Phys105-All-Student-Week01-10

February 8, 2021

1 Introduction to Computational Physics

2 Introduction to Computational Physics - Week 1

2.1 Table of contents week 1

2.2 Introduction to week 1

The aim of the Introduction to Computational Physics (Phys105) module is to provide a practical introduction to programming in Python for Physics students and students on related courses. It takes a "bottom-up" approach: from day one we use examples to develop an understanding of how Python programs can be used to tackle physics problems, rather than first learning a lot of general principles and then trying to apply them. There are two reasons for this. Firstly, you will be using Python in your practical and some other modules soon, so need to be in a position to do some things quite quickly. Secondly, most people learn more effectively by actually doing things than by just hearing how they can be done.

This first Notebook provides some of the basic information you need for the Phys105 module. It shows you how to install the tools we will be using in the course (the Python programming language and Jupyter Notebooks) and provides a few exercises that will help you start to learn how to tackle problems using them. When you have finished working through it, you should understand the basic structure of a Notebook, be able to write Markdown that allows you to present text, figures, formulae and tables attractively in Notebooks and be able to use the program `least_squares` in a Notebook to fit a straight line to a set of data points.

Please take the time to read the information provided here, don't just skip to the exercises. If you have read through the Notebook and still have difficulties with the exercises, or don't understand something, ask one of the demonstrators for help. Further information and explanation is provided in the recommended textbooks (A student's guide to Python for physical modelling, or Learning scientific programming with Python). There is also lots of information on Python and Jupyter Notebooks on the web, so you can ask Google for help!

When you have finished the exercises, ask one of the demonstrators to mark your work.

2.3 Installing Python and Jupyter software

Python and Jupyter Notebooks have aready been installed on the University PCs you will be using in the Phys105 practical sessions.

If you want to use Python and Jupyter Notebooks on your own computer, you can install the

software required for Windows, Mac or Linux as follows:
Go to the Anaconda web site https://www.anaconda.com/download/. Select Windows, macOS or Linux, depending on the operating system running on your computer. Download the version of Anaconda labelled Python 3.7 (not the Python 2.7 version!). Follow the installation instructions on the web site.

Once Anaconda is installed, open it and launch Jupyter Notebook to get started:

- On Windows, click the "Anaconda3" icon in the start menu and then "Jupyter Notebook".

- On Linux, open a terminal window , type in the command "jupyter notebook &" and press *Enter*.
- On a Mac, click on "Anaconda-Navigator" in the LaunchPad and then "Jupyter Notebook" or open a terminal window, type in the command "jupyter notebook &" and press *Enter*.

This "Phys105-Week01-Student.ipynb" file can be opened within Jupyter, which will allow you to work with the code. **It is strongly advised you do this** as opposed to just reading the PDF file!

2.4 Jupyter Notebooks

Jupyter Notebooks consist of cells in which nicely formatted documents can be written and cells which contain computer code. The language used to create the document cells is called Markdown, and we will use the Python programming language in the code cells. We will start off by learning how to create Markdown documents, but learning Python will be the main aim of this course.

When you start a new Notebook, its first cell will be a code cell and any new cell you create will be a code cell by default. You can create a new cell below the current cell by clicking the "+" symbol in the Notebook menu bar or by using the *Insert* menu. (Click on *Insert*, then on *Insert Cell Above*, or *Insert Cell Below*, as appropriate.) To change a cell's type to Markdown, select the cell (by clicking in it) and then click *Cell, Cell Type* and *Markdown*. Alternatively, select the cell, press *Esc*, then *m* (for Markdown). You will have to click inside the cell (or press *Enter*) before it is selected and you are able to type in it!

A Markdown cell can be changed to a code cell using the *Cell* menu, or by typing *Esc, y*.

Cells can be deleted by selecting them and then using the *Edit* menu, or by pressing *Esc, d, d*.

2.4.1 Week 1 exercise 1

Look through the menus in the Notebook screen. Use them to create a new cell below this one. Convert the type of the cell to Markdown, write something in the cell, then delete it. Repeat this exercise using the keyboard shortcuts described above. If you have problems doing this, ask a demonstrator for help!

2.5 Markdown cells

Markdown is a way of writing nicely formatted text, allowing the inclusion of pictures, web links, videos and other features in your Notebooks.

To see how this cell was written using Markdown, double click on it. (Do this now!)

To "run" or "compile" the cell, so the text is formatted nicely, select the cell (click in it) and press *Ctrl + Enter* (i.e. press the *Ctrl* key and hold it down, then press the *Enter* key, then release both keys). Use *Command + Enter* on a Mac. Alternatively, you can click on the *Run* button on the menu bar. If you press *Shift + Enter*, you will run the current cell and move to the next cell (or create a new cell below the current one if there isn't one already there).

Flipping between looking at the Markdown (double click) and the compiled cell (*Ctrl + Enter*) will allow you to see how Markdown can be used.

2.5.1 Entering text and tables

If you want to write some normal text, just type it into the cell. If you want to emphasize the text, you can *write in italics* (using asterisks before and after the section that you want to be in italics), or **make it bold** (using double asterisks). (There are *alternative ways of getting italics* and of **entering bold text**, using single and double underscores, respectively.) You can also ~~cross out text~~ (using two ~ symbols before and after the text to be "deleted").

If you want a new paragraph, press *Enter* twice, so you produce an empty line.

If you want to start a new line without having a new paragraph, leave two spaces at the end of the line, like this:
This is a new line.
So is this.

If you want a numbered list, do this: 1. This is the first entry. 2. This is the second. 3. And this the third.

Bulleted lists are also easy to produce: * This is the first bullet. * And this the second.

If you want to present information in a table, use the following syntax (double click to see the Markdown!):

Number	Angle (degrees)	Cosine of angle
0	0	1
1	30	0.866
2	45	0.707
3	60	0.5
4	90	0.0

Table 1 *The value of the cosine of several angles.*

(You don't have to line up the Markdown columns; when you run the cell the table will be formatted for you, but it does make reading and editing the Markdown easier!) Notice that you have to enter the caption "by hand" below the table; table and figure numbers are not entered automatically.

If you want a title or heading, use the hash symbol. (One hash is a title, two a heading, three a subheading, etc.) For example, here is a subheading…

2.5.2 Including links

Links to pages on the web can be included as is done below for the introduction to Markdown. Alternatively, the link can be direct, e.g. `https://www.liverpool.ac.uk`.

Links can also be "reference style". This is where you can find the UK google home page. You can put the link at the end of the paragraph or cell. This makes the Markdown a little easier to read. There is no visible difference in the text that results when you run the cell.

Links can be made to sections or figures in the Notebook. For example, back to the section Entering text and tables. Double clicking this cell will allow you to see how the section is labelled (using a "#" followed by the section title with the spaces replaced by hyphens) and how a link to that label is created. The label is not allowed to contain any spaces!

2.5.3 Figures

Images stored on your computer can be added using the syntax below (double click on the cell to see it!). The first example uses the path relative to the folder/directory in which the Notebook is saved:

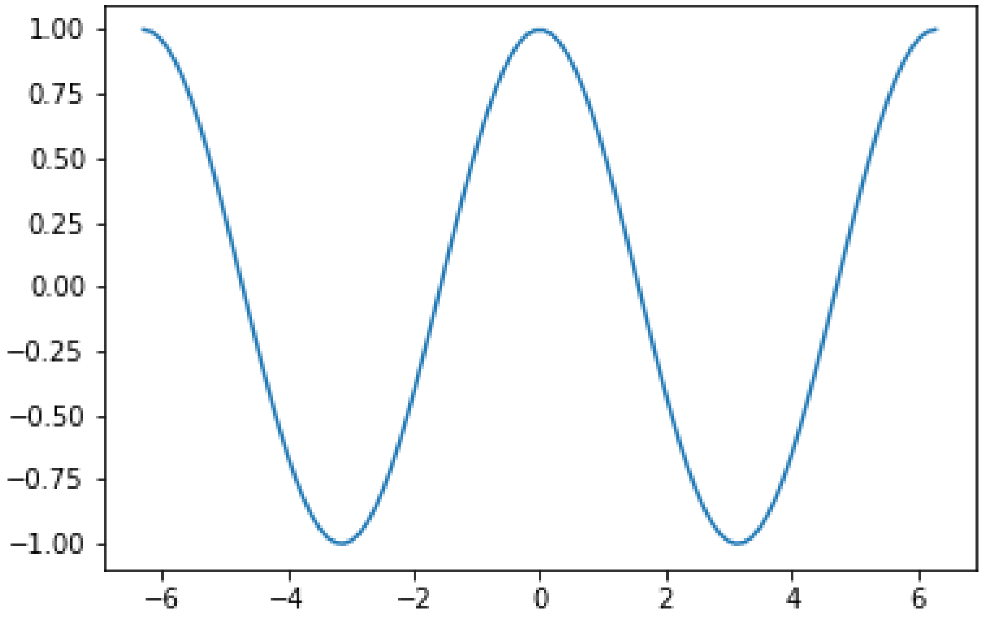


Figure 1 *Cosine as a function of angle*

The absolute path can also be used in Markdown as shown below...though you can't go back further than the Jupyter start folder/directory. These "remote" images are not automatically incorporated if you convert your markdown to PDF, so the following section is "commented out". (Anything between the "<", "!" and "–" and the "–" and ">" symbols is treated as a comment, i.e. an explanatory remark, and is not displayed when the Markdown is compiled.) Remove the comment marks if you want to see what this bit of Markdown produces.

Figures from the web can be added in the markdown using the syntax shown below. (Again, remove the comment marks if you want to compile this Markdown.)

This section is commented out, as there is no automatic location and conversion of web images to PDF, so the above lines would have spoiled the PDF version of this Notebook.

If you want to create a PDF file of this Notebook including the above images, you will have to download the relevant files to your computer and use a local link!

2.5.4 Entering formulae

Mathematical formulae are entered using LaTeX. How this can be done is best illustrated with a few examples. A formula can be entered "inline" like this: $E = mc^2$. Notice how the dollar signs "bracket" the mathematical formula. It is important in Markdown to ensure there are no spaces between the dollar signs and the characters in the formula. Although spaces are allowed by the LaTeX standard, they can cause problems when you try and run LaTex on Notebooks, e.g. when using *File, Export Notebook as..., LaTeX*, or *File, Export Notebook as..., PDF*.

Formulae can also be placed on their own line in the centre of the page using two dollar symbols:

$$\sin^2 x + \cos^2 x = 1.$$

Alternatively, you can use the following syntax, which also allows you to "line up" successive equations:

$$\begin{align}
\int_0^\pi \sin x dx &= \left(-\cos x\right)_0^\pi \tag{1}\\
&= -\left(\cos(\pi) - cos(0)\right) \tag{2}\\
&= -(-1-1) \tag{3}\\
&= 2. \tag{4}
\end{align}$$

The ampersand (&) tells LaTeX where to align the equations and the double backslash tells it where to break the lines. This equation also illustrates the use of the commands (and) to make brackets of the appropriate size (i.e. that expand as they are nested). (The command backslash text changes the font in the region delimited by the curly brackets.)

Notice how functions like sin and cos (and log, exp etc.) are entered, and how superscripts (x^2) and subscripts (p_0) can be obtained. If you need more than one character in a superscript (or subscript), enclose the relevant section in curly brackets, e.g. x_{max}.

Fractions are obtained like this $\frac{1}{2}$, and a vast array of symbols is available, for example: $\sqrt{2}$, $2 \approx 2.5$, $\int_0^1 x^2\,dx$, $\sum_{n=0}^{15} x_n$. (See here for more.) The backslash followed by a comma in the integral produces a "slim space" between the x^2 and the dx, a full space would be obtained using a backslash with a space after it. As already mentioned, brackets which expand to the required height can be entered using () or []. Greek letters are written α, β etc. Another example combining some of these features is the formula for the Fermi function:

$$F(\epsilon) = \frac{1}{\exp\left[\frac{\epsilon-\mu}{kT}\right]+1}. \tag{5}$$

Finally, "inline" segments of computer code can be indicated using reverse quotes, for example: `print("This is a Python 3 print statement")`. Three reverse quotes, with a tag indicating the relevant language, can be used to produce blocks of code. We shall be interested in Python, so we will be seeing things like:

```python
import numpy as np
import matplotlib.pyplot as plt
#
print("This illustrates how Markdown can be used to format a block of code")
#
for n in range(0, nMax + 1):
    print("n =",n)
```

Note that the "three reverse quotes with language tag" produces code with colours highlighting the various elements of the language; much clearer than the inline format.

And that's about all we need to know about Markdown!

2.5.5 Week 1 exercise 2

In the cell below, write a short document with a heading, a line of text, a shopping list with five numbered items, a table with two columns (one of which contains the numbers 1 to 7 and the other the corresponding days of the week), the formula for a gaussian distribution, a link to the Liverpool University web site and a picture of your choice!

2.6 Code cells

Markdown allows us to write attractive documents quickly and efficiently, but the power of Jupyter Notebooks is that they allow these documents to be combined with computer code. We can describe a problem and work on the programs needed to solve it in the same place. Below, we will look at an example of how this can be of use in the context of laboratory measurements, and at the same time see our first Python program. We will fit a straight line to some experimental data using a routine called `least_squares`.

A word of warning. Unless you have done some programming before, you are unlikely to understand the nuts and bolts of all of the code below; we will learn that in later weeks. For now, it is important that you understand how to get data into `least_squares`, run the program and extract the results, as you will need to do this in Practical Physics I (Phys106).

2.6.1 An imaginary experiment

Ten measurements have been made of two quantities, x and y. These are thought to be linearly related, i.e. to lie on a straight line, $y = mx + c$. The x and y values (with their measurement errors) are:

Measurement	x value	Error in x	y value	Error in y
1	1.50	0.21	14.3	2.1
2	2.31	0.11	20.2	1.7
3	2.78	0.43	30.1	3.3
4	3.91	0.10	41.5	1.1
5	3.88	0.07	42.7	0.9
6	4.76	0.08	47.1	0.8

Measurement	x value	Error in x	y value	Error in y
7	5.62	0.05	52.9	0.5
8	7.02	0.09	68.8	0.7
9	8.45	0.17	85.2	1.2
10	9.65	0.11	99.4	2.9

Table 2 *A number of data points which are expected to lie on a straight line.*

We will test whether the expected relationship holds by fitting a straight line to the data and simultaneously determine the values of m and c that best describe the data.

The program in the cells below shows how this can be done. It is broken down into sections interspersed with an explanation of what each section is doing.

2.6.2 Fitting a straight line to data

The first step is to import (load) the Python modules used by the program. Here, we need: * NumPy (Numerical Python), a library of routines for defining arrays of numbers and working with them. * Matplotlib.pyplot, a set of plotting routines for making graphs of various types. * least_squares from scipy.optimize (part of the Scientific Python library), the fitting routine we shall use.

We also use the "line magic" command

`%matplotlib inline`

to ensure that any plots we make appear in the Notebook and don't just get stored as files on the computer.

The following code cell imports the routines we will need. (Note that, in Python, comments are indicated by a "#": anything after the hash symbol is not part of the program but is a "comment" providing information on what the program is doing.) To actually do the importing, you must run the cell. As for Markdown cells, you do this by selecting it and using the menu, or selecting the cell and pressing *Ctrl + Enter*. Alternatively, *Shift + Enter* will run the cell and move to the next one or create a new one if there isn't already a cell after the current one.

[1]:
```python
# <!-- Student -->
import numpy as np
import matplotlib.pyplot as plt
#
# least_squares is a fitting routine
from scipy.optimize import least_squares
#
# This "line magic" ensures plots are displayed in the Notebook
%matplotlib inline
```

2.6.3 Week 1 exercise 3

Run the code cell below and check the data that it prints out against the numbers in Table 2. Look for the mistakes and fix them by editing the Python in the cell. Re-run and re-check the cell until all the data are correctly entered.

[2]:
```
# <!-- Student -->
# nPoints is the number of data points in the fit
nPoints = 10
#
# Define NumPy arrays, initially filled with zeros, to store the x and y data␣
 ↪values
xData = np.zeros(nPoints)
yData = np.zeros(nPoints)
#
# Define arrays to store the errors in x and y
xError = np.zeros(nPoints)
yError = np.zeros(nPoints)
#
# Enter the x data points.
xData[0] = 1.50
xData[1] = 2.31
xData[2] = 2.78
xData[3] = 3.58
xData[4] = 4.08
xData[5] = 4.76
xData[6] = 5.62
xData[7] = 7.02
xData[8] = 8.45
xData[9] = 9.65
#
# Enter the y data points.
yData[0] = 14.3
yData[1] = 20.2
yData[2] = 30.1
yData[3] = 36.5
yData[4] = 42.7
yData[5] = 47.1
yData[6] = 52.9
yData[7] = 68.8
yData[8] = 85.2
yData[9] = 99.4
#
# Enter the errors in the x values
xError[0] = 0.21
xError[1] = 0.11
xError[2] = 0.43
xError[3] = 0.13
xError[4] = 0.17
xError[5] = 0.18
xError[6] = 0.15
```

```
xError[7] = 0.19
xError[8] = 0.17
xError[9] = 0.11
#
# Enter the errors in the y values
yError[0] = 2.1
yError[1] = 1.7
yError[2] = 3.3
yError[3] = 1.1
yError[4] = 0.9
yError[5] = 1.1
yError[6] = 1.5
yError[7] = 0.9
yError[8] = 1.2
yError[9] = 2.9
#
# The code below prints out the data we have entered. We will explain how it␣
 ↪works
# later in the course
print(" ")
print("Check the data points:")
print("Point\t x\t\t\ty\t")
for n in range(0, nPoints):
    print("{:d}\t{:5.2f}\t+-{:5.2f}\t\t{:5.2f}\t+-{:5.2f}".\
          format(n, xData[n], xError[n], yData[n], yError[n]))
```

```
Check the data points:
Point    x                      y
0        1.50   +- 0.21         14.30   +- 2.10
1        2.31   +- 0.11         20.20   +- 1.70
2        2.78   +- 0.43         30.10   +- 3.30
3        3.58   +- 0.13         36.50   +- 1.10
4        4.08   +- 0.17         42.70   +- 0.90
5        4.76   +- 0.18         47.10   +- 1.10
6        5.62   +- 0.15         52.90   +- 1.50
7        7.02   +- 0.19         68.80   +- 0.90
8        8.45   +- 0.17         85.20   +- 1.20
9        9.65   +- 0.11         99.40   +- 2.90
```

The routine we use for fitting is optimize.least_squares from `scipy.optimize`. This requires us to define the function that we want to fit to the data (called `fitLine` here) and a function (`fitChi`), which describes how differences between the data and the fit are treated. (This function is referred to as a *cost* or *merit function* in the SciPy documentation.) The function `fitChi` depends on the derivative of `fitline`, `fitLineDiff`. You should be able to work out why this is the case when you have done some error analysis in Maths for Physics I (Phys107) or from the information on errors given in the Phys106 handbook!

Note that three quotes before and after a section of text are another way of writing comments in

Python - the text between the quotes does not have any effect when the code is run, it serves only to explain what the code does.

The definition of `fitLine` tells us that it accepts two arguments, `p` and `x` (just as though it were a mathematical function fitLine(p, x)). In this case, `p` is a two-component vector, or `array`. The first component, `p[0]`, is the intercept of the straight line, the second component, `p[1]`, is its gradient. That is, `p[0]`$= c$ and `p[1]`$= m$ in the expression $y = mx + c$. Notice again that Python counts array elements from 0, not 1!

[3]:
```
# <!-- Student -->
def fitLine(p, x):
    '''
    Straight line
    '''
    f = p[0] + p[1]*x
    return f
#
def fitLineDiff(p, x):
    '''
    Differential of straight line
    '''
    df = p[1]
    return df
#
def fitChi(p, x, y, xerr, yerr):
    '''
    Cost function
    '''
    e = (y - fitLine(p, x))/(np.sqrt(yerr**2 + fitLineDiff(p, x)**2*xerr**2))
    return e
```

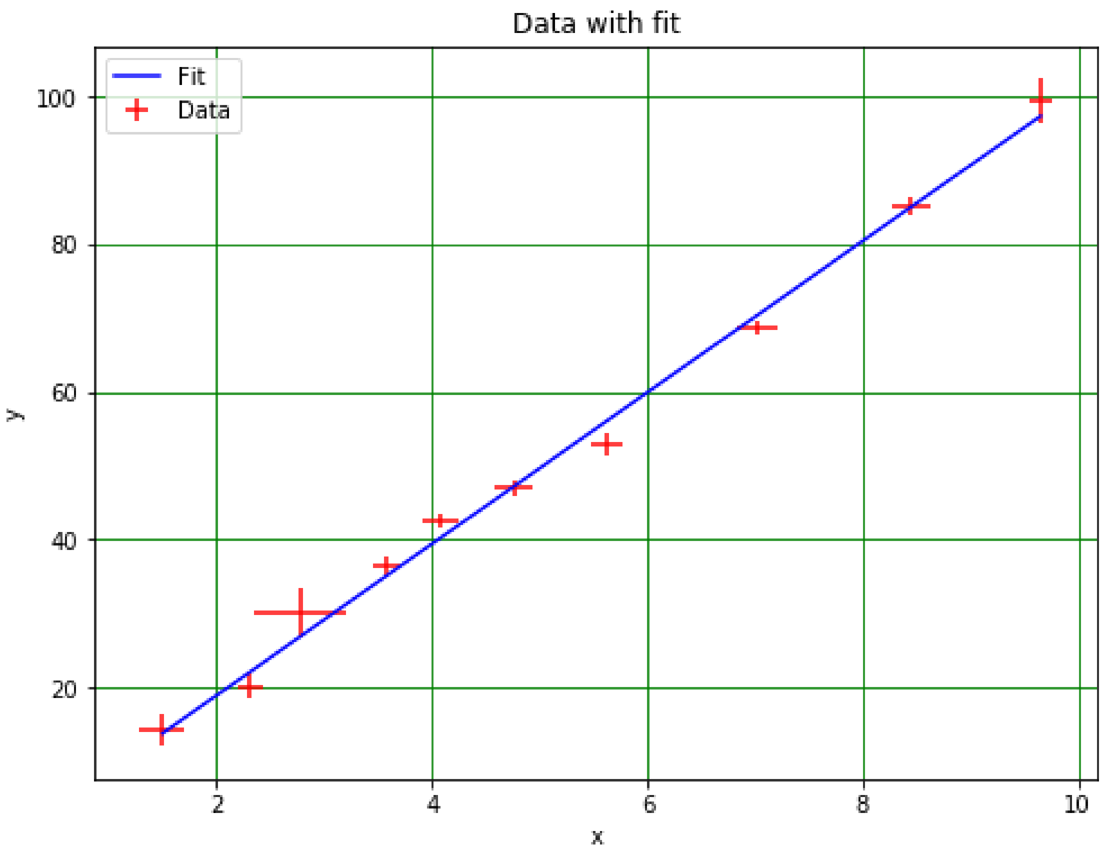
Here, we set our first guesses for the values of the intercept and the gradient, run the fit, and transfer the information on whether the fit has worked to the variable `fitOK`.

[4]:
```
# <!-- Student -->
#
# Set initial values of fit parameters, run fit
pInit = [1.0, 1.0]
out = least_squares(fitChi, pInit, args=(xData, yData, xError, yError))
#
fitOK = out.success
```

Now we check if the fit has worked, and if it has, transfer the fitted parameters into the array `pFinal`. This contains the fitted values of the intercept (`pFinal[0]`) and the gradient (`pFinal[1]`). We also calculate and print out the values of the statistics χ^2 and χ^2/NDF which describe how well the fitted straight line matches the data. We calculate the errors on the fitted parameters by inverting the squared Jacobian matrix to get the covariance matrix `covar`. The diagonal components of `covar` are the squared errors on the fitted intercept (`covar[0, 0]`) and gradient (`covar[1, 1]`). Finally, we plot the data with error bars (using the routine `plt.errorbar` from `matplotlib.pyplot`) and draw the fitted line on the plot (using `plt.plot`).

[5]:
```python
# <!-- Student -->
#
# Test if fit failed
if not fitOK:
    print(" ")
    print("Fit failed")
else:
    #
    # get output
    pFinal = out.x
    cVal = pFinal[0]
    mVal = pFinal[1]
    #
    #   Calculate chis**2 per point, summed chi**2 and chi**2/NDF
    chisqArr = fitChi(pFinal, xData, yData, xError, yError)**2
    chisq = np.sum(chisqArr)
    NDF = nPoints - 2
    redchisq = chisq/NDF
#
    np.set_printoptions(precision = 3)
    print(" ")
    print("Fit quality:")
    print("chisq per point = \n",chisqArr)
    print("chisq = {:5.2f}, chisq/NDF = {:5.2f}.".format(chisq, redchisq))
    #
    # Compute covariance
    jMat = out.jac
    jMat2 = np.dot(jMat.T, jMat)
    detJmat2 = np.linalg.det(jMat2)
    #
    if detJmat2 < 1E-32:
        print("Value of determinat detJmat2",detJmat2)
        print("Matrix singular, error calculation failed.")
        print(" ")
        print("Parameters returned by fit:")
        print("Intercept = {:5.2f}".format(cVal))
        print("Gradient = {:5.2f}".format(mVal))
        print(" ")
        cErr = 0.0
        mErr = 0.0
    else:
        covar = np.linalg.inv(jMat2)
        cErr = np.sqrt(covar[0, 0])
        mErr = np.sqrt(covar[1, 1])
        #
        print(" ")
        print("Parameters returned by fit:")
        print("Intercept = {:5.2f} +- {:5.2f}".format(cVal, cErr))
        print("Gradient = {:5.2f} +- {:5.2f}".format(mVal, mErr))
        print(" ")
    #
    # Calculate fitted function values
    fitData = fitLine(pFinal, xData)
    #
    # Plot data
    fig = plt.figure(figsize = (8, 6))
    plt.title('Data with fit')
    plt.xlabel('x')
    plt.ylabel('y')
    plt.errorbar(xData, yData, xerr = xError, yerr = yError, fmt='r', \
                 linestyle = '', label = "Data")
    plt.plot(xData, fitData, color = 'b', linestyle = '-', label = "Fit")
    #plt.xlim(1.0, 7.0)
    #plt.ylim(0.0, 3.0)
    plt.grid(color = 'g')
    plt.legend(loc = 2)
    plt.show()
```

```
Fit quality:
chisq per point =
 [0.025 0.891 0.33  0.626 1.556 0.003 2.123 0.535 0.008 0.453]
chisq =  6.55, chisq/NDF =  0.82.

Parameters returned by fit:
Intercept = -1.54 +-  1.74
Gradient = 10.24 +-  0.32
```



2.6.4 Results

The data look to be reasonably well described by a straight line (provided you have found and fixed the mistakes in the input!). The value of the statistic χ^2/NDF backs this up. (Values in the range $0.25 < \chi^2$/NDF \$ < 4\$ represent good agreement, the closer to one the better.) The gradient of the straight line and its intercept can be read off from the output of `least_squares`.

2.6.5 Week 1 exercise 4

Fit a straight line to the data in Table 3 (which is the same as Table 2 with two new points added). Do this by copying the relevant cells above and pasting them below this cell (use the *Edit* menu). Make the changes you need to add the new data points.

Measurement	x value	Error in x	y value	Error in y
1	1.50	0.21	14.3	2.1
2	2.31	0.11	20.2	1.7
3	2.78	0.43	30.1	3.3
4	3.91	0.10	41.5	1.1
5	3.88	0.07	42.7	0.9
6	4.76	0.08	47.1	0.8

Measurement	x value	Error in x	y value	Error in y
7	5.62	0.05	52.9	0.5
8	7.02	0.09	68.8	0.7
9	8.45	0.17	85.2	1.2
10	9.65	0.11	99.4	2.9
11	10.25	0.21	110.5	3.1
12	11.85	0.41	122.1	3.3

Table 3 *Extended range of data points*

2.6.6 Week 1 exercise 5

Does the straight line still describe the data? Have the values of the gradient or the intercept changed significantly?

That's the end of Week 1 for Phys105. You will have a chance to practice using the line-fitting routine that we have worked with here in Practical Physics I (Phys 106).

3 Introduction to Computational Physics - Week 2

3.1 Table of contents week 2

3.2 Introduction to week 2

This week, we will start by doing some calculations using Python, look at some of the ways data can be stored and manipulated. We will also look at one way of reading data into Python programs and see how to create hostograms, as you willneed these in Proctical Physics I (Phy106)!

3.3 Calculations with Python

The following cell shows how Python can be used to carry out a simple calculation, the addition of two integers. (Run the cell to get the answer; remember, the easiest way of doing this is to select the cell then press *Ctrl* + *Enter*.)

```
[6]: # <!-- Student -->
     333 + 666
```

```
[6]: 999
```

The answer should appear below the cell you have just run (and it should be 999!).

3.3.1 Week 2 exercise 1

Work out what the following operators do by trying them out on pairs of integers. (Some are obvious, some less so!) Write a very brief description of the function of each operator. 1. `+` 2. `-` 3. `*` 4. `/` 5. `**` 6. `//` 7. `%`

As well as integers, we can enter and manipulate floating point numbers (`floats`). For example, we can calculate the area of a circle with radius 0.2 (in some unspecified units!) as follows:

```
[7]: # <!-- Student -->
     3.1415927*0.2**2
```

```
[7]: 0.125663708
```

We can make this calculation clearer by associating (*binding* in Python-speak) a name with each of the numbers, `r` for radius `pi` for π and `A` for area:

```
[8]: # <!-- Student -->
     r = 0.2
     pi = 3.1415927
     A = pi*r**2
     A
```

```
[8]: 0.125663708
```

Notice that writing a variable name without anything else at the end of a code cell results in the value of that variable being printed out (`A` in the example above).

In addition to the values of the variables, we can provide explanatory text by using Python's `print()` function, as shown below.

```
[9]: # <!-- Student -->
     print("Radius of circle",r)
```

```
Radius of circle 0.2
```

3.3.2 Week 2 exercise 2

Modify the cell above so that it prints out the string "Area of circle" and then the value of `A`.

3.4 Functions in Python

Suppose we want to calculate the sine of an angle, the logarithm of a number, or use some other mathematical function. Can we do this in Python? The answer is (of course) yes, but but we need to import these functions first. We shall use the versions of sine, cosine etc. from the Numerical Python package, NumPy. As we have seen in Week 1, NumPy can be imported and the functions it provides used as shown below:

[10]:
```
# <!-- Student -->
import numpy as np
#
angle = 1.42 # radians
sinAngle = np.sin(angle)
x = 176.4
logx = np.log(x) # logarithm to the base e
print("The sine of",angle,"is",sinAngle)
print("The natural logarithm of",x,"is",logx)
```

```
The sine of 1.42 is 0.9886517628517197
The natural logarithm of 176.4 is 5.172754143572691
```

The functions `np.sin`, `np.cos` etc. expect their arguments to be in radians, not degrees.

Python also allows is to create our own functions. We will see how to do this next week!

