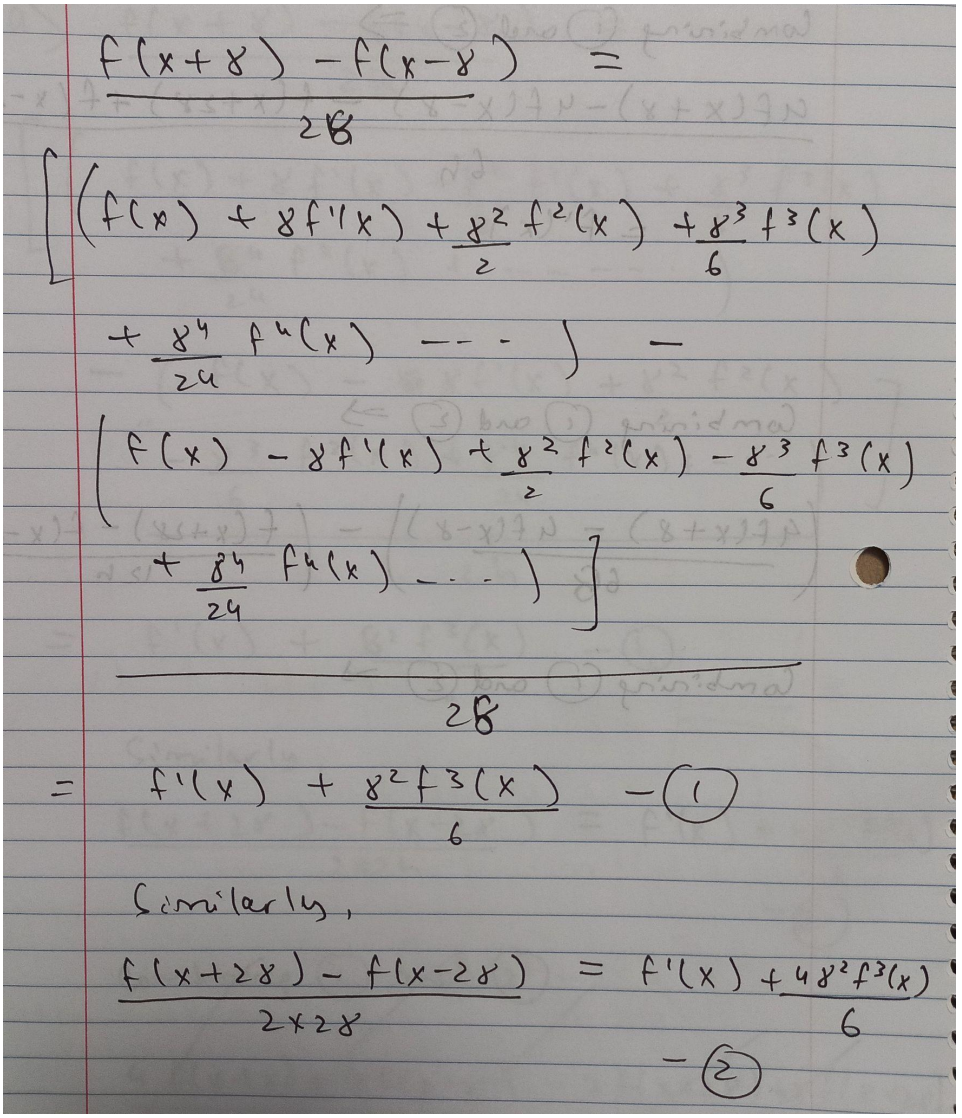
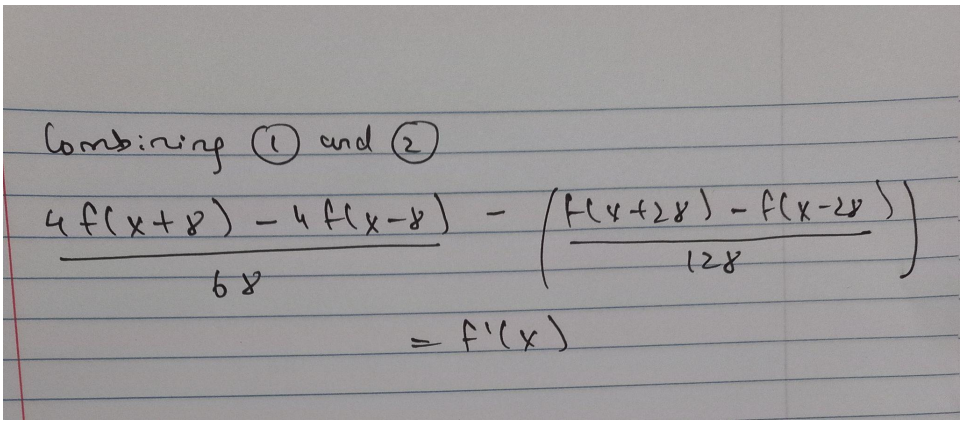
Q1.

