Question 1

a)

We can approximate a derivative with two points using a derivative to the left, and one to the right using

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

or

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

In this problem, I will use the four points $x \pm \delta$ and $x \pm 2\delta$ to approximate the derivative,

To approximate this, we can Taylor expand it:
$f(x \pm \delta) = f(x) \pm \delta f'(x) + \frac{1}{2}\delta^2 f''(x) \pm \frac{1}{3!}\delta^3 f'''(x) + \frac{1}{4!}\delta^4 f^{(4)}(x) \pm \frac{1}{5!}\delta^5 f^{(5)}(x) + \dots$

Similarly, for $f(x \pm 2\delta)$:
$f(x \pm 2\delta) = f(x) \pm 2\delta f'(x) + \frac{2^2}{2}\delta^2 f''(x) \pm \frac{2^3}{3!}\delta^3 f'''(x) + \frac{2^4}{4!}\delta^4 f^{(4)}(x) \pm \frac{2^5}{5!}\delta^5 f^{(5)}(x) + \dots$

This is awful, but we can cancel out some terms: all the non $\pm$ terms will go away if we do $f(x+\delta) - f(x-\delta)$ and $f(x+2\delta) - f(x-2\delta)$ so let's do it. We have:

$$f(x+\delta) - f(x-\delta) \approx 2\delta f'(x) + \frac{1}{3}\delta^3 f'''(x) + \frac{1}{60}\delta^5 f^{(5)}(x), \tag{1}$$

and

$$f(x+2\delta) - f(x-2\delta) = 4\delta f'(x) + \frac{8}{3}\delta^3 f'''(x) + \frac{8}{15}\delta^5 f^{(5)}(x), \tag{2}$$

where I have truncated after the 5th order. We can get a better estimate of $f'(x)$ if we manage to cancel out the $f'''(x)$ term. We need a linear combination of the two derivatives such that:

$$a\left(\frac{f(x+\delta) - f(x-\delta)}{2\delta}\right) + b\left(\frac{f(x+2\delta) - f(x-2\delta)}{4\delta}\right) = \delta f'(x) + c\delta^5 f^{(5)}(x) \tag{3}$$

Putting Equations (1) and (2) in this gives:

$$a\left(f'(x) + \frac{1}{6}\delta^2 f'''(x) + \frac{1}{120}\delta^4 f^{(5)}(x)\right) + b\left(\delta f'(x) + \frac{2}{3}\delta^3 f'''(x) + \frac{2}{15}\delta^5 f^{(5)}(x)\right) \\ = \delta f'(x) + c\delta^5 f^{(5)}(x).$$

This gives the system of equations:

$$\frac{a}{6} + \frac{2b}{3} = 0$$

$$a + b = 1$$

Which has solutions $a = 4/3$ and $b = -1/3$, and this sets c to be $-1/30$. Plugging these values in Equation (3) yields:

$$\frac{4}{3}\left(\frac{f(x+\delta) - f(x-\delta)}{2\delta}\right) - \frac{1}{3}\left(\frac{f(x+2\delta) - f(x-2\delta)}{4\delta}\right) = f'(x) - \frac{1}{30}\delta^4 f^{(5)}(x),$$

and we can solve for $f'(x)$:

$$f'(x) = \frac{8(f(x+\delta) - f(x-\delta)) - (f(x+2\delta) - f(x-2\delta))}{12\delta} + \frac{1}{30}\delta^4 f^{(5)}(x). \qquad (4)$$

The first term is our estimate of the derivative, while the second term ($\Delta = \frac{1}{30}\delta^4 f^{(5)}(x)$) is the error.

b)

Now, to find the best δ possible we need to take into consideration the machine error, which I've ignored until now. Im general, when considering the machine precision (ϵ), we get terms like $g_1 \epsilon f(x)/2\delta$ for Equation (1) and $g_2 \epsilon f(x)/4\delta$ for Equation (2). If we combine these terms together in the same linear combination as in (a) and add them to the error Δ, we get

$$\Delta = \frac{1}{30}\delta^4 f^{(5)}(x) + \frac{\epsilon g}{\delta} f(x), \qquad (5)$$

where I've combined the g_1, g_2, and all the other constants into g. This is our error on the derivative, and what we want to minimize with respect to δ. So let's take the derivative and set it to zero:

$$\frac{d\Delta}{d\delta} = \frac{4}{30}\delta^3 f^{(5)}(x) - \frac{\epsilon g}{\delta^2} f(x) = 0,$$

$$\frac{4}{30}\delta^3 f^{(5)}(x) = \frac{\epsilon g}{\delta^2} f(x)$$

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

$$\delta \approx \left(\frac{15\epsilon}{2} \frac{f(x)}{f^{(5)}(x)}\right)^{1/5} \qquad (6)$$

