MATH3511/6111: Scientific Computing
Mathematical Sciences Institute, Australian National University
Semester 1, 2022

Lab 1: Arrays and Plotting

Lindon Roberts

Due date: 9am, Monday 21 March (week 5)

This lab is based on earlier versions by Stephen Roberts, Graeme Chandler, Jimmy Thomson, Linda Stals and Kenneth Duru.

1 Introduction

This lab will introduce you to linear algebra in Python, using the `numpy` package. NumPy (short for "numeric Python") is the standard package for manipulating vectors and matrices in Python. It comes included as part of the Anaconda distribution.

To check that you have NumPy, load the package (using the standard abbreviation `np`) and check the package version:

```
import numpy as np

print(np.__version__)
```

For more information about anything in NumPy, the online documentation is very helpful: `https://numpy.org/doc/stable/`.

The available NumPy functions and specific syntax don't change much, but are occasionally updated. If there is a mismatch between what the documentation says and what NumPy is allowing you to do, you may want to check the documentation for the specific version of NumPy you have installed (as you checked above) on this page: `https://docs.scipy.org/doc/`.

1.1 Assessment

In each lab, you will be asked some questions. You should provide answers to these questions in your lab book. Marks for these questions will be awarded based on the work presented in your lab book.

Remember this is a mathematics course. The aim is not for you to prove you can produce some sort of output, you must also show that you understand the output. When you have results in your lab book, it is a good idea to include a small discussion about your output: Are the results what you expected? Why or why not? How did you verify your results? What might be some of the reasons why the results are incorrect? During the course we will introduce you to terminology and techniques to help you answer these questions. If you just print the output, you will not be awarded full marks.

Submission Submit your lab book via Wattle. This should include your code, any outputs (e.g. tables of results or figures) and your responses to the questions. I recommend you write a single Python `.py` file which has all your code, and your answers as 'comments': you can print this file to a pdf, and also attach a printout of any outputs (appropriately labelled).

2 Data Types

Every variable in Python stores data of a particular 'type'. There are a few common types you might encounter:

- Integers (`int`)

- Floating-point numbers (`float`) — these represent numbers in $\mathbb{R}$. We will cover these in more detail later in the course.
- Text strings (`str`)
- Boolean truth values `True` and `False` (`bool`).
- Lists (`list`)
- And many more

If you have a variable, the `type` function will tell you what type of data it contains:

```
x = 1
y = 3 / 5
is_positive = (x > 0)
print("x is of type", type(x))
print("y is of type", type(y))
print("is_positive is of type", type(is_positive))
```

Floating-point numbers can also be written in scientific notation, where `e` stands for "$\times 10^{(\cdot)}$":

```
myfloat1 = -2.5e2
myfloat2 = 3.14e-1
print(myfloat1)
print(myfloat2)
```

If it makes sense, you can convert one type to another:

```
x = 1
xfloat = float(x)
print(xfloat)
y = 2.718
yint = int(y)
print(yint)
z = "123.456"
zfloat = float(z)
print(zfloat)
```

Warning! If you convert a `float` to an `int`, you will lose all digits after the decimal place!

In scientific computing, `float` is the most important type. However, Python can also represent complex numbers

```
z1 = complex(1, -1)
z2 = 2 + 3j
print("z1 is of type", type(z1))
print(z1 + z2)
```

You can add/subtract/multiply/divide complex numbers and Python will calculate this correctly.

Warning! Python uses `1j` to represent the complex unit i (this is common notation particularly in engineering and computer science).

The built-in `cmath` module includes complex versions of some common mathematical functions. These are necessary because the `math` module assumes inputs have type `float`. This is true even if the number is real, but of type `complex`:

```
import math
import cmath
z = complex(1, 0)
print(z)
#print(math.exp(z))  # run this once, then comment it out
print(cmath.exp(z))
```

Lab Book 1. Given a complex number in Python how do we extract its real and imaginary parts? Hint: Look at `https://docs.python.org/3/library/cmath.html`. [2 points]

Lab Book 2. Use Python's complex numbers functionality to check Euler's identity $e^{i\pi} = -1$ and the identity $re^{i\theta} = r[\cos(\theta) + i\sin(\theta)]$ for some choice of $r, \theta \in \mathbb{R}$. [3 points]

3 NumPy Vectors

The standard data type for linear algebra is a NumPy "array". This data type includes vectors, matrices and tensors (higher-dimensional versions of matrices). In an array, every entry has to have the same type (usually a floating-point number of type `float`).

