4 - Fitting more General Functions

In the previous notebook we saw how to fit both quadratic and cubic polynomials from data. We had set of x, y observations which we can express in *vector* form

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_N \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_N \end{bmatrix}$$

and our goal was to find some relationship between the two. In the quadratic case we proposed that the following model

$$\hat{y}(x) = a + bx + cx^2$$

that is, for each x_i in our vector, the corresponding y_i can be estimated by the above equation. This means, if we know the values of a, b and c we could construct a vector of *predictions* $\hat{\mathbf{y}}$

$$\hat{\mathbf{y}} \quad = \quad a\begin{bmatrix} 1 \\ 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix} + b\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_N \end{bmatrix} + c\begin{bmatrix} x_1^2 \\ x_2^2 \\ x_3^2 \\ \vdots \\ x_N^2 \end{bmatrix} \quad = \quad \begin{bmatrix} a \\ a \\ a \\ \vdots \\ a \end{bmatrix} + \begin{bmatrix} bx_1 \\ bx_2 \\ bx_3 \\ \vdots \\ bx_N \end{bmatrix} + \begin{bmatrix} cx_1^2 \\ cx_2^2 \\ cx_3^2 \\ \vdots \\ cx_N^2 \end{bmatrix} \quad = \quad \begin{bmatrix} a + bx_1 + cx_1^2 \\ a + bx_2 + cx_2^2 \\ a + bx_3 + cx_3^2 \\ \vdots \\ a + bx_N + cx_N^2 \end{bmatrix}$$

We also showed that this was equivalent to the following *matrix-vector product*

$$\hat{\mathbf{y}} = X\mathbf{w}$$

where

$$\mathbf{w} = \begin{bmatrix} a \\ b \\ c \end{bmatrix}$$

and

$$X = \begin{bmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ 1 & x_3 & x_3^2 \\ \vdots & \vdots & \vdots \\ 1 & x_N & x_N^2 \end{bmatrix}$$

Similarly, the same equation $\hat{\mathbf{y}} = X\mathbf{w}$ holds for the cubic case, except here

$$X = \begin{bmatrix} 1 & x_1 & x_1^2 & x_1^3 \\ 1 & x_2 & x_2^2 & x_2^3 \\ 1 & x_3 & x_3^2 & x_3^3 \\ \vdots & \vdots & \vdots & \vdots \\ 1 & x_N & x_N^2 & x_N^3 \end{bmatrix}$$

Once we have that matrix, we can solve to find the best parameters, w.

Exercise 1

Below is a pre-written function that takes in a set of x-observations, a set of y-observations, and a number defining the *order* of the polynomial you want to fit (1=linear, 2=quadratic, 3=cubic etc) The function then finds the best-fit polynomial of that order and plots it. Read through this function and try to understand each line. Then try calling it using the data provided. What order polynomial do you think is appropriate for this data?

```
import numpy as np
import matplotlib.pyplot as plt
from IPython.display import Audio
from scripts.regression4 import make_graph_1, make_graph_2, make_graph_3
import warnings
```

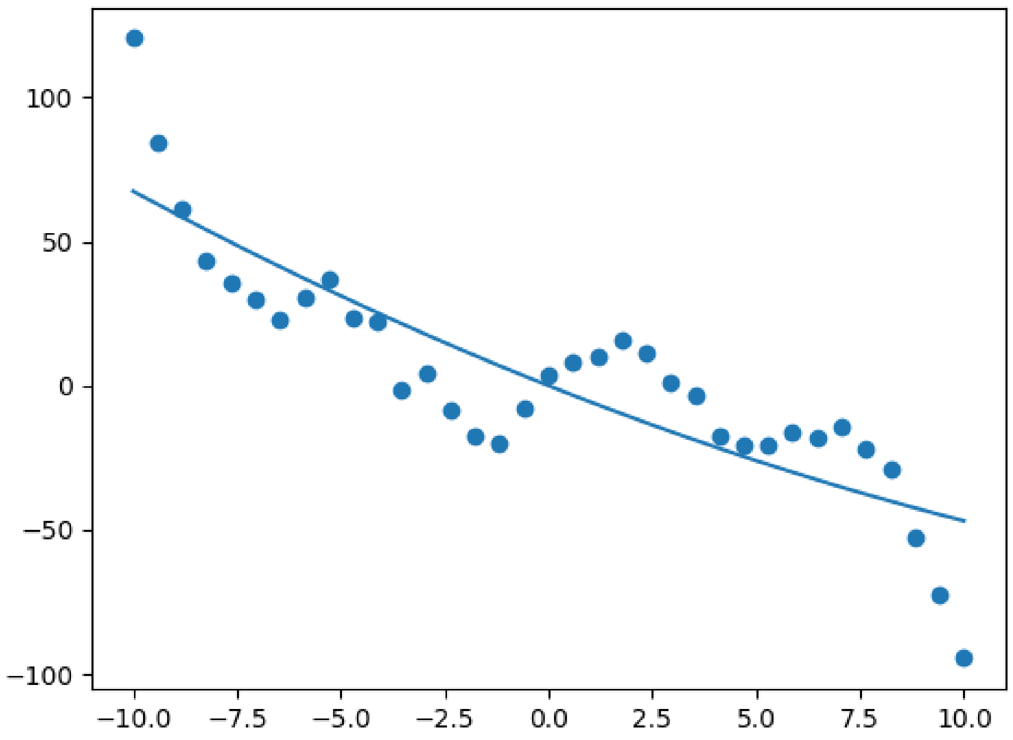
```
np.set_printoptions(precision=3, linewidth=500, threshold=500, suppress=True)
warnings.simplefilter(action='ignore', category=FutureWarning)
```

```
%matplotlib notebook
```

```
%matplotlib notebook
```

```
def fit_polynomial(x_data, y_data, order):
    """
    This function takes in a set of x-observations, and a set of y-observations
and fits a
    polynomial which is defined by the parameter, "order".

    order = 1 --> linear      (y = a + bx)
    order = 2 --> quadratic   (y = a + bx + cx^2)
    order = 3 --> cubic       (y = a + bx + cx^2 + dx^3)
    ...


    Make sure that x_data and y_data are numpy arrays of the same length, and
that
    order is an integer! (a whole number)
    """

    # this is a helper function that creates the X matrix given x_data
    # Note: there's nothing wrong with defining functions *within* other
functions
    def make_X(x):#
        X_shape = (len(x), order + 1)
        X = np.zeros(X_shape)
        for i in range(order + 1):
            X[:, i] = x ** i
        return X

    # create the X matrix
    X = make_X(x_data)

    # fit the parameters
    w_fit = np.linalg.lstsq(X, y_data)[0]
    # scatter the data
    plt.figure()
    plt.scatter(x_data, y_data)

    # create a 250-long 'x_space' variable for plotting the function
    x_space = np.linspace(x_data.min(), x_data.max(), 250)

    # make a y_fit array.
    # Note: '@' means *matrix multiply*
    y_fit = make_X(x_space) @ w_fit

    # plot the fitted function
    plt.plot(x_space, y_fit)
```

```
x_data = np.array([-10, -9.41, -8.82, -8.24, -7.65, -7.06, -6.47, -5.88, -5.29,
-4.71, -4.12, -3.53, -2.94, -2.35, -1.76, -1.18, -0.59, 0, 0.59, 1.18, 1.76,
2.35, 2.94, 3.53, 4.12, 4.71, 5.29, 5.88, 6.47, 7.06, 7.65, 8.24, 8.82, 9.41,
10.])
y_data = np.array([120.5, 84.17, 61, 43.61, 35.8, 29.65, 22.79, 30.28, 36.7,
23.3, 22.27, -1.7, 4.53, -8.21, -17.71, -19.81, -8.12, 3.54, 8.29, 9.92, 15.91,
11.49, 1.45, -3.3, -17.51, -20.91, -20.51, -15.95, -18.12, -14.5, -21.79,
-29.11, -52.83, -72.58, -94.42])

order = 2     # try varying this parameter!

fit_polynomial(x_data, y_data, order)
```