a) As derived below,

$f'(x) = [4 \times (f(x + \delta) - f(x - \delta))/(6 \times \delta)] - [(f(x + 2\delta) - f(x - 2\delta))/(12 \times \delta)]$





b) As derived below, the optimal $\delta = (\frac{210 \times f(x) \times \epsilon}{48 \times f^5(x)})^{1/5}$ where $f^5(x)$ is the fifth derivative of f(x) and ϵ is the machine precision (10^{-16})

$$\frac{4f(x+h)-4f(x-h)}{6h} - \frac{f(x+2h)-f(x-2h)}{12h}$$

$$= \frac{h^9}{4}\left\{ f(x) + f'(x)h + \frac{f''(x)h^2}{2} + \frac{f'''(x)h^3}{6} + \frac{f^4(x)h^4}{24} + \frac{f^5(x)h^5}{120} + \cdots \right\}(3\varepsilon+1)$$

$$- \frac{h^9}{6}\left\{ f(x) - f'(x)h + \frac{f''(x)h^2}{2} - \frac{f'''(x)h^3}{6} + \frac{f^4(x)h^4}{24} - \frac{f^5(x)h^5}{120} + \cdots \right\}(3\varepsilon+1)$$

$$- \frac{1}{12h}\left\{ f(x) + f'(x)2h + \frac{f''(x)2h^2}{2} + \frac{f'''(x)8h^3}{6} + \frac{f^4(x)16h^4}{24} + \frac{f^5(x)32h^5}{120} + \cdots \right\}(3\varepsilon+1)$$

$$+ \frac{1}{12h}\left\{ f(x) - f'(x)2h + \frac{f''(x)2h^2}{2} - \frac{f'''(x)8h^3}{6} + \frac{f^4(x)16h^4}{24} - \frac{f^5(x)32h^5}{120} + \cdots \right\}(3\varepsilon+1) \quad \text{--- (1)}$$

Error $= (1) - f'(x)$

$$= \frac{f(x)3\varepsilon}{12h} - \frac{f^5(x)h^4}{30} \qquad (1 \approx \varepsilon)$$

$$\frac{d(\text{Error})}{dh} = 0 \Rightarrow h \approx \left(\frac{f(x)3}{f^5(x)} \times \frac{90}{210} \right)^{1/5}$$

For f(x)=exp(x), I calculate the derivative at x0=1. I define a range of dx values varying from 10^{-15} to 10^{-1}. I get the optimal dx from the formula derived in b). The analytical derivative (d_anal) is just exp(x):

```
fun=np.exp
x0=1
d_anal = fun(x0)
logdx=np.linspace(-15,-1,100)
dx=10**logdx
best_dx = pow(fun(x0)/fun(x0), 1/5)*pow(10,-3.2)*pow(210/48, 1/5)
```

