08 Plotting

September 13, 2020

1 Introduction

Plotting is used to explore data, and is one of the most important tasks that engineers and scientists use computers for. Plots are used to:

- present and understand data; and
- to check computed results visually.

The most widely used Python module for plotting is Matplotlib (http://matplotlib.org/), and this is what we will use. Matplotlib is extremely powerful and we will use only a small fraction of its functionality. The material in this notebook will provide you with a basis that you can build upon for more advanced plotting by searching online resources. A good place to start for a wide range of examples is the Matplotlib Gallery (http://matplotlib.org/gallery.html).

We will use the tools presented in this notebook to visualise results in subsequent notebooks.

> This is a long activity - this a reflection of the importance of plotting. The next activity is shorter in compensation.

> Some exercises in this notebook are challenging. Use the forum to get help.

1.1 Why do we plot programmatically?

Producing plots via a program is very powerful. It allows you to explore and manipulate data in sophisticated ways. It also allows you to make plotting interactive; from a large and complex data set you can design interfaces for you and others to query the data and have it displayed graphically, possibly over the Internet. It also allows you to automate the fetching and processing of data from online sources or from laboratory equipment.

1.2 Objectives

- Introduction to a programmatic approach to plotting
- Create line plots and bar charts from discrete data sets
- Create line plots of mathematical functions
- Create histograms for distribution-like data
- Develop interactive plots
- Create contour plots of functions of two variables

> This notebook uses interactive widgets - you will need to execute the notebook to see the widgets.

We will be using the `ipywidgets` module for interactive plots. It has already been configured for you on https://notebooks.azure.com/, so the below is necessary only if using other environments.

```
[1]: try:
         import ipywidgets
     except ImportError:
         try:
             !{sys.executable} -m pip -q install ipywidgets
             import ipywidgets
         except ImportError:
             !{sys.executable} -m pip -q --user install ipywidgets
         finally:
             !jupyter nbextension enable --py widgetsnbextension
             print("You will need to refresh your browser page")
```

2 Matplotlib

To use Matplotlib, we need to import it. We will be using the `pyplot` interface to Matplotlib, so we need to import `pyplot` too:

```
[2]: import matplotlib
     import matplotlib.pyplot as plt
```

We will be using NumPy when producing plots, so we import it now:

```
[3]: import numpy as np
```

Matplotlib is a regular Python library that can be used outside of Jupyter notebooks to display plots to the screen or to write plots to a file. To have plots appear inline in a Jupyter notebook, we need the 'magic' command `%matplotlib inline`:

```
[4]: %matplotlib inline
```

If you want to increase the default font size, use:

```
matplotlib.rcParams.update({'font.size': 16})
```

3 Simple line plots

Given the data set

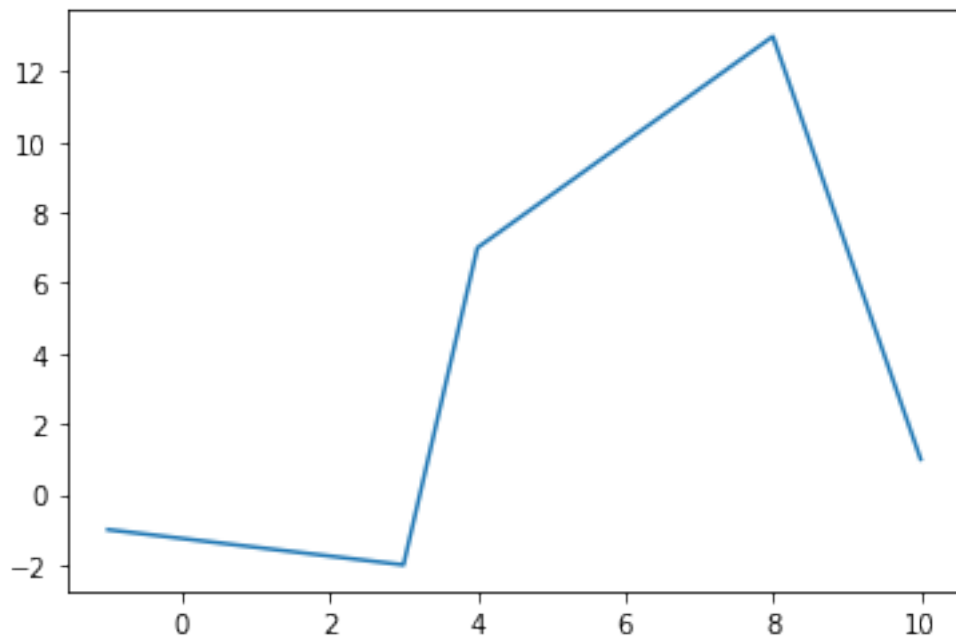| x value	f value
-1	-1
3	-2
4	7
8	13
10	1

we want to create a line plot of x-values (horizontal axis) against the f values (vertical axis).

We first create a list of x values and a list of f values:

```
[5]: x = [-1, 3, 4, 8 , 10]
     f = [-1, -2, 7, 13 , 1]
```

We now use the `plt.plot` function to produce a line plot:

```
[6]: plt.plot(x, f)
     plt.show()
```


Note that x and f are lists, but they could be any ordered array-like structure, e.g. tuple or NumPy array.

