Homework #1

Garcia, Gilberto

ASTRO 5900 - Computational Physics and Astronomy

1 February 2024

Question 1 references the following videos:

1. video 1 (https://www.youtube.com/watch?v=RpxoN9-i7Jc)
2. video 2 (https://www.youtube.com/watch?v=LaolbjAzZvg)
3. video 3 (https://www.youtube.com/watch?v=4VpE9Tbie14)

Complete the following derivations skipped in the video:

1a

for a_i in the linear interpolation in the first video:

- The equation for linear interpolation between two successive points is

$$g_i = a_i(x - x_i) + b_i$$

We have the following two constraints:

1. $g_i(x_i) = y_i$
2. $g_i(x_{i+1}) = y_{i+1}$

So now we apply our constraints to our equation for linear interpolation:

1. $g_i(x_i) = a_i(x_i - x_i) + b_i = y_i \rightarrow \boxed{b_i = y_i}$
2. $g_i(x_{i+1}) = a_i(x_{i+1} - x_i) + y_i = y_{i+1} \rightarrow a_i(x_{i+1} - x_i) = y_{i+1} - y_i$

$\rightarrow \boxed{a_i = \frac{y_{i+1} - y_i}{x_{i+1} - x_i}}$

1b

for equations labeled "3" and "4" for the coefficients of the cubic spline in the second video

- The equation for cubic spline interpolation is

$$g_i(x) = a_i(x - x_i)^3 + b_i(x - x_i)^2 + c_i(x - x_i) + d_i$$

We have the following four constaints:

1. $g_i(x_i) = y_i$
2. $g_i(x_{i+1}) = y_{i+1}$
3. $g'_i(x_{i+1}) = g'_{i+1}(x_{i+1})$
4. $g''_i(x_{i+1}) = g''_{i+1}(x_{i+1})$

Now let $h_i = x_{i+1} - x_i$ and $\eta_i = y_{i+1} - y_i$ for convenience.

We can apply our constraints:

1. $g_i(x_i) = 0 + 0 + 0 + d_i = y_i \rightarrow \boxed{d_i = y_i}$
2. $g_i(x_{i+1}) = a_i(h_i)^3 + b_i h_i^2 + c_i h_i + y_i = y_{i+1} \rightarrow \boxed{a_i h_i^3 + b_i h_i^2 + c_i h_i = \eta_i}$
3. $g'_{i+1}(x_{i+1}) = 3a_i h_i^2 + 2b_i h_i + c_i$

and

$$g_{i+1}(x) = a_{i+1}(x - x_{i+1})^3 + b_{i+1}(x - x_{i+1})^2 + c_{i+1}(x - x_{i+1}) + d_{i+1}$$

$$g'_{i+1}(x) = 3a_{i+1}(x - x_{i+1})^2 + 2b_{i+1}(x - x_{i+1}) + c_{i+1}(x - x_{i+1})$$

so plugging in x_{i+1}:

$$g'_{i+1}(x) = 0 + 0 + c_{i+1}$$

Therefore applying the constraint:

$$\boxed{3a_i h_i^2 + 2b_i h_i + c_i = c_{i+1}}$$

4. $g''_i(x_{i+1}) = 6a_i h_i + 2b_i$

and

$$g''_{i+1}(x) = 6a_{i+1}(x - x_{i+1}) + 2b_{i+1}$$

so plugging in x_{i+1}:

$$g''_{i+1}(x) = 2b_{i+1}$$

so applying the constraint:

$$\boxed{6a_i h_i + 2b_i = 2b_{i+1}}$$

1c

for the final equation all in terms of just the b coefficients, shown in the first part of the third video:

- from video 3 we have the following:

1. $d_i = y_i$
2. $a_i h_i^3 + b_i h_i^2 + c_i h_i = \eta_i$
3. $3a_i h_i + b_i = b_{i+1}$
4. $3a_i h_i^2 + 2b_i h_i + c_i = c_{i+1}$

We first solve (3) for a_i:

$$3a_i h_i + b_i = b_{i+1} \rightarrow a_i = \frac{b_{i+1}-b_i}{3h_i}$$

We then solve (2) for c_i:

$$a_i h_i^3 + b_i h_i^2 + c_i h_i = \eta_i \rightarrow c_i = \frac{\eta_i}{h_i} - a_i h_i^2 - b_i h_i$$

We plug in a_i into the above equation:

$$c_i = \frac{\eta_i}{h_i} - [\frac{b_{i+1}-b_i}{3h_i}]h_i^2 - b_i h_i$$

Simplifying,

$$c_i = \frac{\eta_i}{h_i} - \frac{b_{i+1}h_i}{3} - \frac{2}{3}b_i h_i$$

Similarly we will get that

$$c_{i+1} = \frac{\eta_{i+1}}{h_{i+1}} - \frac{b_{i+2}h_{i+1}}{3} - \frac{2}{3}b_{i+1}h_{i+1}$$

We can now plug in the above results into equation 4:

$$3[\frac{b_{i+1}-b_i}{3h_i}]h_i^2 + 2b_i h_i + [\frac{\eta_i}{h_i} - \frac{b_{i+1}h_i}{3} - \frac{2}{3}b_i h_i] = [\frac{\eta_{i+1}}{h_{i+1}} - \frac{b_{i+2}h_{i+1}}{3} - \frac{2}{3}b_{i+1}h_{i+1}]$$