For the rest of this lab, we will assume you have imported NumPy in the standard way:

```
import numpy as np
```

It is best practice to do this at the top of your `.py` file.

3.1 Vectors

A vector is a one-dimensional array. You can create a vector from a list of numbers (optionally telling NumPy the type of the array):

```
# Create a vector of integers
x1 = np.array([1, 4, 9, 16, 25, 36, 49])
print("x1 =", x1)
# Create a vector of floating-point numbers
x2 = np.array([1.0, 4.0, 9.0, 16.0, 25.0, 36.0, 49.0])
print("x2 =", x2)
# Create a vector of floating-point numbers (alternative)
x3 = np.array([1, 4, 9, 16, 25, 36, 49], dtype=float)
print("x3 =", x3)
```

NumPy has a large number of functions to create common vectors:

```
# Create a length-10 vector of floating-point numbers, filled with zeros
y1 = np.zeros((10,))
print("y1 =", y1)
# Create a length-4 vector of floating-point numbers, filled with ones
y2 = np.ones((4,))
print("y2 =", y2)
# Create a length-8 vector of floating-point numbers, filled with twos
y3 = np.full((8,), 2.0)
print("y3 =", y3)
# Create a sequence of integers, like in range
y4 = np.arange(10)
print("y4 =", y4)
y5 = np.arange(1, 21)
print("y5 =", y5)
# Create a vector of 21 equally spaced floating-point numbers in the interval [-1, 1]
y6 = np.linspace(-1, 1, 21)
print("y6 =", y6)
```

Given a vector, it is often useful to check certain properties:

```
# The data type of the elements of a vector
print(y6.dtype)
# The number of elements in a vector
print(len(y6))
# The 'shape' of a vector
print(y6.shape)
```

The default `float` data type in NumPy is 64-bit floating-point numbers. We will discuss what this means later in the course, but essentially it means numbers are represented in scientific notation with about 15–16 significant figures.

3.2 Shape

The `shape` of a NumPy vector or matrix is a 'tuple' (like a Python list) of integers, representing the number of values in each dimension. Here, `y6.shape` is a tuple with one number, which is the length of the vector. It is written like `(21,)` rather than `21` to distinguish a tuple (with length one) from an integer. For matrices, the shape will have two numbers (e.g. `(21, 3)`), which are the number of rows and columns respectively.

You can check the number of dimensions of an array with

```
print(len(y6.shape))  # 1 for vectors, 2 for matrices, etc.
```

and you can access each entry of the shape just like a list

```
print(y6.shape[0])  # first element of y6.shape
```

Warning! NumPy treats vectors in $\mathbb{R}^n$ (`x.shape = (n,)`) slightly differently to row/column matrices in $\mathbb{R}^{1\times n}$ and $\mathbb{R}^{n\times 1}$ (e.g. `x.shape = (n,1)`). In almost all circumstances, you will only encounter vectors, but very rarely you come across row/column matrices. You can change them using the reshape command with the desired shape (e.g. `y = x.reshape((n,))` to convert into a vector or `y = x.reshape((n,1))` to convert into a column matrix).