We can improve the plot by adding axis labels, a title and dots to indicate data points.

```
[7]: # Plot data with a line and dots at the data points (this is what '-o' does)
     plt.plot(x, f, '-o')

     # Attach labels and title (using LaTeX syntax)
     plt.xlabel('$x$')
     plt.ylabel('$f$')
     plt.title("Simple plot of $x$ vs $f$")
     plt.show()
```

Simple plot of *x* vs *f*
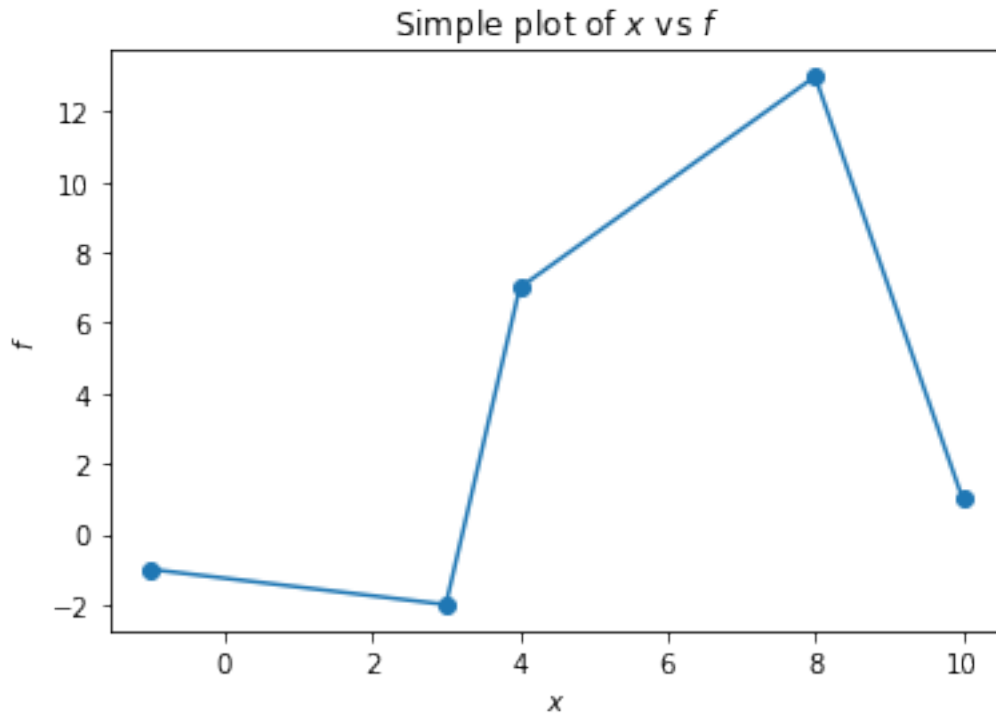
To change the colour of the line to red and use crosses at the data points, and to save the plot to the file `my-plot.png`, we can do the following:

```
[8]:  # Plot data with a red solid line and crosses at the data points (this is what␣
      ↪'-xr' does)
      plt.plot(x, f, '-xr')

      # Attach labels and title
      plt.xlabel('$x$')
      plt.ylabel('$f$')
      plt.title("Simple plot of $f$ against $x$")

      # Save plot to file
      plt.savefig("my-plot.pdf")
      plt.show()
```

Simple plot of *f* against *x*
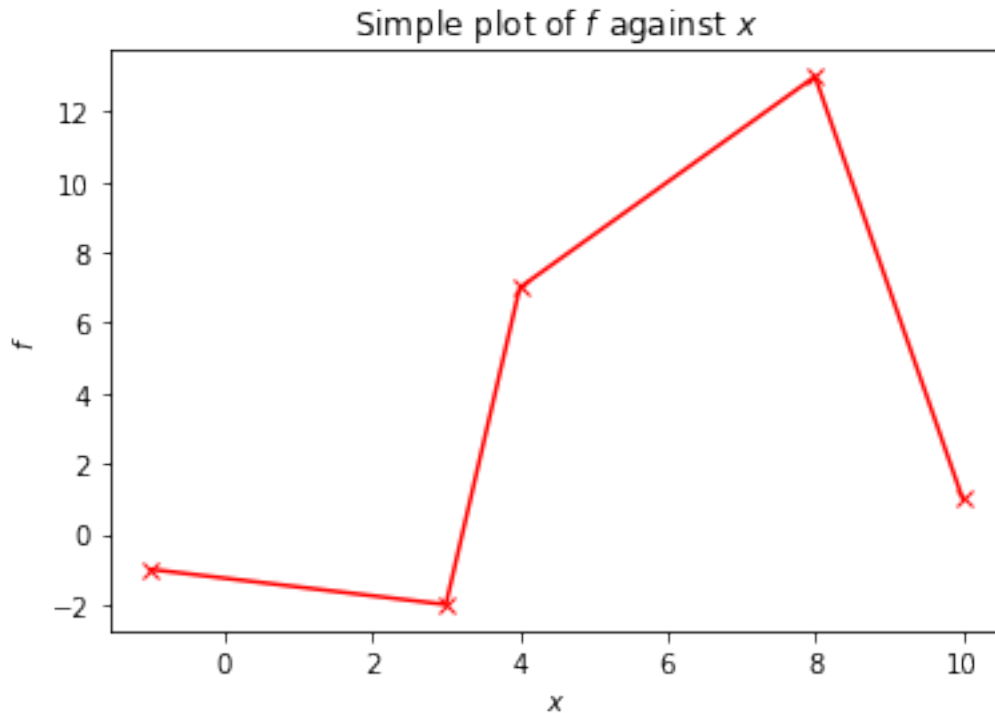
Upon executing the notebook, the file `my-plot.png` should appear in the same directory as the notebook. You could use the file to include the plot in a document or presentation.

4 Plotting mathematical functions of one variable

Line plots of a mathematical function are created by evaluating the function at a large number of points. If we use enough points, the graph of a function will appear smooth.