Simplyfing,

$$(b_{i+1} - b_i)h_i + \frac{4}{3}b_i h_i + \frac{\eta_i}{h_i} - \frac{b_{i+1}h_i}{3} = \frac{\eta_{i+1}}{h_{i+1}} - \frac{b_{i+2}h_{i+1}}{3} - \frac{2}{3}b_{i+1}h_{i+1}$$

$$\frac{2}{3}b_{i+1}h_i + \frac{1}{3}b_i h_i + \frac{\eta_i}{h_i} = \frac{\eta_{i+1}}{h_{i+1}} - \frac{1}{3}b_{i+2}h_{i+1} - \frac{2}{3}b_{i+1}h_{i+1}$$

$$\frac{1}{3}b_i h_i + (\frac{2}{3}b_{i+1}h_i + \frac{2}{3}b_{i+1}h_{i+1}) + \frac{1}{3}b_{i+2}h_{i+1} = \frac{\eta_{i+1}}{h_{i+1}} - \frac{\eta_i}{h_i}$$

$$\boxed{\frac{1}{3}b_i h_i + \frac{2}{3}b_{i+1}(h_i + h_{i+1}) + \frac{1}{3}b_{i+2}h_{i+1} = \frac{\eta_{i+1}}{h_{i+1}} - \frac{\eta_i}{h_i}}$$

question 2

create a function that takes in a set of datapoints (x,y) and returns the y value of any x within the dataset range:

We first import the required libraries:

In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

We read in the given dataset:

In [2]:
```
df = pd.read_csv('HW01_data.txt',sep='\s+',header=0)
```

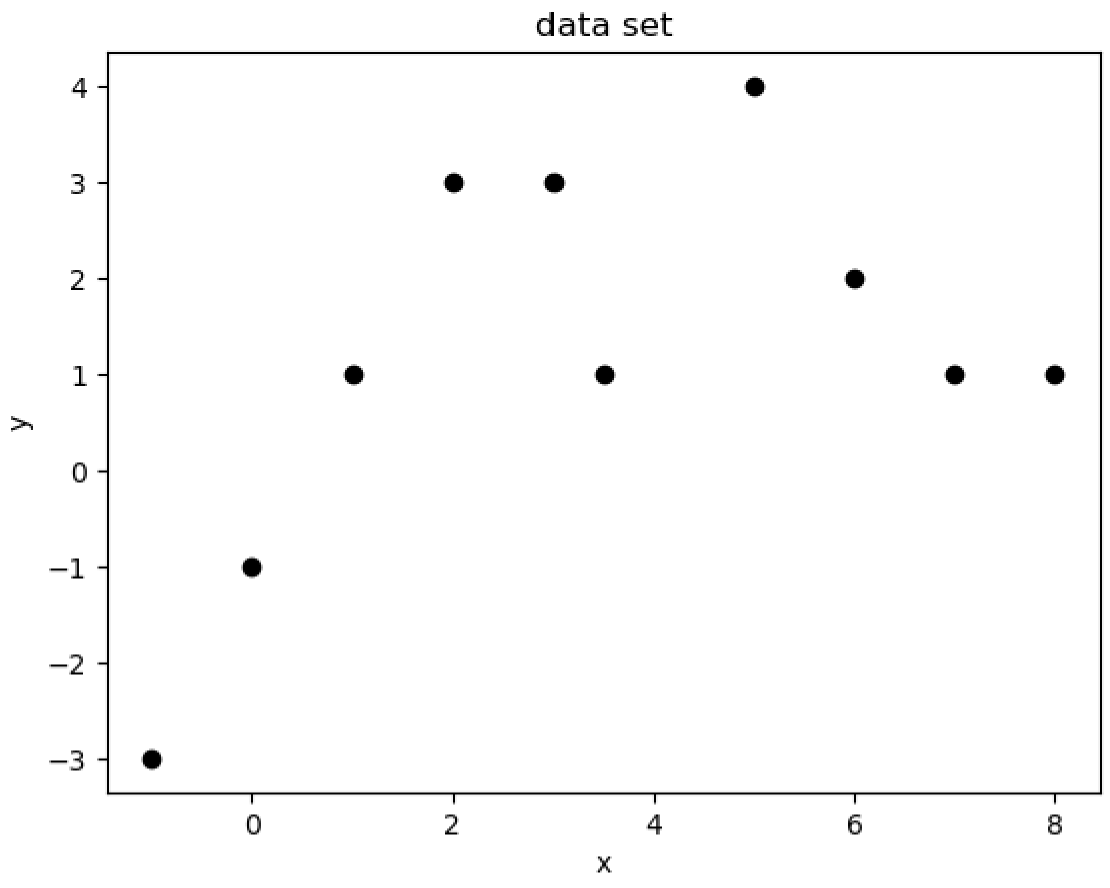
In [3]:
```
#take a lot at the first 5 data points
df.head()
```

Out[3]:

	x	y
0	-1.0	-3
1	0.0	-1
2	1.0	1
3	2.0	3
4	3.0	3

In [4]:
```
#we convert the arrays into lists
x_lst,y_lst = df['x'].to_list(),df['y'].to_list()
```