3.5 Lists and tuples

In addition to the integers and real numbers (`floats`) we have used above, Python has a large range of data types. These include `lists` and `tuples`. Lists can be created by using commas to separate their elements and enclosing them in square brackets. Various types of element are allowed, including integers, floats and strings (lists of characters, delimited by either double or single quotation marks). We can refer to an element in a list using its index (the position of the element, counting from zero). The index is enclosed in square brackets, as shown in the examples below:

[11]:
```
# <!-- Student -->
thisList = [1, 2.278, -8, "cheese", 'blue ', np.pi]
print("thisList =",thisList)
print("thisList[0] =",thisList[0])
print("thisList[0] + thisList[1] =",thisList[0] + thisList[1])
print("thisList[4] + thisList[3] =",thisList[4] + thisList[3])
```

```
thisList = [1, 2.278, -8, 'cheese', 'blue ', 3.141592653589793]
thisList[0] = 1
thisList[0] + thisList[1] = 3.278
thisList[4] + thisList[3] = blue cheese
```

As shown above, we can combine list (or tuple) elements in various ways. For example, we can add two elements. Note that the meaning of add depends on what the type of the element is. Adding numbers does just what you'd expect; adding strings concatenates the two elements (tags one onto the end of the other). You can't add elements where the notion of adding doesn't make sense (e.g. you can't add an integer to a string). Python will give you an error if you try! Note also that the string 'blue' includes a space after the 'e', this is part of the string. Without it,

```
print("thisList[4] + thisList[3] =",thisList[4] + thisList[3])
```

would give the result

```
thisList[4] + thisList[3] = bluecheese
```

The elements in a list can be changed (they are *mutable*). For example, we can assign a new value to element 3:

[12]:
```
# <!-- Student -->
thisList[3] = "car"
#
print("thisList[4] + thisList[3] =",thisList[4] + thisList[3])
```

```
thisList[4] + thisList[3] = blue car
```

A related data type is the tuple. This is similar to the list, but once defined its elements are fixed (they are *immutable*). They can be created as follows - notice that lists are indicated by `[]` and tuples use`()`.

[13]:
```
# <!-- Student -->
thisTuple = (3.14, -9, "orange")
#
print("thisTuple =",thisTuple)
```

```
thisTuple = (3.14, -9, 'orange')
```

The elements in a tuple can be referred to using their indices in exactly the same way as those in a list:

[14]:
```
# <!-- Student -->
#
print("thisTuple[2] =",thisTuple[2])
```

```
thisTuple[2] = orange
```

Because tuples are immutable, if we try and change one of the elements, we get an error:

[15]:
```
# <!-- Student -->
#
thisTuple[2] = 27
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-15-e4e8c77ac3f5> in <module>
      1 # <!-- Student -->
      2 #
----> 3 thisTuple[2] = 27
```

```
TypeError: 'tuple' object does not support item assignment
```

An aside - tuples can also be created without using brackets, as in the example below. We will generally avoid this as it can be confusing! Notice that when it prints out a tuple, Python adds the brackets, even if we didn't put them in!

We will look at more at lists, tuples, strings, and the many other data types available in Python in future weeks. For now, we will concentrate on one of the most important data structures for scientific computing, the NumPy array.

3.6 NumPy arrays

NumPy arrays are one of the most useful ways of manipulating data when using for Python for scientific computing. They allow efficient storage of, and rapid computations with, vectors, matrices or larger dimension arrays of numbers. This efficency comes at a price: all the elements in a NumPy array have to be of the same type; `ints` and `floats` can't be put in the same NumPy array, for example.

3.6.1 One dimensional arrays

One dimensional arrays (vectors) can be created and assigned values in various ways, as is shown below. (Remember how we entered the data for the `leastsq` fit in Week 1!) Note how the index is used to refer to the array element, and that the first index is zero…

[16]:
```python
# <!-- Student -->
import numpy as np
#
arrayFromList = np.array([0, 10, 20, 20, 40])
#
length = 10
arrayOfZeros = np.zeros(length)
arrayOfOnes = np.ones(length)
#
arrayOfNumbers = np.zeros(length)
arrayOfNumbers[0] = 3
arrayOfNumbers[1] = 1
arrayOfNumbers[2] = 4
arrayOfNumbers[3] = 1
arrayOfNumbers[4] = 5
arrayOfNumbers[5] = 9
arrayOfNumbers[6] = 2
arrayOfNumbers[7] = 6
arrayOfNumbers[8] = 5
arrayOfNumbers[9] = 4
#
print("arrayFromList",arrayFromList)
print("arrayOfZeros",arrayOfZeros)
print("arrayOfOnes",arrayOfOnes)
print("arrayOfNumbers",arrayOfNumbers)
```

```
arrayFromList [ 0 10 20 20 40]
arrayOfZeros [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
arrayOfOnes [1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
arrayOfNumbers [3. 1. 4. 1. 5. 9. 2. 6. 5. 4.]
```

[17]:
```
# <!-- Student -->
length = 3
start = 10
stop = 30
arrayOfNumbers = np.linspace(start, stop, length)
print("arrayOfNumbers",arrayOfNumbers)
```

```
arrayOfNumbers [10. 20. 30.]
```

[18]:
```
# <!-- Student -->
print("arrayOfNumbers[1]",arrayOfNumbers[1])
```

```
arrayOfNumbers[1] 20.0
```

Another useful way of filling an array is using the function `np.linspace`. This allows beginning (`beg`) and end (`end`) values to be entered together with a number of steps (`nSteps`). The array starts at `beg` and `nStep` equally spaced values are added until `end` is reached. As there are `nStep` steps, the array contains `nStep + 1` values:

[19]:
```
# <!-- Student -->
beg = 0.0
end = 20.0
nSteps = 10
stepArray = np.linspace(beg, end, nSteps + 1)
print("stepArray =",stepArray)
```

```
stepArray = [ 0.  2.  4.  6.  8. 10. 12. 14. 16. 18. 20.]
```

If you want to know the size of the steps between in the array created by `np.linspace`, you can add the argument `retstep = True` as below (`retstep` is short for return step).

[20]:
```
# <!-- Student -->
beg = 0.0
end = 20.0
nSteps = 10
stepArray, stepSize = np.linspace(beg, end, nSteps + 1, retstep = True)
print("stepArray =",stepArray)
print("stepSize =",stepSize)
```

```
stepArray = [ 0.  2.  4.  6.  8. 10. 12. 14. 16. 18. 20.]
stepSize = 2.0
```

3.6.2 Week 2 exercise 3

Use `np.linspace` to create an array containing the numbers 10.0 to 22.0 with a step of 3.0 between each entry. How many steps will this array contain? Print out your array to check it is correct!

The values of the elements of `stepArray` can be retrieved (or changed) using their indices:

[21]:
```
# <!-- Student -->
print("stepArray[3] =",stepArray[3])
stepArray[3] = 76
print("stepArray[3] =",stepArray[3])
```

```
stepArray[3] = 6.0
stepArray[3] = 76.0
```

Because the first element in the array has index zero, the last element has index `nSteps`, even though there are `nSteps + 1` elements in the array:

[22]:
```
# <!-- Student -->
print("stepArray[nSteps] =",stepArray[nSteps])
```

```
stepArray[nSteps] = 20.0
```

The last (and next-to-last, and next-to-next-to-last) elements of the array can also be accessed by using the indices -1 (and -2, and -3). This can be useful, but can lead to confusion in some circumstances!

[23]:
```
# <!-- Student -->
print("stepArray[-1] =",stepArray[-1])
print("stepArray[-2] =",stepArray[-2])
print("stepArray[-3] =",stepArray[-3])
```

```
stepArray[-1] = 20.0
stepArray[-2] = 18.0
stepArray[-3] = 16.0
```

An alternative NumPy routine for filling arrays with evenly spaced values is `arange`. You can read about arange here.

3.6.3 Two and more dimensional arrays

NumPy arrays with two or more dimensions can also be created. The dimensions are defined by a tuple, e.g. `(3, 4)` for a matrix with three rows and four columns.

[24]:
```
# <!-- Student -->
matrix = np.ones((3, 4))
print("matrix =\n",matrix)   # \n in string "matrix =\n" starts a new line!
```

```
matrix =
 [[1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]]
```

An aside - in defining our one dimensional arrays above, e.g. using ***np.zeros(length)****, we have used a “shorthand”, which allows us to replace the tuple with a number. We could also use a tuple with only one element, as in the example below:*

[25]:
```
# <!-- Student -->
a1Darray = np.zeros((length))
print("a1Darray =",a1Darray)
```

```
a1Darray = [0. 0. 0.]
```

3.6.4 Week 2 exercise 4

Use the *.shape* method described here to check the dimensions of `matrix` are as you would expect.

We can access elements of this matrix using indices that label the row and column of the entry we want (again, both are counted from zero!). For example, we can set the values in our matrix:

[26]:
```
# <!-- Student -->
matrix[0, 0], matrix[0, 1], matrix[0, 2], matrix[0, 3] = 11, 12, 13, 14
matrix[1, 0], matrix[1, 1], matrix[1, 2], matrix[1, 3] = 21, 22, 23, 24
matrix[2, 0], matrix[2, 1], matrix[2, 2], matrix[2, 3] = 31, 32, 33, 34
print("matrix = \n",matrix)
```

```
matrix =
 [[11. 12. 13. 14.]
 [21. 22. 23. 24.]
 [31. 32. 33. 34.]]
```

And we can read them out:

[27]:
```
# <!-- Student -->
print("matrix[1, 3] =",matrix[1, 3])
```

```
matrix[1, 3] = 24.0
```

3.6.5 Slicing NumPy arrays

We have seen that we can access a single element of a 1D or 2D NumPy array using its index, We can also look at ranges of elements in an array. Here is a 1D example:

[28]:
```
# <!-- Student -->
countArr = np.linspace(0, 4, 5)
print("countArr =",countArr)
print("countArr[1:4] =",countArr[1:4])
```

```
countArr = [0. 1. 2. 3. 4.]
countArr[1:4] = [1. 2. 3.]
```

The syntax `countArr[iLow:iHigh]` returns the elements from `iLow` up to `iHigh`, including `iLow` but not `iHigh`.

The situation is similar for 2D (and higher dimensional) arrays.

[29]:
```
# <!-- Student -->
print("matrix \n",matrix)
print("matrix[1:3, 0:2] \n",matrix[1:3, 0:2])
```

```
matrix
 [[11. 12. 13. 14.]
 [21. 22. 23. 24.]
 [31. 32. 33. 34.]]
matrix[1:3, 0:2]
 [[21. 22.]
 [31. 32.]]
```

These "slices" of the arrays can be assigned new names. (Notice that Python doesn't need to be told the shape of the new array, it works it out from the slices we are requesting from the existing array.)

[30]:
```
# <!-- Student -->
square = matrix[1:3, 0:2]
print("square \n",square)
```

```
square
 [[21. 22.]
 [31. 32.]]
```

3.6.6 Week 2 exercise 5

Print out a 2×2 array consisting of the four elements in the lower right corner of the `matrix` array.

3.7 Reading data into a NumPy array

Supposing we have data that we have generated in an experiment and stored in a file. How can we read these into our Python programs and Jupyter Notebooks so we can do some analysis? There are many methods, and we will start with the simplest. This method can be used to read in files created in Excel, for example, as long as they are stored as *.csv* files, contain only numbers, all rows have the same number of elements and all columns have the same number of elements. (We will learn how to read in datasets containing strings etc. later in the module.)

The data set below (the file is called *normDistArr.csv*) consists of a column of 300 numbers from a gaussian (or normal) distribution, which is described by the function:

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left[-\frac{(x-\mu)^2}{2\sigma^2}\right].$$

Here, the mean $\mu = 6$ and the standard deviation $\sigma = 2$. (You can look at the file using MS Excel, or a text editor like notebook on windows computers or gedit on linux systems.) The file can be read into a NumPy array using the np.loadtxt function.

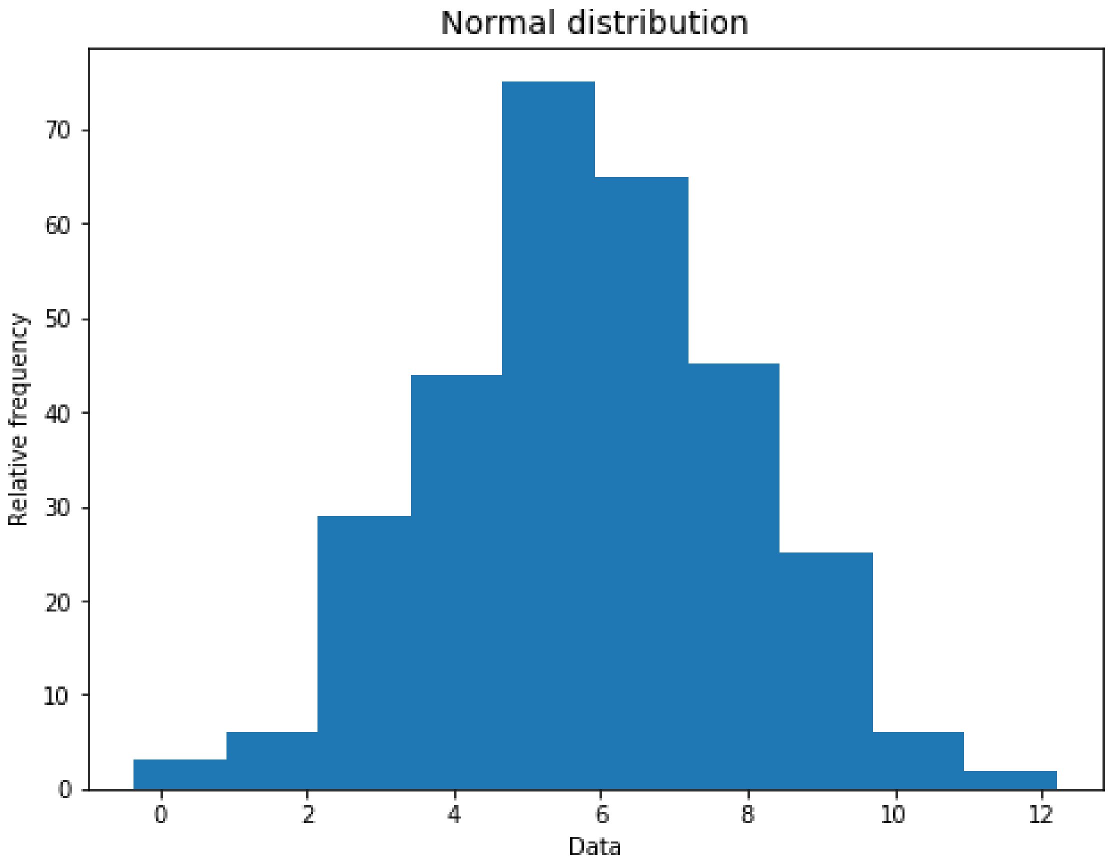
[31]:
```
# <!-- Student -->
gaussArr = np.loadtxt("normDistArr.csv")
print("gaussArr\n",gaussArr)
```

```
gaussArr
 [ 3.501   5.479   6.768   5.229   3.83   10.654   6.862   6.865   4.04    4.736
  7.155   5.75    7.958   9.19    3.596   3.247   8.109   5.922   7.361   8.658
  8.567   2.483   7.229   9.033   5.608   4.366   4.108   6.441   4.799   5.695
  3.625   6.598   4.104   2.313   7.621   4.495   5.127   6.095   5.498   6.334
  7.331   9.394   8.107   6.585   4.351   6.202   6.497   9.431   5.245   1.803
  4.794   4.962   1.575   5.188   5.407   6.543   5.082   6.928   3.945   6.345
  7.187   5.384   4.153   4.594   8.027   9.057   7.052   9.378   6.025   7.681
  5.818   8.064   6.968   5.22    5.796   9.695   7.349   3.214   4.842   5.503
  4.954   6.94    4.159   5.94    6.163   8.071   3.523   5.809   2.616   7.319
  6.732   6.121   7.118   2.56    6.821   4.766   9.242   6.503   9.556   9.467
  5.061   4.749   4.759   6.978   9.18    4.651   5.62    2.469   8.745   6.662
 12.211   4.865   5.729   4.675   6.623   5.843   8.017   5.903   5.347   4.104
  5.465   4.441   5.358   4.93    7.49    8.439   4.319   7.216   6.859   3.971
  3.705   9.736   8.673   2.48    4.591   6.99    5.839   3.619   6.939   5.181
  9.829   9.407   3.157   5.746   4.773   1.779   4.491   6.465   3.288   6.942
  6.025   5.832   9.285   3.981   4.45    7.08    5.502   7.005   4.307   6.729
  9.311   3.968   4.407   7.583   6.509   4.378   4.62    4.975   6.674   6.11
  2.828   4.471   3.714   4.883   3.972   8.306   7.667  10.861   7.494   8.885
  2.344   5.938   3.178  11.319   8.202   8.116   3.119   2.521   5.646   6.015
 -0.354   5.07    4.113   2.907   7.56    5.218   3.773   6.298   5.901   5.581
  5.314   7.076   2.205   8.397   5.726   4.866   7.333   4.92    7.663   5.871
  8.209   2.922   3.285   5.325   4.949   6.844   6.463   7.637   5.401   3.609
  6.564   9.051   4.395   8.109   6.643   5.543   9.829   7.736   5.825   9.994
  1.882   4.939   6.328   4.961   2.762   6.371   4.212   8.454   3.092   5.579
  4.276   8.263   6.508   7.398   3.576   6.471   3.742   5.07    8.728   0.788
  8.671   8.745   2.525   6.692   5.23    3.511   6.853   8.167   6.488   4.903
  7.451   8.079   5.308   5.76    6.334   7.615   2.862   3.362   3.064   6.268
  3.217   7.349   7.339   4.834   6.353   7.384   5.814   6.784   1.493   7.34
 -0.276   5.032   7.896   2.971   5.631   2.391   7.096   6.012   8.051   6.855
  6.131   6.341   1.957   9.503   4.893   7.172   3.262   5.604   4.512   7.244]
```

3.8 Creating a histogram

A histogram of the data stored in the `gaussArray` can be plotted as shown below:

[32]:
```python
# <!-- Student -->
import matplotlib.pyplot as plt
%matplotlib inline
#
plt.figure(figsize = (8, 6))
plt.title('Normal distribution', fontsize = 14)
plt.xlabel('Data')
plt.ylabel('Relative frequency')
plt.hist(gaussArr)
plt.show()
```

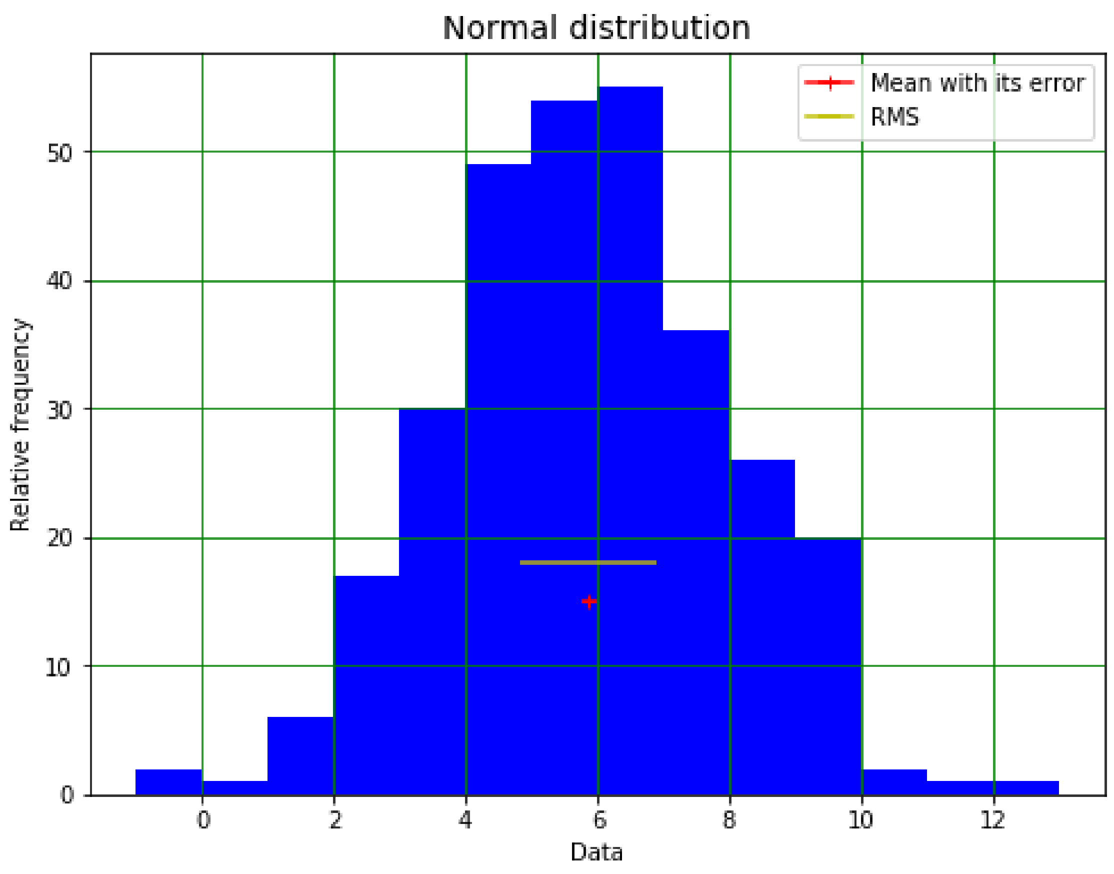


In a histogram, the data are split into "bins", the lower and upper limits of which are indicated by the edges of the bars in the plot. The area of each of the bars is proportional to the number of data points that fall within the bin. (Most histograms use bins of equal width, in which case the height of the bin indicates the number of entries it contains.) The above histogram shows that a gaussian distribution has a bell shape centred on its mean value. The width of the bell is given by the distribution's standard deviation (or root mean square, RMS).

Let's investigate the normal distribution in a little more detail. We'll plot a new histogram of the data, this time entering the bins we would like to use, rather than accepting those calculated by matplotlib, and making some other changes so the plot looks nicer. (A complete description of `plt.hist` is provided here.) We will also calculate the actual mean and standard deviation of the distribution, as well as the error on the mean, and add them to the plot.

[33]:
```
# <!-- Student -->
#
binBot = -1.0
binTop = 13.0
binNumber = 14
binEdges, binWidth = np.linspace(binBot, binTop, binNumber + 1, retstep = True)
print("Histogram bins start at",binBot,"finish at",binTop)
print("Number of bins is",binNumber,"and width of bins is",binWidth)
#
nEvents = len(gaussArr)
mu = np.mean(gaussArr) # calculate arithmetic mean of numbers in array
sigma = np.std(gaussArr) # calculate standard deviation (error on single value)
muError = sigma/np.sqrt(nEvents) # calculate error of mean
yMu = nEvents/20
ySigma = 1.2*nEvents/20
#
plt.figure(figsize = (8, 6))
plt.title('Normal distribution', fontsize = 14)
plt.xlabel('Data')
plt.ylabel('Relative frequency')
plt.hist(gaussArr, bins = binEdges, color = 'b')
plt.errorbar(mu, yMu, xerr = muError, marker = '+', color = 'r', label = 'Mean␣
 ↪with its error')
plt.errorbar(mu, ySigma, xerr = sigma/2, marker = '', color = 'y', label =␣
 ↪'RMS')
plt.grid(color = 'g')
plt.legend()
plt.show()
```

```
Histogram bins start at -1.0 finish at 13.0
Number of bins is 14 and width of bins is 1.0
```



3.8.1 Week 2 exercise 6

Add a line to the histogram above that shows the full width at half maximum (FWHM) of the distribution. You can calculate the FWHM using the formula:

$$\text{FWHM} = 2\sqrt{2\log 2}\sigma.$$

Hint. You can use code similar to that used to show the RMS!

4 Introduction to Computational Physics - Week 3

4.1 Table of contents week 3

4.2 Introduction to week 3

This week we will first see how we can write functions using Python. Then, we will look at how Python allows us to control the flow of programs and introduce boolean variables.

4.3 Functions

We saw last week that Python libraries like Numpy offer a large range of mathematical and other functions. Examples include `np.sin()`, `np.log()` and `np.linspace()`. If we give these the required arguments, they will return the value of the indicated mathematical function, define an array or carry out whatever job they are designed to do.

4.3.1 Week 3 exercise 1

Print out the sine of the angle 77°, the natural logarithm of 17.6 and use `np.linspace` to create a Numpy array that starts at 3, finishes at 7 and has 5 entries. Use the optional parameter in `np.linspace` that returns the step size and check that this is one!

Hint Don't forget to import the Numpy library first, and remember also that `np.sin` wants its argument to be in radians. Look up how to convert degrees to radians using Numpy!

Python also allows us to create our own functions. For example, supposing we need a function to calculate the area of a circle from its radius, we can do this as follows:

[34]:
```
# <!-- Student -->
import numpy as np
#
def circleArea(radius):
    '''
    Calculate the area of a circle given its radius
    '''
    area = np.pi*radius**2
    return area
#
r = 0.2
```

```
A = circleArea(r)
print("Radius of circle",r)
print("Area of circle",A)
```

```
Radius of circle 0.2
Area of circle 0.12566370614359174
```

The start of the definition of the function is indicated by the keyword **def** followed by the name we have chosen for the function, `circleArea`. The arguments (inputs) to the function are given in brackets and the first line closes with a colon (:). The subsequent lines of code that belong to the function are indented, and the end of the function is indicated by the word **return**, followed by the values the function returns (outputs) to the routine that called it (which is the code in the cell below the function definition in this case). The indentation stops at the end of the function.

An aside - Python doesn't specify how much the lines in a function have to be indented, but the amount of indentation has to be the same for all lines. Python also doesn't specify whether the indentation is done using tabs or spaces (both of which are invisible!), but the same method has to be used throughout. It is good practice to pick an amount of indentation and how you achieve it and stick to it. In these Notebooks, I have used four spaces, which is the most common convention.

The function is *called* by using its name and giving it the value of the input parameter it needs. In the case above, this is done as follows:

```
A = circleArea(r)
```

We see that we pass the function the value `r`, which is given the name `radius` in the function. After the function has finished its calculations, the area is returned and given the name `A`.

Note two things: 1. We have used the value of π given by NumPy, `np.pi`. This is more precise and less prone to error (typos) than entering `3.1415926...` every time π is needed. 2. The first few lines of the function consist of a comment describing what the function does, using three quotes to delimit the description. This kind of comment (appearing after a **def** line) is called a *docstring* and can be picked up by Python's documentation system and provided to users of the function who need an explanation of what the function does. More about docstrings can be found here. Including docstrings in your functions is a good habit to get into!

Functions can take more than one argument and can return more than one result. For example, here is how we could calculate the area of a rectangle and the length of its perimeter, given its width and length.

[35]:
```
# <!-- Student -->
def rectangleParams(width, length):
    '''
    Return area and perimeter of rectangle given its width and length
    '''
    area = width*length
    perimeter = 2*(width + length)
    return area, perimeter
#
w = 0.2
l = 0.5
```

```
print("Width of rectangle",w)
print("Length of rectangle",l)
A, peri = rectangleParams(w, l)
print("Area of rectangle",A)
print("Perimeter of rectangle",peri)
```

```
Width of rectangle 0.2
Length of rectangle 0.5
Area of rectangle 0.1
Perimeter of rectangle 1.4
```

As in the function `circleArea`, the names used for the width and length of the rectangle can be different inside and outside the function. (Inside, we have used `width` and `length`, outside, `w` and `l`.) The function decides which is width and which is length using the order in which they appear in the call to the function. If we put the arguments to a function in the wrong order, the results will (usually) be wrong. (Of course, in the case of `rectangleParams`, the order doesn't affect the calculation of the area and the perimeter!)

The values of the variables `width` and `length` can be changed inside the function without affecting the values of `w` and `l` outside it.

The outputs of the function can also be given different names inside and outside the function. Again, Python decides which is which using the order in which they are given.

4.3.2 Week 3 exercise 2

Write a function that takes the width, length and height of a rectangular prism and returns its volume, its surface area and the total length of all of its edges. Print out the results of a test case.

4.4 Python flow control

We have seen how we can do calculations in Python, how we can use library functions and how we can create our own functions. Now we look at how Python allows us to decide how we would like our program to "flow" from one calculation or statement to the next. This allows us to deal with situations where we want to do something different depending on whether a variable is positive or negative, for example. Several ways of steering programs are provided, including the `for`, `while` and `if` statements. We look at these in the following.

4.4.1 For loop

The `for` loop allows us to repeat sections of a program a specified number of times. The full syntax of the loop is shown below:

[36]:
```
# <!-- Student -->
start = 0
stop = 5
step = 2
#
for i in range(start, stop, step):
    print("Index",i)
print("Final value of index",i)
```

```
Index 0
Index 2
Index 4
Final value of index 4
```

The start of the loop is indicated by the `for` statement, which is followed by the name of the index (here we use `i`) which will steer how many times the loop is executed. The values `i` should take are determined by the statement `in range(start, stop, step)` and `i`, `start`, `stop` and `step` must all be integers. The `for` line is terminated using a colon (:).

When the program runs, the value of `i` is first set to `start`, then all the indented statements following the `for` statement are carried out. The value of `i` is then incremented by `step`, and if the resulting value is less than `stop`, the loop is executed again. This continues until adding `step` to `i` would give a value greater than or equal to `stop`.

If `step` is one (which it often is), you can use the simpler version of the `for` loop which omits the `step` parameter:

[37]:
```
# <!-- Student -->
start = 0
stop = 3
for n in range(start, stop):
    print("Index",n)
print("Final value of index",n)
```

```
Index 0
Index 1
Index 2
Final value of index 2
```

Again, the indentation indicates the body of the loop, i.e. the section of the program that is repeated. You can see that the loop doesn't run with `n = stop` and that the value of `n` on leaving the loop is the last "allowed" value.

4.4.2 Week 3 exercise 3

Write a program using a `for` loop that produces two columns, the left column being the integers 0 to 10 and the right column the words "zero" to "ten".

Hint Use a list containing the strings "zero", "one", "two" etc. and recall that you can access the elements of a list using the syntax `list[index]`!

4.4.3 Loops and slices

In Week 2, we introduced the idea of slicing an array. One of the examples we looked at was:

[38]:
```
# <!-- Student -->
import numpy as np
#
```

```
countArr = np.linspace(0, 10, 11)
print("countArr =",countArr)
print("countArr[3:6] =",countArr[3:6])
```

```
countArr = [ 0.  1.  2.  3.  4.  5.  6.  7.  8.  9. 10.]
countArr[3:6] = [3. 4. 5.]
```

This example is slightly modified below to illustrate how the parameters used in slicing are related to those in a `for` loop.

[39]:
```
# <!-- Student -->
countArr = np.linspace(0, 4, 5)
print("countArr =",countArr)
print("countArr[1:4:2] =",countArr[1:4:2])
```

```
countArr = [0. 1. 2. 3. 4.]
countArr[1:4:2] = [1. 3.]
```

4.4.4 Week 3 exercise 4

Explain the role of each of the three indices between the square brackets in the statement `countArr[1:4:2])`.

4.4.5 Alternative forms of the for loop

Loops can also be used to cycle through the elements in a list. An example is shown below:

[40]:
```
# <!-- Student -->
loopList = ["one", "two", "three"]
for var in loopList:
    print(var)
print("End of loop, var is",var)
```

```
one
two
three
End of loop, var is three
```

4.4.6 Week 3 exercise 5

Can you cycle through a tuple in the same was as through a list? Write a program similar to that for the list to test this!

4.4.7 While statement

The `while` statement offers another way of repeatedly using a section of code. An example follows:

[41]:
```
# <!-- Student -->
#
test = 0.3
limit = 1.1
```

```
step = 0.25
while test < limit:
    print("test =",test)
    test = test + step
print("Final value of test is",test)
```

```
test = 0.3
test = 0.55
test = 0.8
test = 1.05
Final value of test is 1.3
```

As in the case of the for loop, the line containing `while` finishes with a colon. The body of the loop, indicated by the indentation, is executed until the condition `test < limit` is false. (It won't execute at all if the condition is false the first time it is checked.) Something must be changed in the body of the loop to ensure that at some point `test < limit` becomes false, or the loop will run for ever. In the example above, the value of `test` increases by `step` each time the while loop runs, because we set `test = test + step`.