Polynomial regression: summary

Hopefully you have noticed the pattern with these examples: fitting a polynomial of a higher order can be done by just extending the matrix X. If we have observations of the form

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_N \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_N \end{bmatrix}$$

then we can find the best fit by making a matrix

$$\text{linear:} \quad X = \begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ 1 & x_3 \\ \vdots & \vdots \\ 1 & x_N \end{bmatrix}$$

$$\text{quadratic:} \quad X = \begin{bmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ 1 & x_3 & x_3^2 \\ \vdots & \vdots & \vdots \\ 1 & x_N & x_N^2 \end{bmatrix}$$

$$\text{cubic:} \quad X = \begin{bmatrix} 1 & x_1 & x_1^2 & x_1^3 \\ 1 & x_2 & x_2^2 & x_2^3 \\ 1 & x_3 & x_3^2 & x_3^3 \\ \vdots & \vdots & \vdots & \vdots \\ 1 & x_N & x_N^2 & x_N^3 \end{bmatrix}$$

$$\text{quartic:} \quad X = \begin{bmatrix} 1 & x_1 & x_1^2 & x_1^3 & x_1^4 \\ 1 & x_2 & x_2^2 & x_2^3 & x_2^4 \\ 1 & x_3 & x_3^2 & x_3^3 & x_3^4 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_N & x_N^2 & x_N^3 & x_N^4 \end{bmatrix}$$

$$\vdots$$

and calling

```
w = np.linalg.lstsq(X, y)[0]
```

to find the best coefficients. Doing regression of a higher polynomial is as simple as adding a new column.

A more general perspective

Consider doing polynomial estimation. What we're really doing is finding

$$\begin{aligned}&\text{something} \times 1\\&\text{something} \times x\\&\text{something} \times x^2\\&\text{something} \times x^3\end{aligned}$$

Those "somethings" define our fit. This would be exactly the same as saying

$$\begin{aligned}&\text{something} \times \phi_1(x)\\&\text{something} \times \phi_2(x)\\&\text{something} \times \phi_3(x)\\&\text{something} \times \phi_4(x)\end{aligned}$$

where we are careful to define that

$$\begin{aligned}\phi_1(x) &= 1\\\phi_2(x) &= x\\\phi_3(x) &= x^2\\\phi_4(x) &= x^3\end{aligned}$$

However, we could use *any* functions , not just the simple ones. In general, we call these functions *basis functions*.

Radial basis functions

Radial basis functions, or RBFs are a common choice of basis function. An RBF is a function of the form