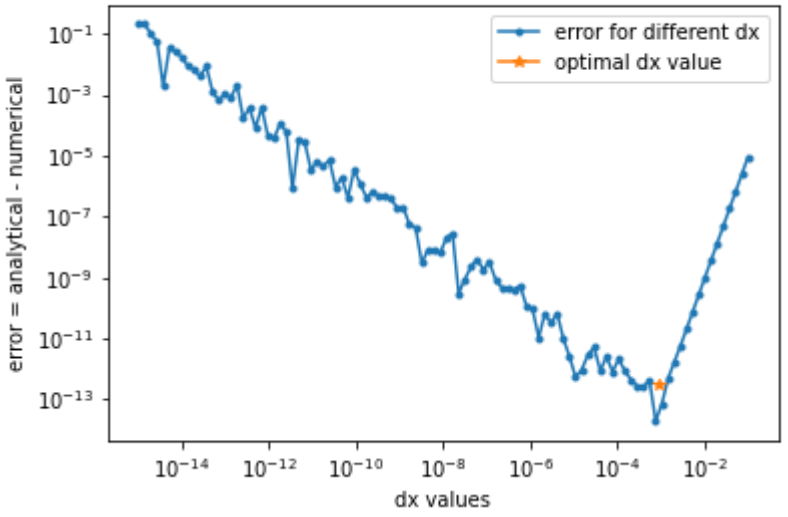
For each of the dx values, I calculate f'(x) using the formula derived in a). The error on the derivative is the difference of this numerically derived derivative and the real analytical derivative:

```
d1=(fun(x0+dx[i]) - fun(x0-dx[i]))/(2*dx[i])
d2=(fun(x0+2*dx[i]) - fun(x0-2*dx[i]))/(2*2*dx[i])
d = (4*d1 - d2)/3
e = np.abs(d-d_anal)
```

Similarly, I derive the the numerical derivative for the optimal dx:

```
best_d1 = (fun(x0+best_dx) - fun(x0-best_dx))/(2*best_dx)
best_d2 = (fun(x0+2*best_dx) - fun(x0-2*best_dx))/(2*2*best_dx)
best_d = (4*best_d1 - best_d2)/3
best_error = np.abs(best_d-d_anal)
```

I plot the error for the various dx values and the optimal dx. As can be seen, the optimal dx I derived is roughly similar to the dx value at which the error is minimum.



For f(x) = exp(0.01x), I calculate the derivative at x0=1. I define a range of dx values varying from 10^{-15} to 10^{2}. The rest of the procedure is same as for exp(x), I just take into account the 0.01 factor in the derivatives (in $f^{5}(x)$ and analytical f'(x)):

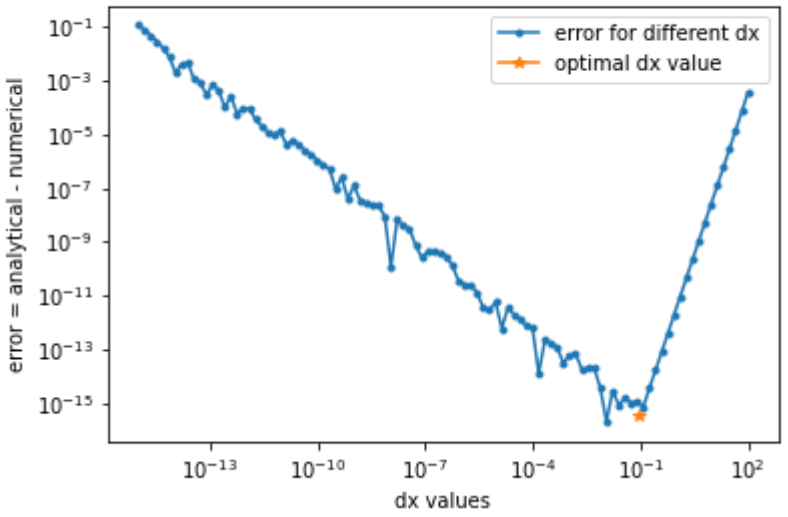
```
fun=np.exp
x0=1
d_anal = 0.01*fun(0.01*x0)
logdx=np.linspace(-15,2,100)
dx=10**logdx
best_dx = pow(fun(0.01*x0)/(pow(0.01,5)*fun(0.01*x0)), 1/5)*pow(10,-3.2)*pow(210/48, 1/5)
```

The derivative for the different dx and their errors are calculated using:

```
d1=(fun(0.01*(x0+dx[i])) - fun(0.01*(x0-dx[i])))/(2*dx[i])
d2=(fun(0.01*(x0+2*dx[i])) - fun(0.01*(x0-2*dx[i])))/(2*2*dx[i])
d = (4*d1 - d2)/3
e = np.abs(d-d_anal)
```

```
best_d1 = (fun(0.01*(x0+best_dx)) - fun(0.01*(x0-best_dx)))/(2*best_dx)
best_d2 = (fun(0.01*(x0+2*best_dx)) - fun(0.01*(x0-2*best_dx)))/(2*2*best_dx)
best_d = (4*best_d1 - best_d2)/3
best_error = np.abs(best_d-d_anal)
```

I plot the error for the various dx values and the optimal dx. As can be seen, the optimal dx I derived is roughly similar to the dx value at which the error is minimum.