The condition that is tested has one of two values, `True` or `False`. If the statement `test < limit` is `True` execution continues, if it is `False`, it stops. As such logical conditions are used so frequently, Python has a data type, `bool` (short for *boolean*), which can take only the values `True` or `False`. A boolean variable (a variable of type `bool`) can be used explicitly in a `while` loop, as shown below:

[42]:
```
# <!-- Student -->
test = 0.3
limit = 1.1
step = 0.25
testVar = True
while testVar:
    print("testVar is",testVar)
    test = test + step
    testVar = test < limit
print("Final value of testVar is",testVar)
```

```
testVar is True
testVar is True
testVar is True
testVar is True
Final value of testVar is False
```

This doesn't make the program any clearer, so it isn't a sensible thing to do in this case, but it does illustrate how `bool` variables can be used!

`While` loops can be supplemented with an `else` statement, which is executed if the condition in the `while` statement is `False`.

[43]:
```
# <!-- Student -->
test = 0.3
limit = 1.1
```

```
step = 0.25
while test < limit:
    print("test is",test)
    test = test + step
else:
    print("Value of test in else section is",test)
print("Final value of test",test)
```

```
test is 0.3
test is 0.55
test is 0.8
test is 1.05
Value of test in else section is 1.3
Final value of test 1.3
```

This looks as though it is superfluous. Surely anything after the while statement will be executed after the tested condition becomes `False` even without an `else` statement? This is indeed the case, but, as we will see below, the `while`, `else` construct is useful in conjunction with other Python control structures, the `continue` and `break` statements.

An aside - If you end up with an endless loop while writing and testing a program, you can stop it by interrupting or restarting the kernel - the computing "engine" of your Notebook - using the appropriate menu commands.

4.4.8 If statement

The `if`, `elif`, `else` statement has the syntax illlustrated below:

[44]:
```
# <!-- Student -->
test = 2.3
if test < 1.0:
    print("This is section A")
elif test > 2.0 and test <= 3.0:
    print("This is section B")
elif test > 3.0 and test <= 4.0:
    print("This is section C")
else:
    print("This is section D")
print("This is the end of the if statement, the value of test is",test)
```

```
This is section B
This is the end of the if statement, the value of test is 2.3
```

Again, the lines starting with the `if`, `elif` (short for *else if*) and `else` statements must end with a colon. The code that is executed when the conditions tested in each of these lines are `True` is indented. Only the first of the sections of the `if`, `elif`, `else` block for which the condition is met is executed. *(Note, Python doesn't check the logic of your control statements, so it won't warn you if the tests in your `if` block don't make sense!)*

Statements can consist of just an `if`, an `if` and an `else`, or an `if` and one or more `elif`s, or, as above, of an `if`, one or more `elif`s and an `else`.

Python allows you to write the above `if`, `elif`, `else` statement in a more natural way (closer to standard mathematical notation) as follows:

[45]:
```python
# <!-- Student -->
test = 2.3
if test < 1.0:
    print("This is section A")
elif 2.0 < test <= 3.0:
    print("This is section B")
else:
    print("This is section C")
print("This is the end of the if statement, the value of test is",test)
```

```
This is section B
This is the end of the if statement, the value of test is 2.3
```

4.4.9 Continue and break

These statements allow you to modify the behaviour of a loop. In a `for` or a `while` loop, `continue` causes control to jump back to the beginning of the loop, without executing the statements after the `continue`. Two examples are shown below.

[46]:
```python
# <!-- Student -->
for letter in "Constantinople":
    if letter in "a, e, i, o, u":
        continue
    print(letter)
print("Final value of letter is",letter)
```

```
C
n
s
t
n
t
n
p
l
Final value of letter is e
```

Note that the string “a, e, i, o u” above could be replaced by “a e i o u” or “aeiou” and the routine would still work. It is checking whether `letter` is in the string enclosed in quotes and as `letter` is never “,” or ” ” (there are no commas or spaces in “Constantinople”), their presence in the string makes no difference!

Here’s another example:

[47]:
```python
# <!-- Student -->
for i in range(0, 6):
    print(i)
    if i > 2:
        continue
    print(10*i)
print("Final value of i is",i)
```

```
0
0
1
10
2
20
3
4
5
Final value of i is 5
```

In contrast, `break` causes control to jump to the end of the loop:

[48]:
```
# <!-- Student -->
for letter in "Constantinople":
    if letter in "a, e, i, o, u":
        break
    print(letter)
print("Final value of letter is",letter)
```

```
C
Final value of letter is o
```

[49]:
```
# <!-- Student -->
for i in range(0, 6):
    print(i)
    if i > 2:
        break
    print(10*i)
print("Final value of i is",i)
```

```
0
0
1
10
2
20
3
Final value of i is 3
```

How the `else` statement can be of use with `while` is apparent in the following examples.

[50]:
```
#<!-- Student -->
#
print(" ")
```

```python
print("While loop with continue statement")
test = 0.3
limit = 0.8
step = 0.2
while test < limit:
    print("test in first position is",test)
    test = test + step
    if test >= limit:
        continue
    print("test in second position is",test)
else:
    print("Value of test in else section is",test)
print("Final value of test",test)
#
print(" ")
print("While loop with break statement")
test = 0.3
while test < limit:
    print("test in first position is",test)
    test = test + step
    if test >= limit:
        break
    print("test in second position is",test)
else:
    print("Value of test in else section is",test)
print("Final value of test",test)
```

```
While loop with continue statement
test in first position is 0.3
test in second position is 0.5
test in first position is 0.5
test in second position is 0.7
test in first position is 0.7
Value of test in else section is 0.8999999999999999
Final value of test 0.8999999999999999

While loop with break statement
test in first position is 0.3
test in second position is 0.5
test in first position is 0.5
test in second position is 0.7
test in first position is 0.7
Final value of test 0.8999999999999999
```

We see that, if the `continue` statement is used, the code in the `else` statement is executed. If the `break` condition is applied, the code in the `else` statement is not run: there is a difference between the code that is executed if a `while` loop runs completely, or if it terminates due to a break statement.

4.4.10 Precision of floating point numbers

In the example above, we also see (at least on my computer…and this can be machine dependent!) that adding 0.2 (the value of step) to 0.7 (one of the values that test takes in the while loop) doesn't give 0.9, but 0.8999999999999999. This happens because computers use binary representations of numbers and for floats this results in limited precision. Similarly, addition, subtraction and other operations cause a loss of accuracy. If you want to read more, see this article.

A consequence of this is that you should never rely on a float having exactly a particular value. For example, the following code is not likely to give the result you want:

```
if test == 0.1397:
```

Instead, you should use something like:

```
if np.abs(test - 0.1397) < 1e-10:
```

You can test how precise the representation of floats is using the following code.

[51]:
```
#<!-- Student -->
#
eps = 1.0
while eps + 1.0 > 1.0:
    eps = eps/2
eps = 2*eps
print("The precision of your computer is", eps)
```

```
The precision of your computer is 2.220446049250313e-16
```

4.4.11 Week3 optional exercise

As an optional exercise, explain the functioning of the while loop that determines your computer's precision!

Python provides a function which returns the precision with which floats are represented in the sys (short for *system*) package:

[52]:
```
#<!-- Student -->
import sys
print("Precision of float is",sys.float_info.epsilon)
```

```
Precision of float is 2.220446049250313e-16
```

Numpy also has a function, described here, which provides information on the representation of floating point numbers:

[53]:
```
#<!-- Student -->
print("Precision of float is",np.finfo(float))
```

```
Precision of float is Machine parameters for float64
---------------------------------------------------------------
```

```
precision =  15   resolution = 1.0000000000000001e-15
machep =    -52   eps =        2.220446049250313e-16
negep =     -53   epsneg =     1.1102230246251565e-16
minexp =  -1022   tiny =       2.2250738585072014e-308
maxexp =   1024   max =        1.7976931348623157e+308
nexp =       11   min =        -max
---------------------------------------------------------------
```

4.4.12 Week 3 exercise 6

Write a program using a `while` loop to calculate and print out the factorial of the first ten integers. The factorial of a number is given by $n! = n(n-1)(n-2)...3 \times 2 \times 1$, for example, $6! = 6 \times 5 \times 4 \times 3 \times 2 \times 1 = 720$.

4.4.13 Week 3 exercise 7

Write a program using a `for` loop to calculate and print out the quantity $\sum_{n=1}^{N} n$ with N taking the values 1...15.

4.5 Booloean variables in Python

As there are many situations where expressions are either true or false, Python has `bool` variables which can ony take these two values. For example:

[54]:
```
# <!-- Student -->
boolVar1 = 3 > 2
print("boolVar1",boolVar1)
boolVar2 = 2 > 3
print("boolVar2",boolVar2)
```

```
boolVar1 True
boolVar2 False
```

Numbers can be compared in a variety of ways, producing the values `True` or `False`:

As mentioned above, there are some potential pitfalls here. While it makes sense to test if two integers are equal, this is very risky (that means don't do it!) for `floats`, as even real numbers that you know should be mathematically identical may be different on your computer because of errors in their calculation or representation. For `floats`, "equality" can be tested in the following way:

```
epsilon = 1E-10
boolVar = np.abs(floatX - floatY) < epsilon
```

Choose a sensible value of `epsilon`, dependent on the errors you expect in the calculation of `floatX` and `floatY`.

Boolean operators (`and`, `or`, `not`) can also be used to act on `bool` objects:

[55]:
```
# <!-- Student -->
print("True and True =",True and True)
print("not True =",not True)
print("False or True =",False or True)
```

```
True and True = True
not True = False
False or True = True
```

Another nice feature of Python is the ease with which strings can be manipulated. One example, looking at the letters in “Constantinople”, is shown above. Another is testing if a particular character or word is `in` (or `not in`) a string as below. (Notice that “Mary” and “mary” are not the same; in Python, size matters!)

[56]:
```
# <!-- Student -->
testString = "Mary had a little lamb"
print("'Mary' is in '",testString,"' is","Mary" in testString)
print("'mary' is in '",testString,"' is","mary" in testString)
print("'Wolf' is not in '",testString,"' is","Wolf" not in testString)
```

```
'Mary' is in ' Mary had a little lamb ' is True
'mary' is in ' Mary had a little lamb ' is False
'Wolf' is not in ' Mary had a little lamb ' is True
```

We will learn next week how to print the above without the superfluous spaces after and before the quotation marks in *’ Mary had a little lamb ’*!

4.5.1 Week 3 exercise 8

Modify the program you have written to calculate $\sum_{n=1}^{N} n$ in exercise 7 above to print out a second column containing the value `True` if the sum is even and `False` if it is odd.

[ ]:
```

```

5 Introduction to Computational Physics - Week 4

5.1 Table of contents week 4

5.2 Introduction to week 4

This week we will look at how output can be more attractively formatted using print statements. The exercises to do with this shouldn't take you too long, so you should have some time to complete any outstanding work from the previous three week's computer classes. Once you have finished these (make sure you get help if you need it!), you can try the *Fizz buzz* problem. This has often been used as part of the selection process for people applying for computing jobs!

5.3 Pretty printing

The outputs we have produced using print statements in the work we have done so far has not been attractively formatted. It would be nice, for example, to make sure that floats were displayed with an appropriate number of significant figures and to be able to line things in tables. We will now look at two ways of doing this. The first is the `format` statement. The second, `f-strings`, was first introduced in Python 3.6, so is relatively new.

5.4 Using format statements

An example illustrating printing using the `format` syntax is shown below, together with the equivalent print statement in the form we have used so far:

[57]:
```
# <!-- Student -->
big = 999.999999
small = 666.666666E-19
number = 33
string = 'letters'
print("Big is",big,"\b, small is",small,"\b, number is",number,"and string␣
 ↪is",string,"\b!")
print("Big is {:f}, small is {:e}, number is {:d} and string is {:s}!"
      .format(big, small, number, string))
```

```
Big is 999.999999, small is 6.66666666e-17, number is 33 and string is letters!
Big is 999.999999, small is 6.666667e-17, number is 33 and string is letters!
```

The format statement allows us to include a "marker" (a *format field*) in the string we want to write which indicates that an `int`, a `float` or a `str` should be included at that point; the varaiables we want to include are put into the `format` statement at the end of the string. The format field is denoted by curly brackets, with the symbols inside the brackets indicating the format of the variable that should be substituted at that point (e.g. `f` for float, `e` for scientific notation, `d` for integer and `s` for string).

Notice that Python will understand we have continued a statement on a new line if we have opened a bracket on one line `print(` above, and closed it on another, `string))` above. If you want to continue a statement on a second line in a situtaiton where Python won't be able to recognise what you are doing, or just because you want to make it clear, you can use a backslash () at the end of the line, as here:

```
print("Big is {:f}, small is {:e}, number is {:d} and string is {:s}!"\
      .format(big, small, number, string))
```

Using `format` above has helped improve the print statement output in some respects (we don't need to use the `\b` backspace character to ensure the comma and fullstop are in the right place), but the code is not as easy to read or write as our "standard" print statement.

A useful feature of the `format` printing methid is that we can steer the number of figures we want after the decimal point (for floats or exponentionals) as follows:

[58]:
```
# <!-- Student -->
print("Big is {:.2f}, small is {:.3e} and number is {:d}.".format(big, small,␣
↪number))
```

```
Big is 1000.00, small is 6.667e-17 and number is 33.
```

Note that numbers are rounded sensibly. (You can't specify the number of figures after the decimal point for integers of course!)

We can also control the length of the number that is printed out (including the spaces in front of the number) by adding an integer in front of the decimal point in the format specifiers:

[59]:
```
# <!-- Student -->
print("Big is {:12.2f}, small is {:12.3e} and number is {:12d}.".format(big,␣
↪small, number))
```

```
Big is      1000.00, small is    6.667e-17 and number is           33.
```

By default, the numbers printed are aligned to the right, but this can be changed using the characters `<`, `>` and `^` for left, right and central alignment, respectively, as illustrated below:

[60]:
```
# <!-- Student -->
print("Big is {:<12.2f}, small is {:^12.3e} and number is {:<12d}.".format(big,␣
↪small, number))
```

```
Big is 1000.00     , small is  6.667e-17   and number is 33          .
```

5.4.1 Week 4 exercise 1

Use the `format` syntax to print out "The number of eggs in a dozen (12) is more than ten.", but using a variable `dozen = 12` for the number in the brackets.

The following example illustrates how `format` can be used to align columns of numbers:

[61]:
```
# <!-- Student -->
nMax = 10
nMin = 0
print("Number Square   Cube")
for n in range(nMin, nMax):
    print("{:6d} {:6d} {:6d}".format(n, n**2, n**3))
```

```
Number Square   Cube
     0      0      0
     1      1      1
     2      4      8
     3      9     27
     4     16     64
```

```
5       25      125
6       36      216
7       49      343
8       64      512
9       81      729
```

Using "tab" characters (`\t`) can also help with formatting (though you might have to do some tweaking to get things to line up as you want). A tab is eight characters long. Here is an example:

[62]:
```python
# <!-- Student -->
print("Months in the year:")
print("1 \t 2 \t 3 \t 4 \t 5 \t 6 \t 7 \t 8 \t 9 \t 10 \t 11 \t 12")
print("Jan \t Feb \t Mar \t Apr \t May \t Jun \t Jul \t Aug \t Sept \t Oct \t␣
 ↪Nov \t Dec")
```

```
Months in the year:
1        2       3       4       5       6       7       8       9       10
11       12
Jan      Feb     Mar     Apr     May     Jun     Jul     Aug     Sept    Oct
Nov      Dec
```

Note, there are lots more tools for writing things prettily using `format`, see here, for example.

5.4.2 Week 4 exercise 2

Use a `format` statement to write a row containing the text strings "One", "two", "three"…"seven". Below it it, write a row listing the days of the week. Use tabs to line up the two rows so "One" is above "Monday", "two" above "Tuesday" etc.

5.5 Using f-strings

The `f-string` (or, to give it its full name, formatted string literal) syntax aims to provide the power of the format statement in a more programmer friendly way. Here is an example:

[63]:
```python
#<!-- Student -->
name = 'John'
year = 2016
print(f"Hello {name}, did you know f-strings were introduced in {year}?")
```

```
Hello John, did you know f-strings were introduced in 2016?
```

The `f` can also be an `F`!

[64]:
```python
#<!-- Student -->
name = 'Fiona'
print(F"No {name}, I didn't!")
```

```
No Fiona, I didn't!
```

The braces can contain expressions that are calculated at run time (i.e. when the print statement is executed), for example:

[65]:
```
#<!-- Student -->
x = 2
y = 7
print(f"The product of {x} and {y} is {x*y}. The larger of {x} and {y} is␣
 ↪{max(x, y)}.")
```

```
The product of 2 and 7 is 14. The larger of 2 and 7 is 7.
```

The length and precision of printed numbers can also be steered using a syntax similar to that in the `format` statement:

[66]:
```
#<!-- Student -->
big = 999.999999
small = 666.666666E-19
number = 33
string = 'letters'
print(f"This number is big {big:.2f}, this one is small {small:.2e}, \
      this one is an integer {number:d} and this is a string: '{string}'.")
```

```
This number is big 1000.00, this one is small 6.67e-17,       this one is an
integer 33 and this is a string: 'letters'.
```

Padding with spaces and aligning also work as before:

[67]:
```
#<!-- Student -->
big = 999.999999
small = 666.666666E-19
number = 33
print(f"This number is big {big:<10.2f}, this one is small {small:^12.2e} and␣
 ↪this one is an integer {number:<d}.")
```

```
This number is big 1000.00   , this one is small   6.67e-17   and this one is an
integer 33.
```

This means that columns of numbers can be aligned using `f-strings`.

5.5.1 Week 4 exercise 3

Reproduce the aligned table above with columns Number, Square and Cube for integers from 0 to 9 inclusive using f-strings.

The good news is that `f-strings` are also the fastest (use the least computing time) of the options for printing that we have looked at and are also faster that the "c-style" option we haven't used (because it's horrid): what's not to like! (Note, even though `f=strings` are the best way of printing prettily, you have to understand the old `format` syntax as most programs - and most of the Notebooks in this module - will have been written using it.)

5.6 The Fizz buzz problem

Write a program that prints the numbers from 1 to 20. But for multiples of three print *Fizz* instead of the number and for multiples of five print *Buzz*. For numbers which are multiples of both three and five print *Fizz buzz*.

No hints or anything to start you off this week, just have a go! When you have a working program, try googling *Fizz buzz Python* to have a look at how other people have solved this problem. There are lots of possible solutions!

And by the way, congratulations are in order. It's only your fourth week and you are tackling a problem that crops up in interviews for professional programmers!

6 Introduction to Computational Physics - Week 5

6.1 Table of contents week 5

6.2 Introduction to week 5

This week we shall use Euler's method and some of the programming techniques we have developed to look at a physics problem that cannot be solved analytically: projectile motion with air resistance. We shall then look at some more plotting routines.

6.3 Projectile motion with no air resistance

We know we can work out the path of a projectile algebraically in the absence of air resistance. We use Newton's Second Law:

$$\begin{aligned}\vec{F} &= m\vec{a} \\ &= m\frac{d}{dt}\vec{v} \\ &= m\frac{d^2}{dt^2}\vec{r},\end{aligned}$$

where, $\vec{F}$ is the force, m the mass, $\vec{v}$ velocity, $\vec{r}$ the position and t the time. Separating the motion into its horizontal (x) and vertical (y) components, for the former we have:

$$\begin{aligned}
\frac{d^2x}{dt^2} &= 0,\\
\Rightarrow \frac{dx}{dt} &= u_x,\\
\Rightarrow x &= u_x t + x_0.
\end{aligned}$$

Here, u_x is the inital velocity and x_0 the initial position of the projectile in the x direction. Taking the origin of the x axis to be at the position from which the projectile is launched at $t = 0$ gives $x_0 = 0$.

In the vertical direction, setting the vertical force to be mg, with $g = -9.81\,\mathrm{ms}^{-2}$, we have:

$$\begin{aligned}
m\frac{d^2y}{dt^2} &= mg,\\
\Rightarrow \frac{d^2y}{dt^2} &= g,\\
\Rightarrow \frac{dy}{dt} &= gt + u_y,\\
\Rightarrow y &= \frac{1}{2}gt^2 + u_y t + y_0,
\end{aligned}$$

where u_y is the inital vertical velocity and we have assumed the height from which the projectile is launched is $y = y_0$.

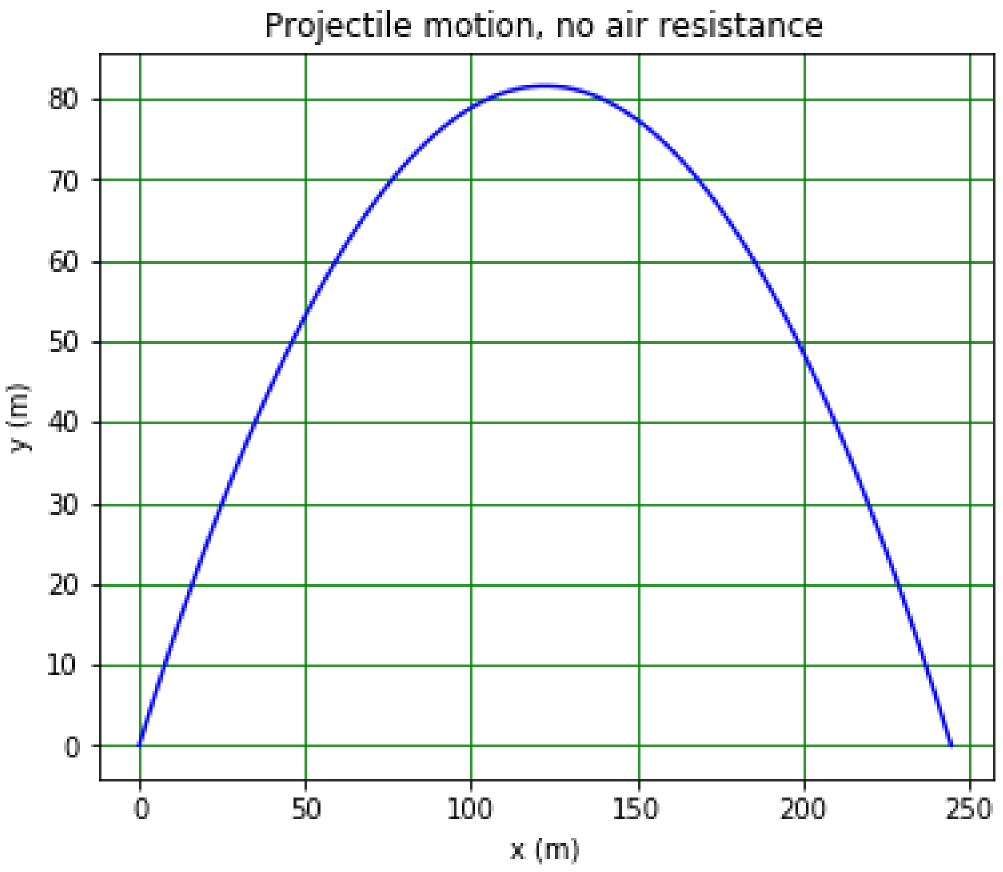
Setting $y_0 = 0$ for simplicity, the times at which the projectile is at ground level can be found from:

$$\begin{aligned}
y &= 0\\
\Rightarrow \frac{1}{2}gt^2 + u_y t &= 0\\
\Rightarrow t\left(\frac{gt}{2} + u_y\right) &= 0\\
\Rightarrow t &= 0 \text{ or } -\frac{2u_y}{g}.
\end{aligned}$$

Taking $u_x = 30\,\mathrm{ms}^{-1}$ and $u_y = 40\,\mathrm{ms}^{-1}$, the trajectory of the projectile can be plotted using the equations for the vertical and horizontal motion in terms of the time.

[68]:
```
# <!-- Student -->
#
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
#
ux = 30 # m/s
uy = 40 # m/s
```

```
g = -9.81 # m/s**2
tMax = -2*uy/g # s
nSteps = 100
tArr = np.linspace(0.0, tMax, nSteps)
#
xArr = ux*tArr
yArr = 0.5*g*tArr**2 + uy*tArr
#
plt.figure(figsize = (6, 5))
plt.title("Projectile motion, no air resistance")
plt.xlabel("x (m)")
plt.ylabel("y (m)")
plt.plot(xArr, yArr, linestyle = '-', color = 'b')
plt.grid(color = 'g')
plt.show()
```



If we want to express y in terms of x, we must eliminate t from the equations for the horizontal and vertical motion. From the expression for the horizontal motion, we see that:

$$t = \frac{x}{u_x}.$$

Substituting this in the equation for the vertical motion gives the equation of a parabola:

$$y = \frac{1}{2}\frac{g}{u_x^2}x^2 + \frac{u_y}{u_x}x + y_0.$$

To find where the projectile hits the ground, we must solve this equation for $y = 0$, that is:

$$\frac{1}{2}\frac{g}{u_x^2}x^2 + \frac{u_y}{u_x}x + y_0 = 0.$$

The roots are:

$$\begin{aligned} x &= \frac{-\frac{u_y}{u_x} \pm \sqrt{\frac{u_y^2}{u_x^2} - 4\frac{1}{2}\frac{g}{u_x^2}y_0}}{2\frac{1}{2}\frac{g}{u_x^2}} \\ &= -\frac{u_x u_y}{g} \pm \frac{u_x}{g}\sqrt{u_y^2 - 2gy_0}. \end{aligned}$$

Setting $y_0 = 0$ gives:

$$\begin{aligned} x &= -\frac{u_x u_y}{g} \pm \frac{u_x u_y}{g} \\ &= 0 \quad \text{or} \quad -2\frac{u_x u_y}{g}. \end{aligned}$$

6.3.1 Week 5 exercise 1

Create a cell below this one and write in it a routine to create a figure which shows (on one plot):

1) The trajectory using the expressions involving time (as is already done in the cell above).

2) The trajectory using the expression involving y as a functon of x which demonstrates that the path is a parabola.

Do you get the same trajectory in both cases?

6.4 Projectile motion with air resistance

What happens of we include the effect of air resistance? In addition to the gravitational force, the projectile then experiences a drag force which acts in the opposite direction to its velocity. The magnitude of the force is given by:

$$D = \frac{1}{2}C_D\rho_{\text{air}}Av^2,$$

where C_D is the drag coefficient (which is dependent on the projectile's shape), ρ_{air} is the density of air and A is the area of the projectile in the plane normal to its velocity. The expression for the horizontal acceleration of the projectile is modified as follows:

$$\begin{aligned} m\frac{d^2x}{dt^2} &= -D\cos\theta \\ \Rightarrow \frac{d^2x}{dt^2} &= -\frac{D}{m}\cos\theta. \end{aligned}$$

Notice that we cannot write down a simple expression for $v_x = \frac{dx}{dt}$ in terms of t: there is no algebraic form for the integral as θ, the angle the velocity makes to the horizontal, is a function of t.

The expression for the vertical acceleration becomes:

$$\begin{aligned} m\frac{d^2y}{dt^2} &= -D\sin\theta + mg \\ \Rightarrow \frac{d^2y}{dt^2} &= -\frac{D}{m}\sin\theta + g. \end{aligned}$$

Again, we cannot write down an algebraic expression for $v_y = \frac{dy}{dt}$.

In order to plot the trajectory with air resistance, we need to resort to numerical methods. The simplest technique is to break the trajectory down into a set of small steps, each of which takes only a small time δt. Given the force, acceleration, velocity and position at the start of each step, we can calculate these quantities at the end of the step. Repeating this process many times allows us to map out the trajectory. Horizontally, for the i^{th} step, the change in velocity is given by:

$$\begin{aligned} \frac{d^2x_i}{dt^2} = \frac{dv_{x\,i}}{dt} &= -\frac{D_i}{m}\cos\theta_i \\ \Rightarrow \delta v_{x\,i} &= -\frac{D_i}{m}\cos\theta_i\,\delta t. \end{aligned}$$

The change in the vertical velocity is:

$$\begin{aligned} \frac{d^2y_i}{dt^2} = \frac{dv_{y\,i}}{dt} &= -\frac{D_i}{m}\sin\theta_i + g \\ \Rightarrow \delta v_{y\,i} &= -\frac{D_i}{m}\sin\theta_i\,\delta t + g\,\delta t. \end{aligned}$$

Hence, at the end of the step, the components of the velocity are:

$$\begin{aligned} v_{x\,i+1} &= v_{x\,i} + \delta v_{x\,i} \\ v_{y\,i+1} &= v_{y\,i} + \delta v_{y\,i}. \end{aligned}$$

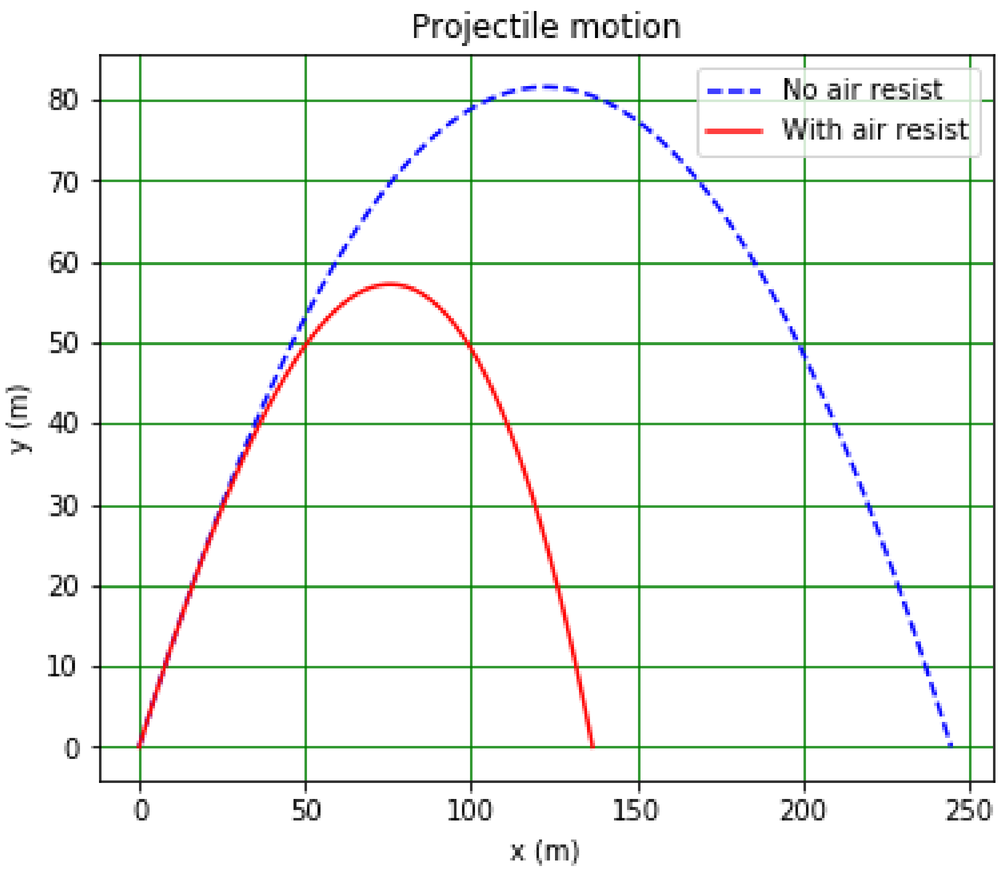
The position at the end of the step is given by:

$$x_{i+1} = x_i + v_{x\,i}\delta t$$
$$y_{i+1} = y_i + v_{y\,i}\delta t.$$

We now use this approach, referred to as the Euler method, to plot the trajectory of the projectile.

