PHYS 512 – Problem Set 1

Jade Ducharme (260929684)

September 2021

Problem 1

a) From class, we have seen that the derivative from $x \pm \delta$ can be expanded into a Taylor series as follows:

$$\begin{aligned} f_1' &\sim \frac{f(x+\delta) - f(x-\delta)}{2\delta} \\ f_1' &\sim [f_0 + f'\delta + \frac{1}{2}f''\delta^2 + \frac{1}{6}f^{(3)}\delta^3 + \frac{1}{24}f^{(4)}\delta^4 + \frac{1}{120}f^{(5)}\delta^5 \\ &\quad - \left(f_0 - f'\delta + \frac{1}{2}f''\delta^2 - \frac{1}{6}f^{(3)}\delta^3 + \frac{1}{24}f^{(4)}\delta^4 - \frac{1}{120}f^{(5)}\delta^5\right)]/2\delta \\ f_1' &\sim f' + \frac{1}{6}f^{(3)}\delta^2 + \frac{1}{120}f^{(5)}\delta^4 \end{aligned} \tag{1}$$

Similarly, the derivative from $x \pm 2\delta$ can be expanded as follows:

$$\begin{aligned} f_2' &\sim \frac{f(x+2\delta) - f(x-2\delta)}{4\delta} \\ f_2' &\sim [f_0 + 2f'\delta + 2f''\delta^2 + \frac{4}{3}f^{(3)}\delta^3 + \frac{2}{3}f^{(4)}\delta^4 + \frac{4}{15}f^{(5)}\delta^5 \\ &\quad - \left(f_0 - 2f'\delta + 2f''\delta^2 - \frac{4}{3}f^{(3)}\delta^3 + \frac{2}{3}f^{(4)}\delta^4 - \frac{4}{15}f^{(5)}\delta^5\right)]/4\delta \\ f_2' &\sim f' + \frac{2}{3}f^{(3)}\delta^3 + \frac{2}{15}f^{(5)}\delta^4 \end{aligned} \tag{2}$$

Now, I need to find a linear combination of f_1' and f_2' that cancels out the $f^{(3)}$ term but

doesn't add any coefficients to the f' term. The obvious choice is $\frac{4}{3}f'_1 - \frac{1}{3}f'_2$:

$$\begin{aligned} f' &= \frac{4}{3}f'_1 - \frac{1}{3}f'_2 \\ &= \frac{4}{3}\left(f' + \frac{1}{6}f^{(3)}\delta^2 + \frac{1}{120}f^{(5)}\delta^4\right) - \frac{1}{3}\left(f' + \frac{2}{3}f^{(3)}\delta^3 + \frac{2}{15}f^{(5)}\delta^4\right) \\ &= \frac{4}{3}f' + \frac{2}{9}f^{(3)}\delta^4 + \frac{1}{90}f^{(5)}\delta^4 - \frac{1}{3}f' - \frac{2}{9}f^{(3)}\delta^4 - \frac{2}{45}f^{(5)}\delta^4 \\ &= f' - \frac{1}{30}f^{(5)}\delta^4 \end{aligned} \tag{3}$$

Therefore, the answer to part a) is that our estimate of the first derivative at x should be $f' \sim \frac{4}{3}f'_1 - \frac{1}{3}f'_2 = \frac{4}{3}\frac{f(x+\delta)-f(x-\delta)}{2\delta} - \frac{1}{3}\frac{f(x+2\delta)-f(x-2\delta)}{4\delta}$.

b)

The round-off error is approximately $\epsilon_r \sim \frac{\epsilon f}{\delta}$ (where ϵ is the machine precision) while the truncation error is on the order of the leading term in the Taylor series: $\epsilon_t \sim \delta^4 f^{(5)}$. We want to minimize $\epsilon_r + \epsilon_t$, so we take the derivative and set it equal to zero:

$$\begin{aligned} \frac{d}{d\delta}\left(\epsilon_r + \epsilon_t\right) \sim -\frac{\epsilon f}{\delta^2} + \delta^3 f^{(5)} &= 0 \\ \frac{\epsilon f}{\delta^2} &= \delta^3 f^{(5)} \\ \delta &= \left(\frac{\epsilon f}{f^{(5)}}\right)^{\frac{1}{5}} \end{aligned} \tag{4}$$

For $\exp(x)$, $f^{(5)} = f$, so the step-size is simply $\delta = \epsilon^{\frac{1}{5}}$. For $\exp(0.01x)$, $f^{(5)} = 10^{-10}f$, so the step-size is $\delta = \left(\frac{\epsilon}{10^{-10}}\right)^{\frac{1}{5}} = 100\epsilon^{\frac{1}{5}}$.