In [5]:
```
#we visually inspect the given data
plt.scatter(x_lst,y_lst,color='black')
plt.title('data set')
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```



2a

we now construct a function that linearly interpolates a set a data point and returns the y value of the given x value given by the user:

In [28]:
```python
#the x values correspond to the y values of the same index
#ensure that the x and y lists are the same length
def interp(x_vals,y_vals,x_value):
    '''
    linearly interpolate a set of data points and
    returns the y value of the desired x value

    input: x_vals (list), y_vals (list), x_value (single float)

    output: a single value that corresponds
    to the y value of the given x value
    '''


    #Sorting algorithm
    #we use pandas's built in sorting function to organize our data
    coords_lst = [x_vals,y_vals]
    df = pd.DataFrame(coords_lst).transpose()
    df.columns = ['x', 'y']
    df_sorted = df.sort_values(by=['x'])

    x_vals,y_vals = df_sorted['x'].to_list(),df_sorted['y'].to_list(


    #interpolation algorithm
    #we first deal with given x values that are outside the range of

    if x_value > max(x_vals) or x_value < min(x_vals):
        raise ValueError("X value is outside the range of the datase

    #we can now work on the interpolation:
    #we apply the fxns from video 1 above
    for i in range(len(x_vals)-1):
        if x_vals[i] <= x_value and x_vals[i+1] >= x_value:
            index = i
            #y-intercept
            b = y_vals[i]
            #slope
            a = (y_vals[i+1] - y_vals[i])/(x_vals[i+1] - x_vals[i])
            break
    #create a lambda fxn for the linear fxn we just interpolated
    y = lambda x: a * (x - x_vals[index]) + b
    return y(x_value)

```

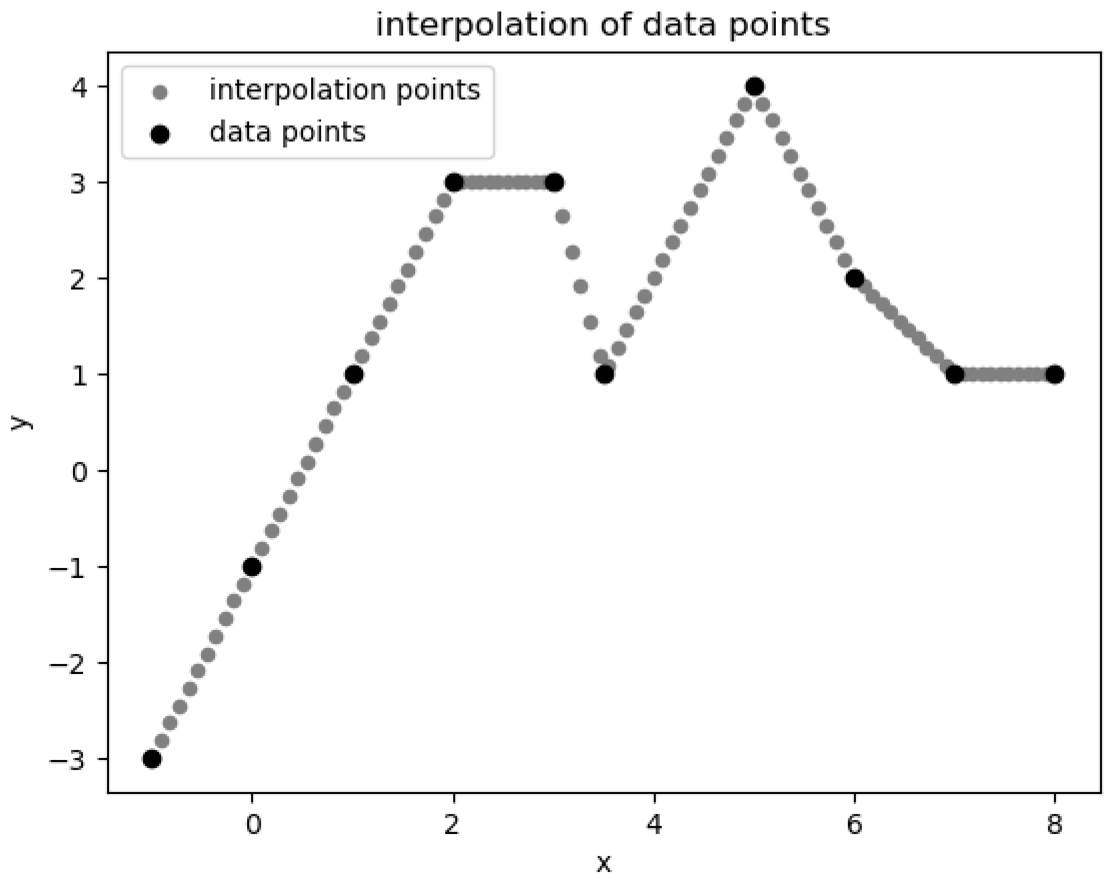
In [7]:
```
#quick test:
interp(x_lst,y_lst,6.5)
```
Out[7]:
```
1.5
```

We now want to plot the original data points and plot the interpolation fo the data to a higher resolution:

In [8]:
```
#we create a list of 100 points that fall within the range of our da
N_pts = 100
interp_pts_x = np.linspace(df['x'].min(),df['x'].max(),N_pts)
```

In [9]:
```
#we find the corresponding y values for each x value in our list usi
interp_pts_y = []
for element in interp_pts_x:
    interp_pts_y += [interp(x_lst,y_lst,element)]
```

In [10]:
```
#we plot our results along with the dataset given