[69]:
```python
# <!-- Student -->
#
def drag(cd, area, rho, velx, vely):
    '''
    Return horizontal and vertical drag force on body given its drag␣
 ↪coefficient, area,
    density of medium in which it's moving and horizontal and vertical velocity.
    '''
    v2 = velx**2 + vely**2
    sinTheta = vely/np.sqrt(v2)
    cosTheta = velx/np.sqrt(v2)
    Dx = -0.5*cd*rho*area*v2*cosTheta
    Dy = -0.5*cd*rho*area*v2*sinTheta
    return Dx, Dy
#
nEuler = 10000
dt = tMax/nEuler
xEuler = np.zeros(nEuler)
yEuler = np.zeros(nEuler)
xE = 0.0 # m
yE = 0.0 # m
vX = ux # m/s
vY = uy # m/s
CD = 0.47
rad = 0.025 # m
Area = np.pi*rad**2 # m**2
rhoAir = 1.2 # kg/m**3
rhoProj = 2000.0 # kg/m**3
mProj = 4/3*np.pi*rad**3*rhoProj # kg
print(" ")
print("Radius {:5.3f} m, mass {:5.3f} kg.".format(rad, mProj))
print("Initial position ({:5.2f}, {:5.2f}) m.".format(xE, yE))
print("Initial velocity ({:5.2f}, {:5.2f}) m/s.".format(vX, vY))
print("Maximum time {:5.3f} s, time step {:5.3e} s.".format(tMax, dt))
#
iStep = 0
while (yE > 0 or iStep == 0) and iStep < nEuler:
    xEuler[iStep] = xE
    yEuler[iStep] = yE
    xE = xE + vX*dt
    yE = yE + vY*dt
    dragX, dragY = drag(CD, Area, rhoAir, vX, vY)
    vX = vX + dragX/mProj*dt
    vY = vY + dragY/mProj*dt + g*dt
    iStep = iStep + 1
#
print("Final step is number {:d}, actual flight time {:5.3f} s.".format(iStep,␣
 ↪iStep*dt))
plt.figure(figsize = (6, 5))
plt.title("Projectile motion")
plt.xlabel("x (m)")
plt.ylabel("y (m)")
plt.plot(xArr, yArr, linestyle = '--', color = 'b', label = 'No air resist')
plt.plot(xEuler[0:iStep], yEuler[0:iStep], linestyle = '-', color = 'r', label␣
 ↪= 'With air resist')
plt.grid(color = 'g')
plt.legend()
plt.show()
```

```
Radius 0.025 m, mass 0.131 kg.
Initial position ( 0.00,  0.00) m.
Initial velocity (30.00, 40.00) m/s.
Maximum time 8.155 s, time step 8.155e-04 s.
Final step is number 8347, actual flight time 6.807 s.
```



6.4.1 Week 5 exercise 2

How could you test that the Euler method is functioning correctly? Make a plot demonstrating your test. Use it to determine the approximate minimum number of steps needed to get a reliable solution for this problem.

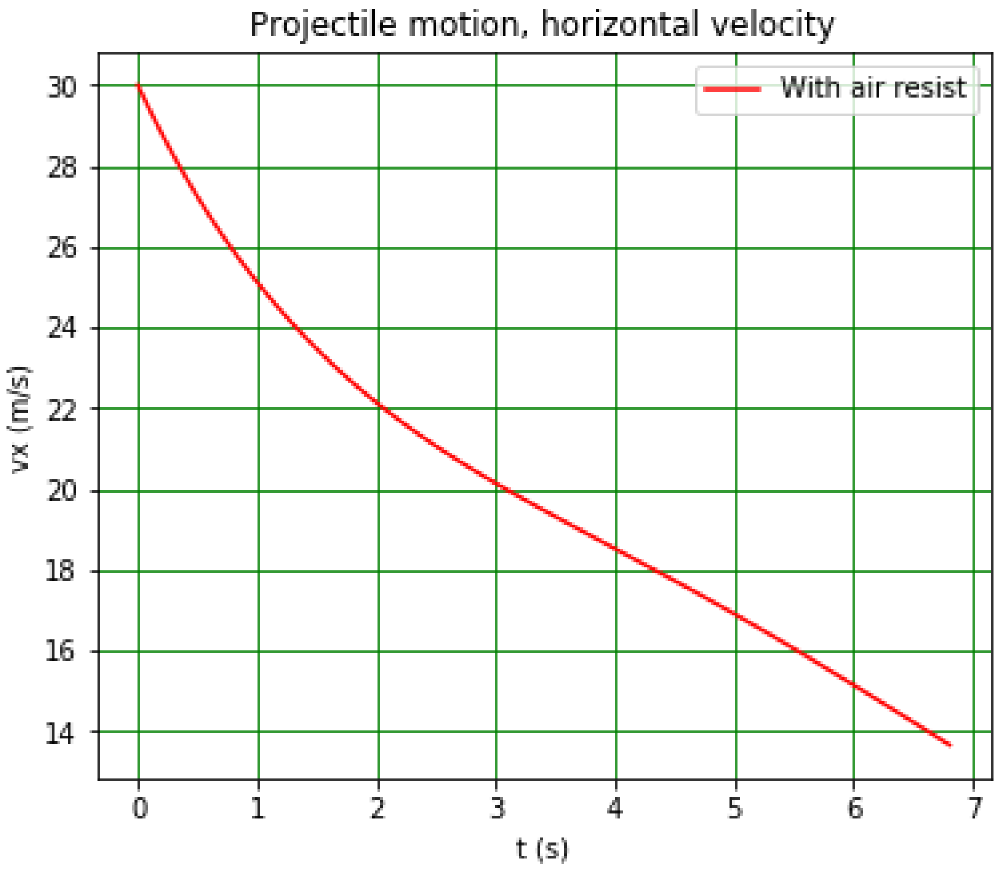
6.5 Projectile motion: behaviour of velocity with time

It is interesting to investigate the behaviour of the x and y components of the velocity with time. The following code allows this to be done for the horizontal component of the velocity.

[70]:
```
# <!-- Student -->
#
nEuler = 10000
dt = tMax/nEuler
xEuler = np.zeros(nEuler)
yEuler = np.zeros(nEuler)
vXEuler = np.zeros(nEuler)
xE = 0.0
yE = 0.0
```

```
vX = ux
vY = uy
CD = 0.47
rad = 0.025
Area = np.pi*rad**2
rhoAir = 1.2
rhoProj = 2000.0
mProj = 4/3*np.pi*rad**3*rhoProj
print(" ")
print("Radius {:5.3f} m, mass {:5.3f} kg.".format(rad, mProj))
print("Initial position ({:5.2f}, {:5.2f}) m.".format(xE, yE))
print("Initial velocity ({:5.2f}, {:5.2f}) m/s.".format(vX, vY))
print("Maximum time {:5.3f} s, time step {:5.3e} s.".format(tMax, dt))
#
iStep = 0
while (yE > 0 or iStep == 0) and iStep < nEuler:
    xEuler[iStep] = xE
    yEuler[iStep] = yE
    vXEuler[iStep] = vX
    xE = xE + vX*dt
    yE = yE + vY*dt
    dragX, dragY = drag(CD, Area, rhoAir, vX, vY)
    vX = vX + dragX/mProj*dt
    vY = vY + dragY/mProj*dt + g*dt
    iStep = iStep + 1
#
print("Final step is number {:d}, actual flight time {:5.3f} s.".format(iStep,␣
 ↪iStep*dt))
tEuler = np.linspace(0.0, tMax, nEuler)
plt.figure(figsize = (6, 5))
plt.title("Projectile motion, horizontal velocity")
plt.xlabel("t (s)")
plt.ylabel("vx (m/s)")
plt.plot(tEuler[0:iStep], vXEuler[0:iStep], linestyle = '-', color = 'r', label␣
 ↪= 'With air resist')
plt.grid(color = 'g')
plt.legend()
plt.show()
```

```
Radius 0.025 m, mass 0.131 kg.
Initial position ( 0.00,  0.00) m.
Initial velocity (30.00, 40.00) m/s.
Maximum time 8.155 s, time step 8.155e-04 s.
Final step is number 8347, actual flight time 6.807 s.
```



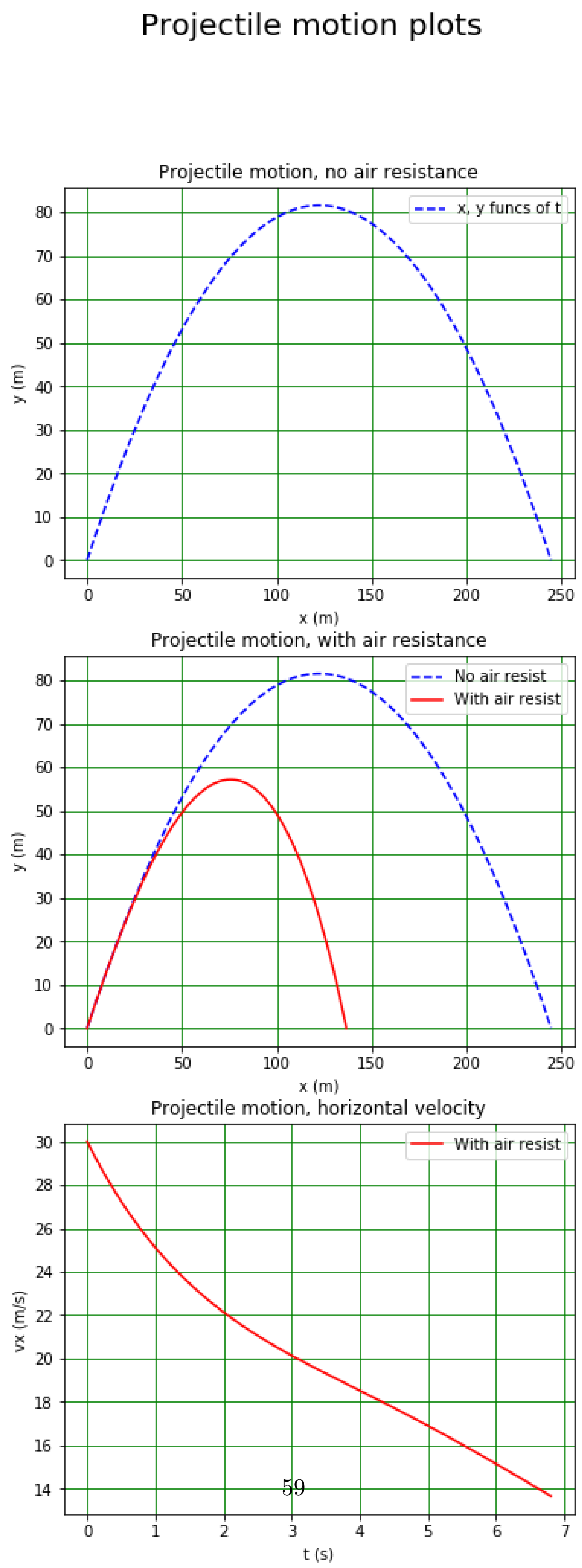
6.5.1 Week 5 exercise 3

Copy the above code into a new cell below this one. Modify the code so that the graph shows how both the horizontal and vertical components of the velocity vary as a function of time.

6.6 Putting multiple plots in one figure

Sometimes it is useful to be able to incorporate more than one plot in a figure. This can be done using `matplotlib.pyplot` as is shown below.

```
[71]: # <!-- Student -->
#
fig = plt.figure(figsize = (6, 16)) # opens a figure
fig.suptitle('Projectile motion plots', fontsize=20) # overall title
#
plt.subplot(3, 1, 1) # creates a 3 row, 1 column grid and starts in the first␣
 ↪(top left) square
plt.title("Projectile motion, no air resistance")
plt.xlabel("x (m)")
plt.ylabel("y (m)")
plt.plot(xArr, yArr, linestyle = '--', color = 'b', label = 'x, y funcs of t')
plt.legend()
plt.grid(color = 'g')
#
plt.subplot(3, 1, 2) # plot in second square (reading from left to right, top␣
 ↪to bottom)
plt.title("Projectile motion, with air resistance")
plt.xlabel("x (m)")
plt.ylabel("y (m)")
plt.plot(xArr, yArr, linestyle = '--', color = 'b', label = 'No air resist')
plt.plot(xEuler[0:iStep], yEuler[0:iStep], linestyle = '-', color = 'r', label␣
 ↪= 'With air resist')
plt.grid(color = 'g')
plt.legend()
#
plt.subplot(3, 1, 3) # plot in third square
plt.title("Projectile motion, horizontal velocity")
plt.xlabel("t (s)")
plt.ylabel("vx (m/s)")
plt.plot(tEuler[0:iStep], vXEuler[0:iStep], linestyle = '-', color = 'r', label␣
 ↪= 'With air resist')
plt.grid(color = 'g')
plt.legend()
#
plt.show()
```


Projectile motion plots
Projectile motion, no air resistance
x, y funcs of t
y (m)
x (m)
Projectile motion, with air resistance
No air resist
With air resist
y (m)
x (m)
Projectile motion, horizontal velocity
With air resist
vx (m/s)
t (s)

The figure is created as before, using `fig = plt.figure(figsize = (6, 16))`. You will have to adjust the size parameters so there is space for all your subplots!

The command `fig.suptitle('Projectile motion plots', fontsize=20)` produces an overall title for the figure.

Each supbplot is created using `plt.subplot(nRows, nCols, nThisPlot)`. Here, the values `nRows` and `nCols` tell Python to draw an array of subplots with (you've guessed it) `nRows` rows and `nCols` columns. Note that almost everywhere, rows and columns are given in that order! The index `nThisPlot` indicates which position the current plot is to fill. The first (with `nThisPlot = 1` is the top left slot, the second (`nThisPlot = 2`) is next slot reading from left-to-right and top-to-bottom, and so on. The commands that we have used for single plots can be used for subplots (titles, axis labels etc.).

The figure is closed by the `fig.show()` command.

6.6.1 Week 5 exercise 4

Copy the code above to a new cell below this one and change the code so the figure contain two rows and two columns. Add a fourth subplot which shows the behaviour of the vertical velocity with time.

6.7 Comments on week 5

This week, we have solved a problem that cannot be tackled algebraically. Many situations in Physics require this kind of numerical approach. The Euler method we have used is a powerful technique in that the way it works is obvious (though actually getting a program using the method to do what you want it to may not be easy!) and is applicable to many equations. The downside of the Euler method is that it requires only works if the step size is small enough, so you must think about how you can check that step size you are using is adequate!

7 Introduction to Computational Physics - Week 6

7.1 Table of contents week 6

7.2 Introduction to week 6

This week, we shall have a first look at how we can generate random numbers in Python programs and how these can be used to create Monte Carlo models and solve physical and mathematical problems.

7.3 Generating random numbers using numpy

In the module `numpy.random`, Python provides a range of tools for generating random numbers. One of the most basic is a way of generating pseudo-random numbers in the x interval $0 \leq x < 1$, `numpy.random.rand`, is described here. We will look at this routine first, as it is a good way of seeing some of the properties of random number generators. It can be used as follows:

[72]:
```python
# <!-- Student -->
import numpy as np
#
# generate one random number
x = np.random.rand()
print("x = ",x)
#
# generate an array of 4 random numbers
x1D = np.random.rand(4)
print(" ")
print("x1D = \n",x1D)
#
# generate a 2D array containing 3 rows and 2 columns of random numbers
x2D = np.random.rand(3, 2)
print(" ")
print("x2D = \n",x2D)
```

```
x =  0.20511122406289795

x1D =
 [0.794 0.879 0.605 0.442]

x2D =
 [[0.266 0.955]
 [0.539 0.534]
 [0.324 0.999]]
```

The routine `np.random.rand` uses the Mersenne Twister algorithm, which (in the NumPy implementation) produces 53-bit precision `floats` and has a period of $2^{19937} - 1$. The algorithm is started with a *seed*, which you can specify as below. The seed must be an integer between 0 and $2^{32} - 1$.

7.3.1 Week 6 exercise 1

Check how the seed works by running the cell below several times for one value of the seed. Note the generated random numbers you get, then change the seed and try again. What do you see?

```
[73]: # <!-- Student -->
#
np.random.seed(1327)
#
# generate one random number
x = np.random.rand()
print("x = ",x)
#
# generate an array of 3 random numbers
x1D = np.random.rand(3)
print(" ")
print("x1D = \n",x1D)
```

```
x =  0.5718839411688054

x1D =
 [0.609 0.11  0.287]
```

If you don't provide a seed, `np.random.rand` will use something like the time of day as a seed, so you will get a different sequence of numbers every time you use it.

Now let's check that the distribution of numbers we get from `np.random.rand` is what we would expect.

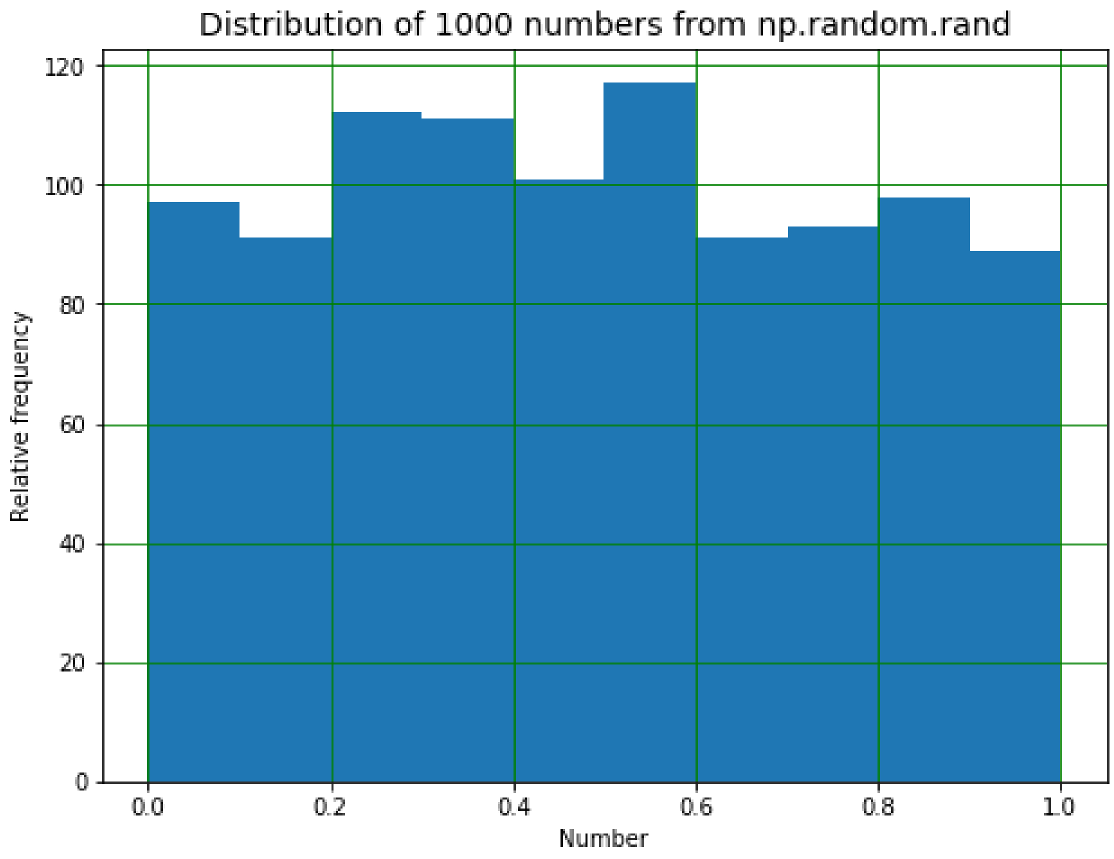
7.3.2 Week 6 exercise 2

Use the code below to display the distribution of 1000 random numbers in the interval from 0 to 1 in a histogram with 10 uniform bins. Then modify the code to add a second histogram to display the distribution resulting from ten million random numbers over the same interval (using the same bin sizes). Explain the differences you observe in the two histograms.

```
[74]: # <!-- Student -->
#
import matplotlib.pyplot as plt
%matplotlib inline
#
binBot = 0.0
binTop = 1.0
binNumber = 10
binEdges = np.linspace(binBot, binTop, binNumber + 1)
binWidth = (binTop - binBot)/binNumber
print("Histogram bins start at",binBot,"finish at",binTop)
print("Number of bins is",binNumber,"and width of bins is",binWidth)
#
randArr = np.random.rand(1000)
```

```
#
plt.figure(figsize = (8, 6))
plt.title('Distribution of 1000 numbers from np.random.rand', fontsize = 14)
plt.xlabel('Number')
plt.ylabel('Relative frequency')
plt.hist(randArr, bins = binEdges)
plt.grid(color = 'g')
plt.show()
```

```
Histogram bins start at 0.0 finish at 1.0
Number of bins is 10 and width of bins is 0.1
```



7.4 More random number generators

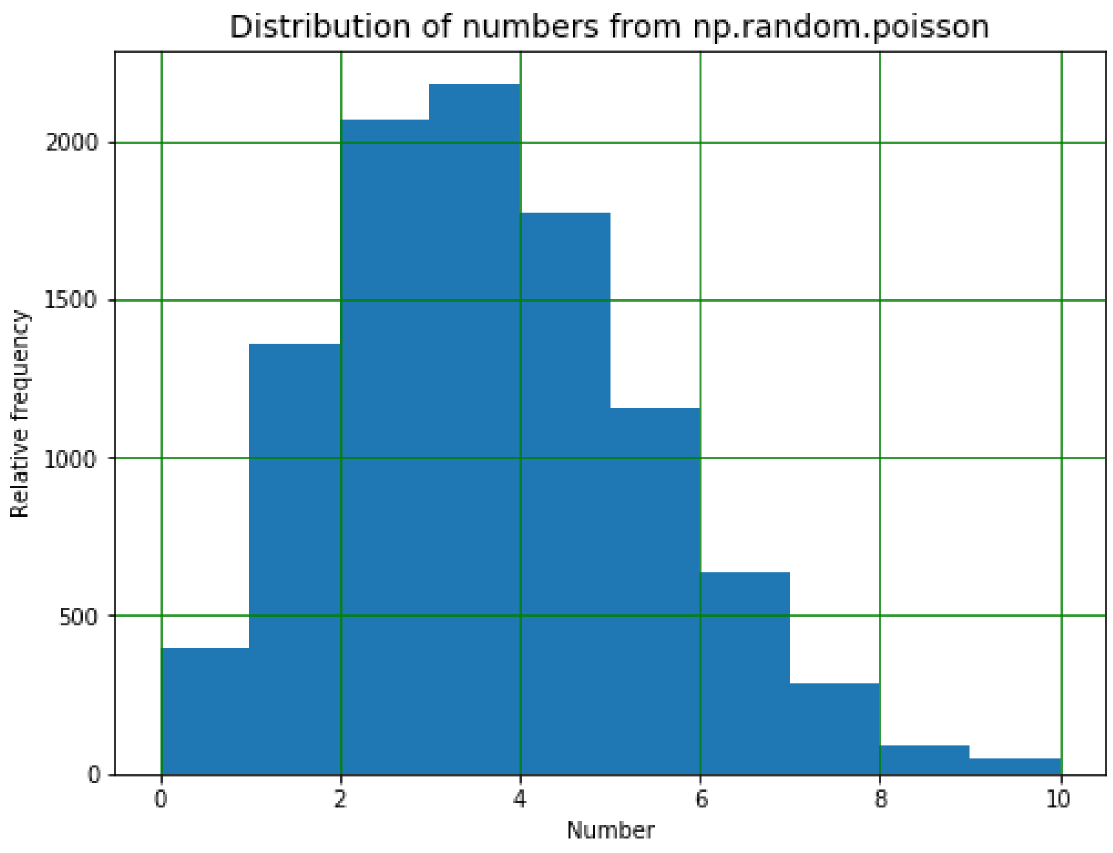
The random number generators available in `numpy.random` are described here. We will look at two further generators.

7.4.1 Poisson distribution

The routine `random.poisson` produces numbers drawn from a Poisson distribution. This distribution occurs, for example, when the number of cars crossing a line on a road every minute is measured. If the average number per minute is 3.2, the numbers measured in the first, second, third etc. minutes could be 3, 3, 3, 4, 5, 2, 2, 5, 4 and 3. We see that, in contrast to `random.rand`, the numbers produced by `random.poisson` are integers. Their distribution can be shown in a histogram as below.

[75]:
```
# <!-- Student -->
#
import matplotlib.pyplot as plt
%matplotlib inline
#
binBot = 0.0
binTop = 10.0
binNumber = 10
binEdges, binWidth = np.linspace(binBot, binTop, binNumber + 1, retstep = True)
print("Histogram bins start at",binBot,"finish at",binTop)
print("Number of bins is",binNumber,"and width of bins is",binWidth)
#
lam = 3.2
nEvents = 10000
randArr = np.random.poisson(lam, nEvents)
print("randArr\n",randArr[0:10])
#
plt.figure(figsize = (8, 6))
plt.title('Distribution of numbers from np.random.poisson', fontsize = 14)
plt.xlabel('Number')
plt.ylabel('Relative frequency')
plt.hist(randArr, bins = binEdges)
plt.grid(color = 'g')
plt.show()
```

```
Histogram bins start at 0.0 finish at 10.0
Number of bins is 10 and width of bins is 1.0
randArr
 [5 5 2 3 2 4 3 1 6 4]
```



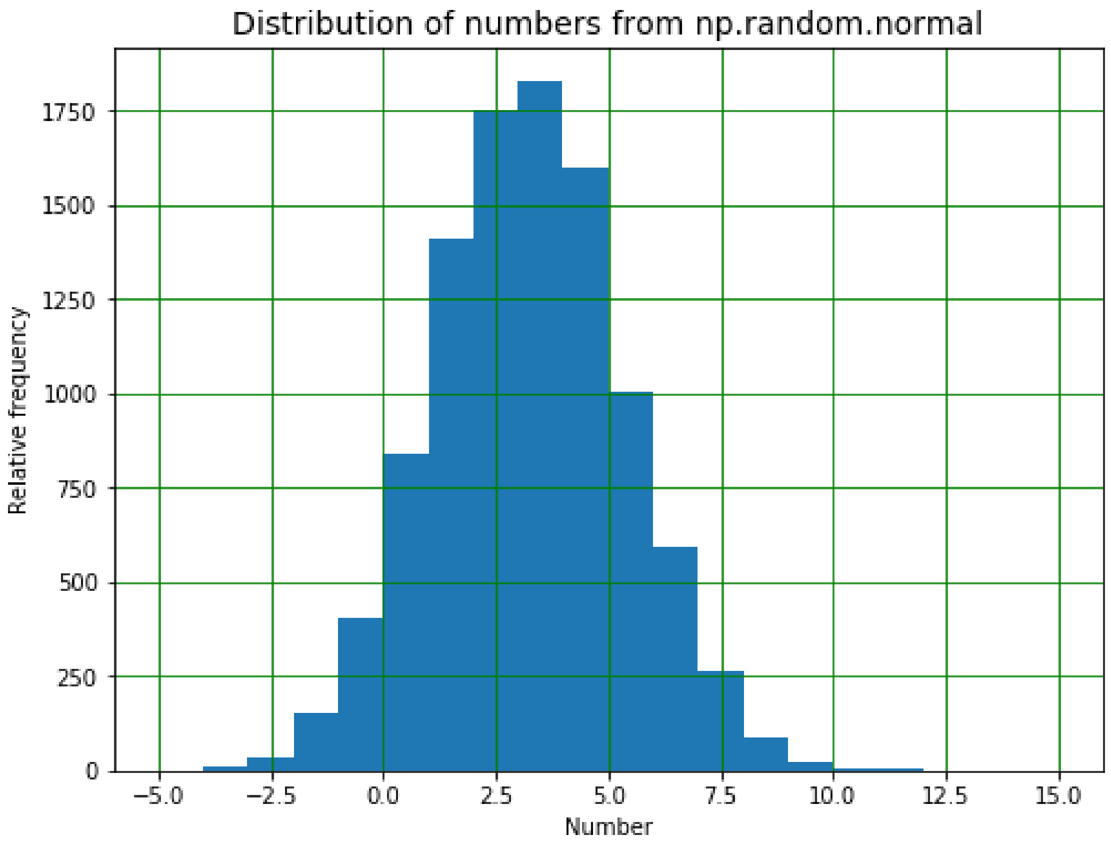
7.4.2 Normal distribution

The Normal or Gaussian distribution is obtained using the routine `random.normal`. The numbers produced are floats, and the routine is used as below.

[76]:
```
# <!-- Student -->
#
import matplotlib.pyplot as plt
%matplotlib inline
#
binBot = -5.0
binTop = 15.0
binNumber = 20
binEdges = np.linspace(binBot, binTop, binNumber + 1)
binWidth = (binTop - binBot)/binNumber
print("Histogram bins start at",binBot,"finish at",binTop)
print("Number of bins is",binNumber,"and width of bins is",binWidth)
#
mean = 3.2
RMS = 2.1
```

```
nEvents = 10000
randArr = np.random.normal(mean, RMS, nEvents)
np.set_printoptions(precision = 4)
print("randArr\n",randArr[0:5])
#
plt.figure(figsize = (8, 6))
plt.title('Distribution of numbers from np.random.normal', fontsize = 14)
plt.xlabel('Number')
plt.ylabel('Relative frequency')
plt.hist(randArr, bins = binEdges)
plt.grid(color = 'g')
plt.show()
```

```
Histogram bins start at -5.0 finish at 15.0
Number of bins is 20 and width of bins is 1.0
randArr
 [6.3188 3.6107 3.8604 1.7905 4.961 ]
```



7.5 Monte Carlo models

Random numbers can be used to produce computer models which descibe physics experiments or other problems. These are usually (for obvious reasons!) called Monte Carlo models. From the discussion above, it is clear that a simple example would be to use the Poisson generator to model situations like the number of cars passing through the Mersey tunnel every minute, or the number of salmon swimming up a stream every day. If the average number in the required time interval is known, the Poisson distribution will give a sequence of numbers (of cars or fish) as expected in real life. A model based on this idea could then be used to find out if a queue is likely to form in the tunnel, or how many bears the stream is likely to be able to feed. (Interestingly, having caught their salmon, the bears retire to the nearest woods, eat most of their fish and indulge in another activity for which bears are renowned. The resulting transfer of nutrients from stream to forest has been found to be essential to maintaining the local flora in some parts of the world - see the Salmon Forest Project for more information!)

Let's look at an example which is closer to home.

Records made by the Royal Liverpool Hospital staff show that, during the winter months, an average of 27 people require a bed in A&E each day. After one day, all A&E patients are moved to other wards. The hospital management have asked you to work out how many beds must be available to ensure that they never run out of spaces in A&E.

This statement is typical of how a question like this might be phrased, but it needs modification. One problem is the word "never". Even if the hospital were to have an enormous number of A&E beds, there will eventually be a winter in which the number will turn out to be too small! Your first job is to persuade the management team that the question has to be how many beds do we need to make sure we don't run out of spaces on (say) 99.7% of days.