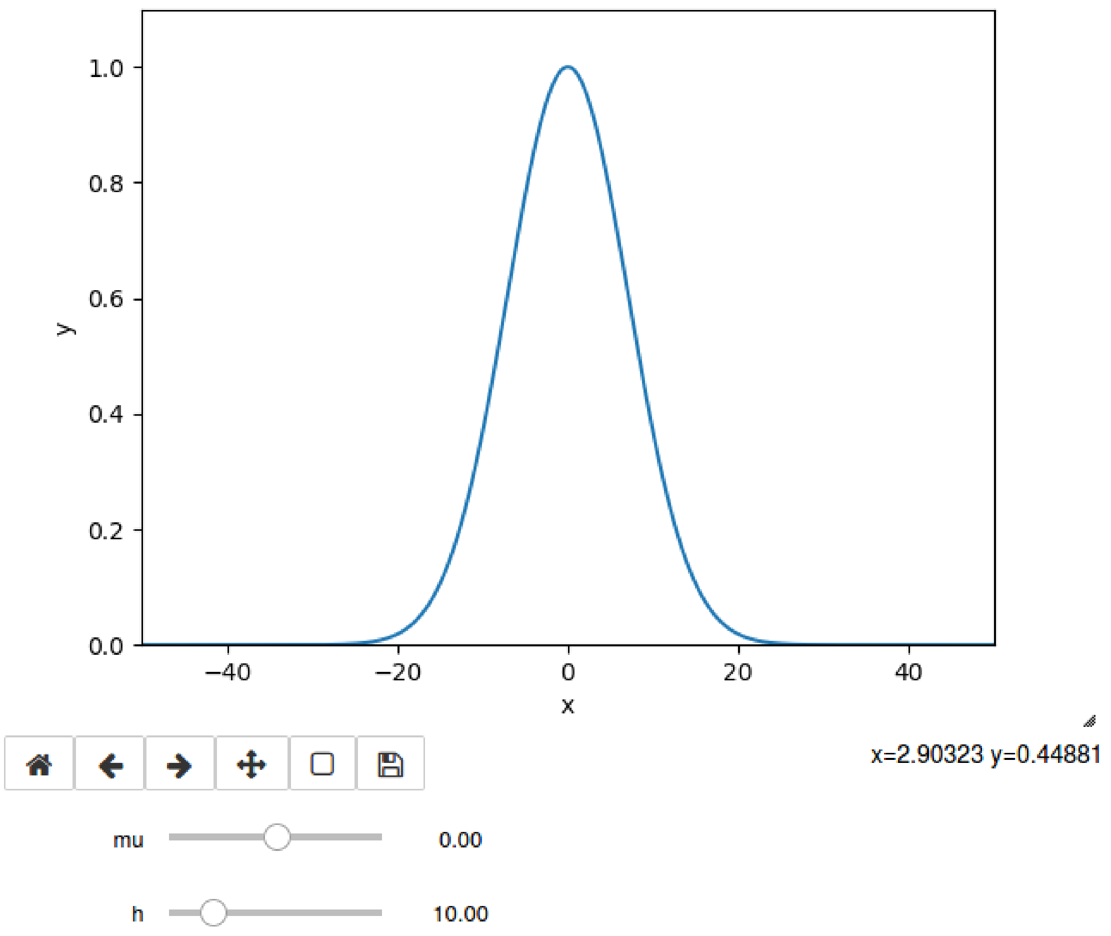
$$\phi(x) = \exp\Big(-\frac{(x-\mu)^2}{h^2}\Big)$$

A radial basis function is defined by two parameters μ and h, and looks very similar to a normal distribution.

Exercise 2

Interact with the plot below to understand how the parameters μ and h change the shape of the radial basis function.

```
make_graph_1()
```



We then typically choose a set of RBFs spaced out over our interval of interest, and then find the parameters that best fit our data.

Lets say

$$\mu = [\mu_1, \mu_2, \dots \mu_k]$$

and we define our RBF as

$$\phi_i(x) = \exp\left(-\frac{(x-\mu_i)^2}{h^2}\right)$$

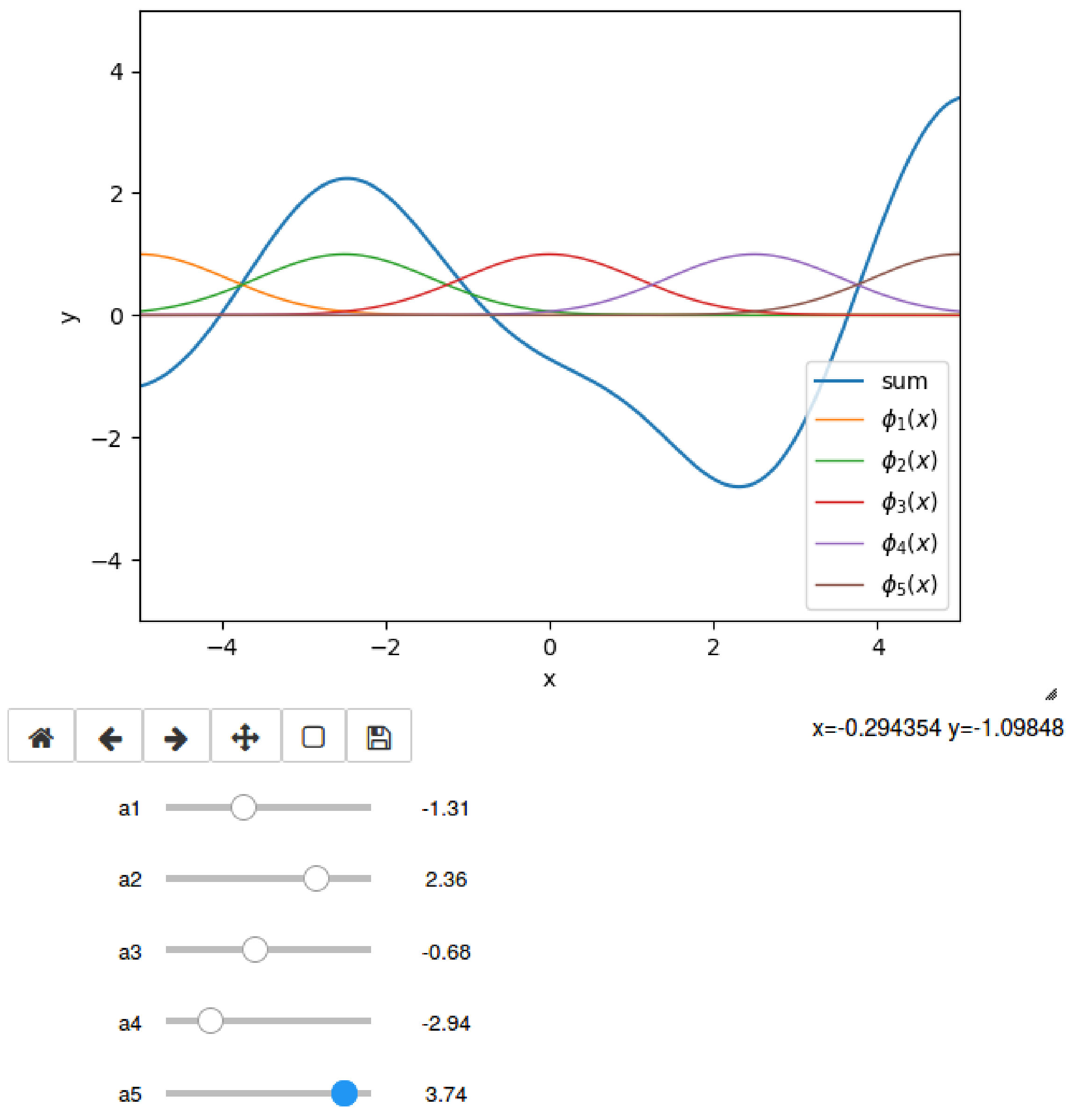
We want to find the parameters $a_1, a_2 \dots a_k$ which best fit our data, which we can approximate as