To plot the function:

$$f(x) = \sin(x)$$

the first step is to create an array of values at which we want to evaluate the function. We can use the NumPy function `linspace` to do this:

```
[9]: num_points = 100
     x = np.linspace(0, 4*np.pi, num=num_points)
```

The above creates a NumPy array with `num_points` equally spaced values between 0 and 4π, including the end points. We can check this for a smaller example:

```
[10]: print(np.linspace(0, 2, num=10))
```
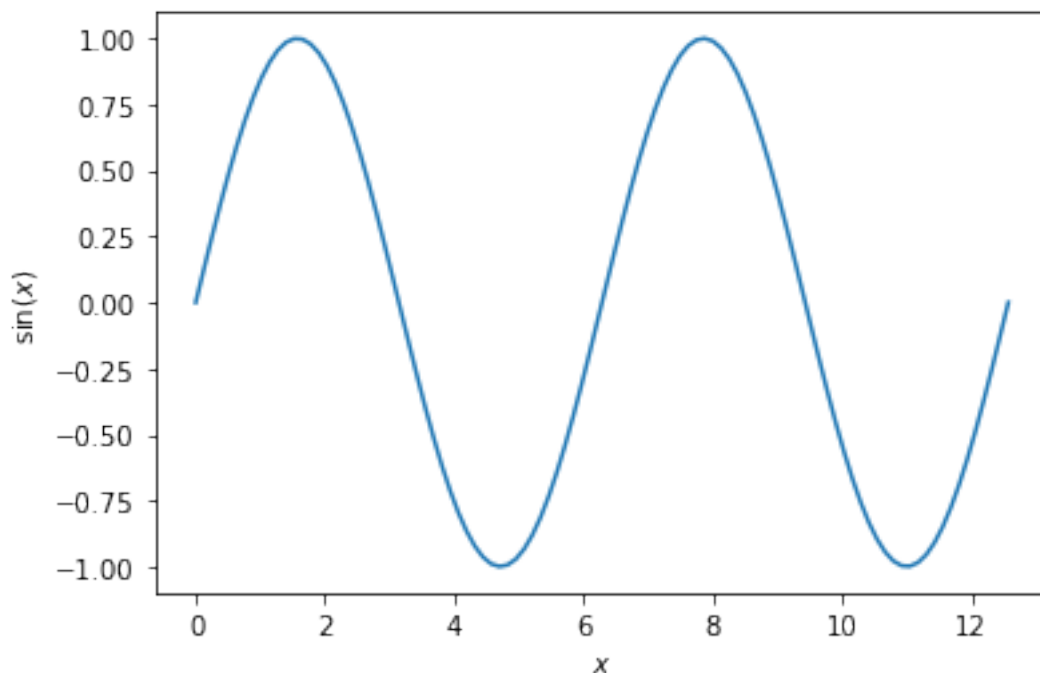
```
[0.          0.22222222 0.44444444 0.66666667 0.88888889 1.11111111
 1.33333333 1.55555556 1.77777778 2.          ]
```

To evaluate $\sin(x)$ for each value in the array `x`, we use the NumPy `sin` function, which can be applied entry-wise to `x`:

```
[11]: f = np.sin(x)
```

We can now plot `f` against `x`.

```
[12]: plt.plot(x, f)
      plt.xlabel('$x$')
      plt.ylabel('$\sin(x)$')
      plt.show()
```


We can improve the appearance of the plot by specifying the x limits:
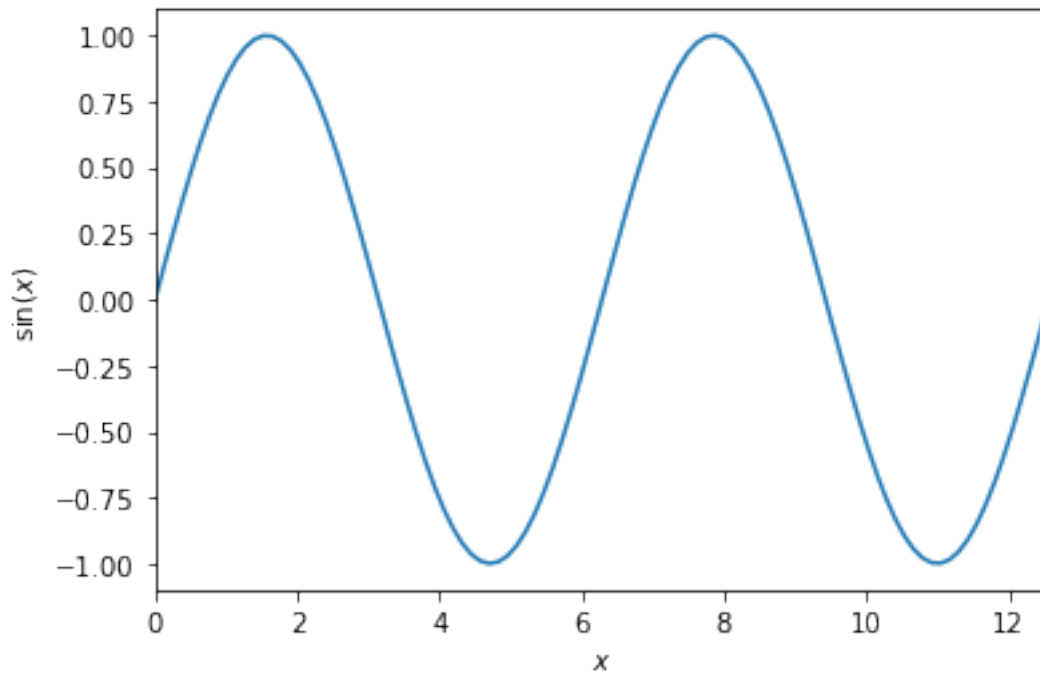
```
[13]: # Plot the graph
      plt.plot(x, f)

      # Label the axis
      plt.xlabel('$x$')
      plt.ylabel('$\sin(x)$')

      # Use the start and end values in x as x limits (recall that x[-1]
      # returns the last value in x)
```

```
plt.xlim(x[0], x[-1])

plt.show()
```