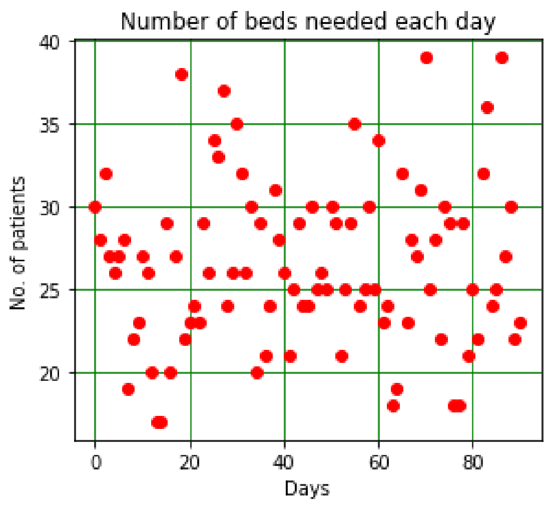
As shown below, we can use a Monte Carlo model to start to answer this question. What we are doing here is not realistic, as we are ignoring problems like the change in average number of people needing a bed with weather conditions, the occurrence of Liverpool and Everton football matches, what happens when the other wards in the hospital are full and someone has to stay in A&E for two days etc. It does, however, illustrate how useful Monte Carlo modelling can be in this kind of situation.

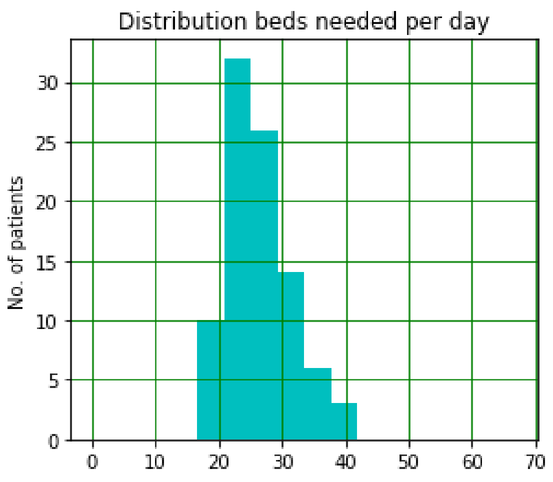
First, generate the numbers of patients expected per day during one winter (91 days) and check that this looks sensible by plotting it!

[77]:
```
# <!-- Student -->
#
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
#
# Average number of pationts per day
nPatsDay = 27
#
# Nmber of days in winter
nDays = 91
tArray = np.linspace(0, nDays - 1, nDays)
```

```
#
# Number of patients per day (i.e. number of beds needed per day), assuming␣
 ↪Poisson dist.
patsPerDay = np.random.poisson(nPatsDay, nDays) #
#
# Plot number of beds needed each day and the distribution of number of beds␣
 ↪needed per day
binBot = 0.0
binTop = int(2.5*nPatsDay)
binNumber = binTop//4
binEdges, binWidth = np.linspace(binBot, binTop, binNumber + 1, retstep = True)
binCentres = (binEdges[1:binNumber + 1] + binEdges[0:binNumber])/2
print("Histogram bins start at",binBot,"finish at",binTop)
print("Number of bins is",binNumber,"and width of bins is",binWidth)
print("Bin centres are:\n",binCentres)
print(" ")
#
plt.figure(figsize = (10, 4))
plt.subplot(1, 2, 1)
plt.title("Number of beds needed each day")
plt.xlabel("Days")
plt.ylabel("No. of patients")
plt.plot(tArray, patsPerDay, color = 'r', linestyle = '', marker = 'o')
plt.grid(color = 'g')
#
plt.subplot(1, 2, 2)
plt.title("Distribution beds needed per day")
plt.hist(patsPerDay, bins = binEdges, color = 'c')
plt.ylabel("Relative frequency")
plt.ylabel("No. of patients")
plt.grid(color = 'g')
#
plt.show()
```

```
Histogram bins start at 0.0 finish at 67
Number of bins is 16 and width of bins is 4.1875
Bin centres are:
 [ 2.0938  6.2812 10.4688 14.6562 18.8438 23.0312 27.2188 31.4062 35.5938
 39.7812 43.9688 48.1562 52.3438 56.5312 60.7188 64.9062]
```



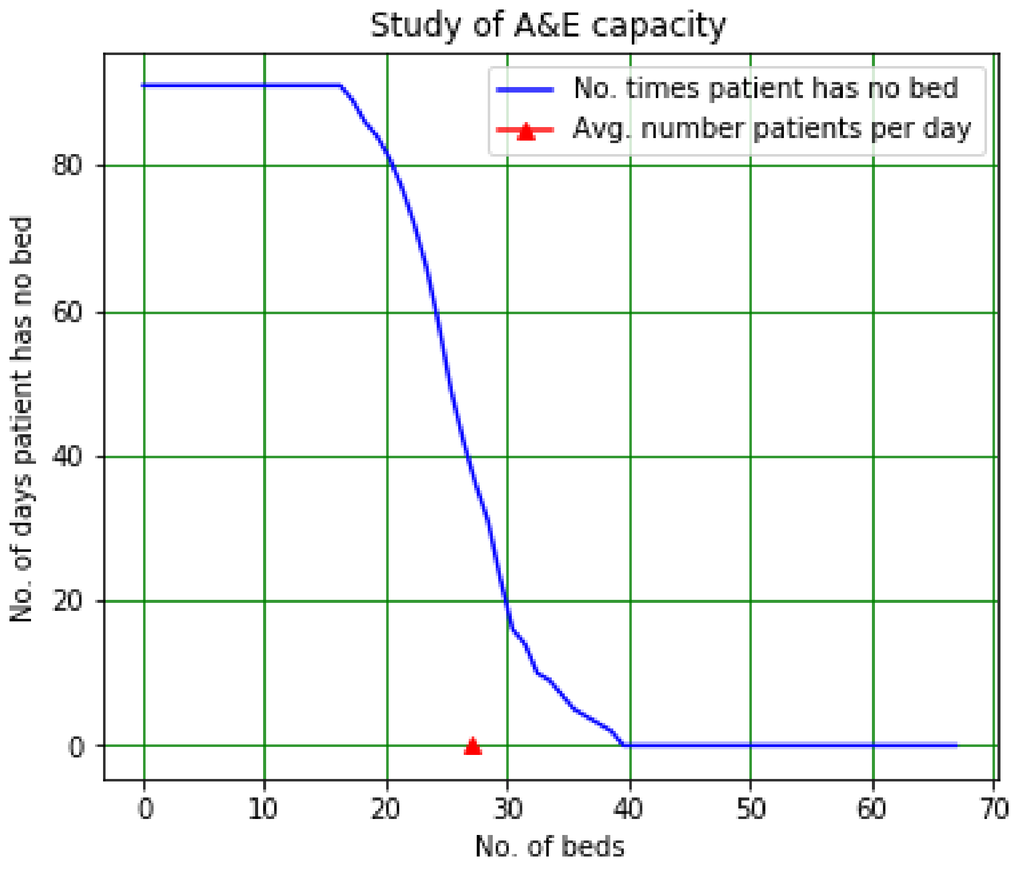


Now add up the number of beds needed each day and test how often the total exceeds the number available, for a range of numbers of available beds.

[78]:
```
# <!-- Student -->
#
maxBedNumber = int(2.5*nPatsDay)
noBedArr = np.zeros(maxBedNumber)
print(" ")
print("Number of beds available \t No days one or more patients has no bed")
for nBeds in range(0, maxBedNumber):
    nWithoutBed = np.sum(patsPerDay > nBeds)
    noBedArr[nBeds] = nWithoutBed
    if nBeds%5 == 0:
        print("\t\t {:d} \t\t\t\t {:d}".format(nBeds, nWithoutBed))
#
bedNumberArr = np.linspace(0, maxBedNumber, maxBedNumber)
#
plt.figure(figsize = (6, 5))
plt.title("Study of A&E capacity")
plt.xlabel("No. of beds")
plt.ylabel("No. of days patient has no bed")
plt.plot(bedNumberArr, noBedArr, color = 'b', label = "No. times patient has no␣
 ↪bed")
plt.plot(nPatsDay, 0.0, color = 'r', marker = '^', label = "Avg. number␣
 ↪patients per day")
plt.legend()
plt.grid(color = 'g')
plt.show()
```

```
Number of beds available         No days one or more patients has no bed
```

```
0                                   91
5                                   91
10                                  91
15                                  91
20                                  81
25                                  49
30                                  16
35                                  5
40                                  0
45                                  0
50                                  0
55                                  0
60                                  0
65                                  0
```



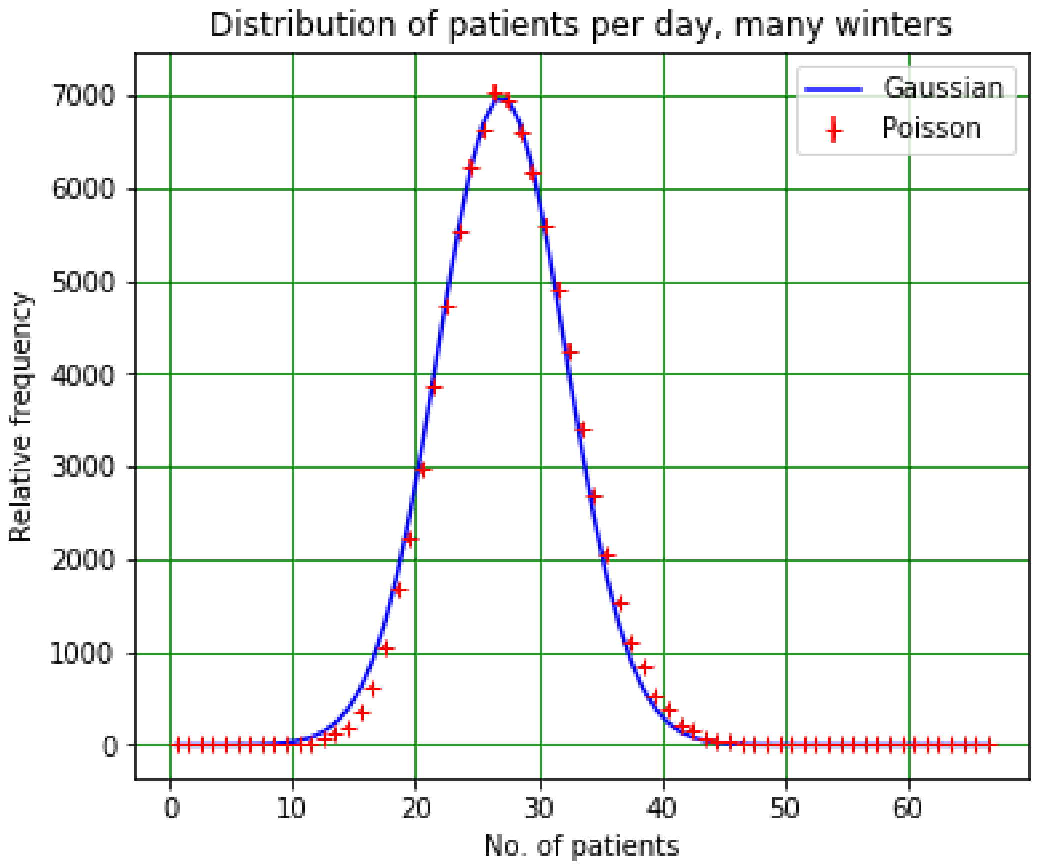
7.5.1 Week 6 exercise 3

Add comments to the code above. Explain in particular what each line of the code in the `for` loop is doing!

Both this plot and the histogram of the number of patients arriving per day show us that something like 35…45 beds are needed in this winter. But what happens next winter? And the winter after that? We need to extend the model to look at the situation over several years if we want to check that the number of occasions on which we run out of beds is small.

[79]:
```
# <!-- Student -->
#
# Can use finer bins as have lots more events
binBot = 0.0
binTop = int(2.5*nPatsDay)
binNumber = binTop
binEdges, binWidth = np.linspace(binBot, binTop, binNumber + 1, retstep = True)
binCentres = (binEdges[1:binNumber + 1] + binEdges[0:binNumber])/2
print("Histogram bins start at",binBot,"finish at",binTop)
print("Number of bins is",binNumber,"and width of bins is",binWidth)
print("Bin centres are:\n",binCentres)
#
nWinters = 1000
summedHist = np.zeros(binNumber)
#
for winter in range(0, nWinters):
    patsPerDay = np.random.poisson(nPatsDay, nDays)
    histThisWinter, bins = np.histogram(patsPerDay, bins = binEdges)
    summedHist = summedHist + histThisWinter
#
bedNumberArr = np.linspace(0, maxBedNumber, maxBedNumber)
#
totalDays = nWinters*nDays
numberOfBeds = 43
daysTooFewBeds = np.sum(summedHist[numberOfBeds:maxBedNumber]).astype(int)
daysEnoughBeds = totalDays -  daysTooFewBeds
propTooFewBeds = daysTooFewBeds/totalDays
propEnoughBeds = daysEnoughBeds/totalDays
print(" ")
print("Too few beds on {:d} days in total of {:d} days (proportion {:5.4e}).".
 ↪format(daysTooFewBeds, totalDays, propTooFewBeds))
print("That is, enough beds on {:d} days (proportion {:5.4e}).".
 ↪format(daysEnoughBeds, propEnoughBeds))
#
mean = nPatsDay
RMS = np.sqrt(nPatsDay)
norm = nWinters*nDays
gaussArr = norm/(np.sqrt(2*np.pi)*RMS)*np.exp(-(binCentres - mean)**2/
 ↪(2*RMS**2))
print(" ")
print("Mean {:5.3f}, RMS {:5.3f}.".format(nPatsDay, RMS))
print("Mean + 3*RMS {:5.3f}.".format(nPatsDay+3*RMS))
print(" ")
#
plt.figure(figsize = (6, 5))
plt.title("Distribution of patients per day, many winters")
plt.ylabel("Relative frequency")
plt.xlabel("No. of patients")
plt.errorbar(binCentres, summedHist/binWidth, yerr = np.sqrt(summedHist)/
 ↪binWidth,
             linestyle = '', marker = '+', color = 'r', label = 'Poisson')
plt.plot(binCentres, gaussArr, color = 'b', label = 'Gaussian')
plt.grid(color = 'g')
plt.legend()
plt.show()
```

```
Histogram bins start at 0.0 finish at 67
Number of bins is 67 and width of bins is 1.0
Bin centres are:
 [ 0.5  1.5  2.5  3.5  4.5  5.5  6.5  7.5  8.5  9.5 10.5 11.5 12.5 13.5
 14.5 15.5 16.5 17.5 18.5 19.5 20.5 21.5 22.5 23.5 24.5 25.5 26.5 27.5
 28.5 29.5 30.5 31.5 32.5 33.5 34.5 35.5 36.5 37.5 38.5 39.5 40.5 41.5
 42.5 43.5 44.5 45.5 46.5 47.5 48.5 49.5 50.5 51.5 52.5 53.5 54.5 55.5
 56.5 57.5 58.5 59.5 60.5 61.5 62.5 63.5 64.5 65.5 66.5]

Too few beds on 213 days in total of 91000 days (proportion 2.3407e-03).
That is, enough beds on 90787 days (proportion 9.9766e-01).

Mean 27.000, RMS 5.196.
Mean + 3*RMS 42.588.
```



Now we see that about 43 beds are needed. In fact, as calculated above, with this number of beds the probablility of there being too few beds on a given day is about 0.3%.

7.5.2 An aside - Gaussian approximation to Poisson distribution

We also see that the Poisson distribution with mean λ looks very like a Gaussian with the same mean ($\mu = \lambda$) and standard deviation $\sigma = \sqrt{\lambda}$. This is true, provided that λ is not too small. (A rule of thumb that is sometimes used is $\lambda > 20$, but what "too small" really means depends on the accuracy of the approximation you need!) The probability that x lies within 3σ of the mean for a Gaussian Prob$(\mu - 3\sigma < \mathrm{x} < \mu + 3\sigma) = 0.9973$, so the probability of getting a result which is larger than the mean (i.e. the upper half of the distrubution only) by more than 3σ is $\frac{1-0.9973}{2} = 0.0013$. The number of 43 beds is roughly $\mu + 3\sigma$, so we would expect the probability of there being too few beds to be roughly 0.13%. Given the approximations involved, the agreement with the above result is reasonable, and from the graph we can see that the Poisson distribution lies above the Gaussian approximation in the region around 40, so we would expect the 0.13% to be an underestimate, as is indeed the case.

7.6 Calculating pi using random numbers

Random numbers can be used to solve a large range of problems, including determining the value of π. This is a method that might have been used by ancient mathematicians in Persia...though I have no idea whether they actually tried it!

Spread out a square blanket of size $2 \times 2\,\mathrm{m}^2$ on some flat ground. Mark out a circle of radius 1 m on the blanket, with the centre of the circle at the centre of the blanket. Now, from some distance, throw rice at the blanket so that the grains are spread uniformly over it. Count the grains that have fallen in the circle, N_C, and the total number of grains on the blanket, N.

As the distribution of rice grains is uniform, the number in the circle and on the blanket is proportional to their respective areas. This means:

$$\begin{aligned}\frac{N_C}{N} &= \frac{\pi \times 1^2}{2 \times 2} \\ &= \frac{\pi}{4}.\end{aligned}$$

We can therefore calculate the value of π by dividing the number of grains in the circle by the number on the blanket and multiplying by 4:

$$\pi = 4\frac{N_C}{N}.$$

As the only time rice is thrown in Liverpool is at weddings, where there aren't too many people whose primary interest is determining the value of π, we will use a random number generator to simulate this experiment.

[80]:
```
# <!-- Student -->
#
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
#
nGrains = 10000
riceX = np.random.rand(nGrains)
riceY = np.random.rand(nGrains)
riceR = np.sqrt(riceX**2 + riceY**2)
nCircle = np.sum(riceR < 1)
pi = 4*nCircle/nGrains
print(" ")
print("Number of grains on blanket {:d}, number in circle {:d}.".
 ↪format(nGrains, nCircle))
print("Value of pi is approximately {:6.5f}".format(pi))
```

```
Number of grains on blanket 10000, number in circle 7825.
Value of pi is approximately 3.13000
```

Now let's determine the error we expect on our value of π. This is a bit trickier than you might expect. We first write the ratio in terms of the independent quantities N_C, the number of grains in the circle and $N_{\bar{C}} = N - N_C$, the number not in the circle:

$$\pi = 4\frac{N_C}{N} = 4\frac{N_C}{N_C + N_{\bar{C}}}.$$

The standard error propagation formula states:

$$(\Delta\pi)^2 = \left(\frac{\partial\pi}{\partial N_C}\Delta N_C\right)^2 + \left(\frac{\partial\pi}{\partial N_{\bar{C}}}\Delta N_{\bar{C}}\right)^2.$$

Using the formula for the derivative of a quotient, the partial derivatives are:

$$\frac{\partial\pi}{\partial N_C} = 4\frac{(N_C + N_{\bar{C}}) - N_C}{(N_C + N_{\bar{C}})^2} = 4\frac{N_{\bar{C}}}{(N_C + N_{\bar{C}})^2} = 4\frac{N_{\bar{C}}}{N^2}$$

and

$$\frac{\partial\pi}{\partial N_{\bar{C}}} = 4\frac{-N_C}{(N_C + N_{\bar{C}})^2} = -4\frac{N_C}{N^2}.$$

As ΔN_C and $\Delta N_{\bar{C}}$ are $\sqrt{N_C}$ and $\sqrt{N_{\bar{C}}}$, respectively, we have:

$$\begin{aligned}(\Delta\pi)^2 &= \left(\frac{4N_{\bar{C}}}{N^2}\right)^2 N_C + \left(\frac{4N_C}{N^2}\right)^2 N_{\bar{C}} \\ &= \frac{4^2}{N^4} N_{\bar{C}} N_C \left(N_{\bar{C}} + N_C\right) \\ &= \frac{4^2}{N}\frac{N_C}{N}\frac{N_{\bar{C}}}{N}.\end{aligned}$$

Using $p = \frac{N_C}{N}$, the probability of a grain landing in the circle, and $q = 1 - p = \frac{N_{\bar{C}}}{N}$, the probability that it doesn't land in the circle, we get:

$$(\Delta\pi)^2 = \frac{4^2}{N}pq \tag{6}$$

$$\Rightarrow \Delta\pi = 4\sqrt{\frac{pq}{N}}. \tag{7}$$

7.6.1 Week 6 exercise 4

Determine the error you expect on the rice grain measurement of π. (Copy the code above into the cell below this one and add your error calculation to it.) Is the measurement consistent with the known value?

7.6.2 Week 6 exercise 5

In the cell below, we look at how the precision of the value of π increases as we increase the number of grains of rice. Unfortunately, there is an error in the code: find it and fix it!

[81]:
```
# <!-- Student -->
#
minGrains = 100
maxGrains = 10000
nSteps = 50
grainArr = np.linspace(minGrains, maxGrains, nSteps)
piArr = np.zeros(nSteps)
DpiArr = np.zeros(nSteps)
#
for n in range(0, nSteps):
    riceX = np.random.rand(grainArr[n])
    riceY = np.random.rand(grainArr[n])
    riceR = np.sqrt(riceX**2 + riceY**2)
    nCircle = np.sum(riceR < 1)
    piArr[n] = 4*nCircle/grainArr[n]
    DpiArr[n] = 4*np.sqrt(nCircle/grainArr[n]*(1 - nCircle/grainArr[n])/
 ↪grainArr[n])
#
plt.figure(figsize = (10,4))
plt.title("$\pi$ value with number of grains")
plt.xlabel("Number of grains")
plt.ylabel("Value of $\pi$")
plt.errorbar(grainArr, piArr, yerr = DpiArr, color = 'r', marker = '.',␣
 ↪linestyle = '')
plt.plot(grainArr, np.pi*np.ones(nSteps), color = 'b', marker = '', linestyle =␣
 ↪'-')
plt.grid(color = 'g')
plt.show()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-81-faf184ee1d50> in <module>
      9 #
     10 for n in range(0, nSteps):
---> 11     riceX = np.random.rand(grainArr[n])
     12     riceY = np.random.rand(grainArr[n])
     13     riceR = np.sqrt(riceX**2 + riceY**2)

mtrand.pyx in numpy.random.mtrand.RandomState.rand()

mtrand.pyx in numpy.random.mtrand.RandomState.random_sample()

_common.pyx in numpy.random._common.double_fill()

TypeError: 'numpy.float64' object cannot be interpreted as an integer
```

7.7 Summary of week 6

This week, we have seen some of the tools Python provides for generating random numbers and looked at a couple of examples illustrating how these can be used. Our illustrations were simple, but some Monte Carlo models are extremely complex. For example, the ATLAS collaboration working on the Large Hadron Collider (LHC) at CERN use a Monte Carlo program consisting of about 250 thousand lines of code. This program, described here, allows them to compare the measurements they make with their detector with theoretical expectations. The Monte Carlo simulates the interactions you would expect to see given the theory and then simulates the signatures you would expect to see in the detector when the particles from those interactions pass through it. It is no exaggeration to say that it would be impossible to perform experiments like ATLAS without random number generators and Monte Carlo programs.

8 Introduction to Computational Physics - Week 7

8.1 Table of contents week 7

8.2 Introduction to week 7

This week we will look at some techniques for debugging programs, but before we do that, we will go through some hints on trying to reduce the number of bugs in your code in the first place!

8.3 Good coding practice

8.3.1 Naming variables

1) Use variable names that are self-explanatory and clearly different. It's not a good idea to use `X` and `x` as labels for arrays containing the measurements of the period of a pendulum and its mass, for example. Names like `periodArr` and `massArr` are much clearer. Many Python programmers favour using `period_arr` and `mass_arr` to "camel case". (I find it often helps to indicate whether a variable is a single value or an array, e.g. `t` could be a single time and `tArr` or `t_arr` an array containing lots of times.) Good variable names can save you having to write lots of comments.

2) If you are using the same quantity two or more times, don't give it a new name each time. That will just be confusing. For example, if you have an array of ten mass values, used three times, don't call it `massArray` the first time, `massArr` the second and `tenMass` the third!

8.3.2 Comments on comments

1) It's a good idea to add units to your code. This can be done with a comment behind the quantity in question.

```
carSpeed = 33.2 # km per hour
hairThickness = 100 # um
```

(Note that um is a frequently used substitute for μm.)

2) Put sensible comments in your code (i.e. explain the things that need to be explained but not the things that don't!). For example, this is a useless comment:

```
#
# Print out the value of y
print("Value of y is",y)
```

Here is a slightly less obvious example of a poor comment. Make sure you explain what your code is supposed to do in a concise way. Often the nitty-gritty of things like how a numpy routine works are best left to the official documentation, particularly if the name makes it clear what the routine is doing!

```
#
# While the value of nRow is less than rowTest (set to 13 here) check how many non-zero
# elements there are in row nRow of the two dimensional array shortData. As soon as the
# number of non-zero elements is bigger than nonZero (which is usually equal to 7), print
# out the value of nRow and leave the while loop. (The numpy routine count_nonzero() counts
# the number of entries in an array that are not zero.)
#
nRow = 0
nonZero = 7
rowTest = 13
```

```
shortData = np.zeros((rowTest + 2, nonZero + 2))
shortData[3, :] = 1.0
#
while nRow < rowTest:
    if np.count_nonzero(shortData, axis = 1)[nRow] > nonZero:
        print("nRow =",nRow)
        break
    nRow = nRow + 1
```

Better would be:

```
#
nRow = 0
nonZero = 7
rowTest = 13
shortData = np.zeros((rowTest + 2, nonZero + 2))
shortData[3, :] = 1.0
#
# Find row in shortData that contains more than nonZero elements which are not 0.
while nRow < rowTest:
    if np.count_nonzero(shortData, axis = 1)[nRow] > nonZero:
        print("nRow =",nRow)
        break
    nRow = nRow + 1
```

You can test the routine to see if it does what you think it should below!

[82]:
```
# <!-- Student -->
#
import numpy as np
#
nRow = 0
nonZero = 7
rowTest = 13
shortData = np.zeros((rowTest + 2, nonZero + 2))
shortData[5, :] = 1.0
#
# Find first row in shortData with more than nonZero non-zero elements
while nRow < rowTest:
    if np.count_nonzero(shortData, axis = 1)[nRow] > nonZero:
        print("nRow =",nRow)
        break
    nRow = nRow + 1
```

```
nRow = 5
```

8.3.3 Week 7 exercise 1

Work out what the statement `nExc = int(np.cumprod(np.linspace(1, number, number))[number - 1])` does. Add an appropriate comment to the code below, or, better still, change the variable name so you don't need to add a comment! Print out an example.

```
[83]: # <!-- Student -->
      #
      import numpy as np
      number = 10
      nExc = int(np.cumprod(np.linspace(1, number, number))[number - 1])
```

8.3.4 Don't hardwire numbers into code

Use variables rather than numbers to control the length of arrays, the number of iterations in loops and so on. If you define a number of arrays to be of length 10, and use `for` loops with upper limit 10 to manipulate them, you'll have to change all the 10s to 12s by hand if you later need bigger arrays. If you had used a variable, you would just have had to change 10 to 12 once.

8.4 Debugging

We will consider three kinds of bug. The first type are the ones that prevent your code from compiling. The second are the bugs that cause it to fail while it's running. These types are both "safe", in the sense that you know something is wrong and you have to fix it before your program will work. More dangerous are the third type, where the program seems to work OK but sometimes does something you don't expect. One famous example of this kind of bug caused the loss of NASA's 330 million dollar Mars Climate Orbiter. This happened when control software written by Lockheed calculated a thrust in pound-force seconds (the standard American unit) instead of newton-seconds (the SI unit) that NASA had specified and used in their own software. The difference caused the Orbiter to get too close to Mars and it broke up in the planet's atmosphere. (Fortunately, the mistakes we make in Phys105 are unlikely to have such costly consequences!)

Python classifies errors as being of various types including:

8.4.1 SyntaxError

What you have written doesn't satisfy the Python language rules (e.g. you are missing a bracket, or a comma is an unexpected place).

8.4.2 IndentationError

Inconsistent indentations have been detected, perhaps in a `for` loop or a function.

8.4.3 TabError

Have you mixed spaces and tabs when indenting the code in a function?

8.4.4 NameError

You probably have tried to use a variable you haven't defined. Often due to a spelling mistake or using a lower case letter where a capital was needed!

8.4.5 TypeError

You have entered a float where any string was expected, or a float where an integer was needed, or something similar.

8.4.6 ValueError

You have entered a value that doesn't allow the requested operation to be performed. E.g. you have entered `int("one")`. You can enter strings into this function, so this is not a *TypeError*. But Python doesn't understand what you want to get as an answer. If you had written `int('1')` that would be OK (it would give you the integer 1), but if the meaning isn't clear, you will get a ValueError.

8.4.7 IndexError

This is what happens, for example, if you try and access `myArray[17]` when `myArray` only has a length of six.

8.4.8 ZeroDivisionError

Guess what this one means!

8.4.9 ModuleNotFoundError

You get this if you ask Python to import a module and it can't find it. It's usually a spelling mistake!

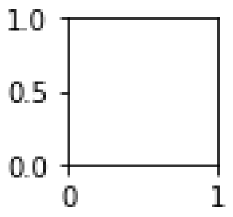
8.4.10 More error types

Further error types are discussed in your textbook, *A Student's Guide to Physical Modeling"*.

8.5 Code that won't compile

Let's look at some examples of things that can go wrong that stop your code compiling.

[84]:
```
# <!-- Student -->
#
fig, ax = plt.subplots(figsize = (1, 1))
```



Python tries to tell us where there is a problem in the code and what that problem is. (Sometimes the messages are long, in which case the most important information is often at the top of the error messages - where the problem occurred - and at the bottom - what Python thinks was wrong). This one is a *NameError*, Python has bumped into an undefined name, and it is easy to find. Python both manages to tell us where the problem is and give us a clear indication of what it is: “name ‘plt’ is not defined”. The solution is to import matplotlib.pyplot...

```
[85]: # <!-- Student -->
      #
      import matplotlib.pyploy as plt
      #
      fig, ax = plt.subplots(figsize = (5, 7))
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
<ipython-input-85-9b629e489c17> in <module>
      1 # <!-- Student -->
      2 #
----> 3 import matplotlib.pyploy as plt
      4 #
      5 fig, ax = plt.subplots(figsize = (5, 7))

ModuleNotFoundError: No module named 'matplotlib.pyploy'
```

8.5.1 Week 7 exercise 2

Whoops, now Python can’t find matplotlib.pyplot! What’s wrong here?

Python can’t always identify the location of an error. An error in one line may cause Python to identify an error in another, often the line after the one with the error.

```
[86]: # <!-- Student -->
      #
      xData = np.full((5, 6)
      print("xData\n",xData)
```

```
  File "<ipython-input-86-eacb6d88a24a>", line 4
    print("xData\n",xData)
        ^
SyntaxError: invalid syntax
```

This time the problem is identified as a *SyntaxError*, something about the program we have written doesn’t obey the rules of the Python language. Here, the error is a missing bracket in line 3. Because there is no closing bracket, Python assumes that the second line is a continuation of the first, in which case the print statement is incorrect, so that is where the error is identified. (Note, if you put the cursor next to a bracket, its partner will be coloured green. Try this now! This helps you identify which bracket is missing its other half.)

Try and fix this one by adding the required bracket.