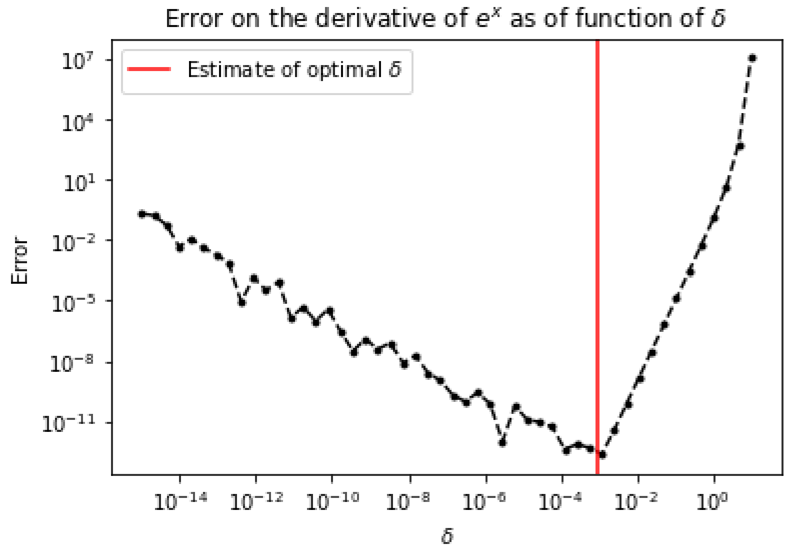
Since we work with exponentials in this question, we can set $f(x)/f^5(x) \approx 1$ since the derivatives of the exponentials look a lot like the exponentials. This gives us

$$\delta \approx \left(\frac{15\epsilon}{2}\right)^{1/5}, \tag{7}$$

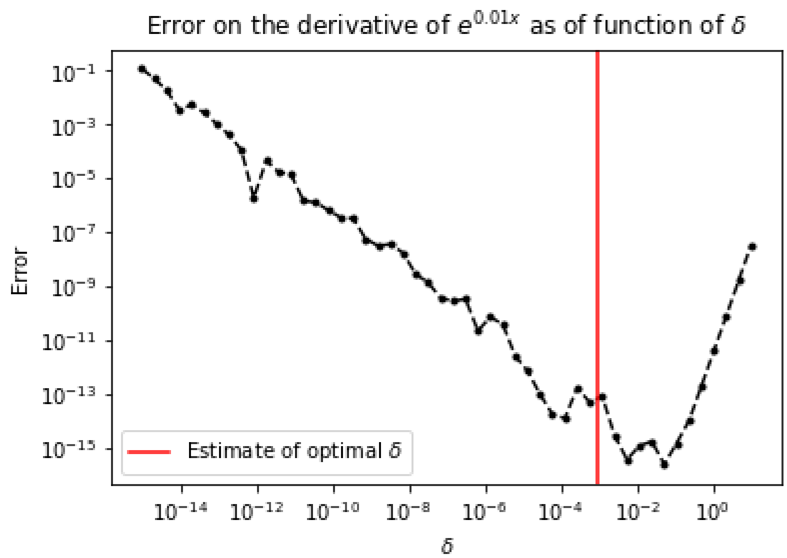
and for double digit precision, $\epsilon = 1e16$. Demonstrations of this below and in q1.ipynb.

