PHYS 512 Assignment 1

Liam Fitzpatrick

September 2022

1 Question 1

a)

$$D_\delta = \frac{f(x+\delta) - f(x-\delta)}{2\delta} = \frac{\big(f(x) + f'(x)\delta + f''(x)\delta^2/2 + ...\big) - \big(f(x) - f'(x)\delta + f''(x)\delta^2/2 + ..\big)}{2\delta}$$

$$= f'(x) + f'''(x)\delta^2/3!...$$

$$\Rightarrow Err(D_\delta) \sim \frac{1}{6} f'''(x)\delta^2$$

$$D_{2\delta} = \frac{f(x+2\delta) - f(x-2\delta)}{4\delta} = \frac{\big(f(x) + f'(x)2\delta + f''(x)4\delta^2/2 + ...\big) - \big(f(x) - f'(x)2\delta + f''(x)4\delta^2/2 + ...\big)}{4\delta}$$

$$= f'(x) + f'''(x)4\delta^2/3! + ...$$

$$\Rightarrow Err(D_{2\delta}) \sim \frac{2}{3} f'''(x)\delta^2$$

In order to cancel errors of order δ^2 for the numerical derivative, we can combine the two derivatives taken above as follows into a new derivative operator:

$$D_{new} = \frac{4D_\delta - D_{2\delta}}{3} = \frac{4f'(x) - f'(x) + \frac{4}{6}f'''(x)\delta^2 - \frac{2}{3}f'''(x)\delta^2}{3} + O(\delta^4) = f'(x) + O(\delta^4)$$

The operator is therefore:

$$D_{new} = \frac{4}{3}\frac{f(x+\delta) - f(x-\delta)}{2\delta} - \frac{1}{3}\frac{f(x+2\delta) - f(x-2\delta)}{4\delta}$$

b)
As in the lectures, the operator is modified a factor of $(1 + g\epsilon)$ multiplying each function call (where ϵ is the machine precision error of 10^{-16}). Due to alternating signs, odd powers of δ cancel and leave only $O(\delta^4)$ terms, therefore the error of the combined derivative operator goes like:

$$Err \sim \frac{\epsilon f(x)}{\delta} + f^{(5)}(x)\delta^4$$

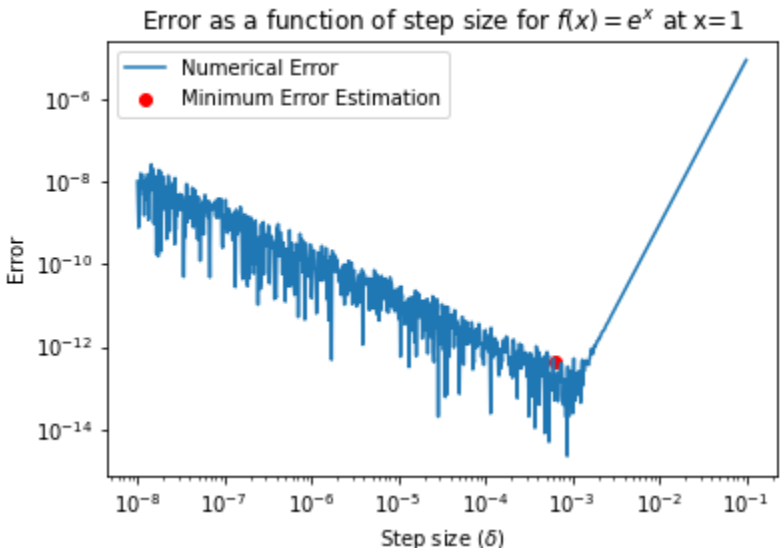
Setting the derivative w.r.t. δ equal to 0:

$$-\frac{\epsilon f(x)}{\delta^2} + 4f^{(5)}(x)\delta^3 = 0 \Rightarrow \delta = \left(\frac{\epsilon f(x)}{4f^{(5)}(x)}\right)^{\frac{1}{5}}$$