3.3 Slicing and Modifying Vectors

Just like for Python lists (see Lab 0), we can extract elements of a vector using indexing. Remember, the first entry of a vector in $\mathbb{R}^n$ has index 0, and the last entry has index $n-1$ (or in Python just -1 for short). We can use indexing to read or change entries in a vector:

```
# Create a vector
x = np.linspace(-1, 1, 11)
print("x =", x)
# Read entries of x
second_element = x[1]
print("The second element of x is", second_element)
last_element = x[-1]
print("The last element of x is", last_element)
# Change the 4th entry of x
x[3] = 2.0
print("After editing, x =", x)
```

Again, just like lists, we can also use slicing to extract (or change) multiple entries.

```
# Create a vector
x = np.linspace(-1, 1, 11)
print("x =", x)
# Basic slicing examples:
print("x[:3] =", x[:3])    # first 3 elements
print("x[-2:] =", x[-2:])  # last 2 elements
print("x[1:4] =", x[1:4])  # indices 1 to 4 (not inclusive of 4)
# Extracting specific elements with a list of indices
indices = [0, 2, 3, 9]
print("x[indices] =", x[indices])
# Change multiple entries to a single value
x[:3] = 100.0
print("After first edit, x =", x)
# Change multiple entries to a new vector of values
x[:3] = np.array([1.0, 2.0, 4.0])
print("After second edit, x =", x)
```

Lab Book 3. Given a vector `x`, write code which sets all elements of `x` to zero except the first and last elements, which should stay unchanged. Verify that your code works on a test vector of your choice. [2 points]

3.4 Vector Arithmetic

NumPy vectors work properly with the standard vector space operations of addition, subtraction, scalar multiplication and linear combinations:

```
# Basic vector arithmetic
x = np.array([1.0, 2.0, 3.0])
y = np.array([-1.0, 4.0, -9.0])
print("x =", x)
print("y =", y)
print("x + y =", x + y)
print("x - y =", x - y)
print("2*x =", 2*x)
print("3.5*x - 2*y =", 3.5*x - 2*y)
```

However, NumPy also allows us to do element-wise multiplication, division and exponentiation:

```
# Element-wise multiplication and division of arrays
print("x * y =", x * y)
print("x / y =", x / y)
# Element-wise exponentiation
print("x**2 =", x**2)
print("2**y =", 2**y)
print("y**x =", y**x)
```

Warning! In NumPy, multiplication and division are always done entry-wise (for vectors and matrices).

In NumPy, the 'at' symbol `@` and the transpose operator `x.T` can be used to compute the vector dot product $x^T y$:

```
# Vector dot product
print("x.T @ y =", x.T @ y)
```

Lab Book 4. What does NumPy do when you try to combine two vectors of different lengths using the above operations? Do you think this is a sensible choice (briefly explain your answer)? [2 points]

3.5 Functions of Vectors

NumPy also has many built-in functions to do calculations on vectors. These are all defined element-wise:

```
# Functions of vectors
x = np.array([1.0, 2.0, 3.0])
print("x =", x)
print("np.sqrt(x) =", np.sqrt(x))
print("np.sin(x) =", np.sin(x))
print("np.exp(x) =", np.exp(x))
```

Of course, there are many more functions available (see `https://numpy.org/doc/stable/reference/routines.math.html`). These are very useful for evaluating a mathematical function for many different inputs (e.g. for plotting).

Warning! You could use the functions in the `math` module and a 'for' loop (see Lab 0) to calculate element-wise functions for a vector. However, in Python this is significantly slower and should be avoided whenever possible.[1]

```
import math
# Slow elementwise sqrt calculation
# **Do not use this, use np.sqrt instead!**
def vector_sqrt(x):
    y = np.zeros((len(x),))  # create an empty vector with the same length as x
    for i in range(n):
        y[i] = math.sqrt(x[i])
    return y
```

There are also routines for looking at summary information about an array. Some examples include:

```
# Summary functions of vectors
x = np.array([1.0, 2.0, 3.0, 4.0])
print("x =", x)
print("Maximum element =", np.max(x))
print("Minimum element =", np.min(x))
print("Sum of elements =", np.sum(x))
print("Product of elements =", np.prod(x))
print("Average of elements =", np.mean(x))
```

Lab Book 5. Given two vectors $x, y \in \mathbb{R}^n$ with $y \neq 0$, we can always write $x = v_1 + v_2$, where v_1 is a vector parallel to y and v_2 is a vector orthogonal to y. Write code which, given NumPy vectors x and y, calculates v_1 and v_2. Your code should work for vectors of any dimension n. By running your code on test vectors `x = np.array([2.0, -3.0])` and `y = np.array([1.0, 1.0])`, show that your function produces the correct answer. [5 points]