plt.scatter(interp_pts_x,interp_pts_y,color = 'gray',s=20,label = 'i
plt.scatter(x_lst,y_lst,color='black',label='data points')
plt.legend()
plt.ylabel('y')
plt.xlabel('x')
plt.title('interpolation of data points')
plt.show()
```



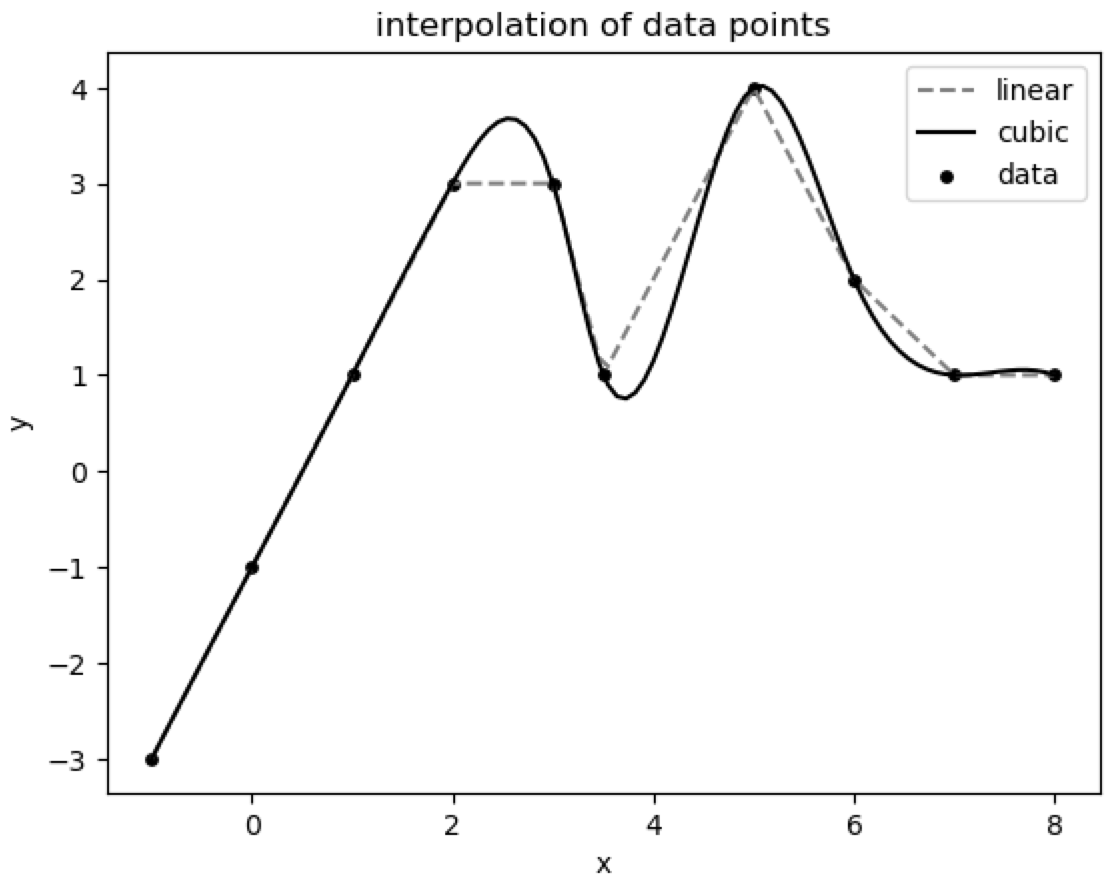
2c

now we import a cubic spline routine from scipy:

In [11]:
```
from scipy.interpolate import CubicSpline
```

In [12]:
```
cs = CubicSpline(x_lst,y_lst)
```

In [13]:
```
#we plot our interplations and the data
plt.plot(interp_pts_x,interp_pts_y,label='linear',color='gray',lines
plt.plot(interp_pts_x,cs(interp_pts_x),label='cubic',color='black',l
plt.scatter(x_lst,y_lst,label='data',color='black',s = 15)
plt.legend()
plt.ylabel('y')
plt.xlabel('x')
plt.title('interpolation of data points')
plt.show()
```



2d

I think that the fact that the cubic spline predicts higher or lower values than the max and min of the dataset is ok in certain situations only.

For example, it would be appropriate to assume continuous and smooth growth for the light curves of stars. So in this case, a cubic spline interpolation of the data would fit best.

On the other hand, it would be wrong to assume the same for data on the population of a city over time. In this case, it can not be assumed that the rate of change of the population changed at all between successive measurements. Thus, a linear interpolation of the data would be best suited here. (Here I was struggling to think of an astro dataset that would be incorrect to use cubic spline).

2e

The main advantage of linear interpolation is that it is easy to build a linear interpolator from scratch, and it is also fairly easy to understand. On the other hand, cubic spline is complex to derive and to code up. Moreover, cubic spline requires additional numerical methods (matrix solver) to work. Linear interpolation does not require any additional numerical methods.

Regarding the quality of the interpolation itself, each are appropriate in different situations. Linear interpolation creates a function that is continuous but not differentiable. Meanwhile, cubic

3

Consider the following equation:

$$y = \sin(\frac{\pi}{2}x) + \frac{x}{2}$$

We first code up the equation and then sample it:

In [14]:
```python
def sin_fxn(x):
    return np.sin((np.pi/2)*x) + (x/2)
```

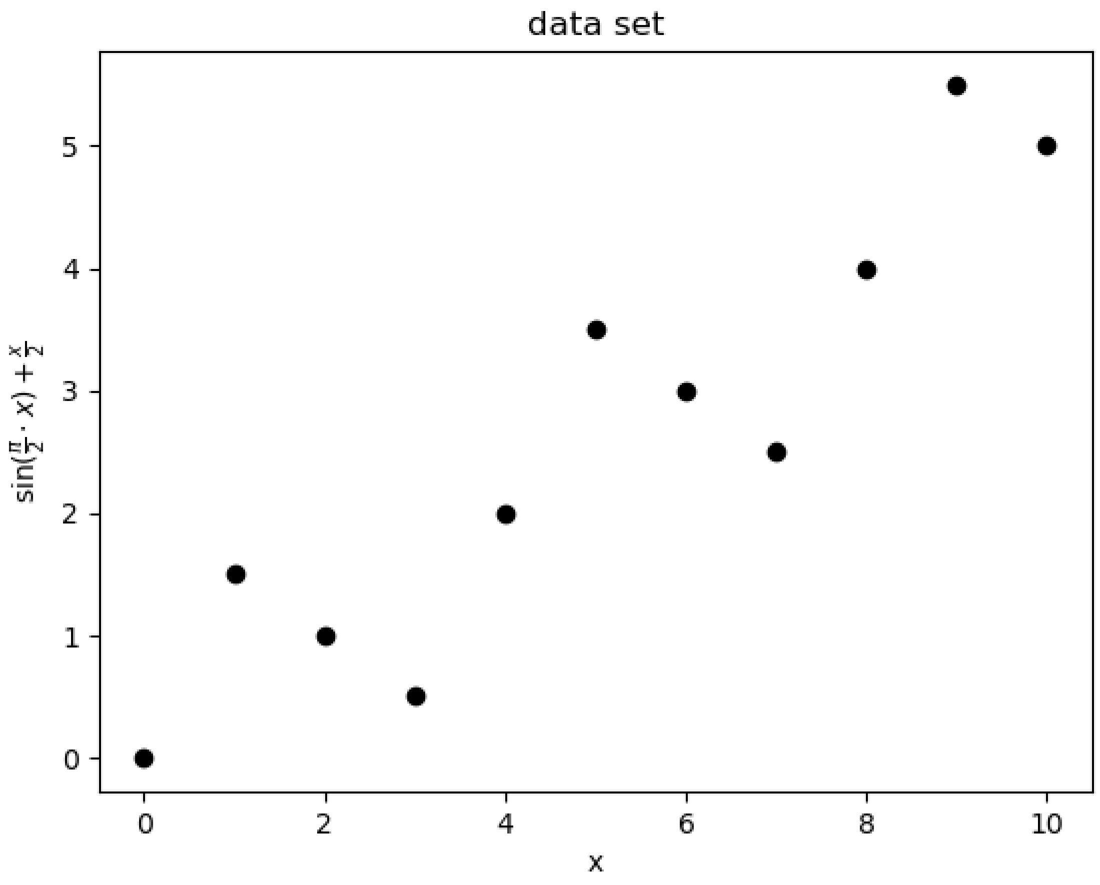
now we want to sample the sin function from 0 to 10:

In [15]:
```python
x_lst_3,y_lst_3 = [],[]