In [ ]:
```
# Question 1 b)

import numpy as np
import matplotlib.pyplot as plt



def f1(x):
    return np.exp(x)

def f2(x):
    return np.exp(0.01*x)

def derivative1(x, dx, f):
    return (8*(f(x+dx)-f(x-dx))-(f(x+2*dx)-f(x-2*dx)))/(12*dx)



# Evaluating derivatives and errors
diff_approx1 = derivative1(1, np.logspace(-15, 1, num=50), f1)
diff_true1 = np.exp(1)

diff_approx2 = derivative1(1, np.logspace(-15, 1, num=50), f2)
diff_true2 = 0.01*np.exp(0.01*1)

error1 = np.abs(diff_approx1-diff_true1)
error2 = np.abs(diff_approx2 -diff_true2)

dx = np.logspace(-15, 1, num=50)
eps = 1e-16 # double point precision

# estimate of optimal delta
delta = np.power(15*eps/2,1/5)

# Plotting stuff
plt.plot(dx, error1, 'k.--')
plt.title('Error on the derivative of $e^x$ as of function of $\delta$')
```

```
plt.xscale("log")
plt.yscale("log")
plt.xlabel('$\delta$')
plt.ylabel('Error')
plt.axvline(x = delta, label = 'Estimate of optimal $\delta$', color = 'r')
plt.legend()
plt.show()
print('The estimated optimal delta is', delta, '. We can see in the plot above that it is quite close to the delta that gives us the smallest error.')

plt.plot(dx, error2, 'k.--')
plt.title('Error on the derivative of $e^{0.01x}$ as of function of $\delta$')
plt.xscale("log")
plt.yscale("log")
plt.xlabel('$\delta$')
plt.ylabel('Error')
plt.axvline(x = delta, label = 'Estimate of optimal $\delta$', color = 'r')
plt.legend()
plt.show()
print('The estimated optimal delta is again', delta, '. This time it is much worse, the best delta is around 1e-1.')
print('This is because when we differentiate exp(0.01x), we pull down factors of 0.01. This makes it so f(x)/f^(5)(x) = 1 is not a good approximation anymore.')
```




```
The estimated optimal delta is 0.0009440875112949016 . We can see in the plot above t
hat it is quite close to the delta that gives us the smallest error.
```




```
The estimated optimal delta is again 0.0009440875112949016 . This time it is much wor
se, the best delta is around 1e-1.
This is because when we differentiate exp(0.01x), we pull down factors of 0.01. This
makes it so f(x)/f^(5)(x) = 1 is not a good approximation anymore.
```

Question 2

We went through how to estimate the optimal dx for the centered derivative in class but I will reproduce it here,

$$\frac{f(x+dx)-f(x-dx)}{2dx} \approx \frac{(f(x)+dxf'(x)+\frac{1}{2}dx^2f''(x)+\frac{1}{6}dx^3f'''(x)+\epsilon g_1 f(x)+\dots)}{2dx} -$$

This gives us

$$\frac{f(x+dx)-f(x-dx)}{2dx} \approx f'(x)+\frac{1}{6}dx^2f'''(x)+\frac{\epsilon g f(x)}{dx}, \tag{8}$$

where I've combined g_1 and g_2 into g. The last two terms of Equation (1) are the error Δ on the derivative, so let's minimize that with respect to dx:

$$\frac{d\Delta}{d(dx)} = \frac{1}{3}dx f'''(x) - \frac{\epsilon g f(x)}{dx^2} = 0,$$

$$\frac{1}{3}dx f'''(x) = \frac{\epsilon g f(x)}{dx^2},$$

$$dx \approx \left(\frac{3\epsilon f(x)}{f'''(x)}\right)^{1/3}. \tag{9}$$

The issue here is that we need to approximate the f'''(x) term. I'm not sure if I was supposed to derived this here, but I looked up online an approximation scheme for the third order derivative using central difference (https://en.wikipedia.org/wiki/Finite_difference_coefficient).

$$f'''(x) = \frac{-f(x-2dx) + 2f(x-dx) - 2f(x+dx) + f(x+2dx)}{2dx}. \tag{10}$$

The idea is to use Equation () with some non-optimal dx to estimate the third order derivative, then use that to find the optimal dx, as well as the error Δ. There are some expamples below with simple numpy functions. The code is Q2.ipynb.