4 Matrices in NumPy

In NumPy, matrices are stored as two-dimensional arrays (i.e. `A.shape` has length 2). Just like vectors, all entries of a matrix must have the same data type. The simplest way to create a matrix is by listing all entries, row-by-row:

```
# Create a matrix of integers from a list of rows
A1 = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]])
print("A1 =", A1)
# Create a matrix of floating-point numbers
A2 = np.array([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0], [7.0, 8.0, 9.0], [10.0, 11.0, 12.0]])
print("A2 =", A2)
# Create a matrix of floating-point numbers (alternative)
A3 = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]], dtype=float)
print("A3 =", A3)
```

Alternatively, NumPy gives some built-in functions for some common matrices:

```
# Create a 2x3 matrix of floating point numbers, filled with zeros
A4 = np.zeros((2, 3))
print("A4 =", A4)
# Create a 3x3 matrix of floating point numbers, filled with ones
A5 = np.ones((3, 3))
print("A5 =", A5)
# Create a 2x2 matrix of floating point numbers, filled with twos
A6 = np.full((2, 2), 2.0)
print("A6 =", A6)
```

[1]This article shows some examples where using NumPy functions is about 70 times faster than a 'for' loop over a list: `https://realpython.com/numpy-array-programming/`

Another way to create a matrix is to take a vector and reshape it into a matrix:

```
x = np.linspace(-1, 1, 21)
A7 = x.reshape((3, 7))
print("x =", x)
print("A7 =", A7)
```

A very important matrix is the identity matrix, which is created with the `eye` function:

```
# Create a 4*4 identity matrix of floating point numbers
I = np.eye(4)
print("I =", I)
```

Sometimes we want to build a matrix with specific entries on one of the diagonals. We can do this with `diag`:

```
# Create a diagonal matrix
diagonal_entries = np.array([1.0, 2.0, 3.0])
D = np.diag(diagonal_entries)
print("D =", D)
# Create a matrix with entries just below the diagonal
subdiagonal_entries = np.array([1.0, 2.0])
D2 = np.diag(subdiagonal_entries, -1)
print("D2 =", D2)
```

With any matrix, we have the same basic properties as vectors:

```
# Data type of a matrix
print(A7.dtype)
# Shape of a matrix
print(A7.shape)
# Number of dimensions
print(len(A7.shape))  # matrix is 2-dimensional
```

Now, matrices have a `shape` with two numbers: the number of rows and the number of columns.

```
nrows = A7.shape[0]
ncols = A7.shape[1]
print("A7 has nrows =", nrows)
print("A7 has ncols =", ncols)
```

4.1 Slicing Matrices

Selecting and editing elements or slices of matrices is exactly the same as vectors, but we need two indices/slices: one for the rows, and one for the columns. Don't forget that indices start from zero.

```
# Create an example matrix
A = np.array([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0], [7.0, 8.0, 9.0], [10.0, 11.0, 12.0]])
print("A =", A)
# Element in second row, first column
print("A[1,0] =", A[1,0])
# Element in the last row, third column
print("A[-1,2] =", A[-1,2])
```

All slicing is the same (note that we just use `:` to refer to all elements in a row/column).

```
# Extract the second row of A (as a NumPy vector)
second_row = A[1,:]
print("Second row =", second_row)
# Extract the last column of A
last_column = A[:,-1]
print("Last column =", last_column)
# Extract the first two rows of A
first_two_rows = A[:2, :]
print("First two rows =", first_two_rows)
# First three elements of the last two columns
top_of_last_columns = A[:3, -2:]
print("Top of last two columns =", part_of_last_columns)
```

Any of the slicing methods described for vectors (section above) can be used for the row/column selection. Whenever we are extracting a single row/column (or part of a row/column), we get a NumPy vector.

