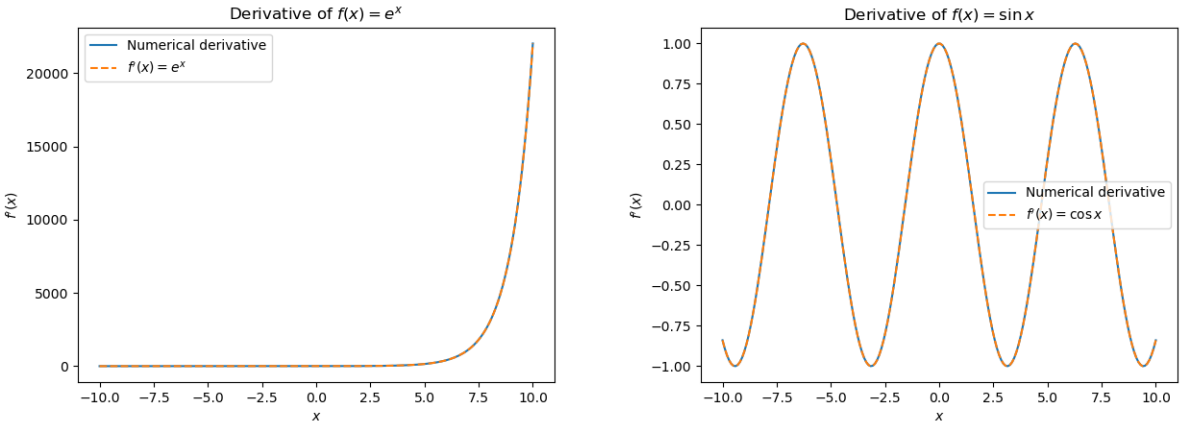
- The numerical derivative output is calculated using the central derivative.formula using the chosen best dx at each point in x.

Estimating Error

- Error estimation is calculated according to the formula for the error of the central derivative: $R \approx \frac{f(x)\epsilon}{dx} + \frac{1}{6}f'''(x)dx^2$ where $\epsilon = 10^{-16}$ is the rounding error.
  - The third derivative for the error formula is crudely calculated at each point of the function:
  - For each point in x, I calculate $f'(x \pm 2dx)$ and use these values to get $f''(x + dx) = \frac{f'(x+2dx)-f'(x)}{2dx}$ and $f''(x - dx) = \frac{f'(x)-f'(x-2dx)}{2dx}$. Finally, the value of $f'''(x) = \frac{f''(x+dx)-f''(x-dx)}{2dx}$ is used to evaluate the formula for R.

The code for this question is in the file named "Q2_numerical_differentiator.py". Testing on the function $f(x) = e^x$ results in actual error of order 10^{-11} or lower. The error estimated by the function is within one order of magnitude to the actual error and generally on the same order of magnitude or even closer. Examples of the function output for e^x and $\sin x$ in Figure 3.

Figure 3: Numerical differentiator sample results



3

My interpolation function uses the SciPy's cubic spline interpolation implementation (scipy.interpolate.CubicSpline) with natural boundary conditions (second derivative is 0 at the endpoints of the interval). The SciPy implementation takes the given data points as input, and computes a cubic spline for the points with the specified boundary conditions. The output is a SciPy function that computes the spline's corresponding polynomial at any given point within the limits of the input data. (https://docs.scipy.org/doc/scipy/reference/generated/scipy.interpolate.CubicSpline.html)

My function estimates the error of the cubic spline by comparing it to a cubic polynomial interpolation since the cubic spline is simply a piecewise collection of cubic polynomial interpolations with special conditions on their derivatives that tend to make it a better fit. Therefore, I expect the actual error to be generally less than or equal to that of a simple cubic polynomial interpolation. Assuming that the input data function is 4 times differentiable, the cubic polynomial interpolation's error over 4 points is:

$$f(x) - P_3(x) = \frac{1}{4!} f''''(\xi_x) \prod_{i=1}^{4} (x - x_i)$$

for some point ξ_x within the interval of the four points [1]. In this case, the fourth derivative is computed by repeatedly using the forward derivative formula using the known first derivatives at the four points around each point of interpolation. For example: if the cubic polynomial interpolation uses the points $\{x_a, x_b, x_c, x_d\}$ and $f'(x)$ is known at all points, then