The results for $f(x) = \exp(x)$ and $f(x) = \exp(0.01x)$ are presented in Figure 1. Please refer to *problem_set_1.ipynb* for the full Python code.

Figure 1 shows that the fractional error is a considerable improvement on the classic derivative ($f' \sim \frac{f(x+\delta)-f(x)}{\delta}$) as well as on the double-sided derivative ($f' \sim \frac{f(x+\delta)-f(x-\delta)}{2\delta}$), both of which we have seen in class for the $\exp(x)$ function. Therefore my estimate of the optimal δ has to be at least roughly correct.

Problem 2

From class, we have seen that the centered derivative

```
For exp(x) and x = 42
For single precision
Numerical derivative: 1.7392747958643146e+18
True derivative: 1.739274941520501e+18
Fractional error: 8.374534865573935e-08
```

```
For exp(x) and x = 42
For double precision
Numerical derivative: 1.739274941520394e+18
True derivative: 1.739274941520501e+18
Fractional error: 6.150635556423367e-14
```

```
For exp(0.01x) and x = 42
For single precision
Numerical derivative: 0.015219614281614367
True derivative: 0.015219615556186337
Fractional error: 8.374534599120409e-08
```

```
For exp(0.01x) and x = 42
For double precision
Numerical derivative: 0.015219615556186557
True derivative: 0.015219615556186337
Fractional error: 1.4432899320127035e-14
```

Figure 1: Numerical derivative, true derivative, and fractional error computed for $x = 42$ using $\exp(x)$ (top row) and $\exp(0.01x)$ (bottom row), using single precision (left column) and double precision (right column).

$$f' \sim \frac{f(x+dx) - f(x-dx)}{2dx}$$

yields an optimal dx of

$$dx \sim \left(\frac{\epsilon f}{f'''}\right)^{\frac{1}{3}}$$

We can compute f''' numerically using the following trick:

$$\begin{aligned} f''' &\sim \frac{f''(x) - f''(x-dx)}{dx} \\ &\sim \frac{\frac{f'(x+dx)-f'(x)}{dx} - \frac{f'(x)-f'(x-dx)}{dx}}{dx} \\ &\sim \frac{f'(x+dx) + f'(x-dx) - 2f'(x)}{dx^2} \\ &\sim \frac{\frac{f(x+dx)-f(x)}{dx} + \frac{f(x)-f(x-dx)}{dx} - 2\frac{f(x+dx)-f(x)}{dx}}{dx^2} \\ &\sim \frac{-f(x+dx) - f(x-dx) + 2f(x)}{dx^3} \end{aligned} \tag{5}$$

Here, we choose a random dx in order to compute f''', which we plug into the equation for dx. Repeating this a few times allows us to zero in on the optimal dx. The error on the centered derivative is given by

$$err = \frac{\epsilon f}{dx} + dx^2 f''' \tag{6}$$

as seen in class. An example of the output of my ndiff function is shown in Figure 2.

Problem 3

```
# Example:
print(ndiff(np.exp, 42, full=True))
```

```
(1.7392748827369134e+18, 1.940262771281828e-08, 1.425624694099209e-23)
```

Figure 2: Output of my ndiff function for $\exp(x)$ at $x = 42$.

I chose the cubic spline interpolation method for Problem 3. I apply my function to the data contained in *lakeshore.txt* for 2000 voltage points I wish to interpolate and present the resulting plot in Figure 3.

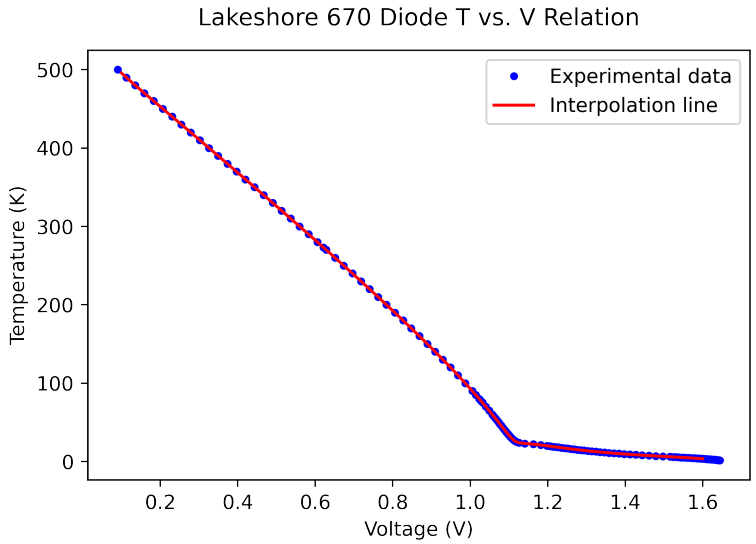


Figure 3: Experimental data is shown in blue. 2000 data points found between the min and max experimental points are interpolated using the cubic spline interpolation method and shown in red.

