PHYS 512 - Assignment 1

André Vallières (260742187)

September 18, 2020

Problem 1

a) First, we let

$$f_1'(x) = \frac{f(x+\delta) - f(x-\delta)}{2\delta}$$
$$f_2'(x) = \frac{f(x+2\delta) - f(x-2\delta)}{4\delta}$$

Ignoring finite machine precision for now, we have, in general

$$f(x+\alpha\delta) = f(x) = \alpha\delta f'(x) + \frac{(\alpha\delta)^2}{2}f''(x) + \frac{(\alpha\delta)^3}{6}f'''(x) + ...$$

Hence,

$$f_1'(x) = \frac{(f(x) + \delta f'(x) + \delta^2 f''(x)/2 + \delta^3 f'''(x)/6 + ...) - (f(x) - \delta f'(x) + \delta^2 f''(x)/2 - \delta^3 f'''(x)/6 + ...)\Big]}{2\delta}$$
$$= f'(x) + \frac{\delta^2}{6}f'''(x) + ...$$

and

$$f_2'(x) = \frac{(f(x) + 2\delta f'(x) + 2\delta^2 f''(x) + 4\delta^3 f'''(x)/3 + ...) - (f(x) - 2\delta f'(x) + 2\delta^2 f''(x) - 4\delta^3 f'''(x)/3 + ...)\Big]}{4\delta}$$
$$= f'(x) + 4\frac{\delta^2}{6}f'''(x) + ...$$

Using these two equations, we get

$$f'_{\text{approx}}(x) = \frac{f_2'(x) - 4f_1'(x)}{-3} + \mathcal{O}(f^{(5)}(x)) \tag{1}$$

which eliminates the third-order term.

b) To find δ in terms of the machine precision (ε) and various properties of the function, we need to account for finite machine precision in the Taylor series and include the fifth-order term.

Expanding from our previous equations, we have

$$f_1'(x) = f'(x) + \frac{\delta^2}{6}f'''(x) + \frac{\delta^4}{120}f^{(5)}(x) + \frac{g_1\varepsilon f(x)}{2\delta} + ...$$
$$f_2'(x) = f'(x) + 4\frac{\delta^2}{6}f'''(x) + \frac{\delta^4}{15}f^{(5)}(x) + \frac{g_2\varepsilon f(x)}{4\delta} + ...$$

Thus, (1) becomes

$$f'_{\text{approx}}(x) = f'(x) + \frac{\delta^4}{15} f^{(5)}(x) + \frac{\varepsilon}{\delta}\Big(\frac{g_2}{4} - 2g_1\Big) f(x) + ... \tag{2}$$

If we let

$$\Delta \equiv \frac{\delta^4}{15} f^{(5)}(x) + \frac{\varepsilon}{\delta}\Big(\frac{g_2}{4} - 2g_1\Big) f(x) = \frac{\delta^4}{15} f^{(5)}(x) + \frac{\varepsilon}{\delta} g f(x)$$

Then the error is minimized if

$$\begin{aligned}
\frac{d\Delta}{d\delta} &= \frac{4\delta^3}{15} f^{(5)}(x) - \frac{g\varepsilon}{\delta^2} f(x) = 0 \\
&\Rightarrow \delta^5 = \frac{15g\varepsilon}{4}\frac{f(x)}{f^{(5)}(x)} \sim \frac{15\varepsilon}{4}\frac{f(x)}{f^{(5)}(x)} \\
&\Rightarrow \delta = \Big(\frac{15\varepsilon}{4}\frac{f(x)}{f^{(5)}(x)}\Big)^{1/5}
\end{aligned}$$

Given $\varepsilon = 10^{-16}$, we have that the error is minimized for

$$\delta \approx \begin{cases} 8 \times 10^{-4} & \text{for } f(x) = \exp(x) \\ 8 \times 10^{-2} & \text{for } f(x) = \exp(0.01x) \end{cases}$$

The code to verify these estimates is in `prob1_b.py`. The results are

$$\delta \approx \begin{cases} 8 \times 10^{-4} & \text{for } f(x) = \exp(x) \\ 1 \times 10^{-2} & \text{for } f(x) = \exp(0.01x) \end{cases}$$

Of course, the estimates are not exact, but at least we see a factor near $1/100$ as expected from the rough calculations.