$$f''(x_b) \approx \frac{f'(x_b) - f'(x_a)}{x_b - x_a}$$
$$f''(x_c) \approx \frac{f'(x_c) - f'(x_b)}{x_c - x_b}$$
$$f''(x_d) \approx \frac{f'(x_d) - f'(x_c)}{x_d - x_c}$$
$$f'''(x_c) \approx \frac{f''(x_c) - f''(x_b)}{x_c - x_b}$$
$$f'''(x_d) \approx \frac{f''(x_d) - f''(x_c)}{x_d - x_c}$$
$$f''''(x_d) \approx \frac{f'''(x_d) - f'''(x_c)}{x_d - x_c}$$

This is a very crude calculation of the fourth derivative and it always uses the fourth derivative at the rightmost point of the interval instead of choosing ξ_x to maximize the error.

[1] Stoer, J., and Bulirsch, R. 2002, Introduction to Numerical Analysis, 3rd ed. (New York: Springer), §2.1.4.1

Figure 4: Interpolation function results

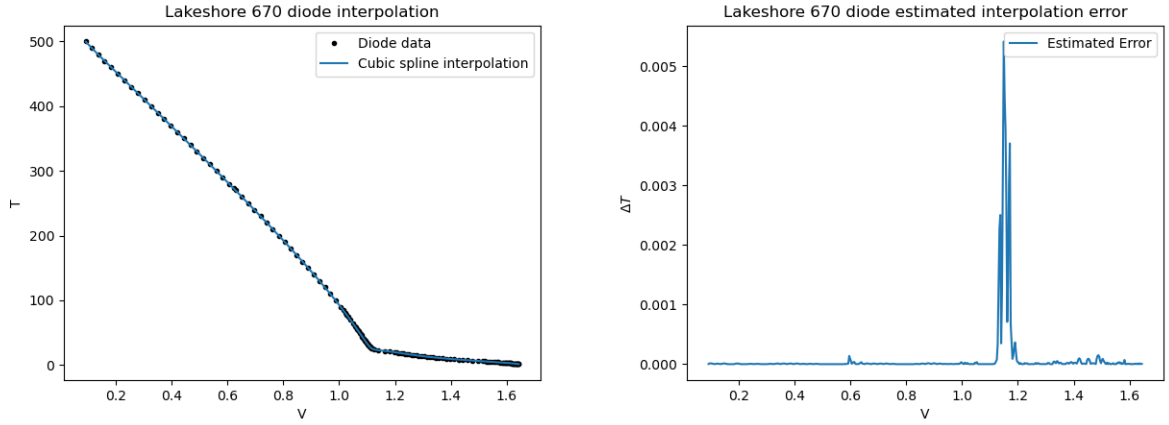


Figure 4 shows the interpolation over the dataset, as well as the error. The interpolation follows the data very closely, which is consistent with the estimated error being small at most points. There is also a large spike in the magnitude of the estimated error near the sharp "elbow" curve in the data, which is logical as the smooth splines will imperfectly represent the sharp change. The code for this question is in the file named "Q3_lakeshore_interp.py".

4

The interpolations were performed using 5 evenly spaced points in the interval $[-\pi/2, pi/2]$. The polynomial fit was computed to over the whole interval to 4th order — the highest order possible for the number of points — using "numpy.polyfit". The cubic spline was computed using "scipy.interpolate.CubicSpline". The rational function interpolation was calculated using the code example from the lecture slides ("interpolate_integrate.pdf"). Code for the $\cos x$ interpolations is in the file named "Q4_cos.py".