$$\hat{y} = a_1\phi_1(x) + a_2\phi_2(x) + \dots + a_k\phi_k(x)$$

Exercise 3

Try interacting with the graph below to see the effect of varying the constants multiplying the radial basis functions.

```
make_graph_2()
```



x=-0.294354 y=-1.09848

a1 -1.31

a2 2.36

a3 -0.68

a4 -2.94

a5 3.74

Fitting radial basis functions to data

It turns out we can fit these parameters in exactly the same way as before

Using this notation, our data matrix X becomes instead

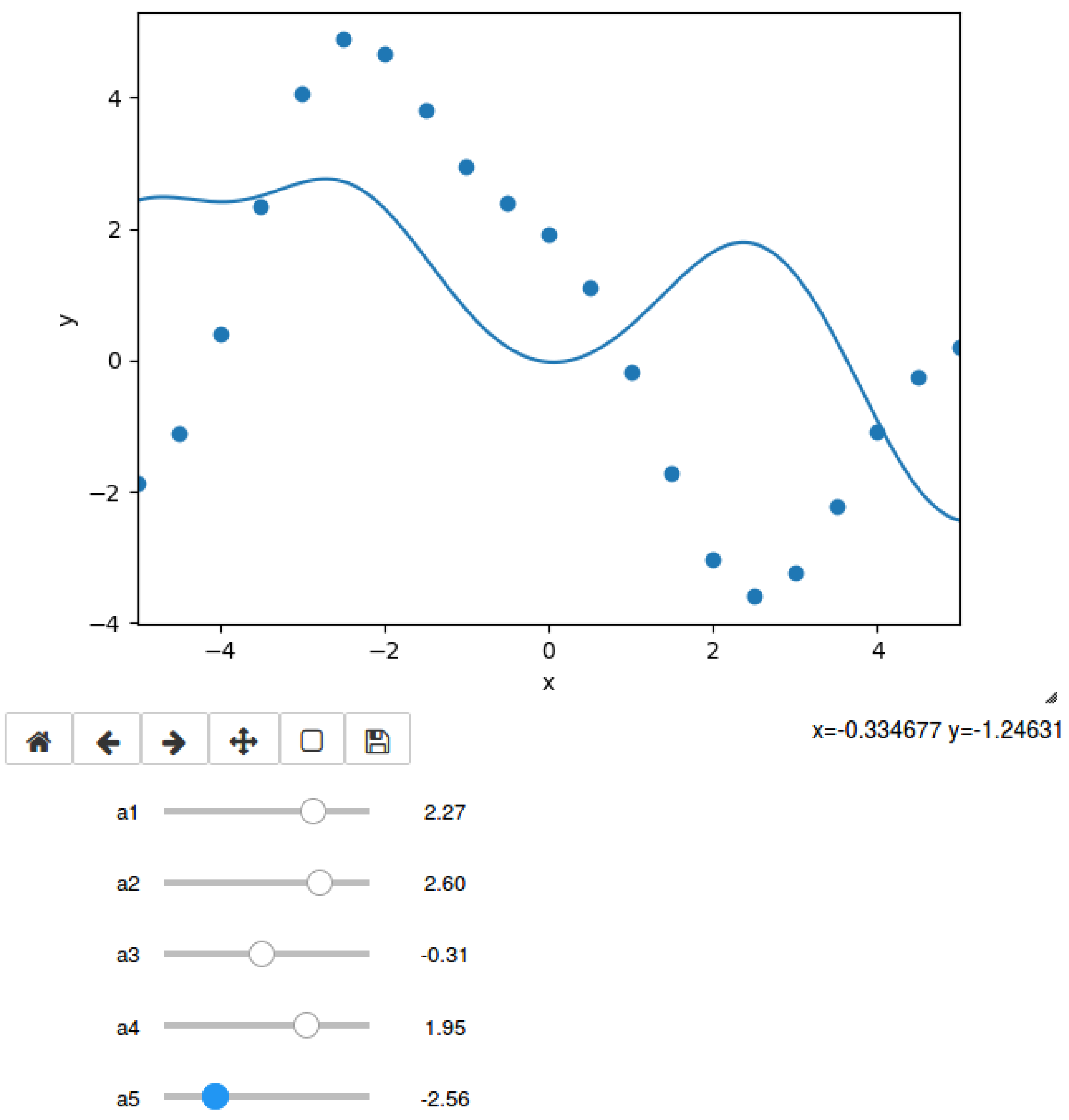
$$X = \begin{bmatrix} | & | & & | \\ \phi_1(\mathbf{x}) & \phi_2(\mathbf{x}) & \cdots & \phi_k(\mathbf{x}) \\ | & | & & | \end{bmatrix} = \begin{bmatrix} \phi_1(x_1) & \phi_2(x_1) & \cdots & \phi_k(x_1) \\ \phi_1(x_2) & \phi_2(x_2) & \cdots & \phi_k(x_2) \\ \phi_1(x_3) & \phi_2(x_3) & \cdots & \phi_k(x_3) \\ \vdots & \vdots & & \vdots \\ \phi_1(x_N) & \phi_2(x_N) & \cdots & \phi_k(x_N) \end{bmatrix}$$