Sometimes we also want to extract the diagonal entries of matrix as a vector:

```
A2 = np.array([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0], [7.0, 8.0, 9.0]])
print("A2 =", A2)
print("Diagonal entries =", np.diag(A2))
```

Warning! The function `np.diag` behaves differently if its input is a vector or a matrix (either creating a matrix with given diagonal entries, or extracting the diagonal entries).

4.2 Matrix Arithmetic

The standard matrix operations (addition, subtraction, scalar multiplication) work as expected:

```
A = np.array([[1.0, 2.0], [3.0, 4.0]])
B = np.array([[-1.0, 0.0], [0.5, 2.0]])
print("A =", A)
print("B =", B)
print("A + B =", A+B)
print("A - B =", A-B)
print("2*A - 0.5*B =", 2*A - 0.5*B)
```

Just like vectors, multiplication, division and exponentiation is element-wise:

```
# Element-wise matrix multiplication, division and exponentiation
print("A * B =", A*B)
print("B / A =", B/A)
print("A**2 =", A**2)
```

All standard matrix multiplication operations use the `@` symbol:

```
# Standard matrix-vector multiplication
x = np.array([1.0, 2.0])
print("A @ x =", A @ x)
# Standard matrix-matrix multiplication
print("A @ B =", A @ B)
```

NumPy also allows common functions to be computed element-wise:

```
# Element-wise functions for matrices
print("np.sqrt(A) =", np.sqrt(A))
print("np.sin(A) =", np.sin(A))
print("np.exp(A) =", np.exp(A))
```

To produce summary data, we can either collect information for each row or column, or for the whole matrix:

```
# Maximum entry in each row
print("Row maxima =", np.max(A, axis=0))
# Maximum entry in each column
print("Col maxima =", np.max(A, axis=1))
# Maximum entry in the whole matrix
print("Overall maximum =", np.max(A))
```

The same `axis` input works for `np.min`, `np.sum`, `np.mean`, etc.

4.3 Linear Algebra Operations

There are several important linear algebra operations that NumPy can compute, such as the transpose, inverse or determinant of a matrix.

```
A = np.array([[1.0, 2.0], [3.0, 4.0]])
print("A =", A)
print("Transpose of A =", A.T)
print("Inverse of A =", np.linalg.inv(A))
print("Determinant of A =", np.linalg.det(A))
print("Eigenvalues of A =", np.linalg.eigvals(A))
```

If we want to get the eigenvectors as well as eigenvalues, we can use the function `np.linalg.eig`.

Lab Book 6. How would you calculate the determinant of the top-left 2×2 block of a 10×10 matrix? [2 points]

Lab Book 7. Recall that the trace of a square matrix is the sum of the diagonal entries: if $A \in \mathbb{R}^{n\times n}$ then $\text{trace}(A) = \sum_{i=1}^{n} A_{i,i}$. (a) Write your own function to compute the trace of a matrix that does not use any loops (i.e. only using fast NumPy functions); (b) Use your function to check the identity $\text{trace}(A^T A) = \sum_{i=1}^{m}\sum_{j=1}^{n} A_{i,j}^2$ for a rectangular matrix $A \in \mathbb{R}^{m\times n}$ of your choice; (c) Prove this identity mathematically. [4 points]

We will see later in the course that to solve a linear system $Ax = b$ it is generally not a good idea to compute A^{-1} and evaluate $A^{-1}b$. Instead, NumPy has a special `solve` function for linear systems:

```
A = np.array([[1.0, 2.0], [3.0, 4.0]])
b = np.array([[3.5, 6.5])
x = np.linalg.solve(A, b)
print("A =", A)
print("b =", b)
print("x =", x)
```

Lab Book 8. Write code to test what happens if you try to solve a linear system with a singular matrix? Do you think this is sensible (briefly explain your answer)? [2 points]

The 'Hilbert matrix' is an $n \times n$ matrix given by

$$A_{i,j} = \frac{1}{i+j+1},$$

where the indices are the same as NumPy (i.e. $i, j = 0, \ldots, n-1$).