Q2.

As we derived in class, the error in the numerical derivative contains the machine precision ϵ = 10^{-16} as the computer cannot differentiate between two values differing by less than ϵ and taylor expansion error in the form of the third derivative of the function ($f^{3}(x)$). Minimising this error gives the optimal dx = $\left(\frac{3\times f(x)\times\epsilon}{4\times f^{3}(x)}\right)^{1/3}$. Inside the function ndiff, I first take an initial guess of dx and use it to calculate $f^{3}(x)$ using the formula

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

Then, using this value of $f^3(x)$, I calculate the optimal dx. I then use the optimal dx to calculate the f'(x) using the formula

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

To get an estimate of the real derivative, I take a range of dx values ranging from 10^{-15} to 10^{-1}. I calculate the f'(x) at these dx using the above formula, and take the mean of all the obtained f'(x) values:

```
logdx=np.linspace(-15,-1,100)
dx=10**logdx
d_num = []
for i in range(len(dx)):
    d_num.append((fun(x+dx[i]) - fun(x-dx[i]))/(2*dx[i]))
d_real = np.mean(d_num)
```

The error in the derivative is then the difference between the numerical derivative and d_real:

```
dx_initial = 0.01
d3 = (fun(x+2*dx_initial) - 2*fun(x+dx_initial) + 2*fun(x-dx_initial) - fun(x-2*dx_initial))/(2*pow(dx_initial,3))
dx_optimal = pow(3/4, 1/3)*pow(fun(x)/d3, 1/3)*pow(10, -5.33)
d = (fun(x+dx_optimal) - fun(x-dx_optimal))/(2*dx_optimal)
error = np.abs(d - d_real)
```

I calculate the derivative of exp(x) at x=1:

```
fun = np.exp
ndiff(fun,1, full=True)
```

```
(2.7182818284440327, 4.2496203725731285e-06, 0.0016672957999142035)
```

The output values are the derivative, the optimal dx and the error.

When full is not set (default is False), then the output is the derivative:

```
fun = np.exp
ndiff(fun,1)
```

```
2.7182818284440327
```

Q3.
I define lakeshore(V, data) such that:

First, to deal with a single value for V (and not an array), I execute this code block in the function:

```
if isinstance(V, float):
    temp = V
    V = np.ndarray(1)
    V[0] = temp
```

To get the error on the predicted temperature values for the input voltage values, I implement bootstrap resampling. I take 10 samples of the data, each with a sample size of 80% the original data. The predicted T values corresponding to each of these samples are stored in the 2D array temp_interp:

```
resamples = 10
sample_len = np.int(0.8*len(data))
temp_interp= [[0]*len(V)]*resamples
```

Then, I randomly get the sample of 80% of the data. Since this is an interpolation, the values in the input array V would lie between the smallest(first) and the largest(last) voltage value in the data. So I make sure that the sample has both these values. (If the sample misses out on the first/last voltage value, this can turn into an extrapolation case for a particular input V value). I then sort the sample:

```
for k in range(resamples):
    data_sample = random.sample(data.tolist(), np.int(0.8*len(data)))
    v = []
    t = []
    for i in range(len(data_sample)):
        v.append(data_sample[i][0])
        t.append(data_sample[i][1])
    if data[0][0] not in v:
        v.append(data[0][0])
        t.append(data[0][1])
    if data[len(data) - 1][0] not in v:
        v.append(data[len(data) -1][0])
        t.append(data[len(data) -1][1])
    voltage = [y for y,_ in sorted(zip(v,t))]
    temp = [y for _,y in sorted(zip(v,t))]
```