We will show how this can be accomplished with code shortly

Exercise 5

Interact with the graph below to see how we can vary the parameters multiplying the radial basis functions to fit some data.

```
make_graph_3()
```



x=-0.334677 y=-1.24631

a1 2.27

a2 2.60

a3 -0.31

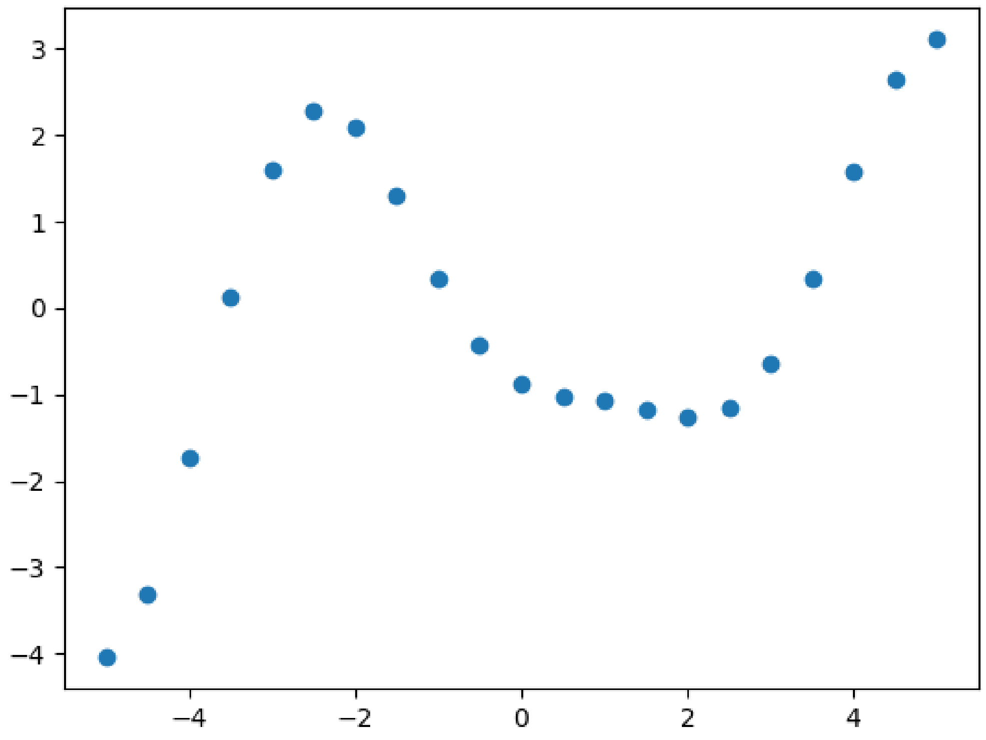
a4 1.95

a5 -2.56

How can we fit these coefficients automatically given some data? Take a look at the code below. First we need to generate some data

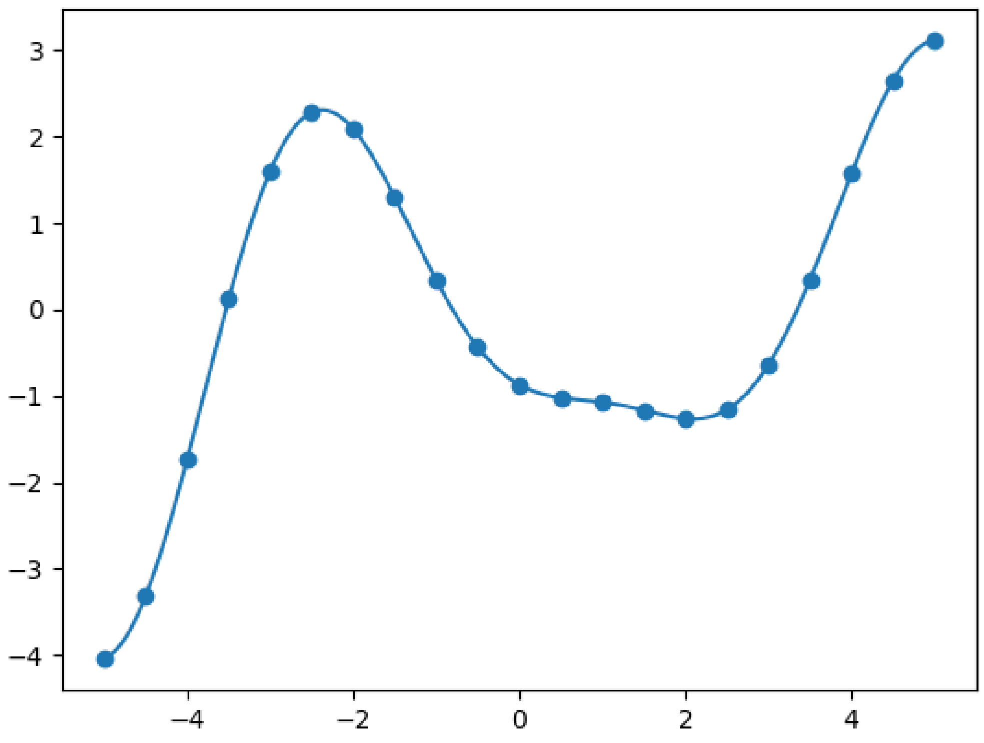
```
def rbf(x, mu, h):
    """
    For a given value of mu and h, return the associated radial basis
    function applied on an array, x
    """
    return np.exp(-((x - mu) ** 2) / h ** 2)

# set some constants
N_rbfs = 5
N_data_points = 21

# set a fixed value of h
h = 1.5

# these will be our different values for mu - they are 5 evenly spaced points
between -5 and 5
mus = np.linspace(-5, 5, N_rbfs)

# generate some x-data: 21 points evenly spaced between -5 and 5
```

```
x_data = np.linspace(-5, 5, N_data_points)

# these are the true factors multiplying each rbf - this is what we want to
recover
true_coefficients = [-4.2,  2.6 , -0.95, -1.3,  3.2]

# initialise y_data as the same shape as x_data
y_data = np.zeros(N_data_points)

# create y-observations by adding contributions from each rbf
for i in range(N_rbfs):
    y_data = y_data + true_coefficients[i] * rbf(x_data, mus[i], h)

# plot to see what we have made
plt.figure()
plt.scatter(x_data, y_data)
```