Problem 2

Using Hermite polynomials for interpolating due to the features at low temperature, we arrive at a pretty accurate model. The code is in `prob2.py`, and performs two tasks: 1) Interpolate with odd points 2) Compute RMS error using even points. Plotting the real and interpolated data simply shows no visible difference, as seen in Figure 1. The RMS error, however, computed as `np.std(true - estimate)`, gives **0.032102**. The reason for using different points for interpolating and computing the error is to minimize bias. Indeed, using all points simply yields a null RMS error for points in the dataset, but the model would obviously not be free of discepancies from reality.

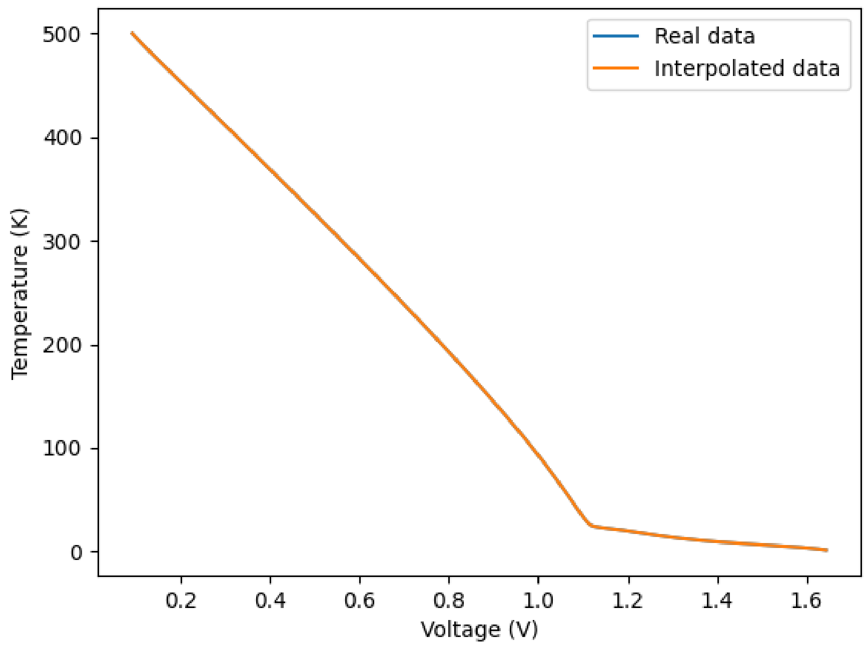
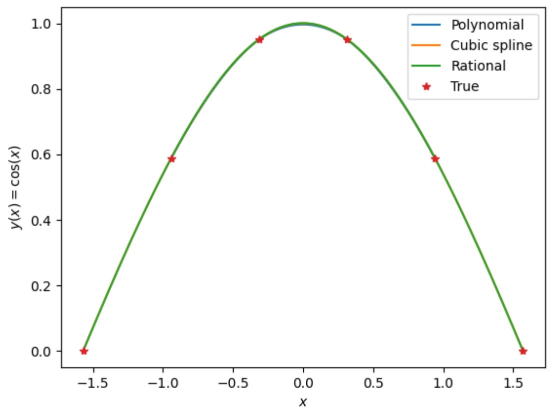


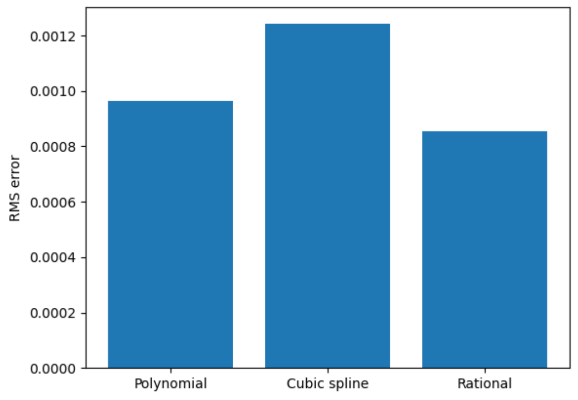
Figure 1: Real and interpolated data for DT-670 silicon diode using Hermite polynomial interpolation.