To plot two functions on one graph, we use `plt.plot` twice:

```
[14]: x = np.linspace(0, 4*np.pi, num=100)

      # Plot sin(x) and cos(x), and add label for each
      plt.plot(x, np.sin(x), label="$\sin(x)$")
      plt.plot(x, np.cos(x), label="$\cos(x)$")

      # Label the x-axis
      plt.xlabel('$x$')

      # Add legend
      plt.legend()

      # Use the start and end values in x as x limits
      plt.xlim(x[0], x[-1])

      plt.show()
```
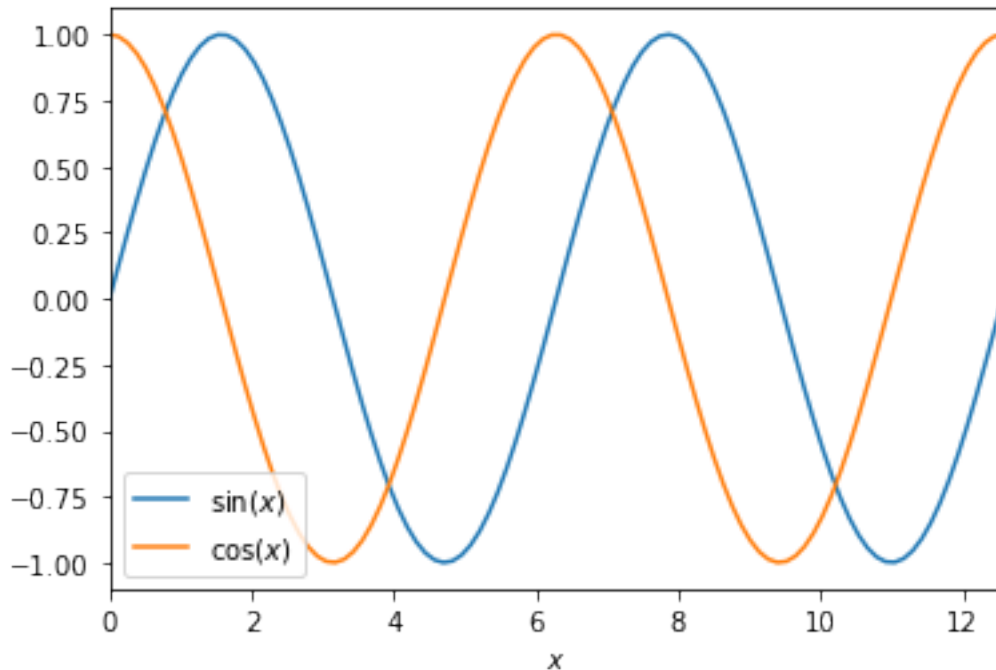
5 Bar charts and histograms

5.1 Bar charts

The numbers of students in each year of the Engineering Tripos, and graduate students are:

```
[15]: cohort = ('IA', 'IB', 'IIA', 'IIB', 'Graduate')
      num_students = (332, 270, 256, 238, 451)
```

We want to represent these figures using a bar chart:

```
[16]: # Create an array with the position of each bar along the x-axis, e.g [0, 1, 2,␣
      ↪3, 4]
      x_pos = np.arange(len(cohort))

      # Produce bar plot
      plt.bar(x_pos, num_students, align='center');

      # Replace the x ticks with the Tripos name, and rotate labels 30 degrees
      plt.xticks(x_pos, cohort, rotation=30)

      # Add axis labels
      plt.xlabel('cohort')
      plt.ylabel('number of students')
```
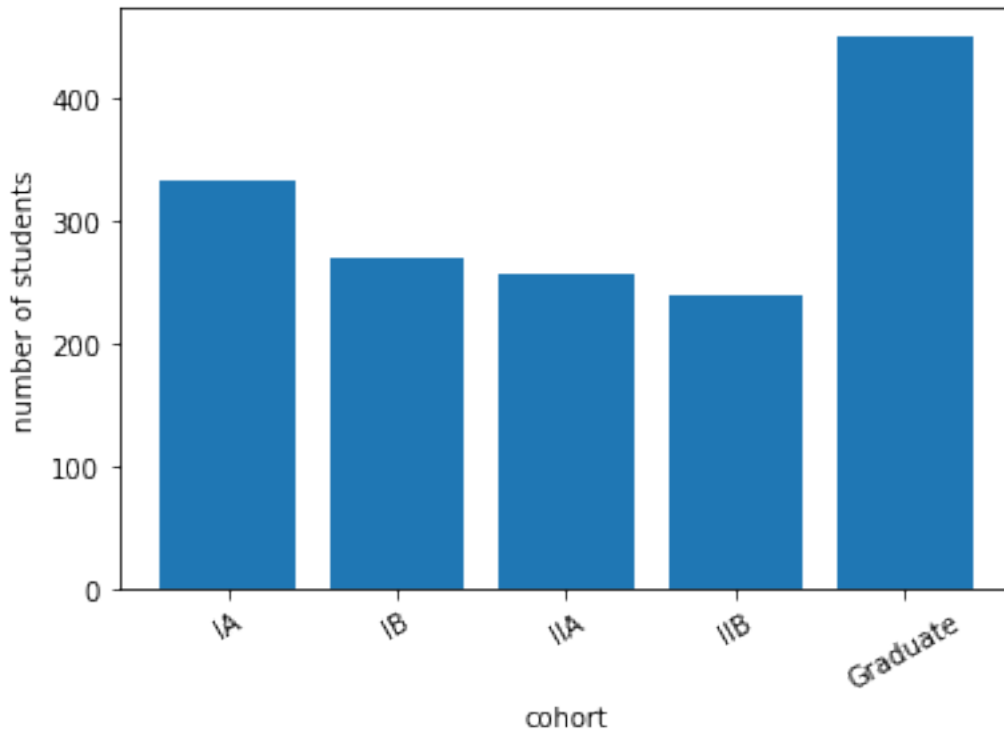
```
plt.show()
```