In [ ]:

```
# Question 2
import numpy as np
import matplotlib.pyplot as plt

def ndiff(fun, x, full = False):

    eps = 1e-16 # double digit precision
    h = 1e-16 # arbitrary, non-optimal dx

    third_derivative_estimate = (-fun(x-2*h)+2*fun(x-
h)-2*fun(x+h)+fun(x+2*h))/(2*h)

    dx = np.abs(np.cbrt((3*eps*fun(x))/third_derivative_estimate)) #
optimal dx
    err_approx = np.abs((dx**2*third_derivative_estimate)/6 +
(eps*fun(x))/dx) # error estimate

    diff_approx = (fun(x+dx)-fun(x-dx))/(2*dx) # numerical derivative
```

```
    if full == False:
        return diff_approx

    else:
        return diff_approx, dx, err_approx

# some examples with numpy functions
print('For exp(x) at x = 1, the numerical derivative is', ndiff(np.exp, 1,
full = True)[0], ', the actual value is', np.exp(1),'.')
print('The optimal dx and the error estimate are
respectively',ndiff(np.exp, 1, full = True)[1],',',ndiff(np.exp, 1, full =
True)[2],'.' )
print('The real error is', np.abs( ndiff(np.exp, 1, full = True)[0]-
np.exp(1)),'.')
print('\n')
print('For sin(x) at x = 3, the numerical derivative is', ndiff(np.sin, 2,
full = True)[0], ', the actual value is', np.cos(2),'.')
print('The optimal dx and the error estimate are
respectively',ndiff(np.sin, 2, full = True)[1],',',ndiff(np.sin, 2, full =
True)[2],'.' )
print('The real error is', np.abs( ndiff(np.sin, 2, full = True)[0]-
np.cos(2)),'.')
print('\n')
print('For ln(x) at x = 2, the numerical derivative is', ndiff(np.log, 2,
full = True)[0], ', the actual value is', 1/(2),'.')
print('The optimal dx and the error estimate are
respectively',ndiff(np.log, 2, full = True)[1],',',ndiff(np.log, 2, full =
True)[2],'.' )
print('The real error is', np.abs( ndiff(np.log, 2, full = True)
[0]-1/(2)),'.')
print('\n')
print('We get really good numerical derivatives!')
```

```
For exp(x) at x = 1, the numerical derivative is 2.7182818284597934 , the actual valu
e is 2.718281828459045 .
The optimal dx and the error estimate are respectively 4.9653677458619375e-06 , 8.211
723584998573e-11 .
The real error is 7.482903185973555e-13 .


For sin(x) at x = 3, the numerical derivative is -0.41614683654057033 , the actual va
lue is -0.4161468365471424 .
The optimal dx and the error estimate are respectively 7.891307997729034e-06 , 5.7613
85990049813e-12 .
The real error is 6.572076216571077e-12 .


For ln(x) at x = 2, the numerical derivative is 0.5000000000062907 , the actual value
is 0.5 .
The optimal dx and the error estimate are respectively 7.2086757948656255e-06 , 1.442
3186732582125e-11 .
The real error is 6.290745702131062e-12 .


We get really good numerical derivatives!
```

Question 3

For the error estimate, what I did is that for every value of T interpolated, I looked for the closest value of T in the data points and computed the difference betweem the two of them. The function lakeshore() returns the interpolated temperatures and the root mean squared error of the differences.

In [ ]:

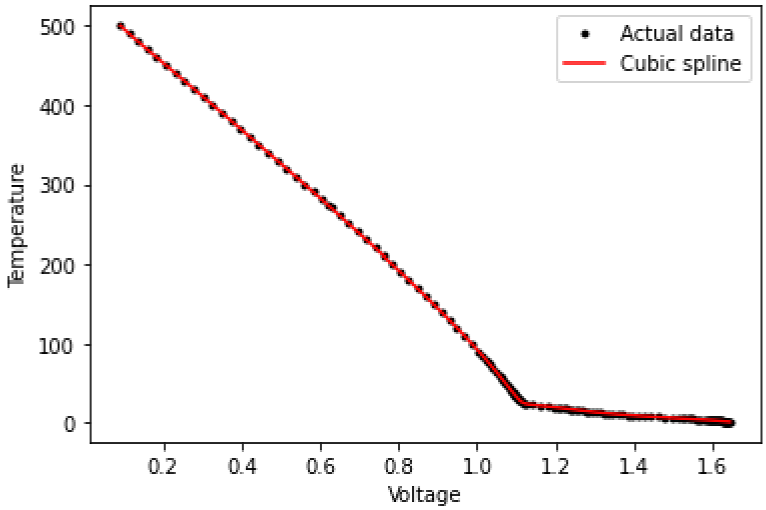
```
# Question 3

import numpy as np
import matplotlib.pyplot as plt
from scipy import interpolate

data = np.loadtxt('lakeshore.txt')

def lakeshore(V, data):
    '''
    data = input data set of temperature vs voltage
    V = voltages you want to know the temperature of (between 0.090681 and
1.644390)

    Function does a cubic spline on the data and returns the interpolated
```

```
temperature and the estimated root
    mean squared error
    '''

    data = data[::-1] # sort in ascending order of voltage
    V_data = data[:,1]
    T_data = data[:,0]

    # Let's do a cubic spline using scipy

    # interpolate.splrep estimates a cubic spline approximation on (x,y)
    spline = interpolate.splrep(V_data,T_data)
    # interpolate.splev returns the spline evaluated at the points given
    T = interpolate.splev(V, spline)

    # estimate of error: take difference between returned interpolated
value and the closest data point for temperature
    err = np.array([])
    for i in range(len(T)):
        all_diff = np.abs(T_data - T[i])
        min_diff = np.min(all_diff)

        err = np.append(err,min_diff)
    rmse = np.sqrt(np.mean(err)**2)
    return T, rmse
```