Problem 3

The code for polynomial, cubic spline, and rational interpolation is in `prob3.py`. To evaluate them on the same ground, 4 points are used for interpolation. The first function to interpolate is $cos(x)$ in the range $[-\pi/2, \pi/2]$. The results are shown in Figure 2.



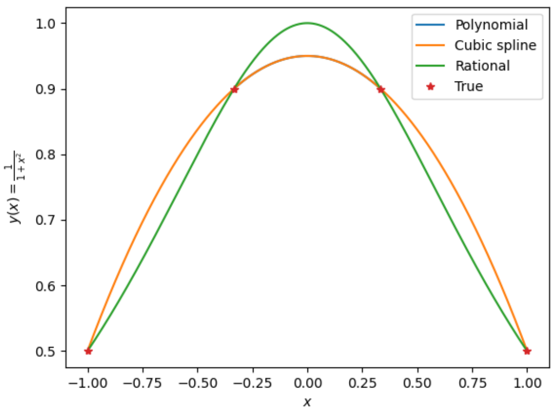
(a)



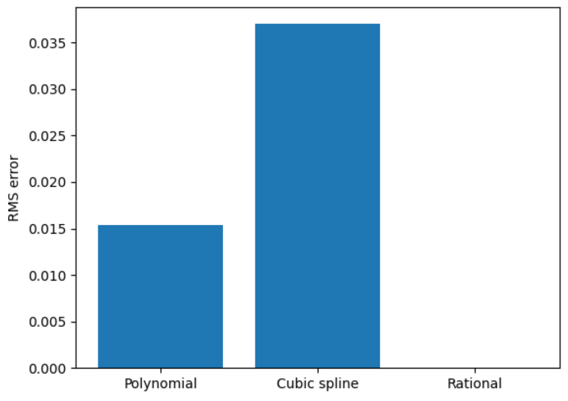
(b)

Figure 2: Cosine interpolation results for cubic polynomial, cubic spline, and rational interpolation with $n = 2, m = 3$. (a) Interpolation of $cos(x)$ in the range $[-\pi/2, \pi/2]$ showing great fit. (b) RMS error for all three interpolation methods. Rational interpolation shows best fit.

The second function to interpolate is a Lorentzian, so $1/(1+x^2)$ in the range $[-1, 1]$. We expect rational interpolation to perform best since a Lorentzian is a polynomial function. Running interpolation with the same parameters as before yields results shown in Figure 3.



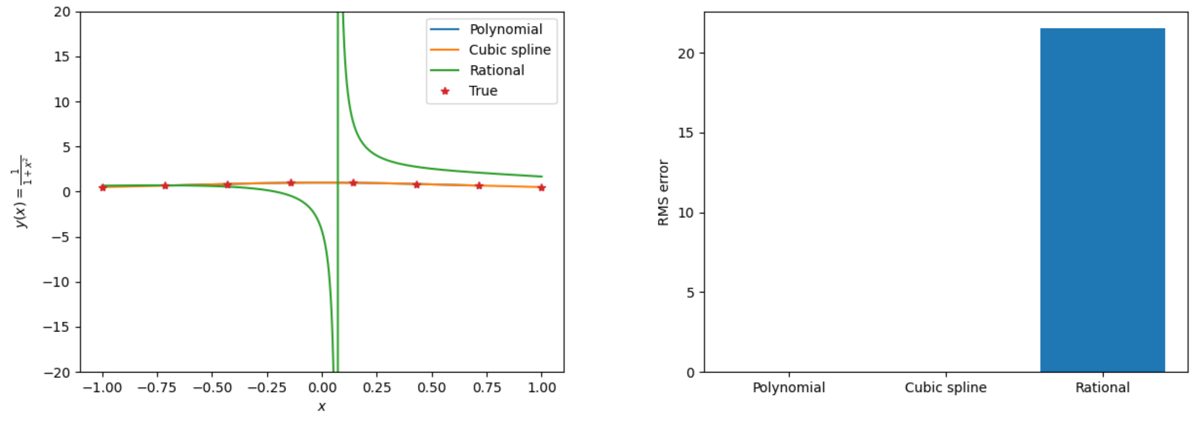
(a)