As shown in figure 5, all three methods have similar mean error on the order of 10^{-4} to 10^{-3}, with the polynomial interpolation having the lowest mean error of all. The same procedure was repeated with the Lorentzian function $f(x) = \frac{1}{1+x^2}$, using 5 evenly spaced points on the interval $[-1, 1]$. Since the Lorentzian function is a rational function by definition, if the rational function interpolation works properly, then the interpolated function should be within some rounding error of the Lorentzian itself. Therefore, the mean error for the rational function interpolation should be equal to 0 within rounding error. This hypothesis is confirmed in figure 6 where the rational function interpolation has mean error on the order of 10^{-16} which is the same order as rounding error for python. Both the polynomial and the cubic spline interpolations have much larger mean error on the order of 10^{-3}. Increasing the order of the rational function interpolation to $n = 4$, $m = 5$, and calculating using 8 points, gives an interpolation that completely fails to match with the Lorentzian function. The higher order rational function is saved however by changing "np.linalg.inv" to "np.linalg.pinv" in the fitting function (figure 7).

The break down of the rational function interpolation with higher orders can be explained by the fact that the rational function will add unneeded higher order terms in the numerator and denominator. These terms would "cancel" analytically, but since they go to zero much faster than the required lower order terms, they introduce random rounding error if they reach the machine precision. This random error is propagated to the rest of the coefficients during the matrix inverse. Swapping "np.linalg.inv" for "np.linalg.pinv" solves the issue by neglecting the very small values in a matrix when taking the inverse, such that they do not affect the final outcome.