Now let's try and solve for the coefficients that we set

```
def make_X(x_points):
    """
    A helper function: given an array of x-observations, create
    the X matrix for these basis functions
    """

    # initialise X
    X = np.zeros((len(x_points), N_rbfs))

    # fill in the columns using the rbf function
    for i in range(N_rbfs):
        X[:, i] = rbf(x_points, mus[i], h)

    return X

# make the X matrix
X = make_X(x_data)

# solve for the best coefficients
w = np.linalg.lstsq(X, y_data)[0]

# make some arrays for plotting the continuous line
x_space = np.linspace(-5, 5, 1001)
y_fit = make_X(x_space) @ w

plt.figure()
plt.scatter(x_data, y_data)
plt.plot(x_space, y_fit)
```

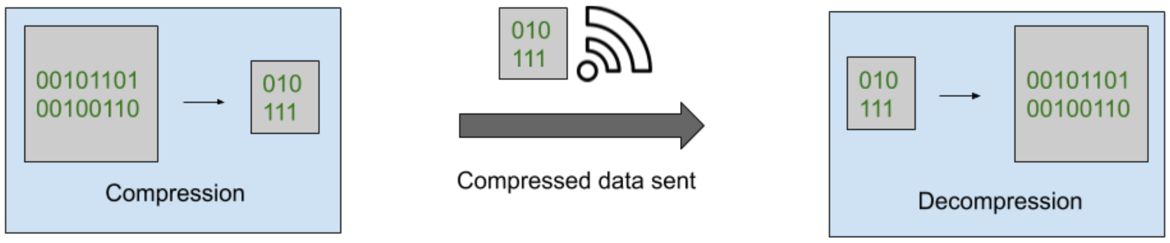


Example: Music Compression

For the rest of this notebook, we are going to use the concept of radial basis function regression to solve a real, useful task: music compression.

What is compression?

All digital files are ultimately made up of zeros and ones. Consider the task of sending a file over the internet made up of 1000 zeros followed by 1000 ones. We could simply send this file as we have found it. However, it would take up less memory for us to send: "0 x 1000, 1 x 1000" and trusting that the person we send this too understands how to reconstruct the original file from this information. This is the essence of compression. We've saved ourselves space by eliminating useless information. Compression is absolutely key for sending files over the internet because they allow us to save memory, increasing speed of transmission and saving money.



Lossless vs lossy compression

The previous situation was an example of lossless compression. Here, we know *exactly* how to reconstruct the file given the instructions - there is no loss of data. However, we can also design techniques that allow us to *approximately* reconstruct the file, with some loss of data, but using even less memory. This is known as lossy compression. Common filetypes you will have seen that are based on lossy compression include mp3, mp4 and JPEG.

Audio data

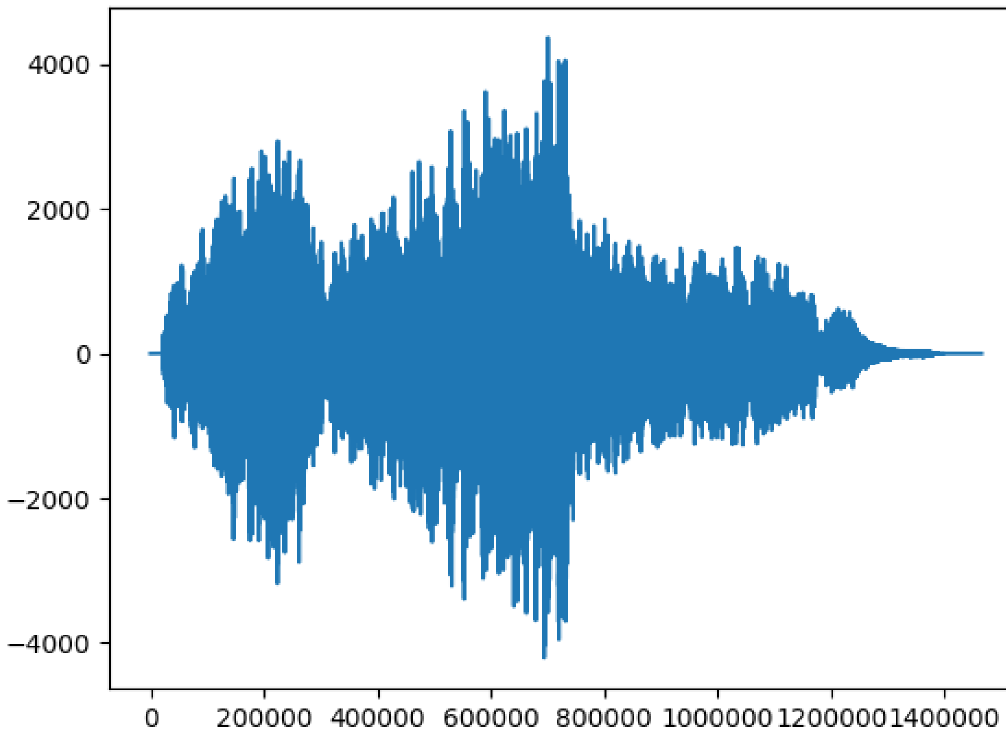
What is audio? Audio fundamentally made up of waves: vibrations in pressure detected by our ears which is understood by our brain as sound. There is a type of file called WAV which holds all the information about this vibration in pressure - you can think of it as a pure time series, giving the exact pressure reading at many finely spaced time points In the data folder you will find a file called `music_uncompressed.wav`. Run the cell below and play the sound.

```
Audio('data/music_uncompressed.wav')
```

0:00:00 / 13:31:36

We can open this file directly and plot the waveform using scipy and matplotlib.

```
from scipy.io.wavfile import read, write

bitrate, waveform = read('data/music_uncompressed.wav')

plt.figure()
plt.plot(waveform)
```