for i in range(11):
    x_lst_3 += [i]
    y_lst_3 += [sin_fxn(i)]
```

We visually inspect our sampling of the data:

In [16]:
```python
plt.scatter(x_lst_3,y_lst_3,color='black')
plt.title('data set')
plt.xlabel('x')
plt.ylabel(r'$\sin(\frac{\pi}{2}\cdot x) + \frac{x}{2}$')
plt.show()
```



3a

Plot the dataset and the linear and cubic spline interpolation of the data:

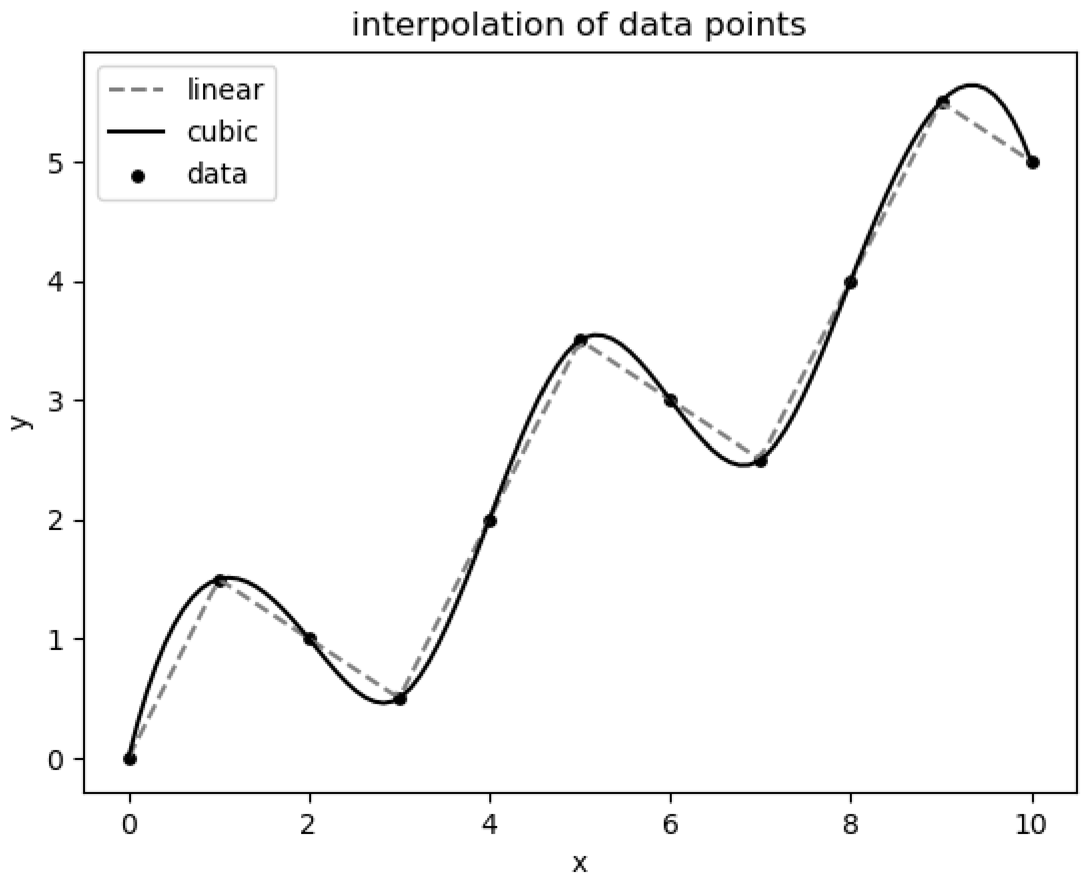
In [17]:
```python
#we will use 100 points:
N_pts_3 = 100
interp_pts_x_3 = np.linspace(min(x_lst_3),max(x_lst_3),N_pts)
```

In [18]:
```python
#we find the corresponding y values for the x values we made