Figure 5: Interpolations of $\cos(x)$

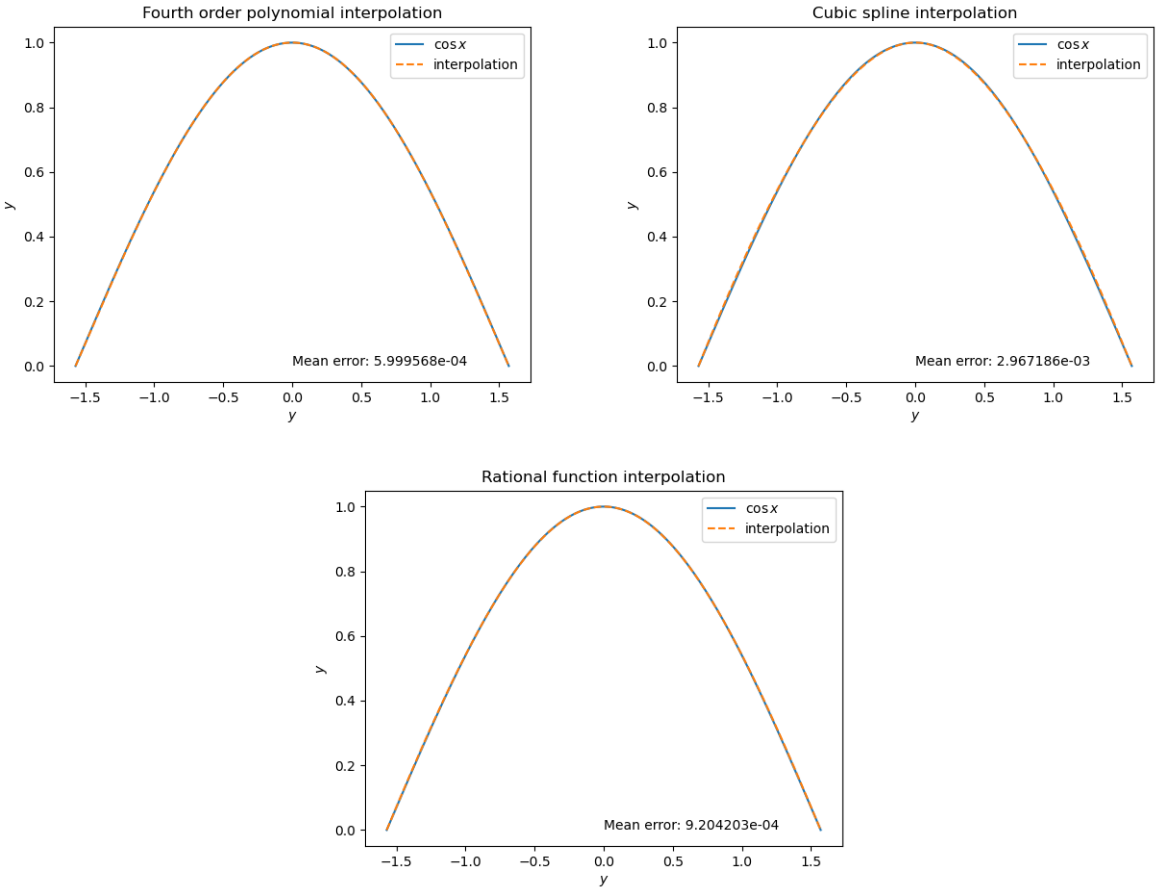


Figure 6: Lorentzian function interpolations

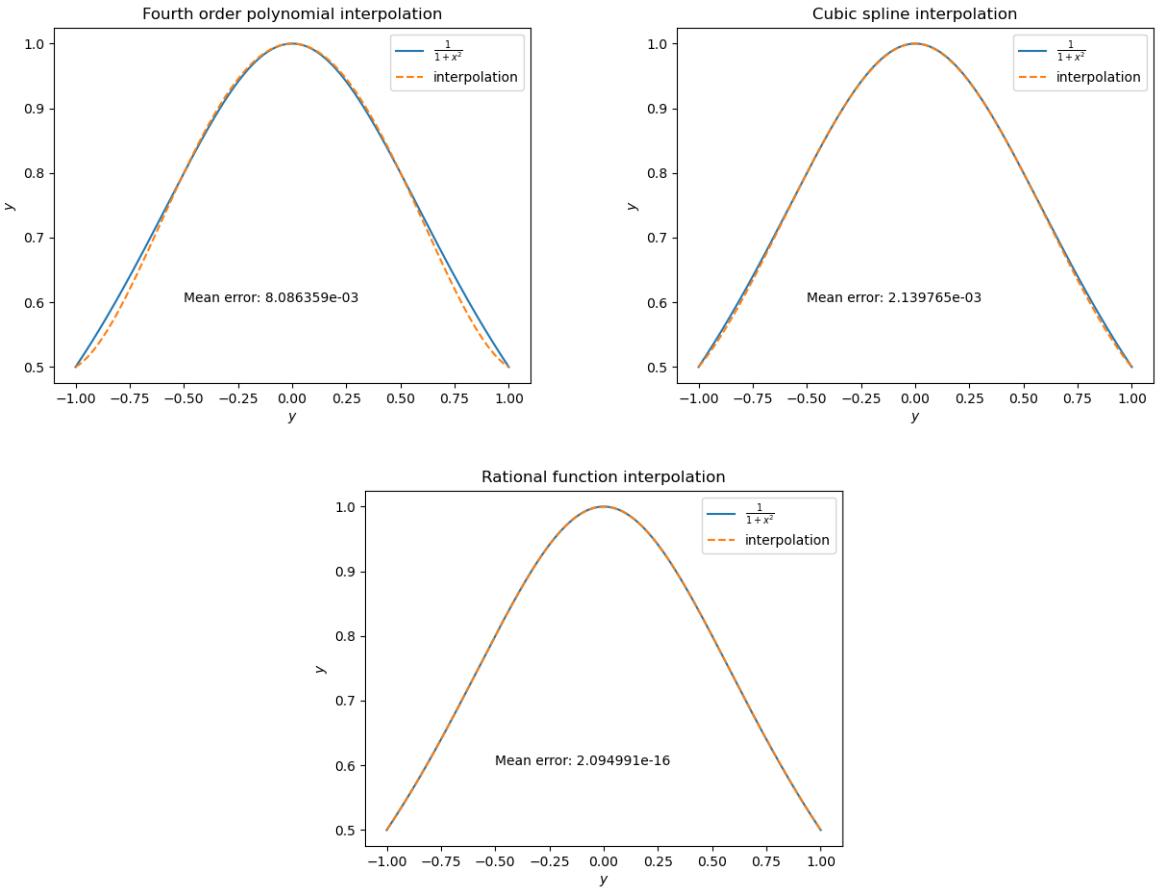


Figure 7: Higher order rational function interpolation of the Lorentzian