(b)

Figure 3: Lorentzian interpolation results for cubic polynomial, cubic spline, and rational interpolation with $n = 2, m = 3$. (a) Interpolation of $1/(1+x^2)$ in the range $[-1, 1]$ showing perfect fit for rational interpolation. (b) RMS error for all three interpolation methods. Rational interpolation again shows best fit with a null RMS error.

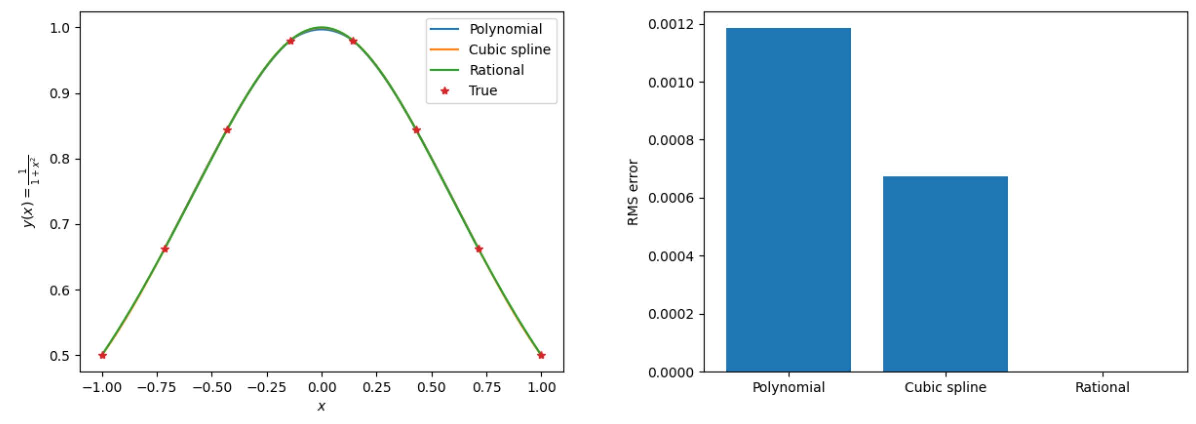
Now, if we were to increase the number of points (to 8, for example), we would expect similar results. Polynomial and cubic spline interpolation may perform better, but rational interpolation should still have perfect fit. However, this is not what happens. In fact, doing interpolation with $n = 4, m = 5$ yields very poor results for rational interpolation as shown in Figure 4.



(a) (b)

Figure 4: Lorentzian interpolation results for cubic polynomial, cubic spline, and rational interpolation with $n = 4, m = 5$. (a) Interpolation of $1/(1 + x^2)$ in the range $[-1, 1]$ showing extremely poor fit for rational interpolation. (b) RMS error for all three interpolation methods. Rational interpolation is so high polynomial and cubic spline interpolation errors show as 0 (although there are around 0.00118 and 0.000675, respectively).