For $f(x) = e^x$, $f(x)$ is of the same order of magnitude as $f^{(5)}(x)$,

$$\delta \sim \epsilon^{\frac{1}{5}} = (10^{-16})^{\frac{1}{5}} = 10^{-3.2}$$

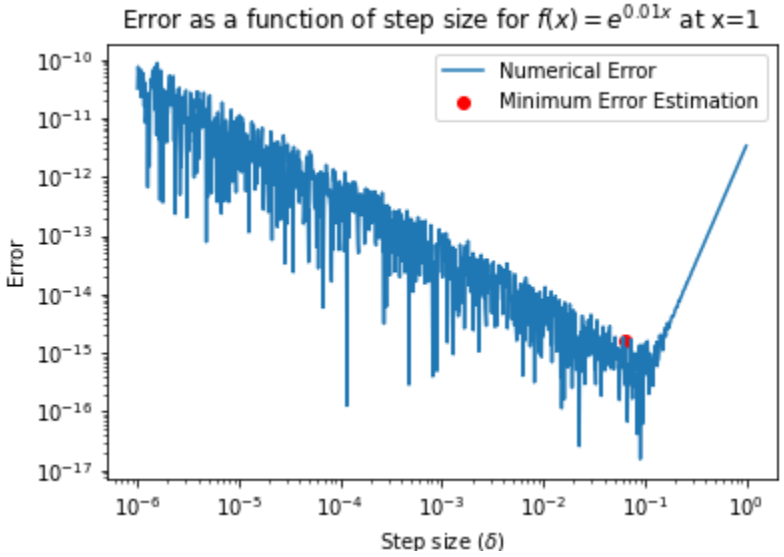
This result is confirmed in the graph below:



For $f(x) = e^{0.01x}$,

$$\frac{f(x)}{f^{(5)}(x)} = 10^{10} \Rightarrow \delta \sim (10^{-16} \cdot 10^{10})^{\frac{1}{5}} = 10^{-1.2}$$

This result is also confirmed in the graph below:



Here is the block of code for $f(x) = e^{0.01x}$:

```
import numpy as np
from matplotlib import pyplot as plt

logdelta=np.linspace(-6, 0, 1001)
delta=10**logdelta #Create points space apart exponentially

def fun(x):
    return np.exp(0.01*x)

x0=1
eps=10**(-16)  #Machine precision error

y0_1=fun(x0-delta)
y1_1=fun(x0+delta)
d_1=(y1_1-y0_1)/(2*delta) #Derivative operator from +/-delta

y0_2=fun(x0-2*delta)
y1_2=fun(x0+2*delta)
d_2=(y1_2-y0_2)/(4*delta) #Derivative operator from +/-2*delta

d=4/3*d_1-1/3*d_2  #Combined derivative operator cancels delta^2 terms

delta_est=10**(-1.2)  #Estimate of delta with minimum error
approx_err=0.01**5*fun(x0)*delta_est**4
# ^ Approximate error arising from delta (0.01^5 from fifth derivtive)

plt.loglog(delta, np.abs(d-0.01*fun(x0)), label='Numerical Error')
plt.scatter(delta_est, approx_err, color='red', label='Minimum Error Estimation')
plt.xlabel(r'Step size ($\delta$)')
plt.ylabel('Error')
plt.title(r'Error as a function of step size for $f(x)=e^{0.01x}$ at x=1')
plt.legend()
```

2 Question 2

As was derived in lecture 1, the order of magnitude estimate for the error on the centered derivative is:

$$Error = \frac{\epsilon f(x)}{dx} + f'''(x)dx^2$$

Setting the derivative equal to 0,

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

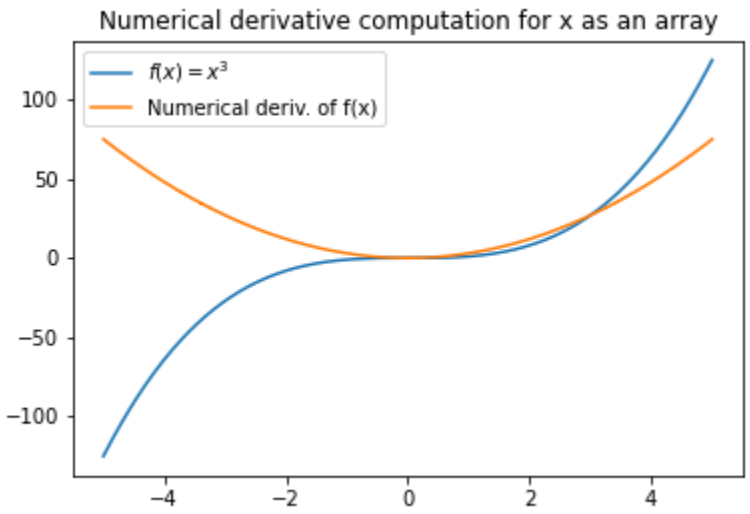
Therefore in order to find the optimal dx, we must first calculate the third derivative numerically which is done as such:

$$f'''(x) \simeq \frac{f(x+2dx) - 2f(x+dx) + 2f(x-dx) - f(x-2dx)}{2dx^3}$$

To calculate this third derivative, a rough estimate of $dx = 10^{-5}$ was used (not important to find optimal dx in this case since it is the "error of the error"). With the optimal dx calculated, the numerical derivative can be evaluated as such:

```python
import numpy as np
from matplotlib import pyplot as plt

eps=10**(-16)  #Machine precision error

def ndiff(fun, x, full=False):
    dx=10**-5

    y2=fun(x+2*dx)
    y1=fun(x+dx)
    y_1=fun(x-dx)
    y_2=fun(x-2*dx)
    d3=(y2-2*y1+2*y_1-y_2)/(2*dx**3) #3rd num. deriv. with dx rough estimate

    dx=np.abs(eps*fun(x)/d3)**(1/3) #Estimate for optimal dx using 3rd deriv.
    d=(fun(x+dx)-fun(x-dx))/(2*dx)
    err=eps*fun(x)/dx+d3*dx**2

    if full==True:
        return d, dx, err
    return d
```

As an example to show that the function works when x is an array, here is the plot of the numerical derivative for $f(x) = x^3$:

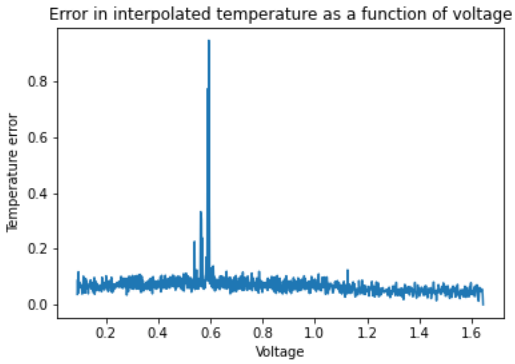


3 Question 3

Using SciPy, the data can be easily interpolated with a spline using scipy.interpolate. In order to estimate the error of an interpolated temperature, a random sub-sampling of intervals of voltage were fitted with splines. These were used take the standard deviation of the temperatures from each spline at the same voltage V. To ensure the intervals had enough points to perform a cubic spline (4), the first random number (start of interval) was made no less than 4 points away from the last voltage point and the second random number (end of interval) was made no less than 4 points away from the first random number. Since the input V to the function lakeshore(V, data) can an array, this whole process was looped through for each possible value of V to avoid comparing arrays.

```
def lakeshore(V, data):
    t=data[:,0] #Temperature
    v=data[:,1] #Voltage

    spl=interp.splrep(t,v)
    t_fine=np.linspace(t[0],t[-1],1001)  #Creating spline
    V_fine=interp.splev(t_fine, spl)
    t_interp=list()
    t_err=list()
    for p in range(len(V_fine)): #Loop through V's individually to avoid comparing arrays
        i=np.abs(V_fine-V[p]).argmin() #Finding index of spline value closest to V
        t_interp.append(t_fine[i])  #Interpolated T given V

        #Calculating error
        t_diff=list()  #Initialize list of T differences
        for k in range(0,25): #25 samples to calculate standard deviation
            n=np.size(t)

            r1=-1  #Initial values so while loop runs
            r2=0
            while V[p]>v[r1] or V[p]<v[r2]:  #Ensure V is in random interval
                r1=random.randint(0,(n-1)-4)  #Leave Room for at least 4 points
                r2=random.randint(r1+4,(n-1)) #Start so that interval has at least 4 points

            sub_spl=interp.splrep(t[r1:r2],v[r1:r2]) #Create sub spline
            sub_t_fine=np.linspace(t[r1],t[r2],1001)
            sub_V_fine=interp.splev(sub_t_fine, sub_spl)

            j=(np.abs(sub_V_fine-V[p])).argmin() #Find index of sub spline value closest to V

            t_diff.append(np.abs(sub_t_fine[j]-t_fine[i]))
        t_err.append(np.std(t_diff))
    return t_interp, t_err

v_fine=np.linspace(data[0,1], data[-1,1], 1001)
plt.plot(v_fine, lakeshore(v_fine, data)[1])
plt.xlabel('Voltage')
plt.ylabel('Temperature error')
plt.title('Error in interpolated temperature as a function of voltage')
plt.savefig('A1Q3_plot.png')
```

Here is the error in the interpolated temperature as a function of voltage to show that the function works when V is an array. The error is of order 10^{-1} for almost all points apart from a few.