[87]:
```
# <!-- Student -->
#
xData = np.full((5, 6))
print("xData\n",xData)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-87-f4dcfc0579d3> in <module>
      1 # <!-- Student -->
      2 #
----> 3 xData = np.full((5, 6))
      4 print("xData\n",xData)

TypeError: full() missing 1 required positional argument: 'fill_value'
```

This is another case where we have lice and fleas (a German saying meaning you can have more than one problem at once!). The second problem is a *TypeError*. Something has been given the wrong type. In this case, the location of the error is correctly identified, and the error message is also clear: the routine `np.full` needs an additional argument. The solution is to look up `np.full` in the numpy documentation or to google something like *numpy full*. You'll then see that we need to specify the shape of the array we want to fill using a tuple as we've done, but we also need to tell numpy what we want to fill the array with. This can be fixed as below:

[88]:
```
# <!-- Student -->
#
xData = np.full((5, 6), 13)
print("xData\n",xData)
```

```
xData
 [[13 13 13 13 13 13]
 [13 13 13 13 13 13]
 [13 13 13 13 13 13]
 [13 13 13 13 13 13]
 [13 13 13 13 13 13]]
```

Other common errors include not declaring variables before they are used (perhaps because the spelling is wrong).

8.5.2 Week 7 exercise 3

Fix the error in the code below.

[89]:
```
# <!-- Student -->
#
littleArray = np.ones(3)
littleArray[0] = 2
```

```
LittleArray[1] = 1
littleArray[2] = 0
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-89-3a04e73ee5da> in <module>
      3 littleArray = np.ones(3)
      4 littleArray[0] = 2
----> 5 LittleArray[1] = 1
      6 littleArray[2] = 0

NameError: name 'LittleArray' is not defined
```

Error messages also tell us when the wrong type of brackets is used. Functions need round brackets around their arguments (error message clear)...

[90]:
```
# <!-- Student -->
#
def thisFunc[y]:
    q = y**2
    return q
```

```
  File "<ipython-input-90-aa52d97c0ed6>", line 3
    def thisFunc[y]:
                ^
SyntaxError: invalid syntax
```

...and arrays, lists and tuples need square brackets around their indices (error message not so obvious).

[91]:
```
# <!-- Student -->
#
littleArray(1) = 3.2
```

```
  File "<ipython-input-91-d22b2a26d1a8>", line 3
    littleArray(1) = 3.2
                        ^
SyntaxError: can't assign to function call
```

8.5.3 Week 7 exercise 4

Explain why Python has produced the error message above! Why is it talking about functions?

The first problem above (the brackets) is fixed below:

[92]:
```
# <!-- Student -->
#
index = 1.0
littleArray[index] = 7.2
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-92-a550aae28d64> in <module>
      2 #
      3 index = 1.0
----> 4 littleArray[index] = 7.2

IndexError: only integers, slices (`:`), ellipsis (`...`), numpy.newaxis (`None`)
 ↪and integer or boolean arrays are valid indices
```

8.5.4 Week 7 exercise 5

Unfortunately, we see that again we have a second error in the code above. Fix it!

8.6 Programs that start but don't finish

Many of the problems that cause a program to start and then fail before it reaches its end are to do with array indices exceeding the boundaries of an array.

[93]:
```
# <!-- Student -->
#
shortArray = np.ones(10)
for i in range(0,  10):
    shortArray[i] = i
print("shortArray\n",shortArray)
newShortArray = np.ones(9)
for i in range(0,  10):
    newShortArray[i] = i
print("newShortArray\n",newShortArray)
```

```
shortArray
 [0. 1. 2. 3. 4. 5. 6. 7. 8. 9.]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-93-736fe2fb7cf9> in <module>
      7 newShortArray = np.ones(9)
      8 for i in range(0,  10):
----> 9     newShortArray[i] = i
     10 print("newShortArray\n",newShortArray)
```

```
IndexError: index 9 is out of bounds for axis 0 with size 9
```

The first section of the routine runs OK, then Python bumps into a problem in the second `for` loop. This is easily fixed, and also easily avoided if variables are used to control the length of arrays, the number of iterations in loops etc. as below:

[94]:
```
# <!-- Student -->
#
nShort = 10
shortArray = np.ones(nShort)
for i in range(0,  nShort):
    shortArray[i] = i
print("shortArray\n",shortArray)
newShortArray = np.ones(nShort)
for i in range(0,  nShort):
    newShortArray[i] = i
print("newShortArray\n",newShortArray)
```

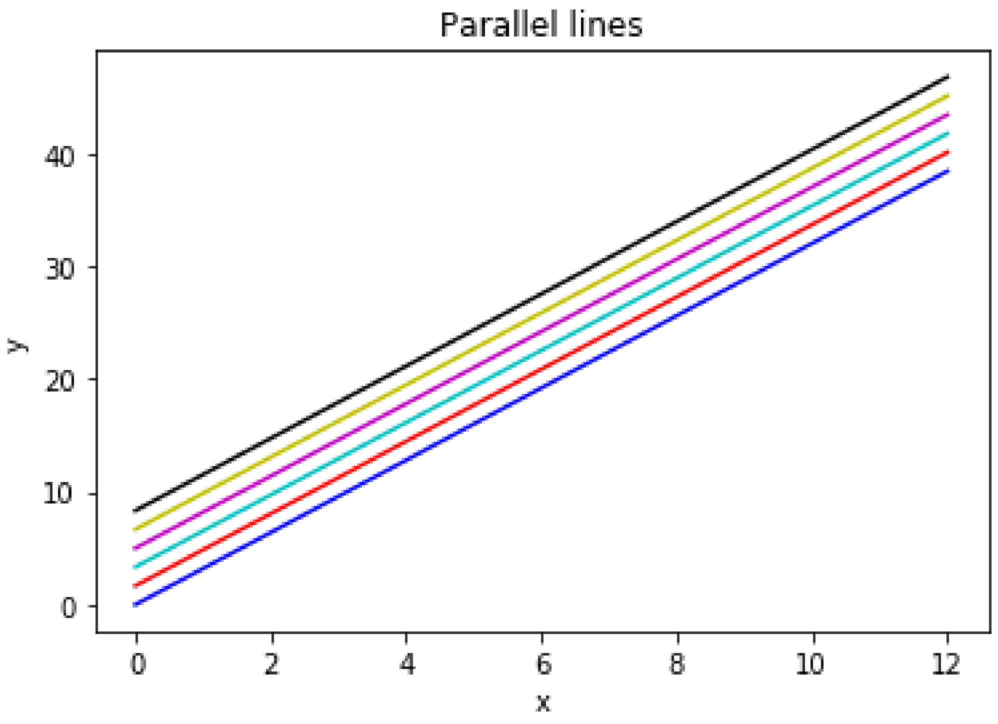
```
shortArray
 [0. 1. 2. 3. 4. 5. 6. 7. 8. 9.]
newShortArray
 [0. 1. 2. 3. 4. 5. 6. 7. 8. 9.]
```

8.6.1 Week 7 exercise 6

This program is supposed to plot a number of parallel lines with changing colours, but doesn't work. Add comments to the code where necessary and find and fix the bug!

[95]:
```
# <!-- Student -->
#
import matplotlib.pyplot as plt
%matplotlib inline
#
nPoints = 12
xArray = np.linspace(0, nPoints, nPoints + 1)
#
nLines = 10
maxConst = 15.0
constArr = np.linspace(0, maxConst, nLines)
grad = 3.2
yLines = np.zeros((nLines, nPoints + 1))
#
nCols = 6
colList = ['b', 'r', 'c', 'm', 'y', 'k']
iCol = 0
#
plt.figure(figsize = (6, 4))
plt.title("Parallel lines")
plt.xlabel("x")
plt.ylabel("y")
#
for n in range(0, nLines):
    yLines[n, :] = grad*xArray[:] + constArr[n]
    plt.plot(xArray, yLines[n, :], linestyle = '-', color = colList[iCol])
    iCol = iCol + 1
    if iCol > nCols:
        iCol = 0
#
plt.grid(color = 'g')
plt.show()
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-95-55bec638e64b> in <module>
     24 for n in range(0, nLines):
     25     yLines[n, :] = grad*xArray[:] + constArr[n]
---> 26     plt.plot(xArray, yLines[n, :], linestyle = '-', color =␣
 ↪colList[iCol])
     27     iCol = iCol + 1
     28     if iCol > nCols:

IndexError: list index out of range
```



8.6.2 The strange case of the shrinking array

There are some features of Python that can cause problems that are difficult to diagnose. One is that the length of arrays can be altered in calculations. This is done to optimise the use of the computer's memory, but if you assume that the lengths of arrays are always as in the original declarations, it can lead to some confusing errors. Here is an example.

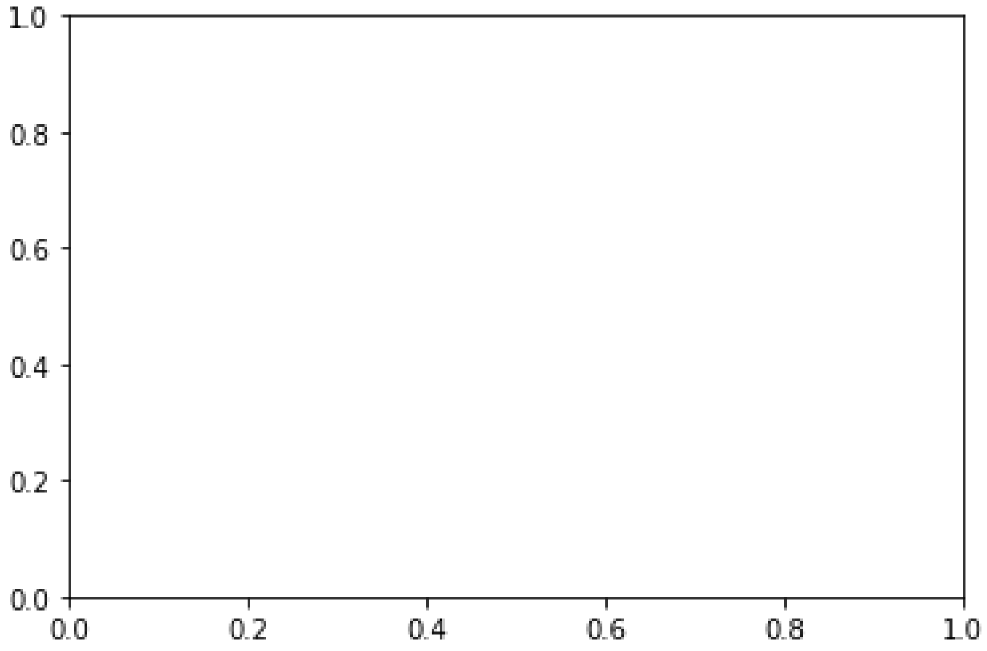
[96]:
```
# <!-- Student -->
#
import matplotlib.pyplot as plt
%matplotlib inline
#
nA = 10
arrayA1 = np.linspace(0, nA - 1, nA)
print(" ")
print("arrayA1 \n",arrayA1)
print("Length of arrayA1",len(arrayA1))
#
arrayA1new = np.linspace(0, nA - 1, nA)
print(" ")
print("arrayA1new\n",arrayA1new)
print("Length of arrayA1new",len(arrayA1new))
#
nB = 3
arrayB = np.linspace(0, nB - 1, nB)
print(" ")
print("arrayB\n",arrayB)
print("Length of arrayB",len(arrayB))
#
arrayA1new=2*arrayB
print(" ")
print("New arrayA1 \n",arrayA1new)
print("Length of new arrayA1",len(arrayA1new))
#
print("Plot arrayA1 against arrayA1new - you might think this should work!")
plt.figure(figsize = (6, 4))
plt.plot(arrayA1, arrayA1new)
plt.show()
```

```
arrayA1
 [0. 1. 2. 3. 4. 5. 6. 7. 8. 9.]
Length of arrayA1 10

arrayA1new
```

```
 [0. 1. 2. 3. 4. 5. 6. 7. 8. 9.]
Length of arrayA1new 10

arrayB
 [0. 1. 2.]
Length of arrayB 3

New arrayA1
 [0. 2. 4.]
Length of new arrayA1 3
Plot arrayA1 against arrayA1new - you might think this should work!
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-96-380a42463f16> in <module>
     28 print("Plot arrayA1 against arrayA1new - you might think this should␣
 ↪work!")
     29 plt.figure(figsize = (6, 4))
---> 30 plt.plot(arrayA1, arrayA1new)
     31 plt.show()

~\Anaconda3\lib\site-packages\matplotlib\pyplot.py in plot(scalex, scaley, data ␣
 ↪*args, **kwargs)
   2794     return gca().plot(
   2795         *args, scalex=scalex, scaley=scaley, **({"data": data} if data
-> 2796         is not None else {}), **kwargs)

   2797
   2798

~\Anaconda3\lib\site-packages\matplotlib\axes\_axes.py in plot(self, scalex,␣
 ↪scaley, data, *args, **kwargs)
   1663         """
   1664         kwargs = cbook.normalize_kwargs(kwargs, mlines.Line2D._alias_ma )
-> 1665         lines = [*self._get_lines(*args, data=data, **kwargs)]
   1666         for line in lines:
   1667             self.add_line(line)

~\Anaconda3\lib\site-packages\matplotlib\axes\_base.py in __call__(self, *args,
 ↪**kwargs)
    223                 this += args[0],
    224                 args = args[1:]
--> 225             yield from self._plot_args(this, kwargs)
    226
    227     def get_next_color(self):

~\Anaconda3\lib\site-packages\matplotlib\axes\_base.py in _plot_args(self, tup,
 ↪kwargs)
```

```
    389                 x, y = index_of(tup[-1])
    390
--> 391             x, y = self._xy_from_xy(x, y)
    392
    393             if self.command == 'plot':

~\Anaconda3\lib\site-packages\matplotlib\axes\_base.py in _xy_from_xy(self, x, y)
    268             if x.shape[0] != y.shape[0]:
    269                 raise ValueError("x and y must have same first dimension,␣
 ↪but "
--> 270                                  "have shapes {} and {}".format(x.shape, y.
 ↪shape))
    271             if x.ndim > 2 or y.ndim > 2:
    272                 raise ValueError("x and y can be no greater than 2-D, but␣
 ↪have "

ValueError: x and y must have same first dimension, but have shapes (10,) and␣
 ↪(3,)
```



Python tells you what the error is (`x`, which is `arrayA1` and `y`, which is `arrayA1new` have different dimensions) but they shouldn't have according to their respective declarations! What Python has done is to set the dimensions of the array on the left-hand side (LHS) of the expression to be the same as the dimensions of the array(s) on the right-hand side (RHS). It does this where it is unambiguous. For example, the following doesn't cause an error as the quantities on the RHS all have the same dimension, so the dimension of `arrayA` (on the LHS) is changed to match that of arrayB (on the RHS).

[97]:
```
# <!-- Student -->
#
import numpy as np
#
arrayA = np.zeros(3)
print(arrayA)
arrayB = np.ones(4)
print(arrayB)
arrayA = 2*arrayB
print(arrayA)
```

```
[0. 0. 0.]
[1. 1. 1. 1.]
[2. 2. 2. 2.]
```

Things can more complicated. If you explicitly tell Python to loop through all the elements in the arrays on the LHS and RHS, it will no longer modify the array lengths. Compare what we have seen above...

[98]:
```
# <!-- Student -->
#
import numpy as np
#
arrayA = np.zeros(3)
print(arrayA)
arrayB = np.ones(4)
print(arrayB)
arrayA = 2*arrayB
print(arrayA)
```

```
[0. 0. 0.]
[1. 1. 1. 1.]
[2. 2. 2. 2.]
```

...with what happens below:

[99]:
```
# <!-- Student -->
#
import numpy as np
#
arrayA = np.zeros(3)
print(arrayA)
arrayB = np.ones(4)
print(arrayB)
arrayA[:] = 2*arrayB[:]
print(arrayA)
```

```
[0. 0. 0.]
```

```
[1. 1. 1. 1.]
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-99-ca24c45307cd> in <module>
      7 arrayB = np.ones(4)
      8 print(arrayB)
----> 9 arrayA[:] = 2*arrayB[:]
     10 print(arrayA)

ValueError: could not broadcast input array from shape (4) into shape (3)
```

This difference is a consequence of how Python stores data in the computer's memory and whether it is actually generating a new copy of the array or not. You get a new array in a new space in memory (which is given the length you ask for) when you explicitly go through the array elements:

```
arrayA[:] = 2*arrayB[:]
```

Whereas the following assigns a second name to an array that already exists in memory, keeping the length that it has:

```
arrayB = 2*arrayA
```

If you aren't sure what Python is doing, or you get error messages that make you think something like the above is going on, print out the length of your arrays using `len(array)` and check that their dimensions are what you expect. (Use `array.shape` for multidimensional arrays.)

8.7 Programs that run but produce incorrect results

As we have mentioned above, bugs which don't cause the program to crash are the most dangerous. They can go unrecognised for years and then pop up and cause a disaster. Books have been written on how to validate code, i.e. to check it does what it should in all circumstances. There are various things you can try to avoid problems (which, by the way, will also help track down bugs that do cause crashes!):

- Read your code carefully.
- When writing a program, do it in steps and test each step.
- Test your code on examples where you know what the answer should be and check you get what you expect!
- Print out the values of variables during running and check that they are as you expect. Do the same for variables after your code has run (the last values will be saved).
- Plot graphs that show your intermediate and final results. Are these sensible?
- Explain your code to someone else. (If there is no-one around, explain it to an imaginary friend!) It's surprising how often this helps you spot a mistake in your logic.
- Look to see if your problem has been encountered before (try google, or look on online forums like stackoverflow.com).

- Use the `assert` statement in your code (described below) to check that things you think should be true really are true!

8.8 Using the assert statement

The assert statement is a debugging tool that is built into Python. It can be used as illustrated in the following example.

Suppose we are calculating the roots of a number of quadratic equations, $ax^2 + bx + c = 0$, with various values of a, b and c. We think all the roots should be real. We could try and do this using the following code.

[100]:
```
# <!-- Student -->
#
nQuadEqs = 4
aArr = np.linspace(1.0, 4.0, nQuadEqs)
b = 2.0
c = 3.0
#
for n in range(0, nQuadEqs):
    discrim = np.sqrt(b**2 - 3*aArr[n]*c)
    root1 = (-b + discrim)/(2*aArr[n])
    root2 = (-b - discrim)/(2*aArr[n])
    print("For a ={:.2f}, b = {:.2f} and c = {:.2f}, roots are {:.2f} and {:.
 ↪2f}.".format(aArr[n], b, c, root1, root2))
```

```
For a =1.00, b = 2.00 and c = 3.00, roots are nan and nan.
For a =2.00, b = 2.00 and c = 3.00, roots are nan and nan.
For a =3.00, b = 2.00 and c = 3.00, roots are nan and nan.
For a =4.00, b = 2.00 and c = 3.00, roots are nan and nan.
```

```
C:\Users\green\Anaconda3\lib\site-packages\ipykernel_launcher.py:9:
RuntimeWarning: invalid value encountered in sqrt
  if __name__ == '__main__':
```

The roots are all given as `nan`. This is Pythonese for *not a number*. We get a strong hint as to where the problem is occurring: *RuntimeWarning: invalid value encountered in sqrt*. It looks as though there is something wrong with the calculation of `discrim`, and of course taking the square root of a negative number might be the problem. (Python does know about imaginary numbers, but the code we have written doesn't deal with them!) We can check this as follows:

[101]:
```
# <!-- Student -->
#
nQuadEqs = 4
aArr = np.linspace(1.0, 4.0, nQuadEqs)
b = 2.0
c = 3.0
#
for n in range(0, nQuadEqs):
    discrim = b**2 - 3*aArr[n]*c
    assert (discrim > 0), "Bumped into negative discrim ({:5.3f}), can't take␣
↪sqrt!".format(discrim)
    root1 = (-b + np.sqrt(discrim))/(2*aArr[n])
    root2 = (-b - np.sqrt(discrim))/(2*aArr[n])
    print("For a ={:.2f}, b = {:.2f} and c = {:.2f}, roots are {:.2f} and {:.
↪2f}.".format(aArr[n], b, c, root1, root2))
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-101-719c2999b3ef> in <module>
      8 for n in range(0, nQuadEqs):
      9     discrim = b**2 - 3*aArr[n]*c
---> 10     assert (discrim > 0), "Bumped into negative discrim ({:5.3f}), can't␣
 ↪take sqrt!".format(discrim)
     11     root1 = (-b + np.sqrt(discrim))/(2*aArr[n])
     12     root2 = (-b - np.sqrt(discrim))/(2*aArr[n])

AssertionError: Bumped into negative discrim (-5.000), can't take sqrt!
```

If the discriminant is negative, the `assert` statement is `False`, so the error message we have written gets printed out and the program stops. In this case, one error is that we have used the wrong value of c; we should have $c = -3$ and have put $c = 3$ instead.

Note, if you prefer, you can use a normal `if` statement together with the routine sys.exit(), as follows.

```
import sys
#
if discrim < 0:
    sys.exit("Bumped into negative discrim!")
```

However, the `assert` statement is quicker when you are debugging!

8.8.1 Week 7 exercise 7

Copy the above code into the cell below. Correct the value of `c` and run the code. It seems to work OK, but something is still wrong. Find out what the problem is and fix it!

9 Introduction to Computational Physics - Week 8

9.1 Table of contents week 8

9.2 Introduction to week 8

Computer programs written in languages like Fortran, C and Java are compiled before they are run. This means the original source code is converted into an executable file. The program is actually run in a separate step, using the executable file. In contrast, Python code is interpreted statement by statement at the time that the program is run. The two approaches have their advantages and disadvantages. The advantage of the separate compilation step is that, as the entire program is compiled in one go, it can be optimised to ensure that it uses the computer's memory and processors as efficiently as possible. The disadvantage is that the development process takes more time. If you change anything, you have to compile the entire program again before you can run it and test it. This is one of the reasons that developing programs using Python is usually considerably quicker than working in C, Fortran or Java. The downside is that Python programs are usually slower. For many applications, this isn't a problem. But in some scientific analyses, for example involving large datasets, it can cause difficulties. This is where numpy can be very useful. It allows programs to be written using Python, but the underlying numpy code has been written in C and compiled, allowing execution at high speed. Learning to use numpy effectively is therefore a useful skill for scientific programmers.

This week we will look at an example of how numpy arrays can be used to solve equations numerically, and then illustrate how the same technique can be used to do some image processing. The idea is not so much that the techniques used are the best for solving equations or edge-finding in images, but that they show how numpy functions can sometimes be used as alternatives to Python control structures in scientific programming.

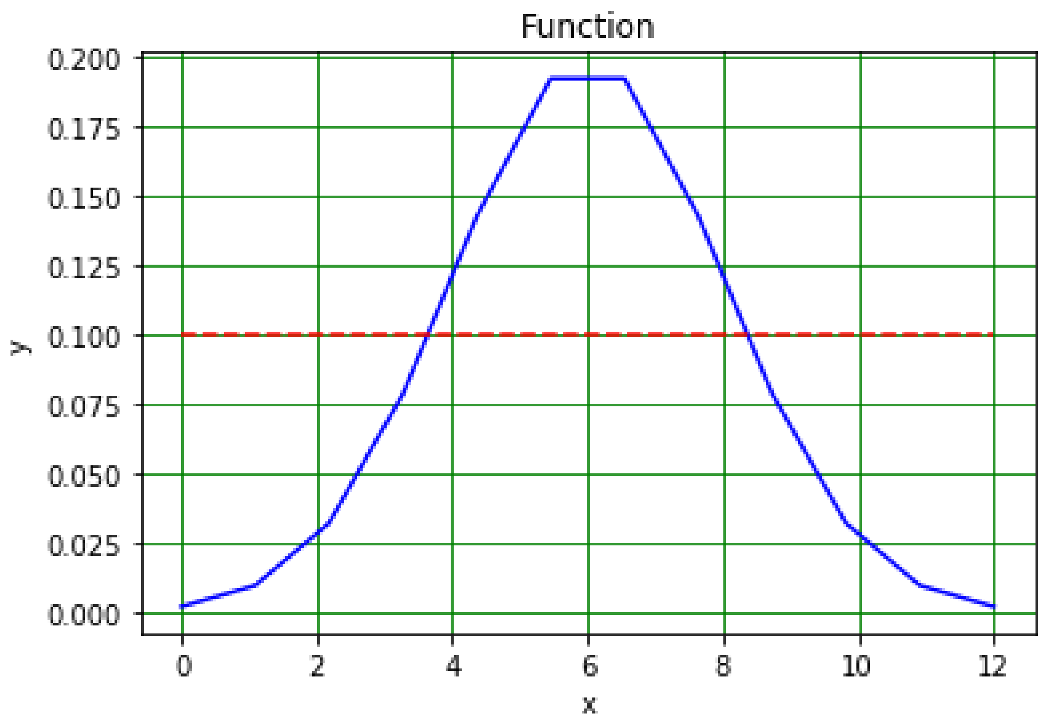
9.3 Numerical solution of equations

9.3.1 Graphical approach

Suppose we have a function $y(x)$ and we want to know the values of x where it crosses a threshold value T, i.e. we want to solve the equation $y(x) = T$. We could do this using a graph:

[102]:
```
# <!-- Student -->
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
#
def gaussFunc(mu, sigma, x):
    g = 1/(np.sqrt(2*np.pi)*sigma)*np.exp(-(x - mu)**2/(2*sigma**2))
    return g
#
mu = 6.0
```

```
sigma = 2.0
#
thresh = 0.1
#
xMin = 0.0
xMax = 12.0
nPoints = 12
xArr = np.linspace(xMin, xMax, nPoints)
#
yArr = gaussFunc(mu, sigma, xArr)
#
plt.figure(figsize = (6, 4))
plt.title("Function")
plt.xlabel("x")
plt.ylabel("y")
plt.plot(xArr, yArr, color = 'b', linestyle = '-')
plt.plot(xArr, thresh*np.ones(nPoints), color = 'r', linestyle = '--')
plt.grid(color = 'g')
plt.show()
```



9.3.2 Week 8 exercise 1

Determine the solutions of the equation $y(x) = T$ from the graph above. How could the precision of the solutions be increased?

9.3.3 Solving equation using numpy

As an alternative to the graphical solution, we could also manipulate the numpy array that contains the values of the function in the following way.

The first step is to make a logical (or boolean) array that is `True` where the function is above the threshold `thresh` and `False` otherwise.

[103]:
```
# <!-- Student -->
#
LyArr = yArr > thresh
```

9.3.4 Week 8 exercise 2

Print out the values of the array `LyArr`. Convert the array from type `bool` to type `int` and print out these values. Use a numpy function to sum the contents of both the boolean and integer arrays. Compare the results!

We then make another version of the logical array in which all the elements are shifted to the left.

[104]:
```
# <!-- Student -->
#
LyArrXL = np.zeros(nPoints).astype(bool)
shiftX = 1
LyArrXL[0:nPoints - shiftX] = LyArr[shiftX:nPoints]
print("LyArr\n",LyArr)
print("LyArrXL\n",LyArrXL)
```

```
LyArr
 [False False False False  True  True  True  True False False False False]
LyArrXL
 [False False False  True  True  True  True False False False False False]
```

Now we take the logical `not` of the first array.

[105]:
```
# <!-- Student -->
#
LyArrNot = np.logical_not(LyArr)
print("LyArrXL\n",LyArrXL)
print("LyArrNot\n",LyArrNot)
```

```
LyArrXL
 [False False False  True  True  True  True False False False False False]
LyArrNot
 [ True  True  True  True False False False False  True  True  True  True]
```

Taking the logical `and` of `LyArrXL` and `LyArrNot`, we can determine the left-hand position in the array which corresponds to the place where the function crosses the threshold, as below.

[106]:
```
# <!-- Student -->
#
```

```
boolThrL = np.logical_and(LyArrNot, LyArrXL)
print("boolThrL\n",boolThrL)
```

```
boolThrL
 [False False False  True False False False False False False False False]
```

We can now get the left-hand x value at which the function crosses the threshold using a further feature of numpy arrays: if we use a set of logical values as the indices of an array, the values that are returned are those with index `True`. This is shown below.

[107]:
```
# <!-- Student -->
#
xThrL = xArr[boolThrL]
np.set_printoptions(precision = 2)
print("xThrL",xThrL)
```

```
xThrL [3.27]
```

9.3.5 Week 8 exercise 3

Make a plot to show that the x value above is (approximately!) in the correct position.

9.3.6 Week 8 exercise 4

How could the precision of the x value of the crossing point be improved?

9.3.7 Week 8 exercise 5

Copy the code above and edit it so that it finds the position of the right-hand point at which the gaussian function crosses the threshold. Plot a graph showing the position of the right-hand crossing point.

9.4 Image analysis with numpy

Numpy can also be used to analyse images. The example below shows how a picture of a telescope can be read into numpy and then manipulated so that only pixels above a certain threshold are displayed.

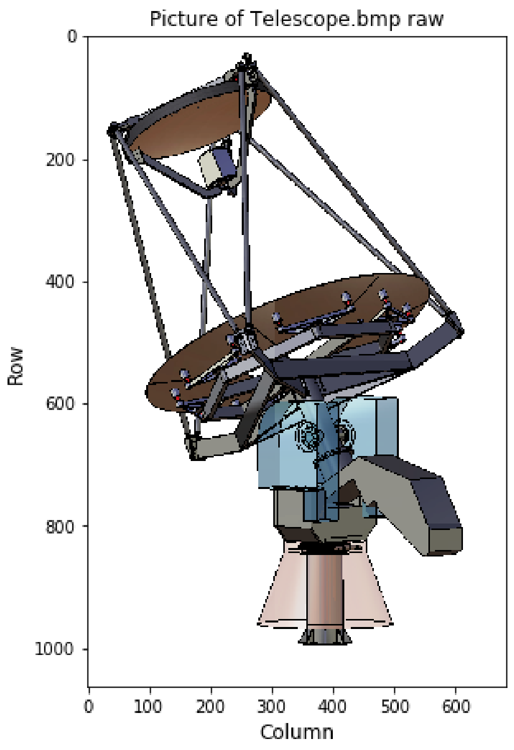
As an aside, if you want to manipluate images using Python, the `scipy.ndimage` package provides a large range of image analysis software available; see here for more information!