Lab Book 9. Write code which generates the 15×15 Hilbert matrix A, and creates the vector b which is the first column of A. Mathematically, what is the true solution to $Ax = b$? Solve the linear system $Ax = b$ using `np.linalg.solve` and by calculating $A^{-1}b$. What do you observe? [5 points]

5 Plotting in Python

The standard module for creating graphs in Python is matplotlib (this also comes pre-installed with Anaconda). This module has lots of functionality (see `https://matplotlib.org/3.1.1/gallery/index.html`), but we will use the `pyplot` sub-module for generating simple plots.

This is usually imported with the `plt` abbreviation, so for the rest of the lab we will assume you have run:

```
import matplotlib.pyplot as plt
```

To add graphs to your lab book, see Section 5.7 for how to save plots as image files. You can show your plots for your lab book in a Word document, for example.

5.1 Basic Plotting

Suppose we want to plot some data points (e.g. from an experiment):

x_k	0.5	0.7	0.9	1.3	1.7	1.8
y_k	0.1	0.2	0.75	1.5	2.1	2.4

First, we need to store our data in NumPy vectors:

```
x = np.array([0.5, 0.7, 0.9, 1.3, 1.7, 1.8])
y = np.array([0.1, 0.2, 0.75, 1.5, 2.1, 2.4])
```

We can then generate a plot of x_k versus y_k with the following commands:

```
plt.figure()     # create a new figure
plt.clf()        # clear any existing plots ('clf' = 'clear figure')
plt.plot(x, y)   # generate the plot
plt.show()       # view the plot
```

In Spyder, you can see the plot in the top-right window (click on the 'Plots' tab). You should see something like Figure 1 below.

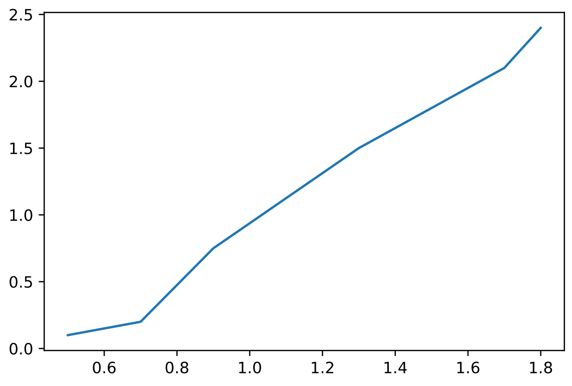

Figure 1. Basic data plot.

For data like this, the default plot is not very useful: we can't see how many data points we have, and the lines between them don't have much meaning. Fortunately, the `plot` command has many different formatting options. For example,

```
plt.figure()
plt.clf()
plt.plot(x, y, 'r.', markersize=10)  # generate alternative plot
plt.show()
```

The 3rd input to `plot` is a string with formatting information: make the plot red, and mark each point with a dot (rather than join with lines). The last argument says how large to make the dots. For a full list of the format string options, see the 'Notes' section of `https://matplotlib.org/3.2.1/api/_as_gen/matplotlib.pyplot.plot.html`.

Instead of using a format string, you can give yourself full control over the plot formatting by using keyword arguments. For example,

```
plt.figure()
plt.clf()
plt.plot(x, y, color='r', linestyle='-', linewidth=1.5, marker='.', markersize=10)
plt.show()
```

A full list of possibilities for line and marker styles is in the `plt.plot` documentation above. Search 'matplotlib named colours' to see the different colour options.

Lab Book 10. Plot the above data with the points shown as green squares and with a dashed line, and show the figure in your lab book. [2 points]

Warning! When making plots to be used in documents, I recommend using `linewidth` at least 1.5 so that it shows up clearly. Yellow tends to be hard to see (and is basically invisible on projectors). You should generally try to design your plots so that they still look reasonable if printed in black and white. Don't forget also that about 1 in 12 men are colourblind (red/green the most common type).