In [ ]:

```
# test of lakeshore(V, data)

test = np.loadtxt('lakeshore.txt')
test = test[::-1]
V_test = test[:,1]
T_test = test[:,0]

VV_test = np.linspace(V_test[0], V_test[-1], 2000)
test1 = test = np.loadtxt('lakeshore.txt')
TT_test = lakeshore(VV_test, test1)[0]

plt.plot(V_test, T_test, 'k.', label = 'Actual data')
```

```
plt.plot(VV_test, TT_test, 'r', label = 'Cubic spline')
plt.xlabel('Voltage')
plt.ylabel('Temperature')
plt.legend(loc = 'best')
plt.show()
print('The (very rough) error estimate is:', lakeshore(VV_test, test1)[1])
print('Doesnt look like a bad interpolation at all!')
```



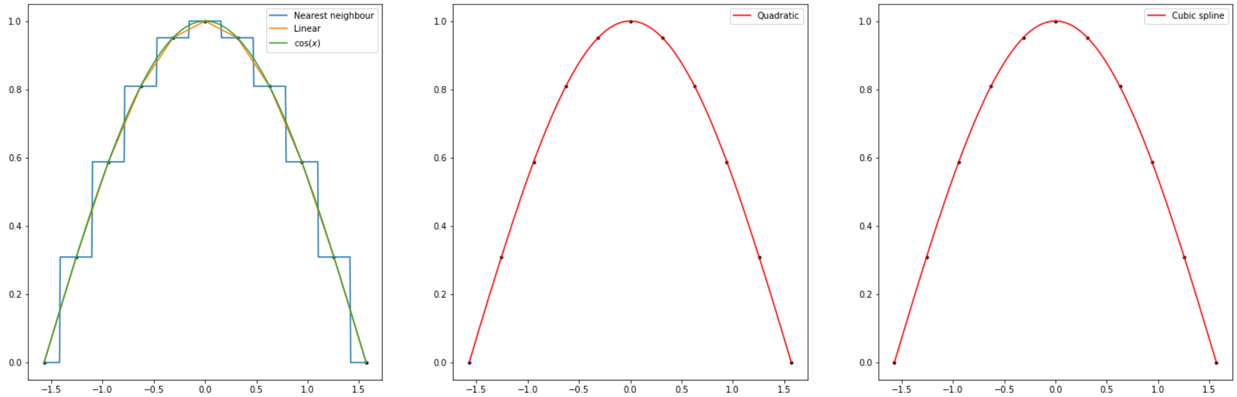

```
The (very rough) error estimate is: 1.5498786095218726
Doesnt look like a bad interpolation at all!
```

Question 4

In [ ]:

```
# Question 4
import numpy as np
import matplotlib.pyplot as plt
from scipy import interpolate

# scipy interpolation for cosine between -pi/2 and pi/2

x = np.linspace(-np.pi/2, np.pi/2, 11)
y = np.cos(x)
x_fine = np.linspace(-np.pi/2, np.pi/2, 500)

# Nearest neighbours
nearest = interpolate.interp1d(x, y, kind = 'nearest')
rmse_nearest = np.sqrt(np.mean((np.cos(x_fine) - nearest(x_fine))**2))

# Linear interpolation
```

```
linear = interpolate.interp1d(x, y, kind = 'linear')
rmse_linear = np.sqrt(np.mean((np.cos(x_fine) - linear(x_fine))**2))

# Quadratic interpolation
quadratic = interpolate.interp1d(x, y, kind = 'quadratic')
rmse_quadratic = np.sqrt(np.mean((np.cos(x_fine) - quadratic(x_fine))**2))

# Cubic spline
cubic = interpolate.interp1d(x, y, kind = 'cubic')
rmse_cubic = np.sqrt(np.mean((np.cos(x_fine) - cubic(x_fine))**2))



fig, ax = plt.subplots(1,3,figsize=(25,8))
ax[0].plot(x,y,'k.')
ax[0].plot(x_fine,nearest(x_fine), label = 'Nearest neighbour')
ax[0].plot(x_fine,linear(x_fine), label = 'Linear')
ax[0].plot(x_fine, np.cos(x_fine), label = '$\cos(x)$')
ax[0].legend(loc ='best')



ax[1].plot(x,y,'k.')
ax[1].plot(x_fine,quadratic(x_fine), 'r', label = 'Quadratic')
ax[1].legend(loc ='best')

ax[2].plot(x,y,'k.')
ax[2].plot(x_fine,cubic(x_fine), 'r', label = 'Cubic spline')
ax[2].legend(loc ='best')
plt.show()



print("The root mean square error on the nearest neighbour interpolation 
is:", rmse_nearest)
print("The root mean square error on linear interpolation is:", 
rmse_linear)
print("The root mean square error on the quadratic interpolation is:", 
rmse_quadratic)
print("The root mean square error on the cubic spline is", rmse_cubic)
```