For the errors, I used the bootstrap resampling method to generate a whole bunch of interpolation values and average them out. For a smaller number of points, I get something like what I show in Figure 4.

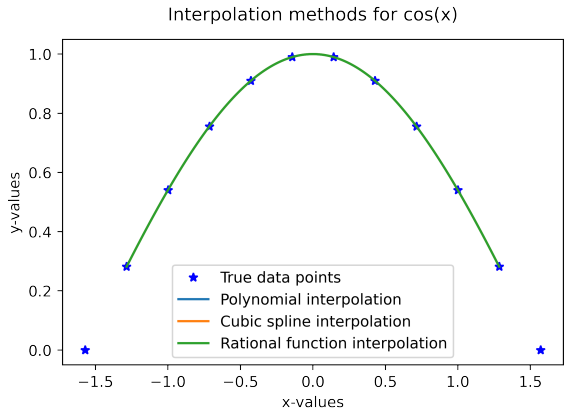
```
Voltage points: [0.1 0.2 0.3 0.4]
Interpolated temperature points: [495.66893911 452.82275501 410.91132944 368.51045156]
Error on each interpolated point: [1.70848689 0.57140311 0.34862139 0.40835209]
Fractional error on each interpolated point: [0.00344683 0.00126187 0.00084841 0.00110812]
Mean fractional error: 0.001666306382244714 +- 0.0010385527543987814
```

Figure 4: Voltage data points interpolated using the cubic spline method. Errors estimated using the bootstrap resampling method.

Problem4

For $\cos(x)$ between $-\frac{\pi}{2}$ and $\frac{\pi}{2}$, the results for the polynomial, cubic spline, and rational

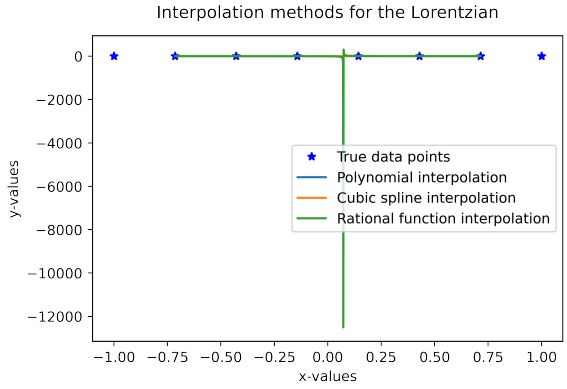
function interpolation are presented in Figure 5.




```
Average error from polynomial interpolation: 7.554360696161543e-05
Average error from cubic spline interpolation: 8.777533254444813e-06
Average error from rational function interpolation: 7.985747410995795e-09
```

Figure 5: (Left) Polynomial, cubic spline, and rational function interpolation for $\cos(x)$ for 1000 x-values between $-\frac{\pi}{2}$ and $\frac{\pi}{2}$. Visually, they all look identical. (Right) Average error for each interpolation method.

For the Lorentzian between -1 and 1, the results for the polynomial, cubic spline, and rational function interpolation are presented in Figure 6. These were performed with $n = 4$ and $m = 5$ for the rational function interpolation. The rational function interpolation looks terrible – For high order, I would expect much smaller errors.




```
Average error from polynomial interpolation: 0.0009317327665490618
Average error from cubic spline interpolation: 0.0002940374073233385 4
Average error from rational function interpolation: 16.556320004189498
```

Figure 6: (Left) Polynomial, cubic spline, and rational function interpolation for the Lorentzian for 1000 x-values between -1 and 1. The rational function interpolation looks terrible! (Right) Average error for each interpolation method.

The plot without the rational function interpolation is given in Figure 7.

Now, if I switch from np.linalg.inv to np.linalg.pinv when doing the rational function interpolation, I get the plot shown in Figure 8. It looks a lot better and the error has improved by ~ 17 orders of magnitude! This is more in line with what I expect for $n = 4$, $m = 5$.