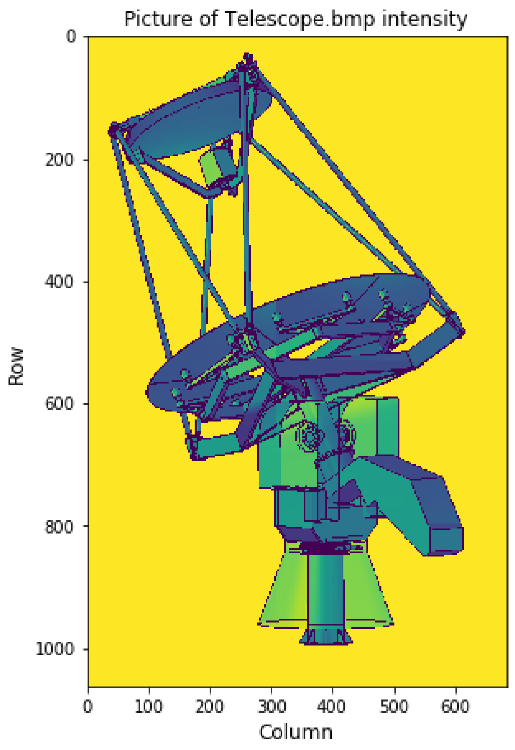
[108]:
```
# <!-- Student -->
#
import datetime
now = datetime.datetime.now()
print("Date and time ",str(now))
#
import numpy as np
import scipy.ndimage as scimg
import matplotlib.pyplot as plt
%matplotlib inline
```

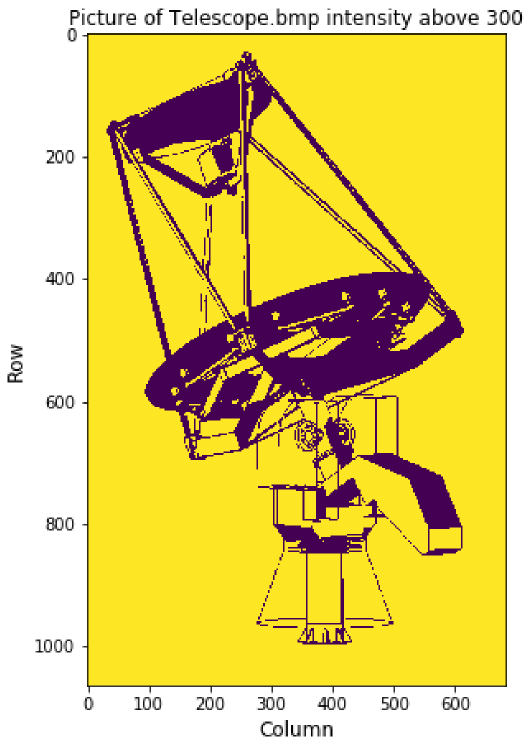
```
#
# Read in image
imFile = "Telescope.bmp"
#imFile = "Shark.jpg"
#
imgRaw = plt.imread(imFile)
nRows = imgRaw.shape[0] # row corresponds to pixel's y coordinate
nCols = imgRaw.shape[1] # col corresponds to pixel's x ccordinate
nDepth = imgRaw.shape[2] # depth == 3 for red, blue green (RBG), == 4 for RBGA␣
 ↪(A is alpha, i.e. transparency)
img = np.zeros((nRows, nCols))
#img[0:nRows, 0:nCols] = imgRaw[0:nRows, 0:nCols, 0].astype(int) # Using depth␣
 ↪[0] gives red
img[0:nRows, 0:nCols] = (imgRaw[0:nRows, 0:nCols, 0].astype(int) +
                         imgRaw[0:nRows, 0:nCols, 1].astype(int) +
                         imgRaw[0:nRows, 0:nCols, 2].astype(int)) # Using depth␣
 ↪[0 + 1 + 2] gives intensity
print("Number of rows",nRows,"of columns",nCols,"of pixels",nRows*nCols,"and␣
 ↪depth",nDepth)
#
thresh = 300  # set threshold for finding edges
imgThr = np.zeros((nRows, nCols))
imgThr = img > thresh
#
print(" ")
fig = plt.figure(figsize=(12, 16))
thisplt = fig.add_subplot(2, 2, 1)
plt.title("Picture of " + imFile + " raw")
plt.xlabel('Column', fontsize = 12)
plt.ylabel('Row', fontsize = 12)
imgplot = plt.imshow(imgRaw)
#
thisplt = fig.add_subplot(2, 2, 2)
plt.title("Picture of " + imFile + " intensity")
plt.xlabel('Column', fontsize = 12)
plt.ylabel('Row', fontsize = 12)
imgplot = plt.imshow(img)
#
thisplt = fig.add_subplot(2, 2, 3)
plt.title("Picture of " + imFile + " intensity above " + str(thresh))
plt.xlabel('Column', fontsize = 12)
plt.ylabel('Row', fontsize = 12)
imgplot = plt.imshow(imgThr)
#
plt.show()
#
then = now
```

```
now = datetime.datetime.now()
print(" ")
print("Date and time",str(now))
print("Time since last check is",str(now - then))
```

```
Date and time  2020-02-05 11:59:14.386635
Number of rows 1064 of columns 684 of pixels 727776 and depth 3
```







```
Date and time 2020-02-05 11:59:14.870341
```

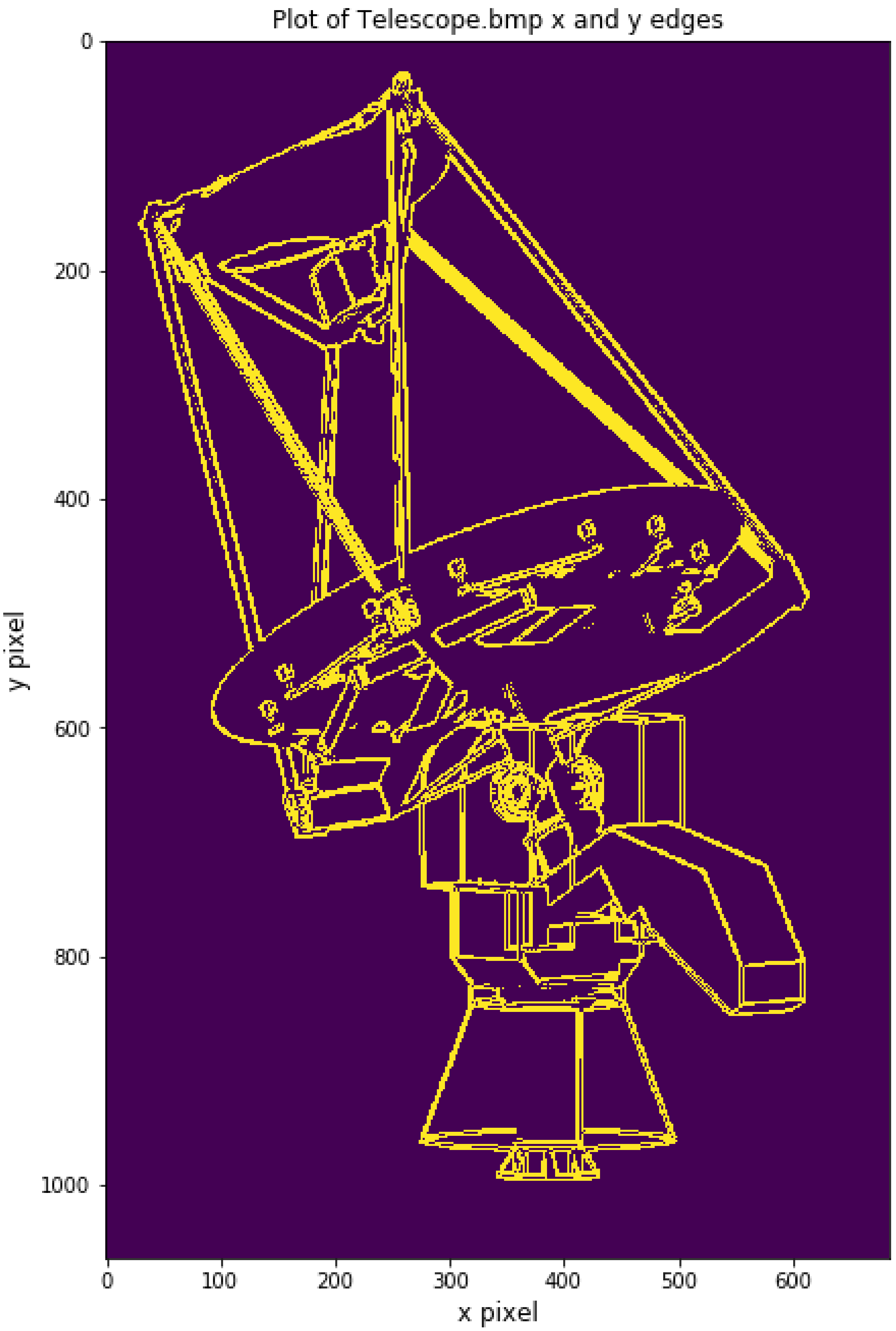
```
Time since last check is 0:00:00.483706
```

This “thresholded” image is a two-dimensional equivalent of the one-dimensional region of the gaussian which is greater that the threshold value above. We can therefore use a two-dimensional version of our “shift left and right” algorithm to find where the threshold is crossed, which will give as the edges of the thresholded figure. Working in two dimensions means we have to both shift left and right and shift up and down.

[109]:
```
# <!-- Student -->
#
import datetime
now = datetime.datetime.now()
print("Date and time",str(now))
#
imgMin = np.amin(img)
imgMax = np.amax(img)
print("Min intensity in image",imgMin,"max intensity",imgMax)
#
shiftR = 3
shiftC = 3
print("nRows",nRows,"nCols",nCols)
print("shiftR",shiftR,"shiftC",shiftC)
imgEdge = np.full((nRows, nCols), False)
imgEdge[0:nRows - shiftR, 0:nCols - shiftC] = \
   np.logical_or(np.logical_or(np.logical_and(imgThr[0:nRows - shiftR, 0:nCols␣
 ↪- shiftC],
                                             np.logical_not(imgThr[0:nRows -␣
 ↪shiftR, shiftC:nCols])),   # horizontal left
                               np.logical_and(np.logical_not(imgThr[0:nRows -␣
 ↪shiftR, 0:nCols - shiftC]),
                                             imgThr[0:nRows - shiftR, shiftC:
 ↪nCols])),                  # horizontal right
                 np.logical_or(np.logical_and(np.logical_not(imgThr[0:nRows -␣
 ↪shiftR, 0:nCols - shiftC]),
                                             imgThr[shiftR:nRows, 0:nCols -␣
 ↪shiftC]),                   # vertical bottom
                               np.logical_and(imgThr[0:nRows - shiftR, 0:nCols␣
 ↪- shiftC],
                                             np.logical_not(imgThr[shiftR:
 ↪nRows, 0:nCols - shiftC])))) # vertical top
imgEdge[0:nRows, 0] = False
imgEdge[0, 0:nCols] = False
imgEdge[0:nRows, nCols - shiftC - 1] = False
imgEdge[nRows - shiftR - 1, 0:nCols] = False
#
print(" ")
fig = plt.figure(figsize=(7.0, 11.0))
```

```python
plt.title("Plot of " +  imFile + " x and y edges", fontsize = 12)
plt.xlabel('x pixel', fontsize = 12)
plt.ylabel('y pixel', fontsize = 12)
plt.imshow(imgEdge)
plt.show()
#
then = now
now = datetime.datetime.now()
print(" ")
print("Date and time",str(now))
print("Time since last check is",str(now - then))
```

```
Date and time 2020-02-05 11:59:16.206986
Min intensity in image 0.0 max intensity 765.0
nRows 1064 nCols 684
shiftR 3 shiftC 3
```


Plot of Telescope.bmp x and y edges
y pixel
x pixel
0
200
400
600
800
1000
0
100
200
300
400
500
600

```
Date and time 2020-02-05 11:59:16.370514
Time since last check is 0:00:00.163528
```

9.4.1 Week 8 exercise 6

Copy the code cell above and change the parameters in it so that only horizontal edges in the image are identified. Does this mean you have to remove the "shift left and right" (shift columns) operation or the "shift up and down" (shift rows) operation from the image manipulation?

9.4.2 Week 8 exercise 7

Copy the relevant code cells from above and edit them so that you read in the image *penguin.jpg*. Adjust the threshold to get the clearest outline you can of the penguin in the image.

10 Introduction to Computational Physics - Week 9

10.1 Table of contents week 9

10.2 Introduction to week 9

This week, we will first see how we can turn the functions we have written in a Jupyter Notebook into a module that can be loaded and used in the same way as the numpy, matplotlib or other libraries. We will then look at how we can provide input to Python programs from the keyboard.

10.3 Module functions

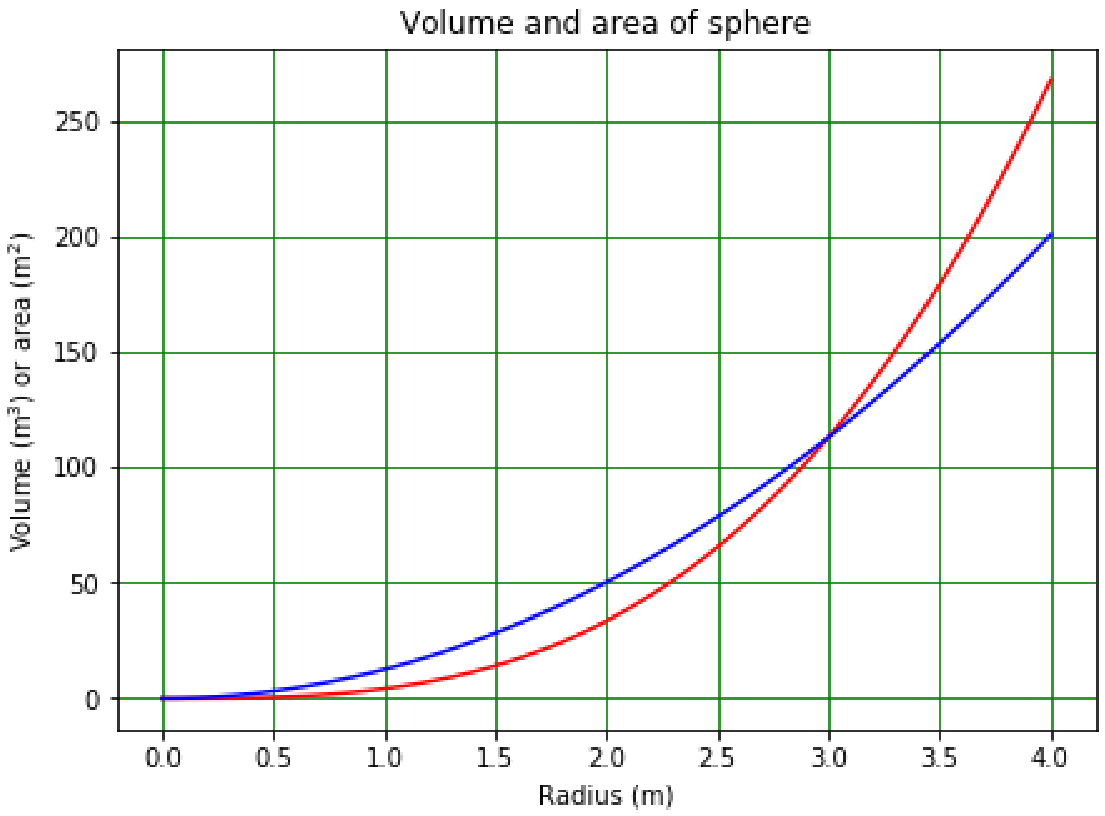
Let's start by writing a few functions that we can use to create a module.

[110]:
```
# <!-- Student -->
#
import numpy as np
#
def circleParams(r):
    '''
    Given the radius of a circle, this function returns its area and␣
↪circumference.
    '''
    A = np.pi*r**2
    c = 2*np.pi*r
    return A, c
#
def rectangleParams(h, w):
    '''
    Given the height and width of a rectangle, this function returns its area␣
↪and perimeter.
    '''
    A = h*w
    p = 2*(h + w)
    return A, p
#
def sphereParams(r):
    '''
    Given the radius of a sphere, this function returns its volume and its␣
↪surface area.
    '''
    V = 4/3*np.pi*r**3
    A = 4*np.pi*r**2
    return V, A
#
def rectPrismParams(h, w, d):
    '''
    Given the height, width and depth of a rectangular prism, this function␣
↪returns its volume,
    surface area and total side length.
    '''
    V = h*w*d
    A = 2*(h*w + w*d + h*d)
    s = 4*(h + w + d)
    return V, A, s
```

As we have seen many times, we can use these functions in the Notebook in which they are defined. For example, here is a plot of the volume, V, and area, A, of a sphere as a function of its radius, r.

[111]:
```
# <!-- Student -->
#
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
#
nArr = 50
rBot = 0.0
```

```
rTop = 4.0
rArr = np.linspace(rBot, rTop, nArr)
Varr, Aarr = sphereParams(rArr)
#
plt.figure(figsize = (7, 5))
plt.title("Volume and area of sphere")
plt.ylabel("Volume (m$^3$) or area (m$^2$)")
plt.xlabel("Radius (m)")
plt.plot(rArr, Varr, linestyle = '-', color = 'r')
plt.plot(rArr, Aarr, linestyle = '-', color = 'b')
plt.grid(color = 'green')
plt.show()
```



10.3.1 Week 9 exercise 1

Make a plot showing the volume, surface area and side length of a rectangular prism of width w and depth d as a function of its height, h. Use $w = 1.5\,\mathrm{m}$, $d = 2.5\,\mathrm{m}$ and h in the range $0 < h < 4.0\,\mathrm{m}$. Include a legend showing what each of the lines on the plot represents!

What we now want to do is to see how we can use these functions in a different Jupyter Notebook, without copying their definitions into that Notebook.

10.4 Creating and using a module

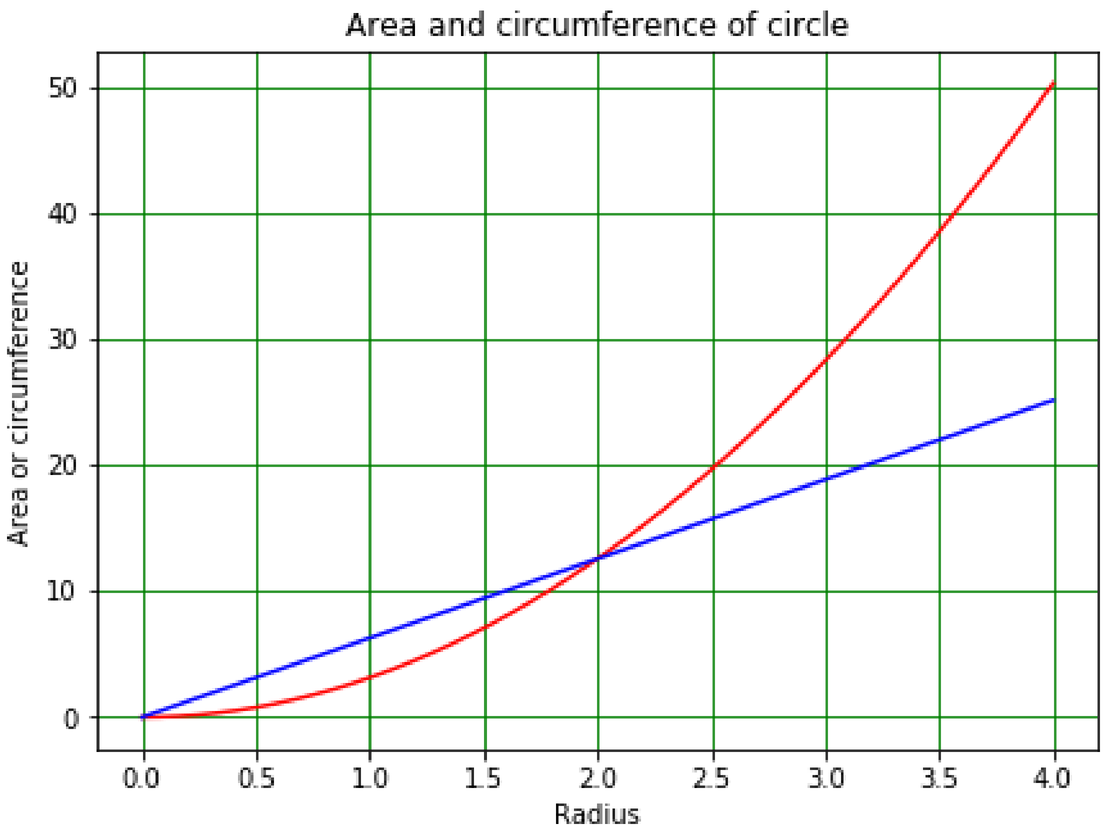
In order to create a module containing routines from this Notebook, click on the *File* menu, then on *Download as* and select *Python*. (If you are running Jupyter Lab rather than Jupyter Notebook, you need to use *File*, *Export Notebook As...* and then *Export Notebook to Executable Script*.) Depending on the security settings on your browser, you may get a warning about the file that is created, saying that it can damage your computer. You can ignore this and click *Keep* or *Save*. In your default download location (usually your *Downloads* folder) you will then have a file called *Phys105-Week09-Student.py*.

Move *Phys105-Week09-Student.py* into your working directory (the directory or folder which contains this Notebook) and rename it. Call it *Phys105W09.py*. Open *Phys105W09.py* by clicking on it in your Jupyter Notebook browser. You will see that it is just a copy of this Notebook written as Python code, i.e. all the Markdown cells have been turned into Python comments by sticking a "#" in front of them. Tidy up the file by deleting the superfluous comment lines - leave the ones that are useful! - and other material that isn't part of the functions. Do not delete the line that reads `import numpy as np`!

You can now use all the functions in *Phys105W09.py* by importing it as a module, as shown in the following example. (The reason you had to rename the file is that hyphens are not allowed in module names in Python, so you wouldn't be able to import the file if it was called *Phys105-Week09-Student.py*.) Note, your file name should have the extension *.py*, but you don't include this in the `import` statement.

Notice that, after doing `import Phys105W09 as ph`, we have called the routine `ph.circleParams` (with a `ph.` in front of the name to indicate it comes from the Phys105W09 module, cf. using `np.cos` to use the cosine function from the numpy library). The version of *circleParams* below is therefore that from the *Phys105W09* module, not the one defined in this Notebook!

[112]:
```python
# <!-- Student -->
#
import numpy as np
import matplotlib.pyplot as plt
import Phys105W09 as ph
#
nArr = 50
rBot = 0.0
rTop = 4.0
rArr = np.linspace(rBot, rTop, nArr)
Aarr, cArr = ph.circleParams(rArr)
#
plt.figure(figsize = (7, 5))
plt.title("Area and circumference of circle")
plt.ylabel("Area or circumference")
plt.xlabel("Radius")
plt.plot(rArr, Aarr, linestyle = '-', color = 'r')
plt.plot(rArr, cArr, linestyle = '-', color = 'b')
plt.grid(color = 'green')
plt.show()
```



We have been careful to include the statement `import numpy as np` at the top of *Phys105W09.py*. This statement is executed when the module is first loaded, so even if we use the functions in *Phys105W09.py* from a program which doesn't import numpy, they will work OK.

10.4.1 Week 9 exercise 2

Copy the code you used to solve exercise 1 into the cell below this one. Alter the code so that it produces a figure with two supbplots. On the left, use the version of *rectPrismParams* in this Notebook, on the right, the version from the module *Phys105W09*. Are the two graphs the same?

10.5 Accessing modules in other folders

What we have done so far only allows us to use functions from a module in the directory in which we are working. We can also get at modules in other directories. In order to try this, make a copy of *Phys105W09.py* and call it *Phys105W09new.py*. Make a new folder *Phys105lib* in your working directory. You can do this using *File Explorer* on Windows, *Finder* on a Mac or the command *mkdir* on a Linux system.) Move the file *Phys105W09new* into the folder *Phys105lib*. Now try to run the cell below.

[113]:
```
# <!-- Student -->
#
```

```
import numpy as np
import matplotlib.pyplot as plt
import Phys105W09new as phnew
#
nArr = 50
rBot = 0.0
rTop = 4.0
rArr = np.linspace(rBot, rTop, nArr)
Aarr, cArr = phnew.circleParams(rArr)
#
plt.figure(figsize = (7, 5))
plt.title("Area and circumference of circle")
plt.ylabel("Area or circumference")
plt.xlabel("Radius")
plt.plot(rArr, Aarr, linestyle = '-', color = 'r')
plt.plot(rArr, cArr, linestyle = '-', color = 'b')
plt.grid(color = 'green')
plt.show()
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
<ipython-input-113-ea0025ae351d> in <module>
      3 import numpy as np
      4 import matplotlib.pyplot as plt
----> 5 import Phys105W09new as phnew
      6 #
      7 nArr = 50

ModuleNotFoundError: No module named 'Phys105W09new'
```

Python can't find the *Phys105W19new* module, because it only looks for it in the current working directory and in directories specified by a system variable called `path`. We can see which directories are in `path` using the `sys.path` command, after we have imported the `sys` module, as follows.

[114]:
```
# <!-- Student -->
#
import sys
#
print("Directories in path are:\n",sys.path)
```

```
Directories in path are:
 ['C:\\Users\\green\\OneDrive\\OneDocuments\\Liverpool\\Teaching\\Phys105-Comp01
-2019\\Phys105-All2019', 'C:\\Users\\green\\Anaconda3\\python37.zip',
'C:\\Users\\green\\Anaconda3\\DLLs', 'C:\\Users\\green\\Anaconda3\\lib',
'C:\\Users\\green\\Anaconda3', '', 'C:\\Users\\green\\Anaconda3\\lib\\site-
packages', 'C:\\Users\\green\\Anaconda3\\lib\\site-packages\\win32',
'C:\\Users\\green\\Anaconda3\\lib\\site-packages\\win32\\lib',
'C:\\Users\\green\\Anaconda3\\lib\\site-packages\\Pythonwin',
'C:\\Users\\green\\Anaconda3\\lib\\site-packages\\IPython\\extensions',
'C:\\Users\\green\\.ipython']
```

The `path` variable is set up when Anaconda is installed. Exactly what you see will depend on your computer's operating system and where Anaconda was installed. The `path` entries will always have the structure *top_level/second_level/third_level*, and this is what you will see on a Macintosh or a Linux system. On a Windows computer, the forward slashes (/) will be replaced by back-slashes (\). These have to be represented by a double back-slash, as the first backslash is treated as an escape character (in both Python and Markdown).

If we want to temporarily add a new directory to `path`, we can do it using `path.append` from the `sys` module as follows.

[115]:
```
# <!-- Student -->
#
sys.path.append('Phys105lib')
print("Directories in path are:\n",sys.path)
```

```
Directories in path are:
 ['C:\\Users\\green\\OneDrive\\OneDocuments\\Liverpool\\Teaching\\Phys105-Comp01
-2019\\Phys105-All2019', 'C:\\Users\\green\\Anaconda3\\python37.zip',
'C:\\Users\\green\\Anaconda3\\DLLs', 'C:\\Users\\green\\Anaconda3\\lib',
'C:\\Users\\green\\Anaconda3', '', 'C:\\Users\\green\\Anaconda3\\lib\\site-
packages', 'C:\\Users\\green\\Anaconda3\\lib\\site-packages\\win32',
'C:\\Users\\green\\Anaconda3\\lib\\site-packages\\win32\\lib',
'C:\\Users\\green\\Anaconda3\\lib\\site-packages\\Pythonwin',
'C:\\Users\\green\\Anaconda3\\lib\\site-packages\\IPython\\extensions',
'C:\\Users\\green\\.ipython', 'Phys105lib']
```

You will see that *Phys105lib* has now been added to *path*. Now `import Phys105W09new as phnew` will work.

10.5.1 Week 9 exercise 3

Copy the cell which uses the routine `phnew.circleParams` above and insert it below. Run it to prove that your modified *path* variable is doing what it should!

The addition to `path` we have made above will allow us to use anything in the module *Phys105lib* if it is in our current working directory. If we want to be able to use routines from *Phys105lib* from *any* directory, we have to add the full description of its location to `path`. On my computer, this implies…

[116]:
```
# <!-- Student -->
#
import sys
#
sys.path.append('C:/Users/green/OneDrive/OneDocuments/Liverpool/Teaching/
 ↪Phys105-Comp01-2019/Phys105-Week09/Phys105lib')
print("Directories in path are:\n",sys.path)
```

```
Directories in path are:
 ['C:\\Users\\green\\OneDrive\\OneDocuments\\Liverpool\\Teaching\\Phys105-Comp01
-2019\\Phys105-All2019', 'C:\\Users\\green\\Anaconda3\\python37.zip',
'C:\\Users\\green\\Anaconda3\\DLLs', 'C:\\Users\\green\\Anaconda3\\lib',
'C:\\Users\\green\\Anaconda3', '', 'C:\\Users\\green\\Anaconda3\\lib\\site-
packages', 'C:\\Users\\green\\Anaconda3\\lib\\site-packages\\win32',
'C:\\Users\\green\\Anaconda3\\lib\\site-packages\\win32\\lib',
'C:\\Users\\green\\Anaconda3\\lib\\site-packages\\Pythonwin',
'C:\\Users\\green\\Anaconda3\\lib\\site-packages\\IPython\\extensions',
'C:\\Users\\green\\.ipython', 'Phys105lib', 'C:/Users/green/OneDrive/OneDocument
s/Liverpool/Teaching/Phys105-Comp01-2019/Phys105-Week09/Phys105lib']
```

…which is a bit of a mouthful. (You can work out what the full description of the location of *Phys105lib* should be on your computer by looking at the existing entries in your `path` variable.) Notice that I can use forward slashes in the `sys.path.append` command; Python changes these to the format that is relevant for my operating system. (Because I am working on a Windows machine, I could have used the double back-slash notation, it's just a bit clumsier.)

There are (system dependent) ways of permanently adding folders like `mylib` to `path`, but getting this wrong can cause problems, so we will use the above method. The downside is that before using any of the routines in the library `mylib`, we have to include the statement:

```
import sys
sys.path.append('path to mylib')
```

The upside is that when we shut down our Jupyter Notebook, or restart the kernel, `path` returns to its original value and we don't influence how anything else on the computer works.

10.6 Keyboard input to Python

Python programs can read input from the keyboard. They do this with the function `input()`, as is shown in the following example.

```
[1]: # <!-- Student -->
     #
     name = input("What's your name?")
     print("Nice to meet you " + name + "!")
     age = input("How old are you?")
     print("So, you are already",age,"years old",name,"\b!")
```

```
What's your name? Tim

Nice to meet you Tim!

How old are you? 123

So, you are already 123 years old Tim!
```

We can check the type of the input as below:

```
[2]: # <!-- Student -->
     #
```

```
print("Type of name is",type(name))
print("Type of age is",type(age))
```

```
Type of name is <class 'str'>
Type of age is <class 'str'>
```

Everything (whether numbers or letters) is read as strings. This means the following code will not work (try it!):

```
[3]: # <!-- Student -->
     #
     retirementAge = 67
     #
     name = input("What's your name?")
     print("Nice to meet you " + name + "!")
     age = input("How old are you?")
     #
     retireIn = retirementAge - age
     if retireIn > 0:
         print("I guess you will retire in",retireIn,"years,",name,"\b.")
     else:
         print("I guess you retired",-retireIn,"years ago,",name,"\b.")
```

```
What's your name? Tim

Nice to meet you Tim!

How old are you? 123
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-3-29728eb4f808> in <module>
      7 age = input("How old are you?")
      8 #
----> 9 retireIn = retirementAge - age
     10 if retireIn > 0:
     11     print("I guess you will retire in",retireIn,"years,",name,"\b.")

TypeError: unsupported operand type(s) for -: 'int' and 'str'
```

10.6.1 Week 9 exercise 4

Copy the above cell and insert it below this one. Modify the code so that it runs wthout errors!

10.6.2 Week 9 exercise 5

Copy your answer to exercise 4 into the cell below this one and modify it so it checks that the age entered is in the range $0 < \text{age} < 120$ years. If the age is out of range, ask that it be re-entered. If an out-of-range answer is given more than 3 times in a row, print an error message and stop the program!

Hint 1 You could do this using the `while`, `break` and `else` control structure, or using a `for` loop. Remember also the `sys.exit()` command we have mentioned in previous weeks!

Hint 2 Remember that Python reads the input as a string. Does the statement `if age > 0` when `age` is a string make sense?

11 Introduction to Computational Physics - Week 10

11.1 Table of contents week 10

11.2 Introduction to week 10

This week we will have a quick look at the Symbolic Python or sympy library, which is described in full here. This library provides Python tools for solving equations, differentiating and integrating functions, expanding functions as series and other symbolic manipulations.

11.3 Sympy introduction - defining equations

In order to use sympy, we have to import it in the same way we import numpy. Let's do that and then look at a first example.