5.2 Histograms

To create a histogram we first need a data set. We will use 4000 random values drawn from a normal (Gaussian) distribution with zero mean and standard deviation of one. We can use NumPy to create such an array using:

```
[17]: # loc=mean, scale=stddev, size=num samples
      x = np.random.normal(loc=0.0, scale=1.0, size=5000)
```

We can visualise the distribution of the values using a histogram. A histogram works by collecting data into intervals (bins) along the horizontal axis. The number of values that fall within a 'bin' is then displayed on the vertical axis. Using 20 bins, we can visualise how x is distributed:

```
[18]: # Produce histogram with 20 bins (try changing the number of bins)
      n, bins, patches = plt.hist(x, 20, facecolor='green');

      # Add label
      plt.xlabel('value')
      plt.ylabel('frequency')

      plt.show()
```
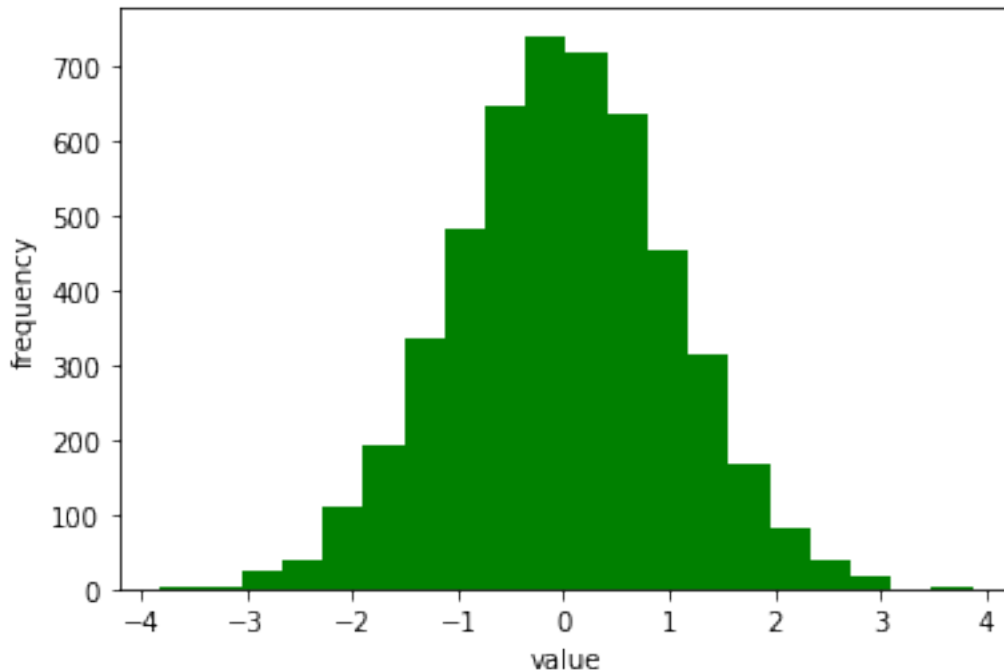
Matplotlib has automatically created 20 bins of equal width, and computed the number of entries in each bin. We can inspect the bin intervals

```
[19]: print(bins)
```

```
[-3.81535766 -3.43115268 -3.0469477  -2.66274272 -2.27853774 -1.89433277
 -1.51012779 -1.12592281 -0.74171783 -0.35751285  0.02669213  0.4108971
  0.79510208  1.17930706  1.56351204  1.94771702  2.33192199  2.71612697
  3.10033195  3.48453693  3.86874191]
```

And the number of values in each bin:

```
[20]: print(n)
```

```
[  2.   3.   26.   38. 109. 193. 334. 482. 646. 740. 717. 635. 454. 314.
  167.  81.  40.  16.   1.   2.]
```

We can mix plot types on a single graph - in the case of a histogram it can be useful to compare against a probability density function. Since we drew the values in x from a normal distribution, we expect a normal probability distribution function to be a good fit. Let's test.

We can use the module `scipy.stats` to compute the normal probability distribution function. To compare the histogram to a probability density function, the histogram needs to be 'normalised' (so the integral is equal to 1).

```
[21]: # Import norm from SciPy stats module to compute norm probability distribution␣
      ↪function
```