4 Question 4

In order to interpolate, the code from lectures for each of the interpolation methods were adapted to functions, which were then called to graph each interpolation and its corresponding error. Below are the graphs for each interpolation method on cosine using order=10 (interpolation on the right, error on the left):

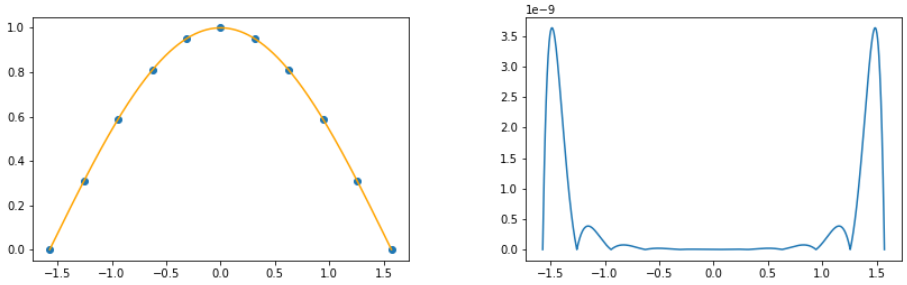


Figure 1: Polynomial interpolation for cosine, maximum error is 3.6×10^{-9}

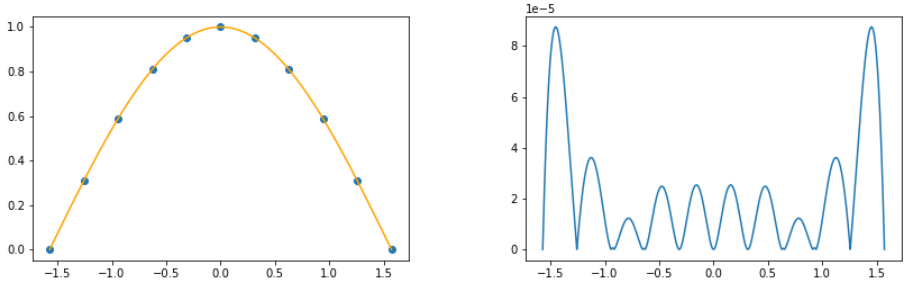


Figure 2: Cubic spline interpolation for cosine, maximum error is 8.8×10^{-5}

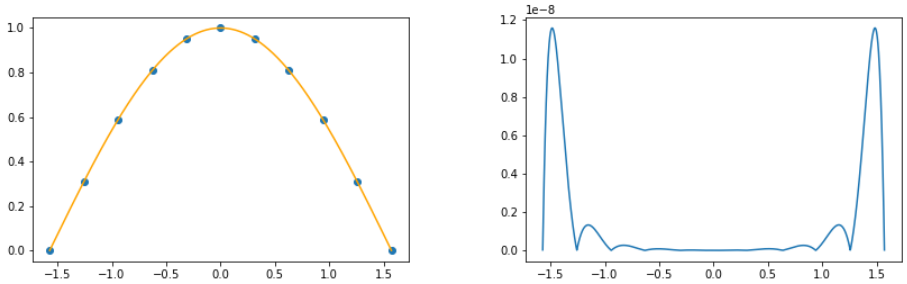


Figure 3: Rational function interpolation for cosine (n=4, m=6 chosen for minimum error), maximum error is 1.2×10^{-8}

For the Lorentz function, the order was chosen as 5 instead. Below are the graphs:

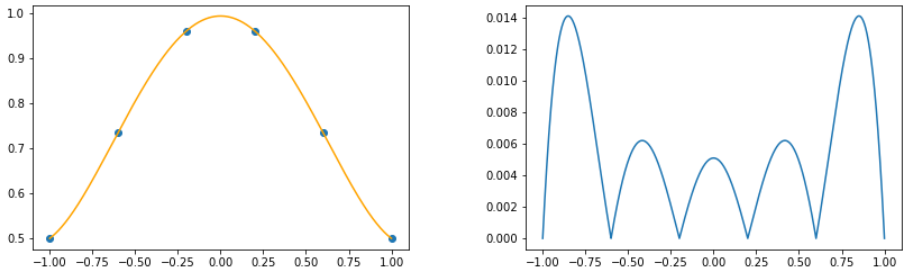


Figure 4: Polynomial interpolation for cosine, maximum error is 1.4×10^{-2}

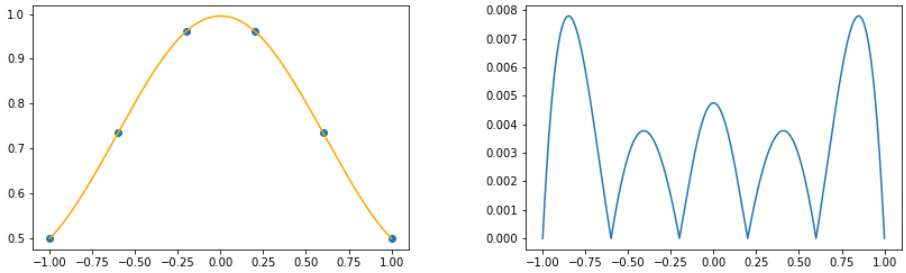


Figure 5: Cubic spline interpolation for cosine, maximum error is 7.8×10^{-3}

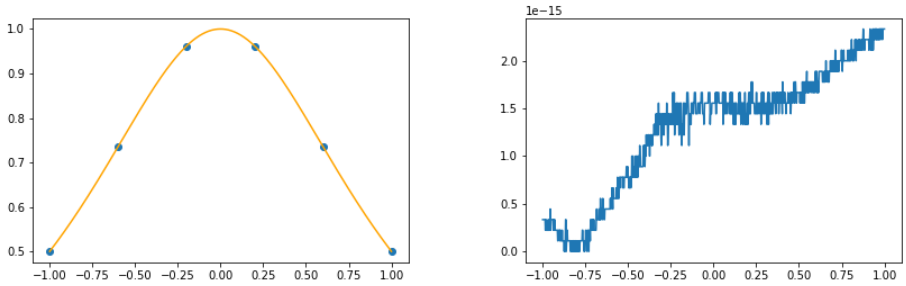


Figure 6: Rational function interpolation for cosine (n=3, m=2 chosen for minimum error), maximum error is 2.3×10^{-15}

As is seen in the rational function interpolation of the Lorentz function, the error is extremely small and the error graph looks very discontinuous and "digital", therefore the error is only due to machine precision. Since the Lorentz function is a rational function, this makes sense as this particular interpolation corresponds to the exact function being interpolated.

When the order is set higher, the function was not able to be fit properly and had very large peaks at certain locations. This indicates over fitting, which is consistent with the fact that the function was able to be perfectly fit using a lower order.

Below is the code for the functions for each method used to interpolate:

```
def lorentz(x):
    return 1/(1+x**2)

def poly_interp(fun, x_fine, x):  #Polynomial interpolation from lectures
    p_sum=0
    for i in range(len(x)):
        x_use=np.append(x[:i],x[i+1:])
        x0=x[i]
        mynorm=np.prod(x0-x_use)
        p0=1.0
        for xi in x_use:
            p0=p0*(xi-x_fine)
        p0=p0/mynorm*fun(x[i])
        p_sum+=p0
    return p_sum

def spline_interp(fun, x_fine, x): #Spline interpolation from lectures
    y=fun(x)
    spl=interp.splrep(x,y)
    y_fine=interp.splev(x_fine,spl)
    return y_fine

def rational_interp(fun, x_fine, x, n): #Rational function interpolation from lectures
    m=ord-n
    y=fun(x)

    pcols=[x**k for k in range(n+1)]
    pmat=np.vstack(pcols)

    qcols=[-x**k*y for k in range(1,m+1)]
    qmat=np.vstack(qcols)
    mat=np.hstack([pmat.T,qmat.T])
    coeffs=np.linalg.inv(mat)@y

    p=0
    for i in range(n+1):
        p=p+coeffs[i]*x_fine**i
    qq=1
    for i in range(m):
        qq=qq+coeffs[n+1+i]*x_fine**(i+1)
    y_pred=p/qq
    return y_pred


#Cosine interpolation
x=np.linspace(-np.pi/2,np.pi/2,ord+1)
x_fine=np.linspace(x[0],x[-1],1001)

plt.scatter(x, np.cos(x))
plt.plot(x_fine, poly_interp(np.cos,x_fine,x), color='orange')
plt.savefig('A1Q4_cos_polyfit.png')
plt.clf()
```

5 References

`https://en.wikipedia.org/wiki/Finite_difference_coefficient` (for numerical derivatives)