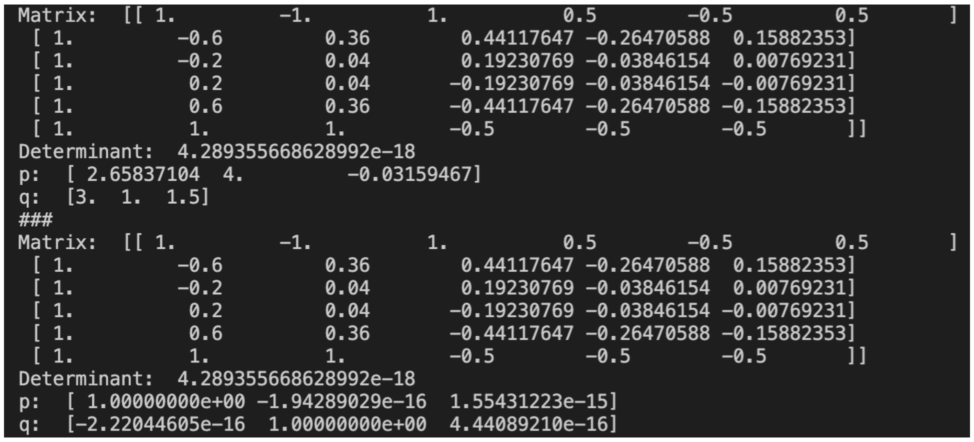
By simply changing `np.linalg.inv` to `np.linalg.pinv` in the `rat_fit` function solves the issue, as shown in Figure 5. The change makes sure the fitting step can deal with singular matrix inversion.



(a) (b)

Figure 5: Lorentzian interpolation results for cubic polynomial, cubic spline, and rational interpolation with $n = 4, m = 5$ using `np.linalg.pinv`. (a) Interpolation of $1/(1 + x^2)$ in the range $[-1, 1]$ showing back perfect fit for rational interpolation. (b) RMS error for all three interpolation methods. Rational interpolation is so high polynomial and cubic spline interpolation errors show as 0 (although there are around 0.00118 and 0.000675, respectively.

Investigating a bit on why it failed miserably before, as shown in Figure 4, we look at the `rat_fit` function. We notice that the determinant of the matrix that we build is really close to zero, but not exactly due to numerical imprecision. For example, with $n = 3, m = 4$, the determinant is of the order of 4.3×10^{-18}. This is effectively zero and no real inverse can be computed. This is why we needed `np.linalg.pinv` to compute the pseudo-inverse. However, practically, the determinant is not exactly zero thus the inversion process carries on but the calculated matrix inverse will have unusually large numbers due to the $1/\det(M)$ factor. In fact, this could turn practically zero values (i.e., really near zero, but not exactly due to numerical imprecision) into appreciable values, which would screw up the fitting process. If we look at the entries in p and q, some entries are greater than 1, which should not happen. As we increase n and m, those values increase to higher numbers which cause the poor fitting. For example, in Figure 6, the top portion corresponds to calculations with `np.linalg.inv` and the bottom with `np.linalg.pinv`. Notice the huge difference of magnitude between the entries of p and q.



```
Matrix:  [[ 1.         -1.          1.          0.5        -0.5         0.5       ]
 [ 1.         -0.6         0.36        0.44117647 -0.26470588  0.15882353]
 [ 1.         -0.2         0.04        0.19230769 -0.03846154  0.00769231]
 [ 1.          0.2         0.04       -0.19230769 -0.03846154 -0.00769231]
 [ 1.          0.6         0.36       -0.44117647 -0.26470588 -0.15882353]
 [ 1.          1.          1.         -0.5        -0.5        -0.5       ]]
Determinant:  4.289355668628992e-18
p:  [ 2.65837104  4.         -0.03159467]
q:  [3.  1.  1.5]
###
Matrix:  [[ 1.         -1.          1.          0.5        -0.5         0.5       ]
 [ 1.         -0.6         0.36        0.44117647 -0.26470588  0.15882353]
 [ 1.         -0.2         0.04        0.19230769 -0.03846154  0.00769231]
 [ 1.          0.2         0.04       -0.19230769 -0.03846154 -0.00769231]
 [ 1.          0.6         0.36       -0.44117647 -0.26470588 -0.15882353]
 [ 1.          1.          1.         -0.5        -0.5        -0.5       ]]
Determinant:  4.289355668628992e-18
p:  [ 1.00000000e+00 -1.94289029e-16  1.55431223e-15]
q:  [-2.22044605e-16  1.00000000e+00  4.44089210e-16]
```

Figure 6: Top: Rational function fitting using `np.linalg.inv`. Bottom: Rational function fitting using `np.linalg.pinv`. Notice the huge difference in magnitudes of the entries of p and q for those two approaches.