10

```python
from scipy.stats import norm

# Plot normed histogram of data
n, bins, patches = plt.hist(x, 20, normed=1.0, facecolor='green', alpha=0.75);

# Create array of 100 equally spaces points, stating from the first value in
 ↪the bins
# through to the last values in the bins
xf = np.linspace(bins[0], bins[-1], 100)

# Evaluate normal probability density function (pdf) at each xf
f = norm.pdf(xf)

# # Create line plot of normal probability distribution function (blue line)
plt.plot(xf, f, 'b')

# # Add labels
plt.xlabel('value')
plt.ylabel('frequency')

plt.show()
```
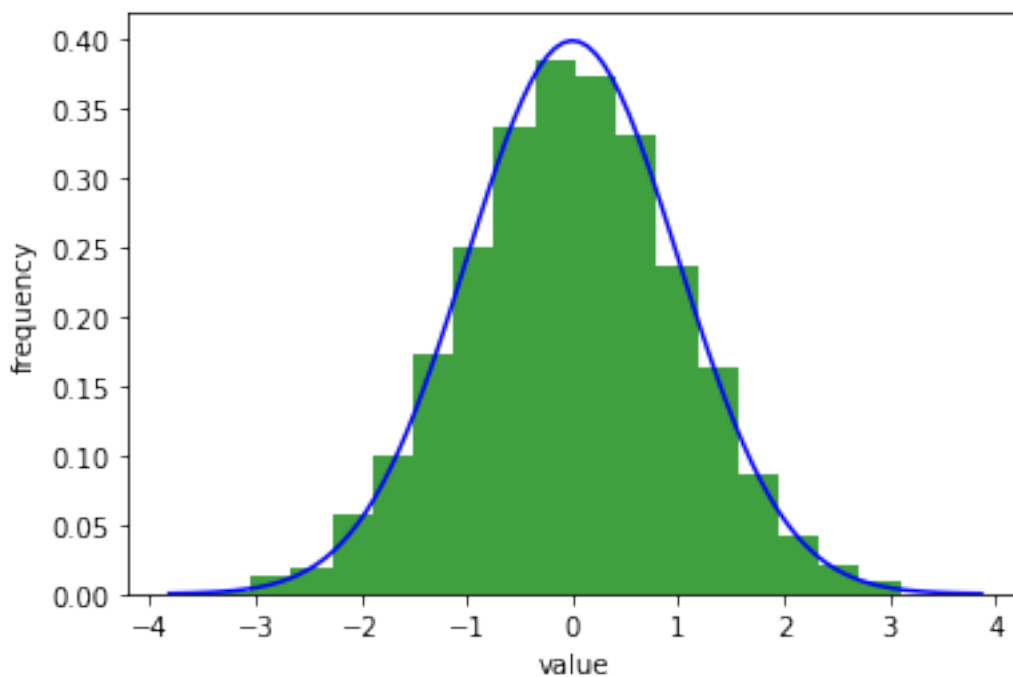
```
/Users/garth/local/python-venv/lib/python3.7/site-
packages/ipykernel_launcher.py:5: MatplotlibDeprecationWarning:
The 'normed' kwarg was deprecated in Matplotlib 2.1 and will be removed in 3.1.
Use 'density' instead.
  """
```

Experiment with the histogram plots by changing the number of samples and the number of bins.

6 Pie charts

At the time of the 2011 census, the population of Cambridgeshire by district was:

District	Population
Cambridge City	123900
East Cambridgeshire	83800
Fenland	95300
Huntingdonshire	169500
South Cambridgeshire	148800

As part of a presentation to the Cambridge City council, we need to produce a pie chart of the Cambridgeshire population by district, indicating the percentage of residents in each district and highlighting the Cambridge City 'slice'.

To do this, we first enter the data into tuples :

```
[22]: district = ('Cambridge City', 'East Cambridgeshire', 'Fenland',
       'Huntingdonshire',
                'South Cambridgeshire')
      population = (123900, 83800, 95300, 169500, 148800)
```

Tuples are appropriate because the number of districts is fixed (recall that we can't change the length of a tuple).

We next produce a colour pie chart:

```
[23]: # Specify slice colours
      # (see http://stackoverflow.com/questions/22408237/named-colors-in-matplotlib)
      colors = ['yellowgreen', 'gold', 'lightskyblue', 'lightcoral', 'red']

      # "explode" the 1st slice (Cambridge City)
      explode = (0.1, 0.0, 0, 0, 0)

      # Produce pie chart. The slices will be ordered and plotted counter-clockwise.
      plt.pie(population, explode=explode, labels=district, colors=colors,
              autopct='%1.1f%%', shadow=True, startangle=90)

      # Add title
      plt.title("2011 population distribution in Cambridgeshire")

      plt.show()
```

2011 population distribution in Cambridgeshire
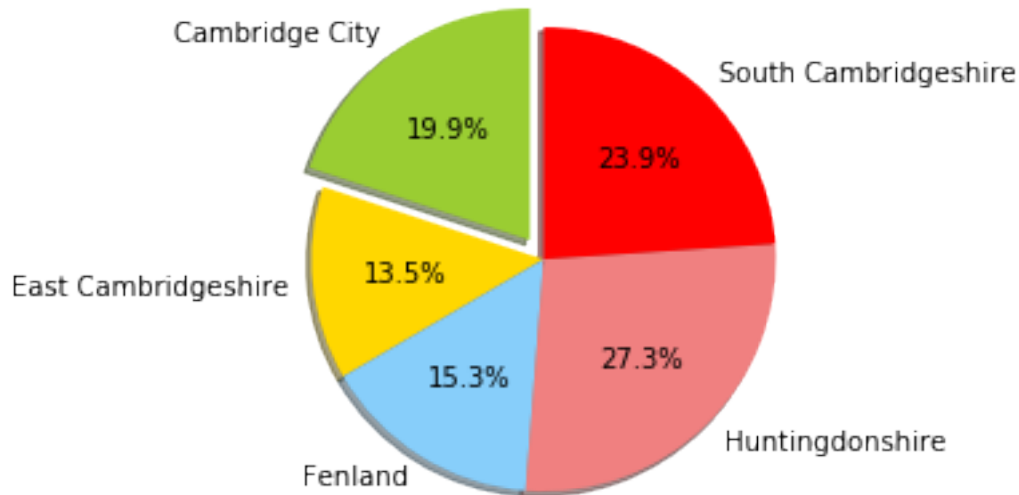
7 Interactive plots

Interactive plots are great ways to explore the influence of parameters on a response. As a simple example, we want to plot

$$f(t) = t^\alpha \sin(\omega t)$$

where α and ω are parameters. We would like to explore interactively the influence of the two parameters.