I then implement polynomial fit for each of the input V values. I find the index (ind) of the voltage value in the sample which is the closest (on the left) to V[i]. I then take 2 closest points to the left and 2 closest points to the right of V[i] and fit them with a cubic polynomial. If the ind=0, then there is only one point to the left of V[i]. Similarly, if ind+2=len(data), then there is only one point to the right of V[i]. In these cases, I take the 3 points closest to V[i] (one point on the left/right and two on the other side) and fit them with a 2 order polynomial. The value of V[i] is then estimated based on the fit:

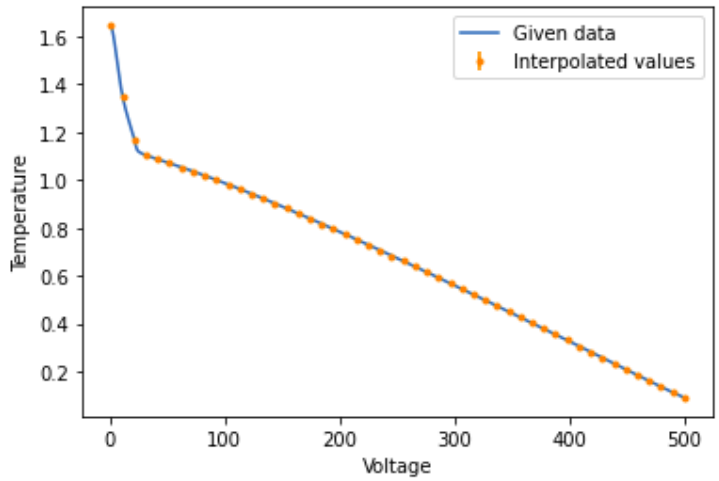
```
for i in range(len(V)):
    ind=np.max(np.where(V[i]>=voltage)[0])
    if (ind-1)>=0 and (ind+2)<len(voltage):
        V_use=voltage[ind-1:ind+3]
        T_use=temp[ind-1:ind+3]
        pars=np.polyfit(V_use,T_use,3)
        T_pred=np.polyval(pars,V[i])
        temp_interp[k][i]=T_pred
    if ind==0:
        V_use=voltage[ind:ind+3]
        T_use=temp[ind:ind+3]
        pars=np.polyfit(V_use,T_use,2)
        T_pred=np.polyval(pars,V[i])
        temp_interp[k][i]=T_pred
    if (ind+2)==len(voltage):
        V_use=voltage[ind-1:ind+2]
        T_use=temp[ind-1:ind+2]
        pars=np.polyfit(V_use,T_use,2)
        T_pred=np.polyval(pars,V[i])
        temp_interp[k][i]=T_pred
```

The predicted temperature values are the mean of the values obtained for the different samples and the uncertainty is the standard deviation:

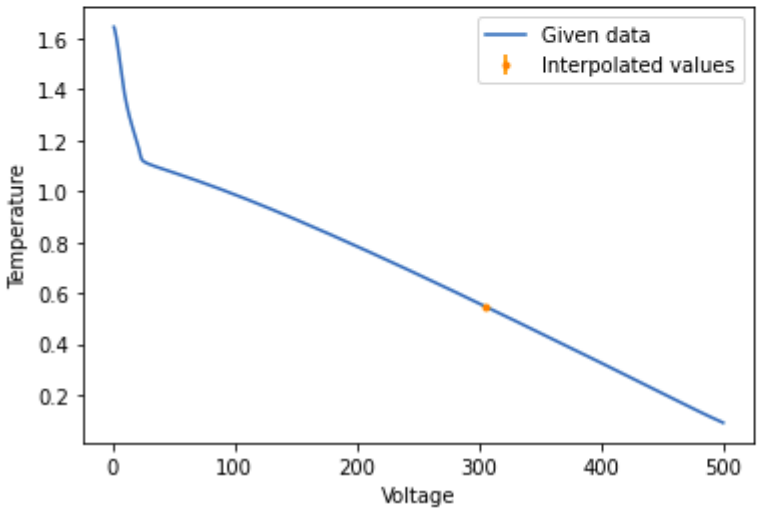
```
temp_pred = []
temp_std = []
for i in range(len(V)):
    temp_resample = []
    for k in range(resamples):
        temp_resample.append(temp_interp[k][i])
    temp_pred.append(np.mean(temp_resample))
    temp_std.append(np.std(temp_resample))
return temp_pred, temp_std
```