Problem 4

From Griffiths's solutions, the electric field from an infinitesimally thin spherical shell of charge with radius R is given by

$$E = \frac{\sigma}{4\pi\epsilon_0} \int_0^{\pi} \frac{(z - R\cos\theta)(2\pi R\sin\theta)}{z^2 + R^2 - 2zR\cos\theta}^{3/2} R d\theta \tag{3}$$

where z is the distance from the center. In cgs units, we can simply discard the $4\pi\epsilon_0$ factor, and for the purpose of this demonstration, we let $\sigma = 1$.

The integrator I've decided to use is the variable step size integrator shown in class due to its ability to find sharp features, such as the electric field at the spherical shell boundary. Running through the integration with different z, we notice a problem with the singularity at $z = R$. This is due to the discontinuity of the electric field at the boundary which means the integrand's denominator is 0 during the integration. To solve this issue, I first simply ignored integration that returned NaN, but that would give a discontinuity in the electric field versus distance plot. Hence, I decided to instead change the electric field formula to

$$E = \int_0^{\pi} \frac{(z - R\cos\theta)(2\pi R\sin\theta)}{z^2 + R^2 - 2zR\cos\theta + \epsilon}^{3/2} R d\theta \tag{4}$$

where I've set $\epsilon = 10^{-16}$. Doing so, the integration at $z = R$ is correctly handled and the results are shown in Figure 7. Interestingly, no such trick is needed with the `scipy.integrate.quad` function, which correctly handles singularities.

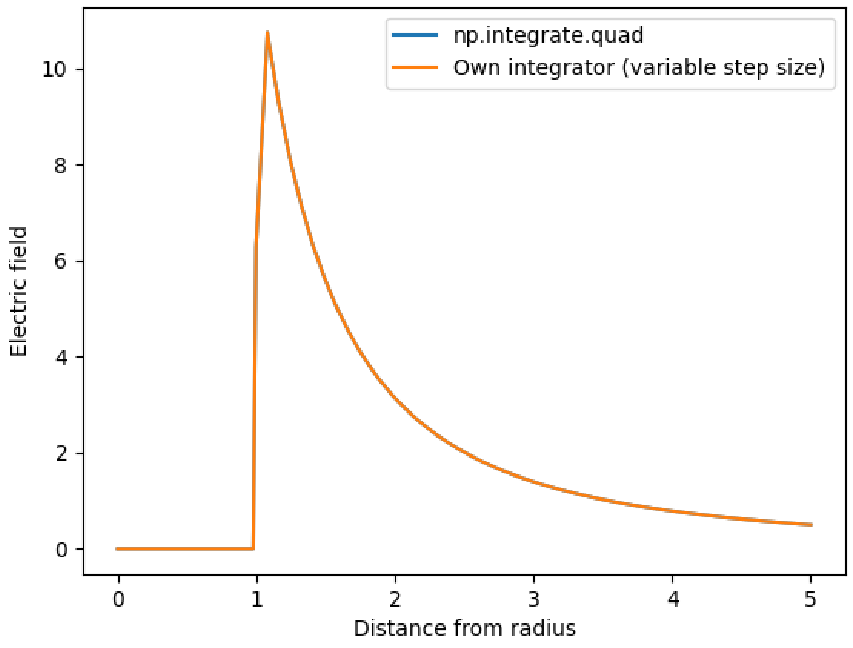


Figure 7: Electric field as a function of the distance from the radius, R, which is set to 1. This is found by integration following (4). The RMS difference between results found with `scipy.integrate.quad` or `my_integrate` is around 9.3×10^{-5}.