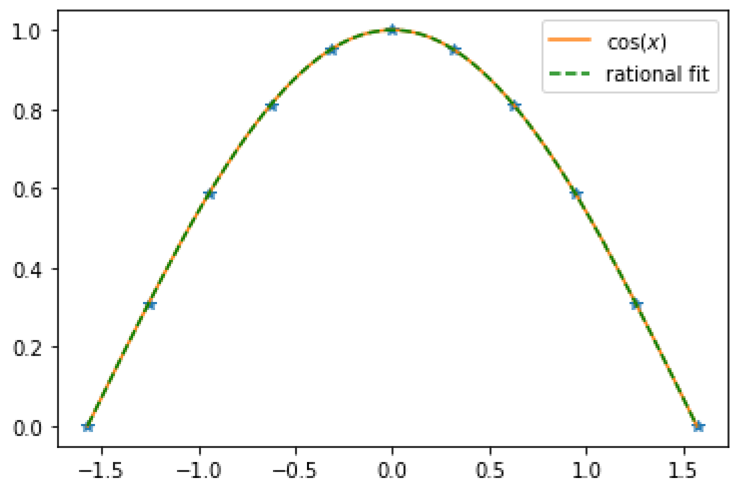



```
The root mean square error on the nearest neighbour interpolation is: 0.0639339392182
442
The root mean square error on linear interpolation is: 0.006350728317241847
The root mean square error on the quadratic interpolation is: 0.00046929553987613945
The root mean square error on the cubic spline is 3.140124235578087e-05
```

In [ ]:

```
# Code for rational interpolation (taken from class)

def rat_eval(p,q,x):
    top=0
    for i in range(len(p)):
        top=top+p[i]*x**i
    bot=1
    for i in range(len(q)):
        bot=bot+q[i]*x**(i+1)
    return top/bot

def rat_fit(x,y,n,m):
    assert(len(x)==n+m-1)
    assert(len(y)==len(x))
    mat=np.zeros([n+m-1,n+m-1])
    for i in range(n):
        mat[:,i]=x**i
    for i in range(1,m):
        mat[:,i-1+n]=-y*x**i
    pars=np.dot(np.linalg.inv(mat),y)
    p=pars[:n]
    q=pars[n:]
    return p,q

# Rat fit for cosine bewteen -pi/2 and pi/2
n=5
```

```
m=7
x=np.linspace(-np.pi/2,np.pi/2,n+m-1)
y=np.cos(x)
p,q=rat_fit(x,y,n,m)
xx=np.linspace(x[0],x[-1],500)
y_true=np.cos(xx)
pred=rat_eval(p,q,xx)
plt.clf();plt.plot(x,y,'*')
plt.plot(xx,y_true, label = '$\cos(x)$')
plt.plot(xx,pred, 'g--', label = 'rational fit')
plt.legend()
plt.show()



rmse_rat = np.sqrt(np.mean((y_true - pred)**2))
print("The root mean squared error on the rational interpolation is:",
rmse_rat)
```



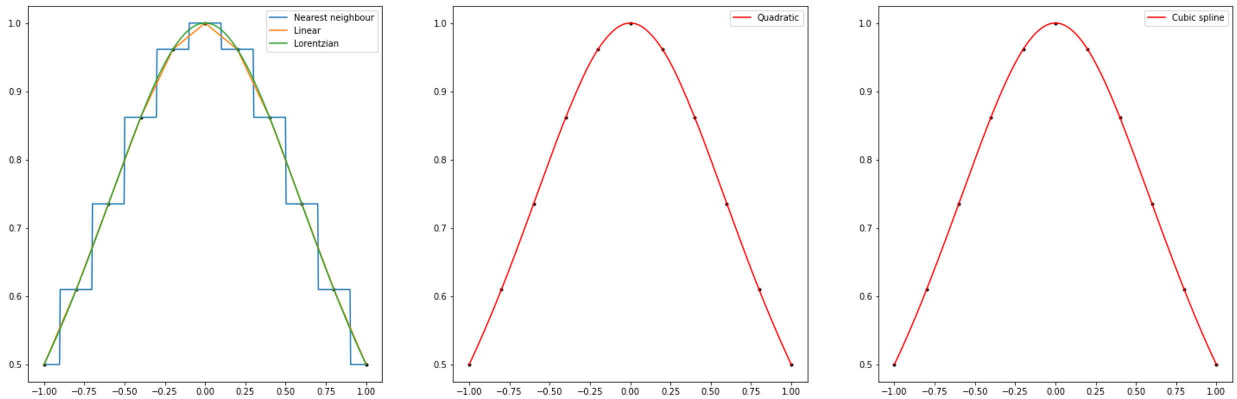

```
The root mean squared error on the rational interpolation is: 3.3758978990759764e-09
```