As can be seen, the interpolated values closely follow the given data and the error bars are very small:

```
dat=np.loadtxt("lakeshore.txt")
voltage = []
temp = []
for i in range(len(dat)):
    voltage.append(dat[i][0])
    temp.append(dat[i][1])
V = np.linspace(dat[0][0]+0.01, dat[len(dat)-1][0]-0.01, 50)
pred, std = lakeshore(V, dat)
plt.plot(voltage, temp, label='Given data')
plt.errorbar(V, pred, yerr=std, fmt='.', label='Interpolated values')
plt.legend()
plt.xlabel("Voltage")
plt.ylabel("Temperature")
plt.savefig("Q2_2.png")
```



```
dat=np.loadtxt("lakeshore.txt")
voltage = []
temp = []
for i in range(len(dat)):
    voltage.append(dat[i][0])
    temp.append(dat[i][1])
V = 305.23
pred, std = lakeshore(V, dat)
plt.plot(voltage, temp, label='Given data')
plt.errorbar(V, pred, yerr=std, fmt='.', label='Interpolated values')
plt.legend()
plt.xlabel("Voltage")
plt.ylabel("Temperature")
plt.savefig("Q2_2.png")
```



Q4.

For cos(x), I implement the rational fit with the order of P(x)=n and Q(x)=m. I take the code from Jon's code. Since, we are fitting the ratio of these two polynomials to the data points, we would ideally need n+1+m+1 data points to fit the polynomial. But, we set the constant term in Q(x) to 1. Thus, only n+1+m points are needed to get the coefficients of P(x) and Q(x). I use the same number of data points to also fit the cubic spline and polynomial functions to the data:

```
xfine=np.linspace(-np.pi/2,np.pi/2,1001)
fun=np.cos
y_true=fun(xfine)
```

```
#rational fit
n=5 #numerator order
m=4 #denominator order
x=np.linspace(-np.pi/2,np.pi/2,n+m+1)
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]*xfine**i
qq=1
for i in range(m):
    qq=qq+coeffs[n+1+i]*xfine**(i+1)
y_pred_rat=p/qq
rat_error = np.abs(y_true - y_pred_rat)
```

The function is fitted using n+m+1 data points and then the predicted values of y are calculated at the xfine points using the fitted function. The error is the difference between the true value of y (defined by cos(x)) and the predicted value.

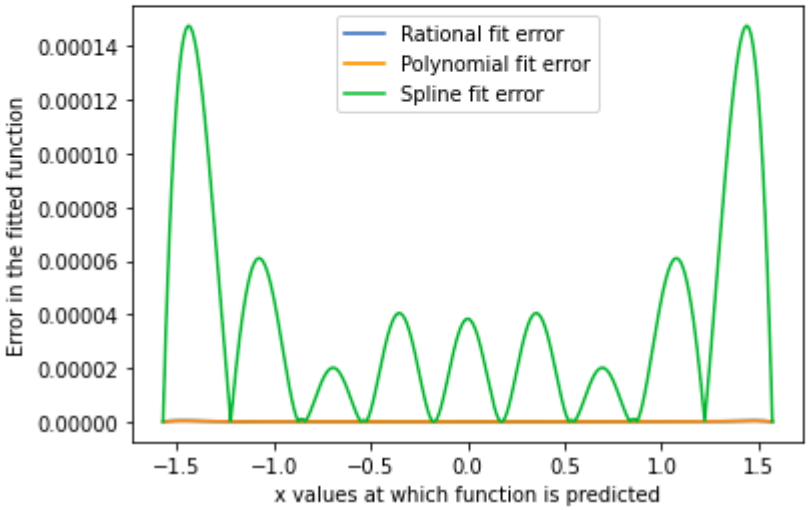
Similarly for the polynomial fit, the order of the polynomial is chosen to be the number of data points we are fitting with - 1, the maximum order polynomial we can fit to those many data points. The polynomial and spline fits are executed as:

```
#polynomial fit
poly_coeffs=np.polyfit(x,y,len(y)-1)
y_pred_poly=np.polyval(poly_coeffs,xfine)
poly_error = np.abs(y_true - y_pred_poly)

#cubic spline
spl=interp.splrep(x,y)
y_pred_spline=interp.splev(xfine,spl)
spline_error = np.abs(y_true - y_pred_spline)
```