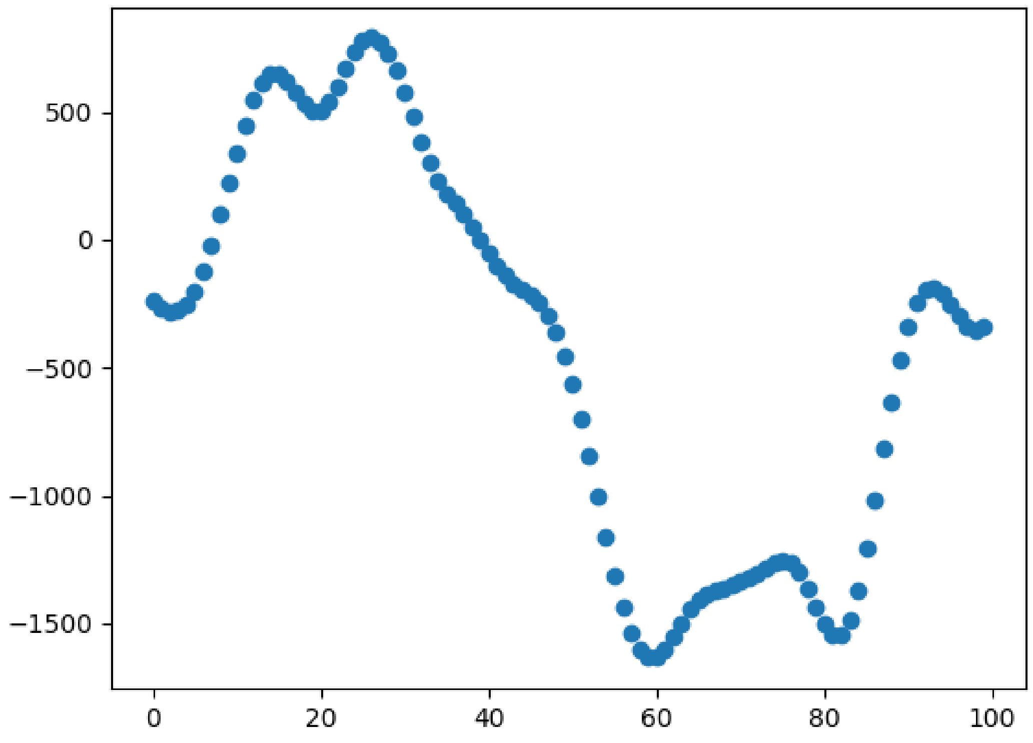


We can find out the initial size of this file using the following function

```
def get_size(array):
    return '{:.2f}MB'.format(array.nbytes / 1e6)

print('Initial size of the file is', get_size(waveform))
```

Let's look at a specific section of the waveform in detail

```
plt.figure()
section = waveform[200000:200100]
plt.scatter(range(100), section)
```

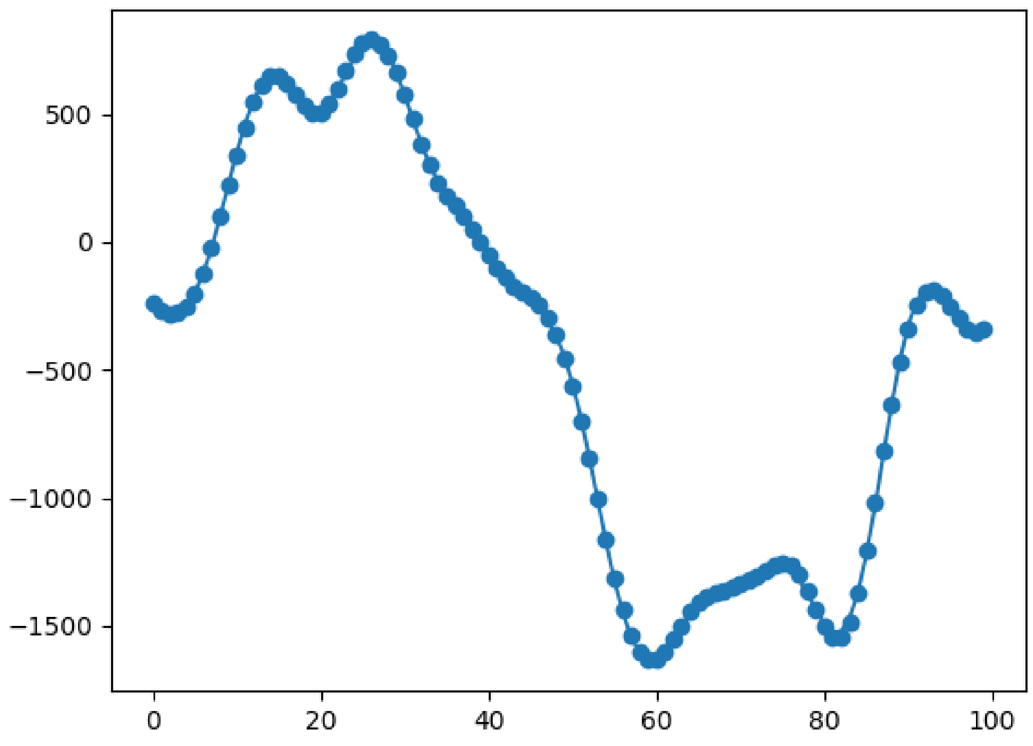


Compression using RBF regression

Our goal will be to go through the whole of this waveform in sections like this, and fit the data using RBFs. That way, instead of saving every single point into memory, we can save an array of coefficients. We will hopefully need far fewer coefficients than original data points (taking up less memory), and given these coefficients we should be able to approximately reconstruct the entire waveform. This means we will have written a compression algorithm!

```python
# lets take sections of the waveform 100 at a time
section_length = 100

# this is the number of rbfs we will use
n_rbfs = 20

# this will be our x-data: just evenly spaced points between 0 and 99
x = np.linspace(0, section_length-1, section_length)

# these will be the centres of the rbfs: evenly spaced points over our interval
centres = np.linspace(0, section_length, n_rbfs)

# we now create the X matrix by filling the columns with the RBFs
X = np.zeros((section_length, n_rbfs))
for i in range(n_rbfs):
    # fill each column with the i-th rbf, applied on x
    X[:, i] = rbf(x, centres[i], 9)

# solve for the optimal coefficients
w = np.linalg.lstsq(X, section)[0]

# calculate a fit
fit = X @ w
```

```
# plot the original data and our fit
plt.figure()
plt.scatter(x, section)
plt.plot(x, fit)
```