In [ ]:

```
# Scipy interpolation for a lorentzian between -1 and 1


x = np.linspace(-1, 1, 11)

def lorentzian(x):
    return 1/(1+x**2)

y = lorentzian(x)
x_fine = np.linspace(-1, 1, 500)
```

```
# Nearest neighbours
nearest = interpolate.interp1d(x, y, kind = 'nearest')
rmse_nearest = np.sqrt(np.mean((lorentzian(x_fine) - nearest(x_fine))**2))

# Linear interpolation
linear = interpolate.interp1d(x, y, kind = 'linear')
rmse_linear = np.sqrt(np.mean((lorentzian(x_fine) - linear(x_fine))**2))

# Quadratic interpolation
quadratic = interpolate.interp1d(x, y, kind = 'quadratic')
rmse_quadratic = np.sqrt(np.mean((lorentzian(x_fine) -
quadratic(x_fine))**2))

# Cubic spline
cubic = interpolate.interp1d(x, y, kind = 'cubic')
rmse_cubic = np.sqrt(np.mean((lorentzian(x_fine) - cubic(x_fine))**2))

fig, ax = plt.subplots(1,3,figsize=(25,8))
ax[0].plot(x,y,'k.')
ax[0].plot(x_fine,nearest(x_fine), label = 'Nearest neighbour')
ax[0].plot(x_fine,linear(x_fine), label = 'Linear')
ax[0].plot(x_fine, lorentzian(x_fine), label = 'Lorentzian')
ax[0].legend(loc ='best')



ax[1].plot(x,y,'k.')
ax[1].plot(x_fine,quadratic(x_fine), 'r', label = 'Quadratic')
ax[1].legend(loc ='best')

ax[2].plot(x,y,'k.')
ax[2].plot(x_fine,cubic(x_fine), 'r', label = 'Cubic spline')
ax[2].legend(loc ='best')
plt.show()

print("The root mean square error on the nearest neighbour interpolation
is:", rmse_nearest)
print("The root mean square error on linear interpolation is:",
rmse_linear)
print("The root mean square error on the quadratic interpolation is:",
```

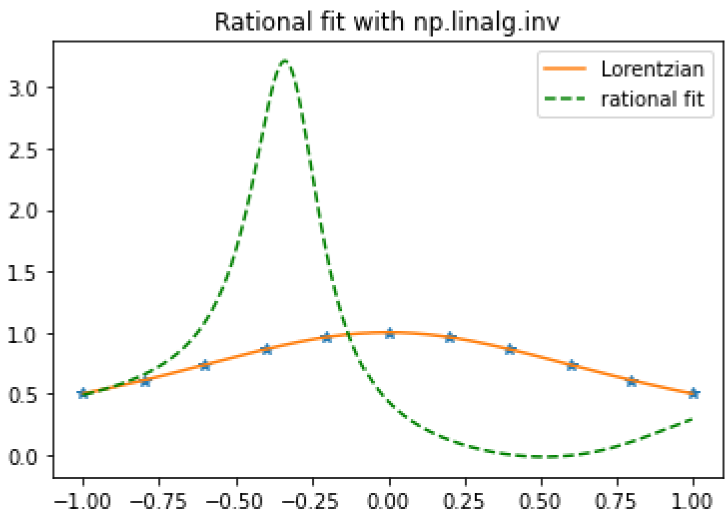
```
rmse_quadratic)
print("The root mean square error on the cubic spline is", rmse_cubic)
```




```
The root mean square error on the nearest neighbour interpolation is: 0.0304642702460
18672
The root mean square error on linear interpolation is: 0.0036913982724037374
The root mean square error on the quadratic interpolation is: 0.00018091510510278392
The root mean square error on the cubic spline is 0.00011125141313629842
```