The errors for the different fit:

```
plt.plot(xfine,rat_error, label='Rational fit error')
plt.plot(xfine, poly_error, label='Polynomial fit error')
plt.plot(xfine, spline_error, label='Spline fit error')
plt.xlabel("x values at which function is predicted")
plt.ylabel("Error in the fitted function")
plt.legend()
plt.savefig("Q4_1.png")
```



As can be seen, the cubic spline error is the maximum among all the other fits. The spline fit can be made better if we increase the number of data points we are fitting with.

For lorentzian, first attempting to fit the data points with a rational polynomial of n = 4, m = 2:

```
xfine=np.linspace(-1,1,1001)
y_true=[]
for i in xfine:
    y_true.append(1/(1+i**2))
```

```
#rational fit
n=4 #numerator order
m=2 #denominator order

x=np.linspace(-1,1,n+m+1)
y = []
for i in x:
    y.append(1/(1+i**2))

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
print("coefficients for P(x) and Q(x):")
print(coeffs)
p=0
for i in range(n+1):
    p=p+coeffs[i]*xfine**i
qq=1
for i in range(m):
    qq=qq+coeffs[n+1+i]*xfine**(i+1)

y_pred_rat=p/qq
rat_error = np.abs(y_true - y_pred_rat)
```

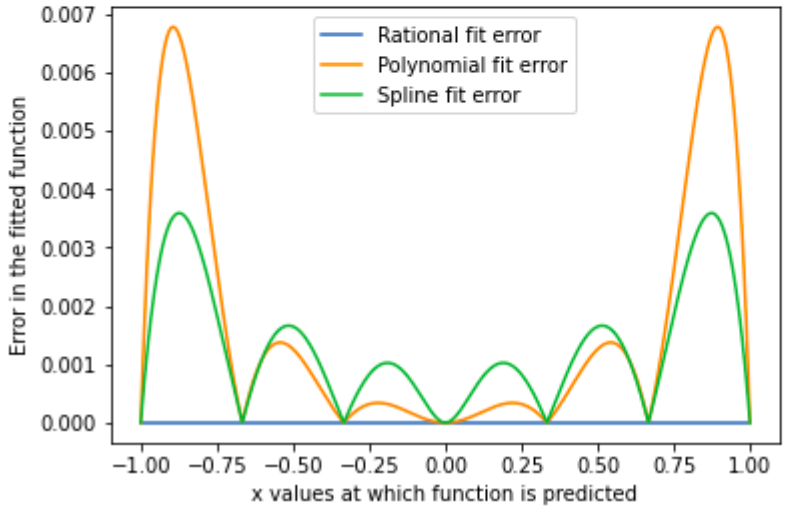
```
#polynomial fit
poly_coeffs=np.polyfit(x,y,len(y)-1)
y_pred_poly=np.polyval(poly_coeffs,xfine)
poly_error = np.abs(y_true - y_pred_poly)

#cubic spline
spl=interp.splrep(x,y)
y_pred_spline=interp.splev(xfine,spl)
spline_error = np.abs(y_true - y_pred_spline)
```

The errors for the different fits are along with the fitted coefficients for the rational polynomials are:

```
plt.plot(xfine,rat_error, label='Rational fit error')
plt.plot(xfine, poly_error, label='Polynomial fit error')
plt.plot(xfine, spline_error, label='Spline fit error')
plt.legend()
plt.xlabel("x values at which function is predicted")
plt.ylabel("Error in the fitted function")
plt.savefig("Q4_2.png")
```