In the above section, we have fit the signal pretty accurately. To do this we only require 20 numbers, as opposed to the original 100! This means we have achieved a 5-fold compression: not bad! Now lets do the same for every single section

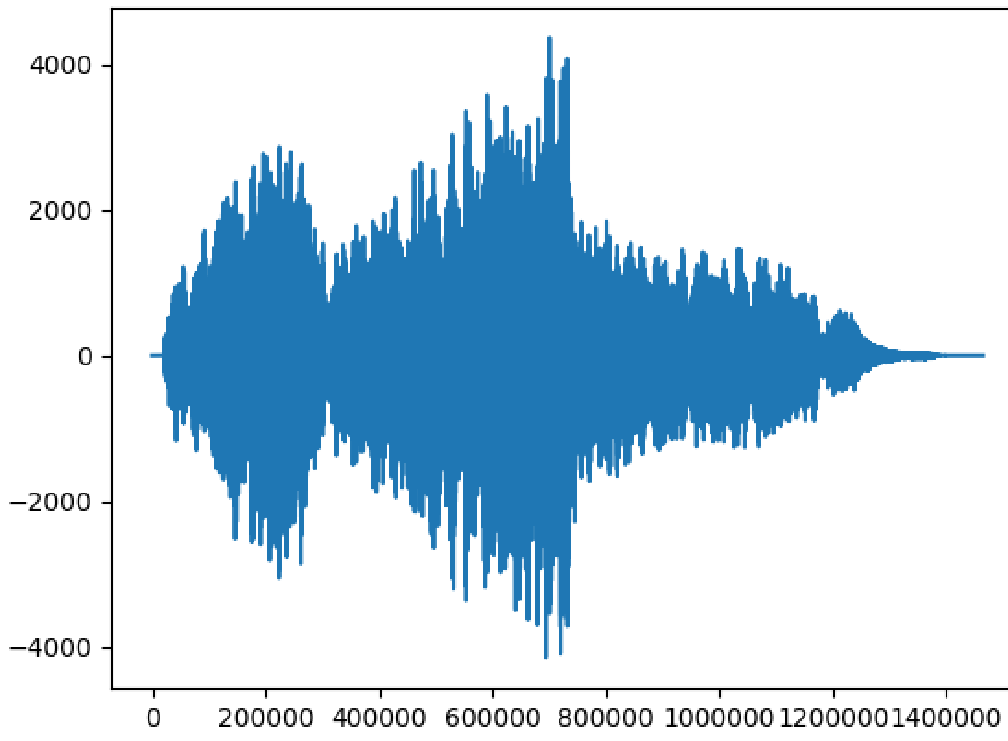
```
signal_length = len(waveform)
n_sections = signal_length // section_length

# create an array to hold the coefficients
W = np.zeros((n_sections, n_rbfs))

for i in range(n_sections):

    # take a slice of the waveform
    section = waveform[i*section_length:(i+1)*section_length]

    # fit the coefficients of this section
    w = np.linalg.lstsq(X, section)[0]

    # add these to our holding array
    W[i, :] = w

# convert internal datatype to 16-bit (don't worry too much)
W = W.astype(np.float16)

print('Compressed file size:', get_size(W))
```

We have now successfully completed the compression stage! Let's try and decompress this out now and save the new wav file.

```
# this array will hold our decompressed signal
decompressed_signal = np.zeros(signal_length)

for i in range(n_sections):

    # fill in the relevant section of the waveform
    decompressed_signal[i*section_length:(i+1)*section_length] = X @ W[i, :]

# make the underlying data type 16-bit
decompressed_signal = decompressed_signal.astype(np.int16)
```

Let's take a look at our decompressed signal

```
plt.figure()
plt.plot(decompressed_signal)
```



ooks roughly ok... How does it sound?

```
# save the file into the data folder
write('data/music_compressed.wav', bitrate, decompressed_signal)

# play the file
Audio('data/music_compressed.wav')
```

0:00:00 / 13:31:36

Final exercise

Below is a function which performs all these steps in one (with a few added things). You don't need to understand every line, but try varying the input parameters to see what qualitative effects this has on the sound created

```
def compress(section_length=100, compression_ratio=5):

    signal_length = len(waveform)
    n_sections = signal_length // section_length
    n_rbfs = int(section_length // compression_ratio)

    x = np.linspace(0, section_length-1, section_length)
    centres = np.linspace(0, section_length, n_rbfs)
    X = np.concatenate([rbf(x[:, None], centres, 9), 0.01 * np.eye(n_rbfs)])

    W = np.zeros((n_sections, n_rbfs))

    for i in range(n_sections):

        section = waveform[i*section_length:(i+1)*section_length]
        W[i, :] = np.linalg.lstsq(X, np.concatenate([section,
np.zeros(n_rbfs)]))[0]

    W = W.astype(np.float16)

    print('Successfully compressed file from {} to
{}'.format(get_size(waveform), get_size(W)))

    decompressed_signal = np.zeros(signal_length)

    for i in range(n_sections):
        decompressed_signal[i*section_length:(i+1)*section_length] = X[:-n_rbfs,
:] @ W[i, :]

    decompressed_signal = decompressed_signal.astype(np.int16)
    write('data/music_compressed.wav', bitrate, decompressed_signal)

    return Audio('data/music_compressed.wav')
```

```
compress(section_length=500, compression_ratio=5)
```

Question

What do you think is the origin of the strange buzzing you can hear in this audio? How might we eliminate it? Hint: how does varying the section length affect this?