[4]:
```
# <!-- Student -->
#
import sympy as sp
#
x, a, b, c = sp.symbols('x a b c')
quadEq =  sp.Eq(a*x**2 + b*x + c, 0)
```

```
print("The quadratic equation is",quadEq)
```

```
The quadratic equation is Eq(a*x**2 + b*x + c, 0)
```

As we can see above, the first thing we have done is defined the symbols we want to use. The statement:

```
x, a, b, c = sp.symbols('x a b c')
```

tells sympy that `x`, `a`, `b` and `c` are to be treated as algebraic symbols. (These don't need to be single letters, for example, we could define `theta = sp.symbols('theta')`.)

Our next step was to define the quadratic equation $ax^2 + bx + c = 0$. To do this, we can't write `a*x**2 + b*x + c = 0`, because Python uses the `=` character to assign values to variables (e.g. `y = 18.0` or `y = np.cos(1.3)`). We also can't define the quadratic equation by writing `a*x**2 + b*x + c == 0`, because `==` tests whether what's to its left is identical to whatever's to its right and returns the value `True` or `False` accordingly. Hence the syntax:

```
sp.Eq(a*x**2 + b*x + c, 0)
```

is used to indicate that the left hand side (LHS) `a*x**2 + b*x + c` is equal to the right hand side (RHS) `0`.

11.4 Solving equations

Having defined an equation, can we solve it?

[5]:
```
# <!-- Student -->
#
import sympy as sp
quadSol = sp.solve(quadEq, x)
print("Solution of",quadEq,"is x =",quadSol)
```

```
Solution of Eq(a*x**2 + b*x + c, 0) is x = [(-b + sqrt(-4*a*c + b**2))/(2*a),
-(b + sqrt(-4*a*c + b**2))/(2*a)]
```

The `sp.solve` statement solves the equation we have set up (`sp.solves`'s first argument) for the variable indicated (`sp.solve`'s second argument), and we see that it has returned the expected solutions for the quadratic equation:

$$x = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

and

$$x = \frac{-b - \sqrt{b^2 - 4ac}}{2a}.$$

We can find out from sympy how many solutions there are and get at the solutions separately as follows.

[6]:
```
# <!-- Student -->
#
nQuadSols = len(quadSol)
```

```
for n in range(0, nQuadSols):
    print("Solution",n,"is",quadSol[n])
```

```
Solution 0 is (-b + sqrt(-4*a*c + b**2))/(2*a)
Solution 1 is -(b + sqrt(-4*a*c + b**2))/(2*a)
```

Note that in order to use `sp.solve`, the RHS of the equation has to be zero. For example, if we were given the equation $ax^3 + bx^2 = cx$ to solve, we would first have to rewrite it as $ax^3 + bx^2 - cx = 0$, then we could determine the solution as follows.

[7]:
```
# <!-- Student -->
#
cubicSol = sp.solve(a*x**3 + b*x**2 - c*x, x)
nCubicSols = len(cubicSol)
for n in range(0, nCubicSols):
    print("Solution",n,"is",cubicSol[n])
```

```
Solution 0 is 0
Solution 1 is -b/(2*a) - sqrt(4*a*c + b**2)/(2*a)
Solution 2 is -b/(2*a) + sqrt(4*a*c + b**2)/(2*a)
```

11.4.1 Week 10 exercise 1

Use sympy to solve the equation $4 \sin\theta + 5 \cos\theta = 1$.

Note, as we have done above (e.g. `quadSol = sp.solve(quadEq, x)`) we can assign a symbolic expression to a variable. Here is another example.

[8]:
```
# <!-- Student -->
#
y = a*x**2 + b*x + c
print("Solution of",y,"= 0 is x =",sp.solve(y, x))
```

```
Solution of a*x**2 + b*x + c = 0 is x = [(-b + sqrt(-4*a*c + b**2))/(2*a), -(b +
sqrt(-4*a*c + b**2))/(2*a)]
```

This does not define a function $y(x)$, it just assigns the symbolic expression `a*x**2 + b*x + c` to the variable `y`. The following expression will therefore produce an error message.

[9]:
```
# <!-- Student -->
#
print("y(2) =",y(2))
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-9-3a32d7c3435a> in <module>
      1 # <!-- Student -->
      2 #
----> 3 print("y(2) =",y(2))
```

```
TypeError: 'Add' object is not callable
```

If we want to define a symbolic function, we can use the normal Python syntax for creating functions (having defined the relevant symbols):

[10]:
```
# <!-- Student -->
#
x, a, b, c = sp.symbols('x a b c')
#
def quad(x):
    f = a*x**2 + b*x + c
    return f
#
print("Solution of quad(x) = 0 is x =",sp.solve(quad(x), x))
```

```
Solution of quad(x) = 0 is x = [(-b + sqrt(-4*a*c + b**2))/(2*a), -(b +
sqrt(-4*a*c + b**2))/(2*a)]
```

We can now use this Python function to evaluate `quad(x)` for a specific value of `x`:

[11]:
```
# <!-- Student -->
#
print("quad(2) =",quad(2))
```

```
quad(2) = 4*a + 2*b + c
```

Note that we get a symbolic answer, as `a`, `b` and `c` are symbols. We can check this, e.g. for `a`, by looking at the type of `a`:

[12]:
```
# <!-- Student -->
#
print("Type of a is",type(a))
```

```
Type of a is <class 'sympy.core.symbol.Symbol'>
```

If we assign numerical values to `a`, `b` and `c`, `quad(2)` will return numbers for the solutions of the equation:

[13]:
```
# <!-- Student -->
#
a = 2
b = 3
c = 1
print("quad(2) =",quad(2))
print("Solution of quad(x) = 0 is x =",sp.solve(quad(x), x))
```

```
quad(2) = 15
Solution of quad(x) = 0 is x = [-1, -1/2]
```

The disadvantage of giving `a`, `b` and `c` specific values in this way is that we can no longer use them as symbolic objects. E.g. if we re-check the type of `a`, we will find that it is now an integer.

[14]:
```
# <!-- Student -->
#
print("Type of a after assigning integer value is",type(a))
```

```
Type of a after assigning integer value is <class 'int'>
```

There is an alternative method. Let's first turn `a`, `b` and `c` back into symbols and check that they are behaving as we want them to.

[15]:
```
# <!-- Student -->
#
a, b, c = sp.symbols('a b c')
print("Type of a after setting as symbol is",type(a))
print("quad(2) =",quad(2))
```

```
Type of a after setting as symbol is <class 'sympy.core.symbol.Symbol'>
quad(2) = 4*a + 2*b + c
```

Now substitute the numerical value 3 for `a` using `subs(a, 3)`. We can check that `a` is given the value 3 when we do this, and that `a` is still a symbol, as follows:

[16]:
```
# <!-- Student -->
print("quad(x).subs(a, 3) =",quad(x).subs(a, 3))
print("quad(2).subs(a, 3) =",quad(2).subs(a, 3))
print("Type of a is",type(a))
```

```
quad(x).subs(a, 3) = b*x + c + 3*x**2
quad(2).subs(a, 3) = 2*b + c + 12
Type of a is <class 'sympy.core.symbol.Symbol'>
```

We can substitute more than one numerical value into an expression:

[17]:
```
# <!-- Student -->
#
print("quad(x).subs(a, 3).subs(b = -2.4).subs(c = 17/3) =",quad(x).subs(a, 3).
 ↪subs(b, -2.4).subs(c, 17/3))
print("quad(2).subs(a, 3).subs(b = -2.4).subs(c = 17/3) =",quad(2).subs(a, 3).
 ↪subs(b, -2.4).subs(c, 17/3))
```

```
quad(x).subs(a, 3).subs(b = -2.4).subs(c = 17/3) = 3*x**2 - 2.4*x +
5.66666666666667
quad(2).subs(a, 3).subs(b = -2.4).subs(c = 17/3) = 12.8666666666667
```

We can also substitute another symbolic value:

[18]:
```
# <!-- Student -->
#
print("quad(x).subs(a, x) =",quad(x).subs(a, x))
print("quad(x).subs(a, y) =",quad(x).subs(a, y))
```

```
quad(x).subs(a, x) = b*x + c + x**3
quad(x).subs(a, y) = b*x + c + x**2*(a*x**2 + b*x + c)
```

11.5 Solveset and the future of solve

In time, the authors of sympy hope to replace `sp.solve` with a new routine, `sp.solveset`(see here for a description), which is used in a similar but not quite identical way. For the moment `sp.solve` can cope with more types of equations than `sp.solveset`. An example using `sp.solveset` is shown below. Notice that it removes the restriction that the equation must be presented with the RHS being zero.

```
[19]: # <!-- Student -->
#
z, p, q, r = sp.symbols('z p q r')
newQuadEq =  sp.Eq(p*z**2 + q*z, -r)
print("This quadratic equation is",newQuadEq)
#
newQuadSol = sp.solveset(newQuadEq, z)
print("Its solution is",newQuadSol)
```

```
This quadratic equation is Eq(p*z**2 + q*z, -r)
Its solution is FiniteSet(-q/(2*p) - sqrt(-4*p*r + q**2)/(2*p), -q/(2*p) +
sqrt(-4*p*r + q**2)/(2*p))
```

The curly brackets indicate that the solutions are not given as a list as with `sp.solve`, but as (for us) a new data type, a set. This allows `sp.solveset` to deal with situations where there is an infinite number of solutions, but means these have to be accessed in a slightly different way. If the number of solutions is finite, the following works.

```
[20]: # <!-- Student -->
#
nSol = 0
for sol in newQuadSol:
    print("Solution",nSol,"is",sol)
    nSol += 1
```

```
Solution 0 is -q/(2*p) - sqrt(-4*p*r + q**2)/(2*p)
Solution 1 is -q/(2*p) + sqrt(-4*p*r + q**2)/(2*p)
```

11.6 Fractions

Fractions (rational numbers) can be entered as below.

```
[21]: # <!-- Student -->
#
alpha = sp.Rational(1, 137)
print("alpha =",alpha)
```

```
alpha = 1/137
```

They can be added, subtracted, multiplied and divided.

[22]:
```
# <!-- Student -->
#
half = sp.Rational(1, 2)
quarter = sp.Rational(1, 4)
print("half + quarter =",half + quarter)
print("half - quarter =",half - quarter)
print("half*quarter =",half*quarter)
print("half/quarter =",half/quarter)
```

```
half + quarter = 3/4
half - quarter = 1/4
half*quarter = 1/8
half/quarter = 2
```

Of course, you don't have to assign variable names to fractions before manipulating them, and fractions can be turned into floats.

[23]:
```
# <!-- Student -->
#
ans = sp.Rational(3, 137) - sp.Rational(17, 13)
print("ans =",ans)
print("ans as float =",float(ans))
```

```
ans = -2290/1781
ans as float = -1.2857944974733295
```

11.7 Differentiation

Sympy provides a routine for differentiating. You must give it the expression you want to differentiate and the variable with respect to which the differentiation should be peformed, as shown below.

[24]:
```
# <!-- Student -->
#
print("The differential of a*x**5 + b*x**2 + c/x**3 w.r.t. x is",sp.diff(a*x**5␣
 ↪+ b*x**2 + c/x**3, x))
print("The differential of quad(x) w.r.t. x is",sp.diff(quad(x), x))
```

```
The differential of a*x**5 + b*x**2 + c/x**3 w.r.t. x is 5*a*x**4 + 2*b*x -
3*c/x**4
The differential of quad(x) w.r.t. x is 2*a*x + b
```

The second (and higher) differentials can also be calculated (in several ways).

[25]:
```
# <!-- Student -->
#
print("Second differential is",sp.diff(sp.diff(a*x**5 + b*x**2 + c/x**3, x),x))
print("Second differential is",sp.diff(a*x**5 + b*x**2 + c/x**3, x, x))
print("Second differential is",sp.diff(a*x**5 + b*x**2 + c/x**3, x, 2))
```

```
Second differential is 20*a*x**3 + 2*b + 12*c/x**5
Second differential is 2*(10*a*x**3 + b + 6*c/x**5)
Second differential is 2*(10*a*x**3 + b + 6*c/x**5)
```

Notice the output from the first of these methods is written differently to that from the latter two, but we can check they are the same by subtracting them and using `sp.simplify` to simplify the result!

[26]:
```
# <!-- Student -->
#
sp.simplify(sp.diff(sp.diff(a*x**5 + b*x**2 + c/x**3, x),x) - sp.diff(a*x**5 +␣
 ↪b*x**2 + c/x**3, x, 2))
```

[26]:
```
0
```

11.7.1 Week 10 exercise 2

Find the first derivative w.r.t x of the function:

$$s(x) = A\left(\tanh\left[\frac{x-x_0}{L}+\frac{w}{2L}\right]-\tanh\left[\frac{x-x_0}{R}-\frac{w}{2R}\right]\right).$$

Using the values $A = 2.0$, $x_0 = 0.5$, $L = 1.1$, $R = 0.5$ and $w = 0.4$, plot the function and its derivative in the range $-2 < x < 3$ using numpy and matplotlib.

Hint, you can copy the output from the differentiation and paste it into your code for doing the plotting. It will need some editing, but not much! This can save you a lot of time and helps avoid typos.

11.8 Integration

The syntax for performing indefinite integration (i.e. without limits) is similar to that for differentiation. Note that sympy does not add a constant of integration to indefinite integrals, you have to think about what you need to do about these yourself!

[27]:
```
# <!-- Student -->
#
intQuad = sp.integrate(quad(x), x)
print("Integral of",quad(x),"w.r.t. x is",intQuad)
```

```
Integral of a*x**2 + b*x + c w.r.t. x is a*x**3/3 + b*x**2/2 + c*x
```

If we want to evaluate this integral with particular values of `a`, `b`, `c` and `x` we must `sp.subs` all of these, including `x`, as `intQuad` is not a function.

[28]:
```
# <!-- Student -->
#
```

```
print("Integral of",quad(x),"with a = 1, b = 2, c = -1 and x = 4 is",intQuad.
↪subs(a, 1).subs(b, 2).subs(c, -1).subs(x, 4))
```

```
Integral of a*x**2 + b*x + c with a = 1, b = 2, c = -1 and x = 4 is 100/3
```

If you want to do a definite integral, give `sp.integrate` a tuple containing the variable and the range over which the integration should take place.

```
[29]: # <!-- Student -->
#
intQuadLim = sp.integrate(quad(x), (x, -1, 2))
print("Integral of",quad(x),"w.r.t. x over range -1 to 2 is",intQuadLim)
```

```
Integral of a*x**2 + b*x + c w.r.t. x over range -1 to 2 is 3*a + 3*b/2 + 3*c
```

Numerical values of `a`, `b`, and `c` can be provided as previously described. The substitution can be made before or after the integration.

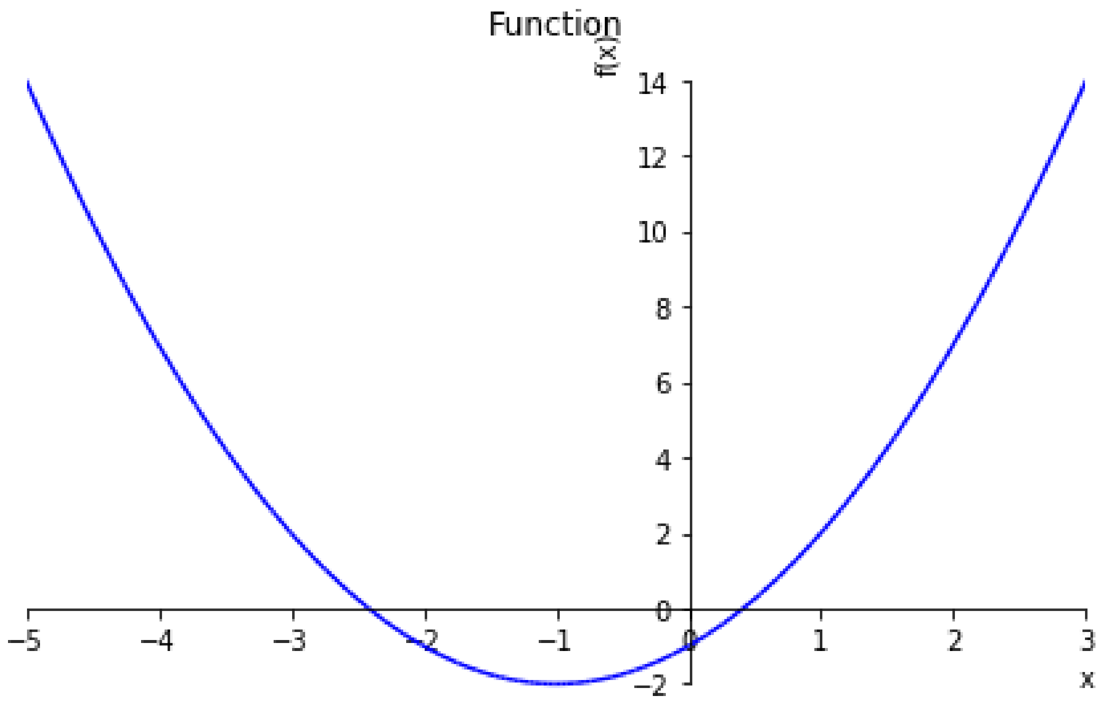
```
[30]: # <!-- Student -->
#
intQuadLim1 = sp.integrate(quad(x).subs(a, 1).subs(b, 2).subs(c, -1), (x, -1,␣
↪2))
intQuadLim2 = sp.integrate(quad(x), (x, -1, 2)).subs(a, 1).subs(b, 2).subs(c,␣
↪-1)
print("Integral of",quad(x),"w.r.t. x over range -2 to 2 is",intQuadLim1)
print("Integral of",quad(x),"w.r.t. x over range -2 to 2 is",intQuadLim2)
```

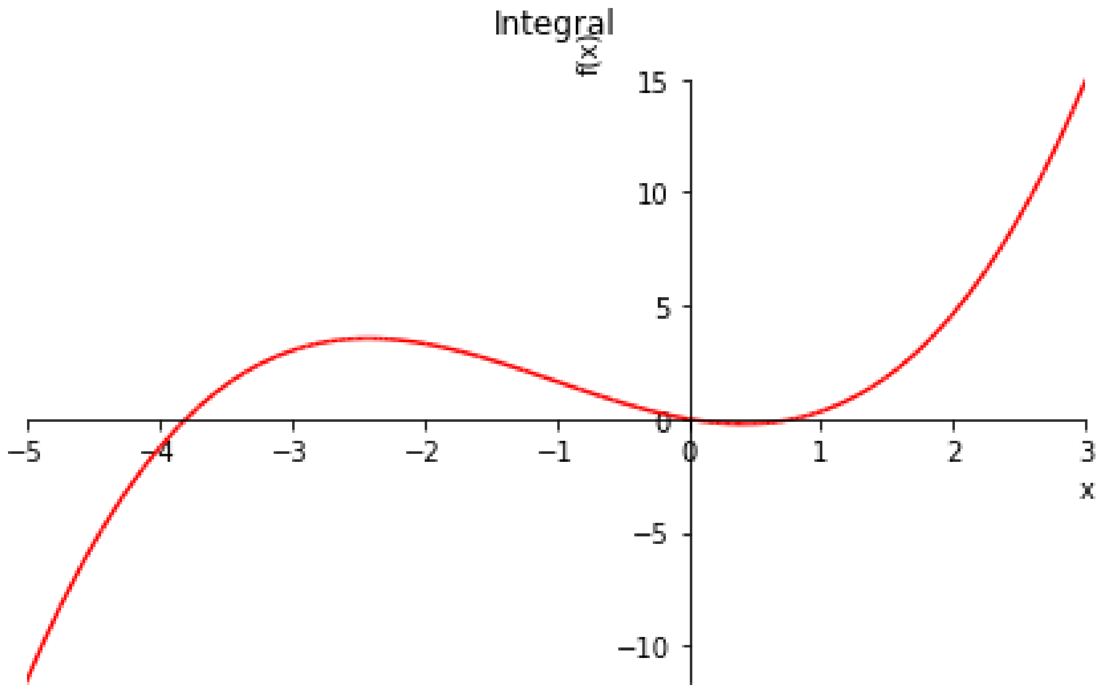
```
Integral of a*x**2 + b*x + c w.r.t. x over range -2 to 2 is 3
Integral of a*x**2 + b*x + c w.r.t. x over range -2 to 2 is 3
```

11.9 Plotting with sympy

We can demonstrate the plotting capabilities of sympy by plotting `quad` and its integral. Sympy does this via an interface to matplotlib, but the syntax for making the plots is a little different to that we have seen so far. Here are the two plots as two separate figures.

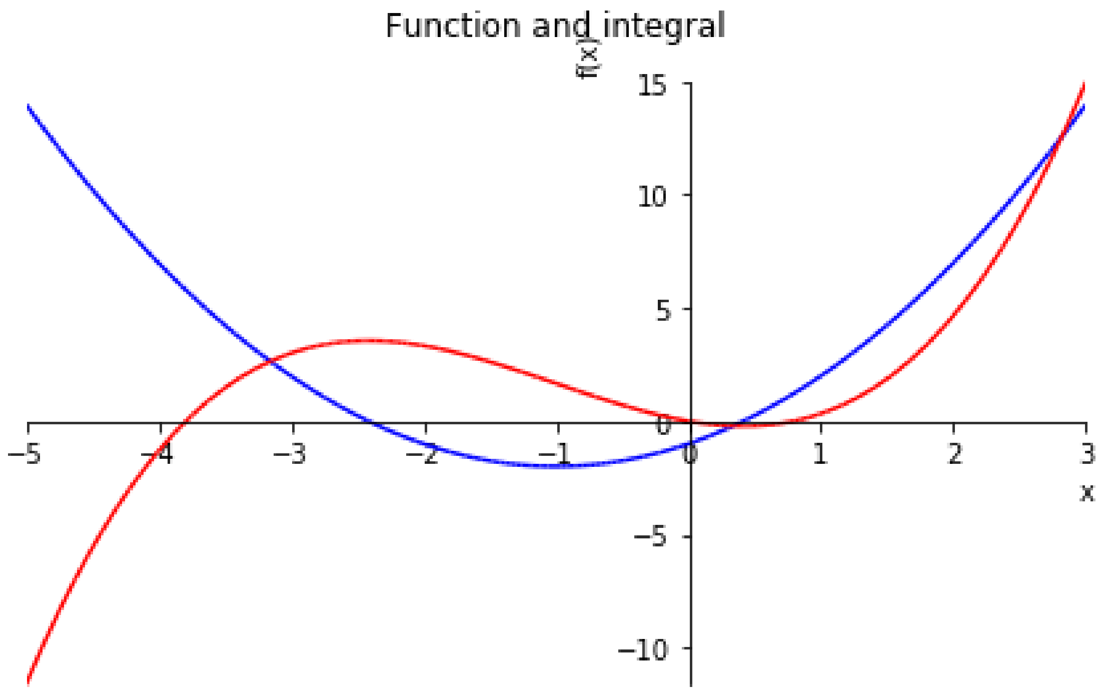
```
[31]: # <!-- Student -->
#
import matplotlib.pyplot as plt
%matplotlib inline
#
sp.plot(quad(x).subs(a, 1).subs(b, 2).subs(c, -1),(x, -5, 3), line_color =␣
↪'blue', title = "Function")
sp.plot(intQuad.subs(a, 1).subs(b, 2).subs(c, -1).subs(x, x),(x, -5, 3),␣
↪line_color = 'red', title = "Integral")
plt.show()
```


Function
f(x)
14
12
10
8
6
4
2
0
−2
−5
−4
−3
−2
−1
0
1
2
3
x


Integral
f(x)
15
10
5
0
−5
−10
−5
−4
−3
−2
−1
0
1
2
3
x

And here they are plotted together. (Getting two different line colours is a little convoluted!)

[32]:
```
# <!-- Student -->
#
thisPlot = sp.plot(quad(x).subs(a, 1).subs(b, 2).subs(c, -1),
                   intQuad.subs(a, 1).subs(b, 2).subs(c, -1).subs(x, x),
                   (x, -5, 3), show = False, title = "Function and integral")
thisPlot[0].line_color = 'blue'
thisPlot[1].line_color = 'red'
thisPlot.show()
```



11.10 Solving differential equations

A range of ordinary differential equations can be solved with sympy using `dsolve`. We will look at the equation:

$$\frac{dg}{dt} = \exp(-t) - g,$$

with the inital condition $g(0) = 1$. In order to set up this, or any, differential equation in sympy, we first have to define the function we want to solve for, in this case g. As we don't know what this will be until we have solved the equation, we can't write down an explicit form for it. Sympy allows us to define a general function using `sp.Function` as below. When defined like this, g can be a function of anything and can have any functional form. We have to make sure we refer to it consistently; in this example, we must refer to it explicitly as a function of t.

[33]:
```
# <!-- Student -->
#
t = sp.symbols("t")
g = sp.Function("g")
diffEq = sp.Eq(g(t).diff(t), sp.exp(-t) - g(t))
solDiffEq = sp.dsolve(diffEq, g(t))
print("Solution of differential equation is",solDiffEq)
```

```
Solution of differential equation is Eq(g(t), (C1 + t)*exp(-t))
```

Notice how we have defined the derivative w.r.t. t in the above: `g(t).diff(t)`.

An alternative method of solving the equation requires that we rewrite it so that the RHS is zero:

$$\frac{dg}{dt} - \exp(-t) + g = 0.$$

We can then solve it as below. Note also the alternative way of defining $\frac{d}{dt}g(t)$.

[34]:
```
# <!-- Student -->
#
g_ = sp.Derivative(g(t), t)
print("Solution of differential equation is",sp.dsolve(g_ - sp.exp(-t) + g(t),␣
 ↪g(t)))
```

```
Solution of differential equation is Eq(g(t), (C1 + t)*exp(-t))
```

Which ever method we choose, we now have to find the arbitrary constant `C1`. This can be done using symbolic methods as below. Note that the statement `Eq(1, C1)` is an equation that tells us $1 = C1$, as described above!

[35]:
```
# <!-- Student -->
#
equationForC1 = solDiffEq.subs([(t, 0), (g(0), 1)])
print("Equation for C1 is",equationForC1)
```

```
Equation for C1 is Eq(1, C1)
```

11.10.1 Week 10 optional exercise

Solve symbolically the equation that describes radioactive decay,

$$\frac{dN}{dt} = -\frac{N}{\tau},$$

using the fact that at time $t = 0$ the number of nuclei is N_0.

(This equation describes the situation in which the number of decays per unit time in a sample is proportional to the number of remaining nuclei, N. Hence the rate of change in N with time, $\frac{dN}{dt}$ is proportional to $-N$.)

11.11 Series

Sympy knows about Taylor series. To use them, we must specify the function we want to express as a series and the variable in which we want the expansion to be done. We can also specify the point around which the expansion should be made ($\pi/4$ in the example below) and the maximum power in the expansion (the 3 below means we get terms up to the power 2). Look at the documentation for more information! The example is the determination of the Taylor expansion for the sine function.

[36]:
```
# <!-- Student -->
sinSeries = sp.sin(x).series(x, sp.pi/4, 3)
print("First terms in Taylor series for sine function about pi/4 are",sinSeries)
```

```
First terms in Taylor series for sine function about pi/4 are sqrt(2)/2 +
sqrt(2)*(x - pi/4)/2 - sqrt(2)*(x - pi/4)**2/4 + O((x - pi/4)**3, (x, pi/4))
```

11.11.1 Week 10 exercise 3

Using numpy and matplotlib, plot a graph showing a cosine function in the θ range $0...\pi$. Superimpose on this a plot of the sum of the Taylor series for the cosine up to and including the θ^3 term, expanded about $\pi/2$.

11.12 Matrices and linear equations

This section is here for information only as you have yet to look at matrices in maths!

Matrices can be defined and inverted as shown here.

[37]:
```
# <!-- Student -->
#
A = sp.Matrix(([3, 7], [4, -2]))
invA = A.inv()
AinvA = A*invA
invAA = invA*A
print("This is the matrix A\n",A)
print("Its inverse is\n",invA)
print("A multplied by its inverse is\n",AinvA)
print("The inverse of A multplied y A is\n",invAA)
print("As expected, both multiplications give the unit matrix!")
```

```
This is the matrix A
 Matrix([[3, 7], [4, -2]])
Its inverse is
 Matrix([[1/17, 7/34], [2/17, -3/34]])
A multplied by its inverse is
 Matrix([[1, 0], [0, 1]])
The inverse of A multplied y A is
```

```
 Matrix([[1, 0], [0, 1]])
As expected, both multiplications give the unit matrix!
```

Systems of linear equations can be defined and solved using matrices. For example, consider the equations:

$$\begin{aligned} 3x + 4y &= 7 \\ x - 3y &= 2. \end{aligned}$$

These can be written:

$$\begin{pmatrix} 3 & 4 \\ 1 & -3 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 7 \\ 2 \end{pmatrix}$$

Defining $M = \begin{pmatrix} 3 & 4 \\ 1 & -3 \end{pmatrix}$, $\vec{x} = \begin{pmatrix} x \\ y \end{pmatrix}$ and $\vec{c} = \begin{pmatrix} 7 \\ 2 \end{pmatrix}$, we have $M\vec{x} = \vec{c}$. The solution is obtained by multiplying through by M^{-1}, the inverse of M, and using the fact that $M^{-1}M = I$, the unit matrix:

$$\begin{aligned} M^{-1}M\vec{x} &= M^{-1}\vec{c} \\ \vec{x} &= M^{-1}\vec{c}. \end{aligned}$$

Using sympy, this can be done as follows. (Note we start by redefining x and y so we do not use the assignments above!)

[38]:
```python
# <!-- Student -->
#
x, y = sp.symbols('x y')
M = sp.Matrix(([3, 4], [1, -3]))
X = sp.Matrix((x, y))
C = sp.Matrix((7, 2))
invM = M.inv()
solVect = invM*C
print("Equations to solve",M*X,'=',C)
print("Solution is",X,"=",solVect)
```

```
Equations to solve Matrix([[3*x + 4*y], [x - 3*y]]) = Matrix([[7], [2]])
Solution is Matrix([[x], [y]]) = Matrix([[29/13], [1/13]])
```

The solution is therefore:

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} \frac{29}{13} \\ \frac{1}{13} \end{pmatrix},$$

or

$$x = \frac{29}{13}$$
$$y = \frac{1}{13}.$$

This system of equations can of course easily be solved by hand, but sympy will cope with systems of equations that would cause even Bradley Cheal to despair!

11.13 Odds and ends

You can print equations from sympy in LaTeX format, so they can be copied and pasted into Markdown cells or documents. An example is shown below.

```
[39]: # <!-- Student -->
      #
      # For TJG: must comment folliwing line out when merging notebooks to form␣
       ↪summary!
      #print(sp.latex(sp.Integral(sp.sqrt(1/x), x)))
```

You can also get sympy to write its results to the screen in a more legible format by giving the command `sp.init_printing()`. This looks on your computer to find the most attractive printing mode available (e.g. LaTeX if you have it installed) and uses that. The disadvantage of using this mode is that you can't copy and paste the output into code cells.

```
[40]: # <!-- Student -->
      #
      import sympy as sp
      sp.init_printing()
      #
      # For TJG: must comment folliwing line out when merging notebooks to form␣
       ↪summary!
      #sp.solve(a*x**2 + b*x + c, x)
```

If you want to turn "posh printing" off, you can do:

```
[41]: # <!-- Student -->
      #
      sp.init_printing(pretty_print = False)
      #
      sp.solve(a*x**2 + b*x + c, x)
```

```
[41]: [(-b + sqrt(-4*a*c + b**2))/(2*a), -(b + sqrt(-4*a*c + b**2))/(2*a)]
```