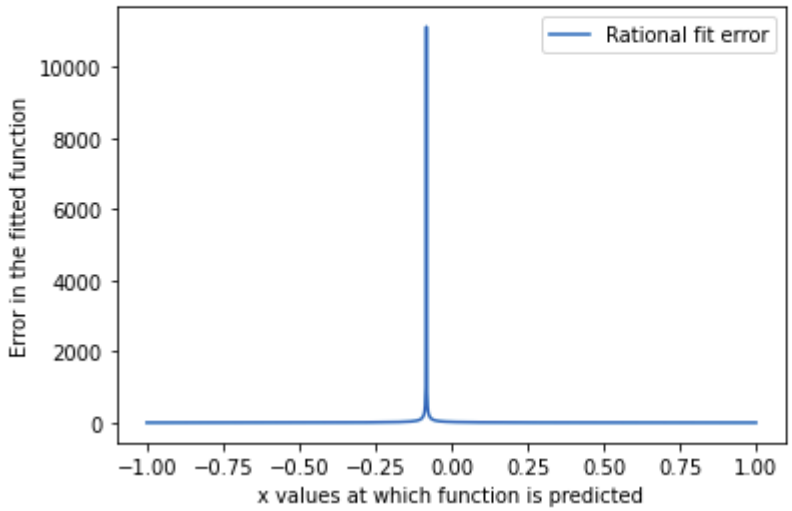
```
coefficients for P(x) and Q(x):
[ 1.00000000e+00 -2.66453526e-15  1.42108547e-14  8.88178420e-16
 -7.10542736e-15  1.77635684e-15  1.00000000e+00]
```



If the order of Q(x) is 2, I would expect the rational function to give almost a zero error. This is because, the lorentzian = $\frac{1}{1+x^2}$ has a second order polynomial in the denominator. So the rational function should perfectly fit the data point. As seen from the above graph, the error for the rational fit is almost zero!

Fitting with rational function of higher order (n=4, m=5):

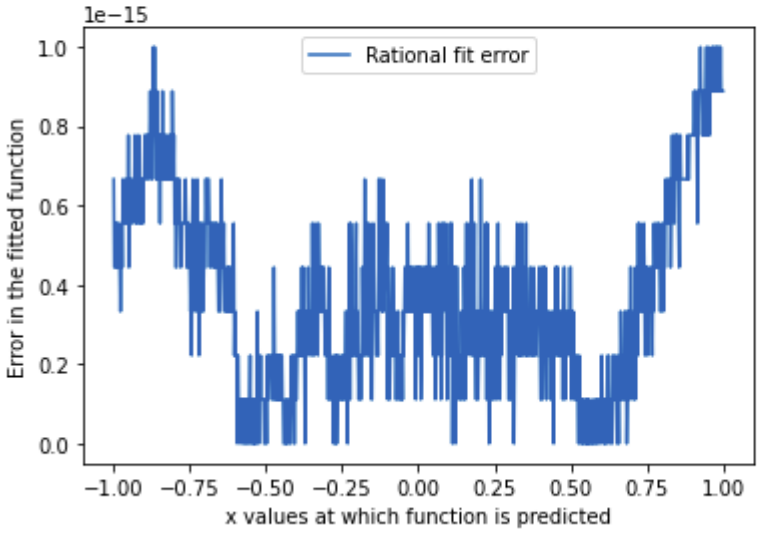
```
coefficients for P(x) and Q(x):
[-21.75840561  12.          -4.          -2.           0.57033844
  12.          -1.5         7.          -0.5         -4.         ]
```



The error shoots up at a particular x and is far from the expected value of 0. As we go to higher orders, it becomes difficult to invert the matrix mat defined in the code above. In that case, the code returns very rough estimates of the coefficients of P(x) and Q(x) leading to a huge error.

Switching from np.linalg.inv to np.linalg.pinv for the same order of P(x) and Q(x) (n=4, m=5) and for the same data points:

```
coefficients for P(x) and Q(x):
[ 1.00000000e+00 -1.11022302e-15 -3.33333333e-01 -6.24500451e-16
 -5.32907052e-15 -1.11022302e-15  6.66666667e-01  0.00000000e+00
 -3.33333333e-01 -2.88657986e-15]
```



As can be seen, pinv handled the singular (non-invertible) matrix and gave much finer and accurate values of the coefficients of P(x) and Q(x) leading to an almost negligible error.