7.1 Installing widget extensions

Interactive plots be can created using the interactive widgets module `ipywidgets`. Interactive widgets have been configured for you on https://notebooks.azure.com/. To install the widgets on your own computer, follow the instructions at https://ipywidgets.readthedocs.io/en/latest/user_install.html.

We first import a function from the `ipywidgets` module:

```
[24]: from ipywidgets import interact
```

7.2 Creating interactive plots

To create an interactive plot, we need to encapsulate our plot code inside a function, with the plot parameters as arguments to the function:

```python
[25]:  def plot( =1,  =0):
           t = np.linspace(0, 2*np.pi, 200)
           plt.xlabel('$t$')
           plt.ylabel('$f$')
           plt.title(r"$\alpha$ = {}, $\omega$ = {}".format( ,  ))
           plt.plot(t, (t** )*np.sin( *t))
           plt.show()
```

(Remember, to get ω and α use `\omega + Tab` key and `\alpha + Tab` key.) We have used default function arguments - these will be the initial parameters for out interactive plot.

Next, we use the `interact` function, passing to it the name of our function that does the plotting (`plot` in this case), and a tuple for each parameters:

```python
[26]:  interact(plot,  =(-10, 10, 0.25),  =(0, 2, 0.25));
```

```
interactive(children=(FloatSlider(value=1.0, description=' ', max=10.0, min=-10.0, step=0.25),
```

If you have interactive widgets working, you should see two sliders above the plot that you can adjust. The plot should dynamically change.

The tuple for each parameters has the form (`end_point0, end_point1, step_size`) for the slider.

7.3   Advanced interactive plot creation (optional)

There is a compact way of constructing interactive plots using function *decorators*. Decorators are not required for this course, so this section is optional. If you are interested, you can find lots of information on decorators online.

Below is the interactive plot example constructed using a decorator.

```python
[27]:  @interact(  =(-10, 10, 0.25),  =(0, 2, 0.25))
       def plot( =1,  =0):
           t = np.linspace(0, 2*np.pi, 200)
           plt.xlabel('$t$')
           plt.ylabel('$f$')
           plt.title(r"$\alpha$ = {}, $\omega$ = {}".format( ,  ))
           plt.plot(t, (t** )*np.sin( *t))
           plt.show()
```

```
interactive(children=(FloatSlider(value=1.0, description=' ', max=10.0, min=-10.0, step=0.25),
```

The decorator

```python
@interact(  =(-10, 10, 0.25),  =(0, 2, 0.25))
```

is a 'shortcut' that creates the line

```python
interact(plot,  =(-10, 10, 0.25),  =(0, 2, 0.25))
```

for us.

8   Contour plots

Consider the function of two variables:

$$f(x, y) = \sin(x) \cos\left(\frac{xy^2}{10}\right)$$

It is not easy to visualise the form of this function. One way to visualise the function is to create a contour plot. To do this, we first create points in the x and y directions at which we want to evaluate the function:

[28]:
```
N = 50   # 50 points in each direction
x = np.linspace(-np.pi, np.pi, N)
y = np.linspace(-np.pi, np.pi, N)
```

Next, we use NumPy to create a 'grid' of (x, y) points:
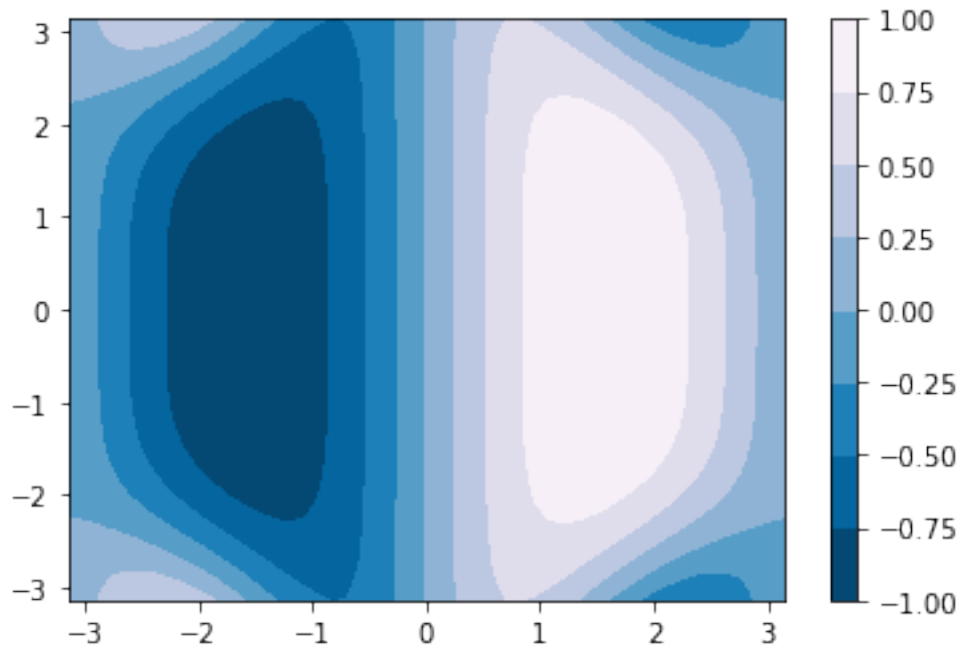
[29]:
```
X, Y = np.meshgrid(x, y)
```

If you want to explore what `meshgrid` returns, try making N smaller and printing X and Y to the screen.