#linear interpolation
interp_pts_y_3 = []
for element in interp_pts_x_3:
    interp_pts_y_3 += [interp(x_lst_3,y_lst_3,element)]

#cubic spline interpolation
cs_3 = CubicSpline(x_lst_3,y_lst_3)
```

In [20]:
```
#we now plot all our data and interpolations
plt.plot(interp_pts_x_3,interp_pts_y_3,label='linear',color='gray',l
plt.plot(interp_pts_x_3,cs_3(interp_pts_x_3),label='cubic',color='bl
plt.scatter(x_lst_3,y_lst_3,label='data',color='black',s = 15)
plt.legend()
plt.ylabel('y')
plt.xlabel('x')
plt.title('interpolation of data points')
plt.show()
```



3b

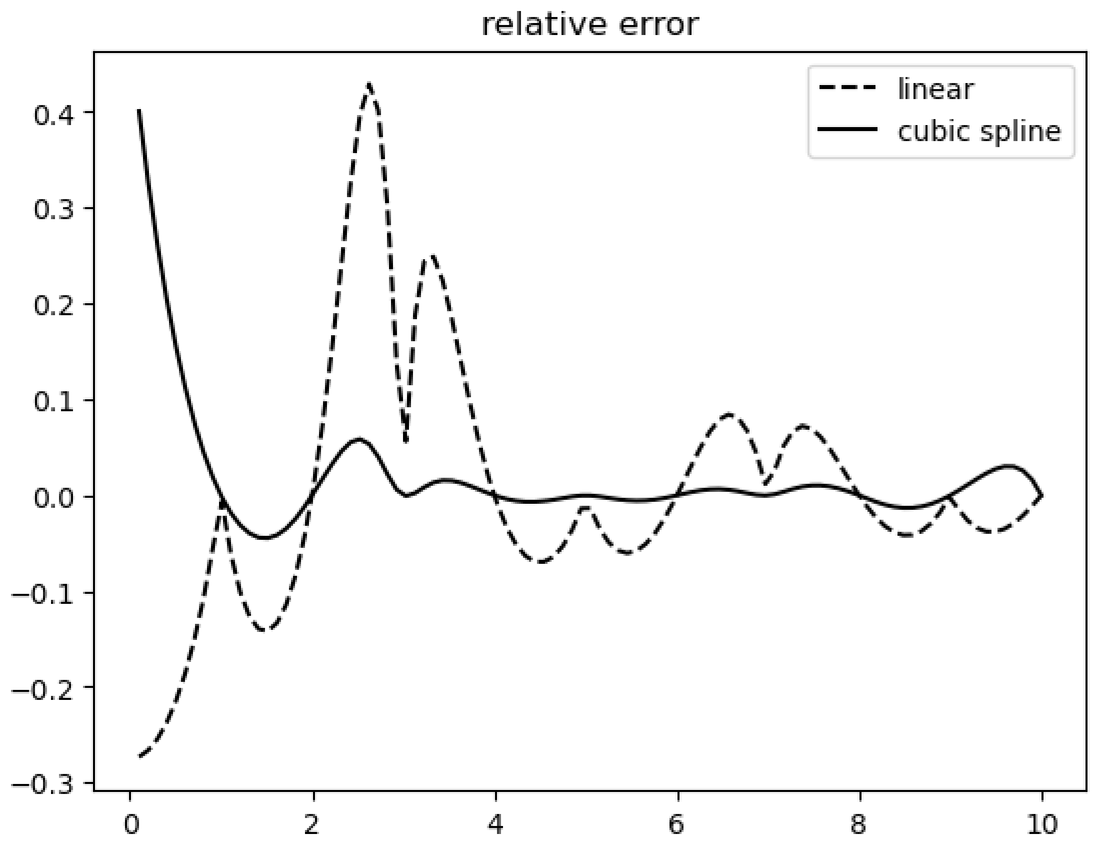
plot the relative error and comment on the result:

In [21]:
```
#we first find the difference between the interpolations and the true
true_values = sin_fxn(interp_pts_x_3)
linear_diff = interp_pts_y_3 - true_values
cubic_spline_diff = cs_3(interp_pts_x_3) - true_values
```

In [22]:
```
#we not find the relative error
linear_rel_err = linear_diff / true_values
cubic_rel_err = cubic_spline_diff / true_values
```