5.2 Multiple Lines

It is very easy to plot multiple lines/datasets on the same graph. You just call the `plt.plot` function multiple times. In our case, since the data is almost linear, let's try to draw a trendline. We could try to calculate a proper line of best fit, but for now we can just see that the line passing through $(0.5, 0)$ and $(2, 2.7)$ would give a good approximation to the data.[2] We can plot our data and the trendline using:

```
# Raw data
x = np.array([0.5, 0.7, 0.9, 1.3, 1.7, 1.8])
y = np.array([0.1, 0.2, 0.75, 1.5, 2.1, 2.4])
# Trend line
xtrend = np.array([0.5, 2.0])
ytrend = np.array([0.0, 2.7])
# Make plot
plt.figure()
plt.clf()
plt.plot(x, y, 'r.', markersize=10)  # plot raw data with red dots
plt.plot(xtrend, ytrend, 'b--')      # plot trend as blue dashed line
plt.show()
```

Of course, when we show multiple datasets on a single plot, we should include a legend. We can do this with

```
# Make plot with legend
plt.figure()
plt.clf()
plt.plot(x, y, 'r.', markersize=10, label='Data')
plt.plot(xtrend, ytrend, 'b--', label='Trend line')
plt.legend(loc='best')  # display legend
plt.show()
```

We have to give a `label` to each dataset, and then add the `legend` command to draw the legend. The `loc` option tells matplotlib where to put the legend: here, we ask matplotlib to put it wherever it thinks is best (but you can say things like 'upper left' or 'lower right' if you want to).

5.3 Formatting

There are several more commands we can use to make our plot look nicer. We can add axis titles and a plot title with the commands

```
# Add plot titles
plt.xlabel('x-axis title')
plt.ylabel('y-axis title')
plt.title('My plot title')
```

and we can control the limits of the axes

```
# Set axis limits
plt.xlim(0, 3)  # lower and upper bounds for x-axis
plt.ylim(0, 3)  # lower and upper bounds for y-axis
```

[2]NumPy can calculate trend lines with `np.polyfit`.

To let matplotlib automatically determine some (or all) of the bounds, use `None` instead of a number, like:

```
plt.xlim(None, 3)  # default lower bound, upper bound = 3 for x-axis
```

To add gridlines, you run

```
# Add gridlines
plt.grid()
```

Of course, you have to run all of these commands before `plt.show()`, otherwise they won't appear!

Lab Book 11. Add reasonable plot titles, gridlines and set some axis limits to the trendline plot above, and show the resulting figure in your lab book. [2 points]

5.4 Different Axis Types

In the previous example, it was easy to see the relationship between x and y from a basic plot. In more complicated situations, it may be necessary to use different scales to show the data more clearly. Consider the data

n_k	3	5	9	17	33	65
s_k	0.257	0.0646	0.0151	3.96×10^{-3}	9.78×10^{-4}	2.45×10^{-4}

A plot of n_k versus s_k shows no obvious trend:

```
n = np.array([3, 5, 9, 17, 33, 65])
s = np.array([2.57e-1,  6.46e-2,  1.51e-2, 3.96e-3, 9.78e-4,  2.45e-4])

plt.figure()
plt.clf()
plt.plot(n, s)
plt.show()
```

However, if we put both axes on a log scale, things become much clearer:

```
plt.figure()
plt.clf()
plt.loglog(n, s)  # log scale for both axes
plt.show()
```

If we only want one axis to be logscale, we can use `plt.semilogx` (x-axis only in log scale) or `plt.semilogy` (y-axis only in log scale).

5.5 Graphing Functions

Another common task is to graph a mathematical function $f(x)$ on an interval $[a, b]$. We can do this by:

1. Creating a list of x values between a and b.
2. Evaluate f at each of these x values.
3. Plot the list of x values and $f(x)$ values, connected by lines.

Basically, this does linear interpolation to our x values, but if we have enough x values it looks like a proper graph.

This process is made easier by NumPy's `np.linspace` function (to create a vector of x values) and the elementwise functions (from Section 3.5). For example, if we wanted to plot

$$f(x) = \frac{\sin(x)}{1 + \sqrt{x}}$$

on the interval $x \in [1, 10]$, we could run:

```
# Choose 101 equally-spaced x-values between 1 and 10
x = np.linspace(1, 10, 101)
# Evaluate f(x) at each x (using NumPy's element-wise functions rather than a for loop)
fx = np.sin(x) / (1 + np.sqrt(x))
# Plot x versus fx
plt.figure()
plt.clf()
plt.plot(x, fx)
plt.show()
```

We get the plot shown in Figure 2 below.