Now I will look at p and q in order to understand what happened. The terms for when I use np.linalg.inv vs. when I use np.linalg.pinv are shown in Figure 9. Here the 'good' terms are much smaller than the 'bad' terms for both p and q.

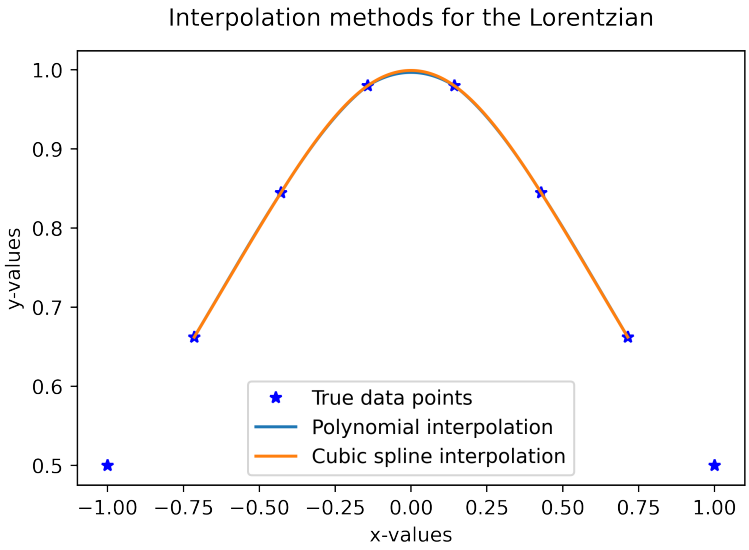
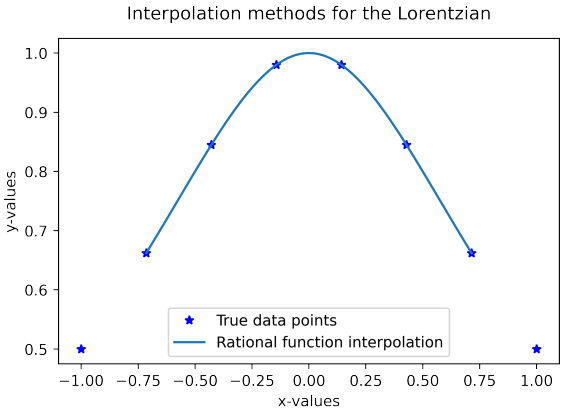


Figure 7: Polynomial and cubic spline interpolation for the Lorentzian for 1000 x-values between -1 and 1. They don't look too bad!




```
Average error from rational function interpolation: 2.6008721206253665e-16
```

Figure 8: (Left) Rational function interpolation for the Lorentzian for 1000 x-values between -1 and 1 while using np.linalg.pinv. (Right) Average error.

```
p term in rational function: [ 1.00000000e+00  1.33226763e-15 -3.33333333e-01 -1.77635684e-15]
q term in rational function: [ 1.77635684e-15  6.66666667e-01 -8.88178420e-16 -3.33333333e-01]
```

(a)

```
p term in rational function: [ -4.29862069 -20.          2.          3.96729867]
q term in rational function: [-14.   4.  -2.   0.]
```

(b)

Figure 9: (a) p and q terms for the rational function interpolation when np.linalg.pinv is used. (b) p and q terms for the rational function interpolation when np.linalg.inv is used.

I can understand this by looking at the function that computed the rational function interpolation, shown in Figure 10.

```python
def rat_eval(p,q,x):
    ''' From PHYS 512 Lecture 2'''
    top=0
    for i in range(len(p)):  # Evaluate the top polynomial
        top=top+p[i]*x**i
    bot=1
    for i in range(len(q)):  # Evaluate the bottom polynomial
        bot=bot+q[i]*x**(i+1)
    return top/bot  # Return the rational function
```

Figure 10: Rational function evaluator (from Lecture 2).

We have to fix the *bot* variable to 1 before beginning the iteration because, otherwise, any x values close to zero would give a very very small denominator, which in turn would make the rational function huge. For the 'good' rational fit, p and q terms are several orders of magnitude apart, while for the 'bad' fit, they are barely one order of magnitude apart. Since the 'bad' coefficients are also on average larger, getting multiplied by powers of x would have affected them a lot more, leader to the weird pattern we can observe in Figure 6. On the other hand, very small values of p and q won't be bothered to much by an extra factor of x^i or x^{i+1}.