```
/var/folders/16/30hs3l693m58vxsvd5r6_5jh0000gn/T/ipykernel_3214/5048922
69.py:1: RuntimeWarning: invalid value encountered in divide
  linear_rel_err = linear_diff / true_values
/var/folders/16/30hs3l693m58vxsvd5r6_5jh0000gn/T/ipykernel_3214/5048922
69.py:2: RuntimeWarning: invalid value encountered in divide
  cubic_rel_err = cubic_spline_diff / true_values
```

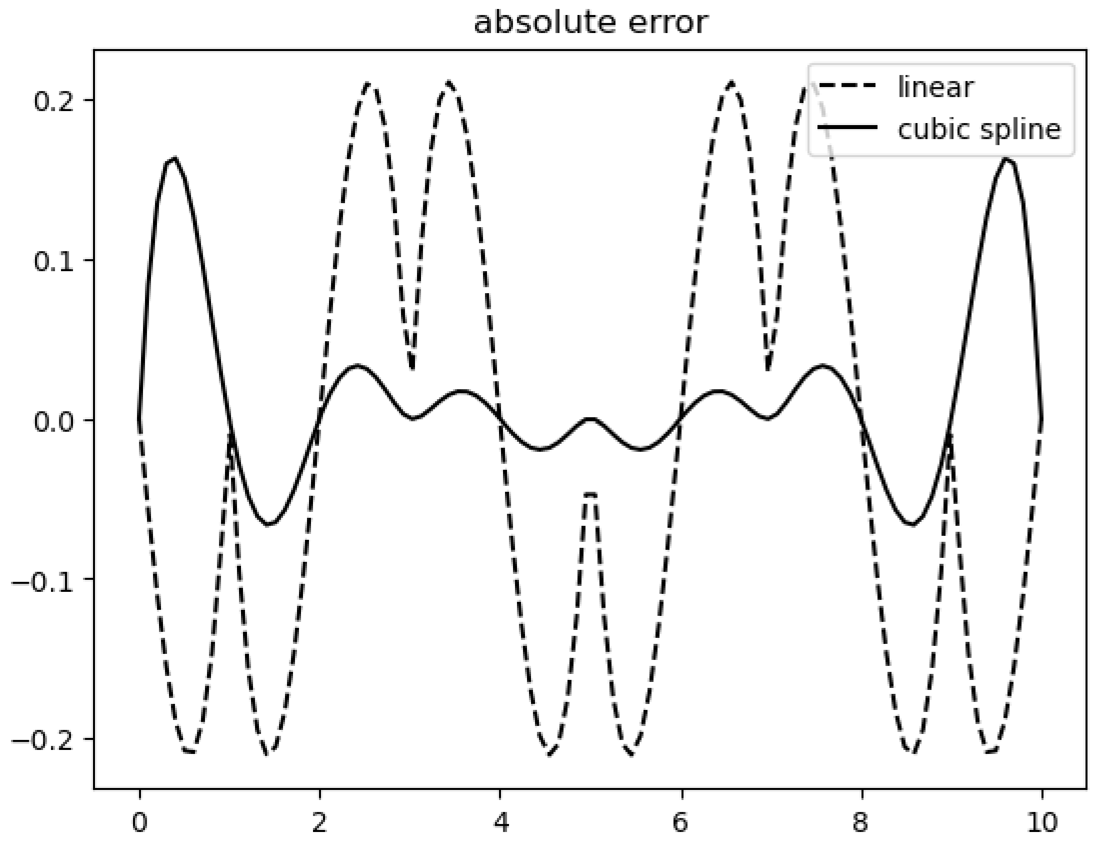
In [23]:
```
#we plot the relative errors:
plt.plot(interp_pts_x_3,linear_rel_err,label='linear',color='black',
plt.plot(interp_pts_x_3,cubic_rel_err,label='cubic spline',color='bl
plt.legend()
plt.title('relative error')
plt.show()
```



It is clear that the relative error is not great at small numbers for either interpolator. However, cubic spline quickly improves and has a better relative error than the linear interpolator at larger numbers. This makes sense given that we are fitting a sine function.

Below we also plot the absolute error (difference in interpolator and true value). Again, the cubic spline interpolator is a better fit overall.

In [24]:
```python
plt.plot(interp_pts_x_3,linear_diff,label='linear',color='black',lin
plt.plot(interp_pts_x_3,cubic_spline_diff,label='cubic spline',color
plt.legend()
plt.title('absolute error')
plt.show()
```

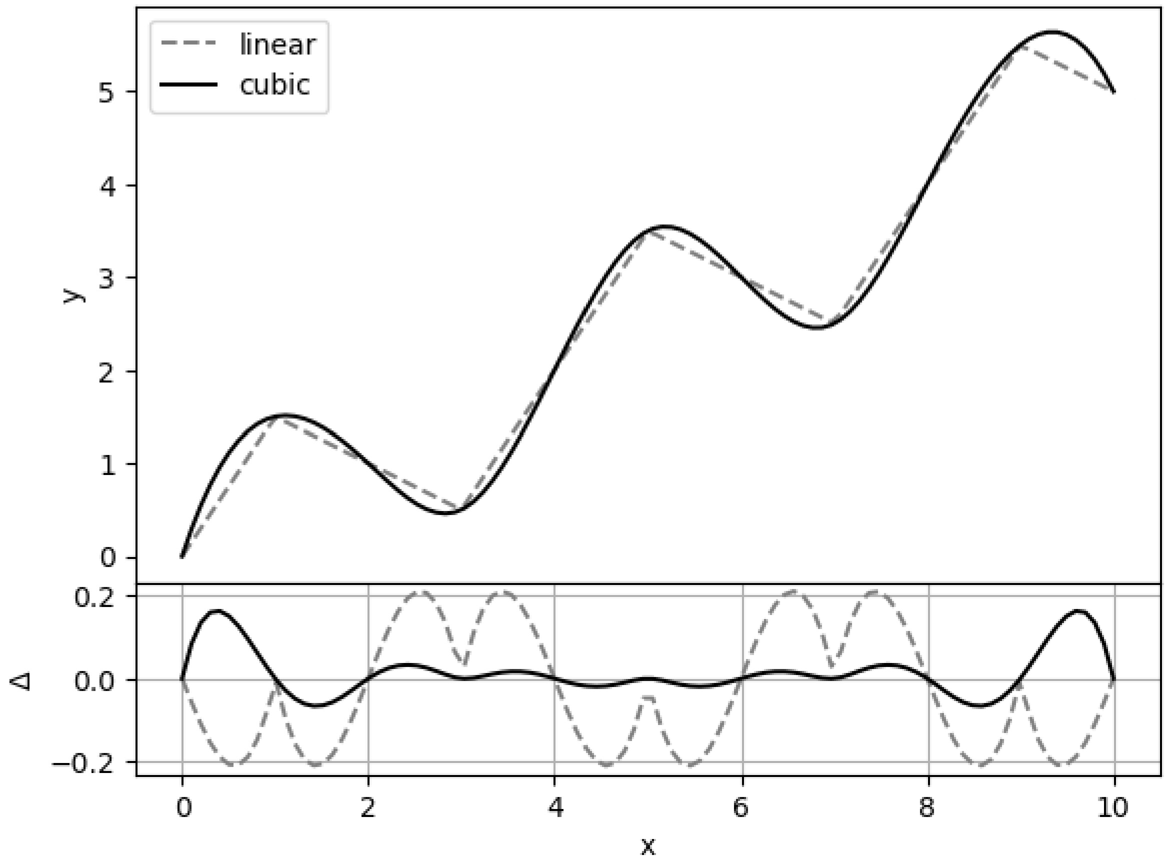


To bring everything all together, I make the following plot of the interpolations along with the residual plot below:

In [36]:
```
#PLOT
fig1 = plt.figure(1)
frame1=fig1.add_axes((.1,.3,.8,.6)) #keep
#plt.plot(interp_pts_x_3,true_values,label='true',color='blue',lines