We now evaluate the function f at each point:

[30]:
```
f = np.sin(X)*np.cos((X*Y**2)/10)
```

and then create a filled contour plot using `plt.contourf`:

[31]:
```
plt.contourf(X, Y, f,  cmap=matplotlib.cm.PuBu_r)
plt.colorbar()  # Add colour bar (legend)
plt.show()
```
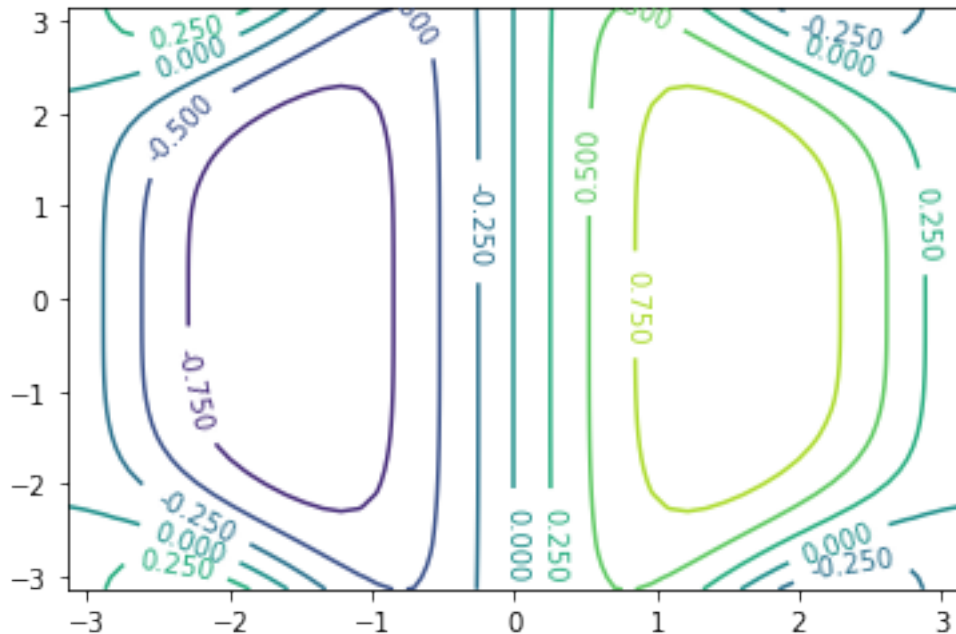
Above, we have chosen a built-in Matplotlib colour map (`matplotlib.cm.PuBu_r`). Without passing the `cmap` argument we would have seen the default colour map, which is `matplotlib.cm.jet` (and which is ugly).

To create a contour map using contour lines:

```
[32]:  # Plot
       p = plt.contour(X, Y, f)

       # Add contour labels
       plt.clabel(p, inline=1)

       plt.show()
```

There is more we could do, for example controlling the numbers of contour lines.

9 Three-dimensional plot (optional)

This is an advanced example. Skip over this if you wish.

Another way to represent a function of two variables is to use a three-dimension plot, with axes for x, y, and f. Plotting in three dimensions is considerably more complex that in two dimensions.

To create a three-dimensional plot, we need to import some extra modules from Matplotlib:

```
[33]: from mpl_toolkits.mplot3d import axes3d
      from matplotlib import cm
```

The below plots the function $f(x, y)$ in three-dimensions:
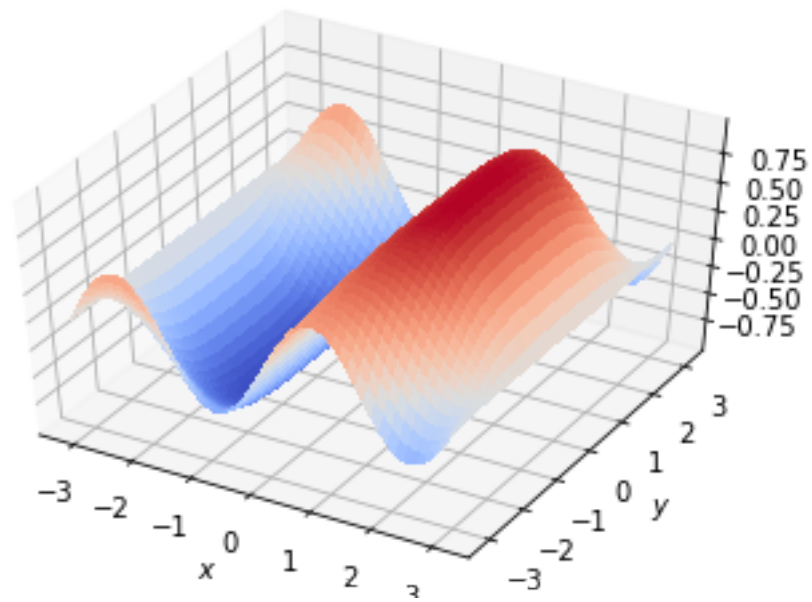
```
[34]: fig = plt.figure()
      ax = fig.gca(projection='3d')

      surf = ax.plot_surface(X, Y, f, rstride=1, cstride=1, cmap=cm.coolwarm,
                             linewidth=0, antialiased=False)

      ax.set_xlabel('$x$')
      ax.set_ylabel('$y$')

      ax.view_init(elev=45)
```

```
plt.show()
```


10   Exercises

Complete now the 08 Exercises notebook.