In [ ]:

```
# rational function fit on the lorentzian


n=5
m=7
x=np.linspace(-1,1,n+m-1)
y=lorentzian(x)
p,q=rat_fit(x,y,n,m)
xx=np.linspace(x[0],x[-1],500)
y_true=lorentzian(xx)
pred=rat_eval(p,q,xx)
plt.clf();plt.plot(x,y,'*')
plt.plot(xx,y_true, label = 'Lorentzian')
plt.plot(xx,pred, 'g--', label = 'rational fit')
plt.title('Rational fit with np.linalg.inv')
plt.legend()
plt.show()


p_inv = p
q_inv = q


rmse_rat = np.sqrt(np.mean((y_true - pred)**2))
print("The root mean squared error on the rational interpolation is:",
rmse_rat)
print('Wow that is really not good!')
```

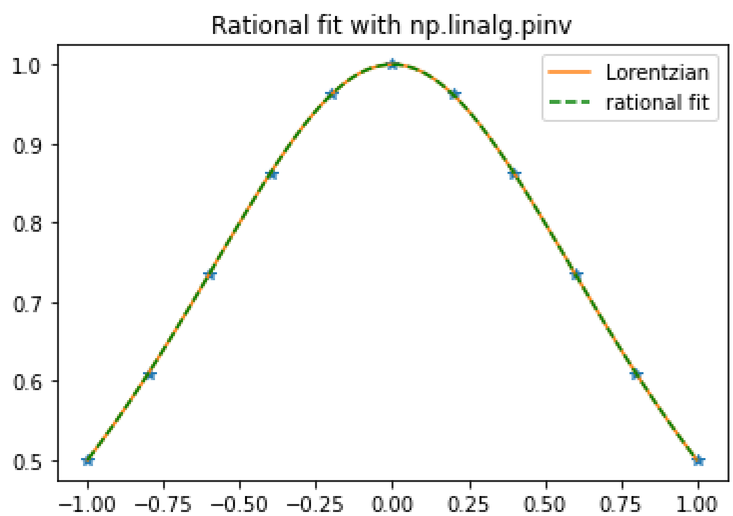



```
The root mean squared error on the rational interpolation is: 0.8500774166773788
Wow that is really not good!
```

In [ ]:

```
# Let's try using np.linalag.pinv in the rational fit


def rat_fit_new(x,y,n,m):
    assert(len(x)==n+m-1)
    assert(len(y)==len(x))
    mat=np.zeros([n+m-1,n+m-1])
    for i in range(n):
        mat[:,i]=x**i
    for i in range(1,m):
        mat[:,i-1+n]=-y*x**i
    pars=np.dot(np.linalg.pinv(mat),y)
    p=pars[:n]
    q=pars[n:]
    return p,q


n=5
m=7
x=np.linspace(-1,1,n+m-1)
y=lorentzian(x)
p,q=rat_fit_new(x,y,n,m)
xx=np.linspace(x[0],x[-1],500)
y_true=lorentzian(xx)
pred=rat_eval(p,q,xx)
plt.clf();plt.plot(x,y,'*')
plt.plot(xx,y_true, label = 'Lorentzian')
```

```
plt.plot(xx,pred, 'g--', label = 'rational fit')
plt.title('Rational fit with np.linalg.pinv')
plt.legend()
plt.show()

p_pinv = p
q_pinv = q

rmse_rat = np.sqrt(np.mean((y_true - pred)**2))
print("The root mean squared error on the rational interpolation is:",
rmse_rat)
print('This much better! Lets look at p and q to try and figure out what
changed.')
print('p and q obtained from np.linalg.inv:', p_inv, q_inv)
print('p and q obtained from np.linalg.pinv:', p_pinv, q_pinv)
```




```
The root mean squared error on the rational interpolation is: 4.134146169365445e-16
This much better! Lets look at p and q to try and figure out what changed.
p and q obtained from np.linalg.inv: [ 0.43701074 -0.75       -1.         -2.
6.        ] [ 4.25  4.   -4.    2.   -4.    6.  ]
p and q obtained from np.linalg.pinv: [ 1.00000000e+00  0.00000000e+00 -3.75000000e-0
1 -4.30211422e-16
  1.25000000e-01] [ 4.4408921e-16  6.2500000e-01  4.4408921e-16 -2.5000000e-01
  4.4408921e-16  1.2500000e-01]
```

There's an enormous difference between the p's and q's obtained using np.linalg.inv (inv) and np.linalg.pinv (pinv). Most of the entries obtained from inv are greater than 1, which is not normal, while the entries obtained from pinv are all less than 1, with most of them being zero. This is probably because the matrix in that case is singular (determinant is zero or close to zero), and np.linalg.pinv is able to handle that.