plt.plot(interp_pts_x_3,interp_pts_y_3,label='linear',color='gray',l
plt.plot(interp_pts_x_3,cs_3(interp_pts_x_3),label='cubic',color='bl
plt.ylabel('y')
plt.legend()
#frame1.set_xticklabels([]) #Remove x-tic labels for the first frame

#Residual plot
frame2=fig1.add_axes((.1,.1,.8,.2))
plt.plot(interp_pts_x_3,linear_diff,label='linear difference',color=
plt.plot(interp_pts_x_3,cubic_spline_diff,label='cubic spline differ
plt.xlabel('x')
plt.ylabel(r'$\Delta$')
plt.grid()

plt.show()
```



4

With precise measurements of a simple harmonic oscillator at discrete points in time, we would expect the linear interpolation to not uphold the conservation of energy. There are multiple reasons for this. First, the fit itself will overfit (or underfit) the oscillations at certain points, especially if the measurements are not taken at consistent time intervals. This will in turn cause an over estimation (or underestimation) of the potential energy. The kinetic energy, which requires the velocity (i.e - the derivative of the position measurements), will be discontinuous at points where there are kinks in the linear interpolation. So the kinetic energy will not be calculated at these points.

The cubic spline on the other hand has a continuous derivative so a kinetic energy will be able to be computed at all points. The potential energy is also able to be computed at all points. Now, cubic spline will be able to be conserve energy if the simple harmonic oscillator motion is finely sampled with lots of points. However, if the sampling is sparce it will likely not conserve the energy.

In [ ]:

```

```