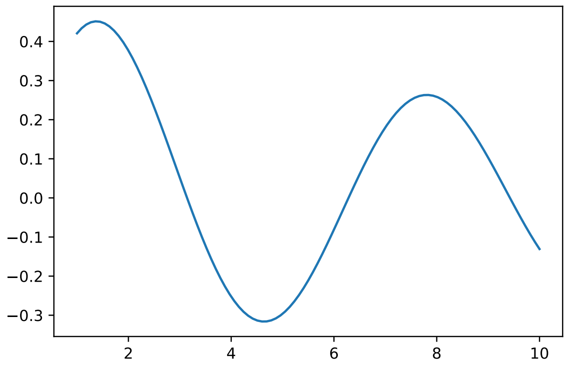

Figure 2. Plot of $f(x) = \sin(x)/(1+\sqrt{x})$.

Lab Book 12. Plot the graphs of $f(x) = 1/(1+e^{ax})$ for $x \in [-4, 4]$, where $a = 0.5, 1, 2$, all in one figure (with appropriate legends and axis ranges). [4 points]

5.6 Multiple Plots

Sometimes we want to plot different data on different plots, all within one figure. We can do this using the `plt.subplot` function.

```
# Initialise an empty figure
plt.figure()
plt.clf()

# Create a grid of axes (1 row and 2 columns)
nrows = 1
ncols = 2

# Draw first subplot
plt.subplot(nrows, ncols, 1)
plt.plot(...)
plt.xlabel(...)
...

# Draw second subplot
plt.subplot(nrows, ncols, 2)
plt.plot(...)
plt.xlabel(...)
...

# Finished drawing all subplots
plt.show()
```

5.7 Saving Plots

To save a plot as an image file, replace the `plt.show()` command with `plt.savefig`, like in the below:

```
plt.figure()
plt.clf()
plt.plot(x, y)
#plt.show()  # not used when saving to file
plt.savefig('myfigure.png')
```

You can choose from a number of file types: I recommend `.png` for regular usage (including for Word documents or websites), and `.pdf` for formal documents written in LaTeX (e.g. honours or masters thesis).

Sometimes the margins on the saved file can be very large, which doesn't look so nice. To save the plot with very narrow margins, use

```
plt.savefig('myfigure.png', bbox_inches='tight')
```

5.8 Exercise: Data Plotting

For this exercise you will generate a plot of new COVID-19 cases in NSW. The file `lab1_nsw_covid_case_data.csv` is a text file containing the number of new COVID-19 cases reported in NSW from 1 July 2021 until 7 February 2022 inclusive (the first row is for 1 July, etc.).[3] You can open the file in Excel or Notepad to inspect it.

Lab Book 13. Read this data into a NumPy vector, and calculate a new vector which has the 7-day moving average of new cases. Make a plot showing both data series with appropriate legends/axis labels, etc. [5 points]

Notes:

- NumPy has functions to read data from text files: you will have to find a suitable choice and read the documentation to figure out how to use it.
- NumPy does not have a moving average function, you will have to write a function to do this yourself. The 7-day moving average is the average of the last 7 days' of data (e.g. for the 7th date 7 July 2021, the number of new cases is 37 but the 7-day average is 31; for the last day 7 February 2022, there were 6136 new cases but the 7-day average is 9240.4). The 7-day moving average is not defined for the first 6 days, so your vector will be shorter.
- For the plots, you can just use 0,1,2,... for the x-axis values (but be sure to use a suitable axis label). Make sure the moving average line is correctly aligned with the original data (i.e. no moving average is plotted for the first 6 days).

[3] Based on data from `https://data.nsw.gov.au/search/dataset/ds-nsw-ckan-3dc5dc39-40b4-4ee9-8ec6-2d862a916dcf/details?q=`, accessed 9/2/22. The ACT Government doesn't have its data as readily available.