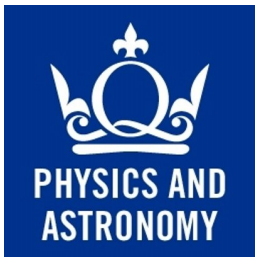
Planet Hunting with Python



Richard P. Nelson & Gavin Coleman
School of Physics & Astronomy, Queen Mary University of London

Abstract

NASA's Kepler spacecraft was launched in 2009 and spent approximately 4 years staring at 150,000 stars, searching for planets orbiting around them using the transit detection method. The mission resulted in the discovery of 4717 exoplanet candidates, of which 2303 have been confirmed as *bona fi*de planets, and a number of which are found to reside in multi-planet systems. During your research, you will learn to write computer programmes in the Python programming language, and you will use your newly developed skills to write routines to analyse data from the Kepler mission using simplified algorithms. You will analyse light curves that have been downloaded from the Kepler spacecraft, and determine the key physical parameters of the planetary systems including orbital periods, orbital inclinations to the line of sight, orbital semi-major axes (i.e. the distance between the star and the planet), and the physical radii of the planets. You will learn about fitting models to data, and techniques for optimising the accuracy of these models. Based on your analysis, you will be able to draw conclusions about the observed population of exoplanets, including whether or not they look like the planets in our Solar Systems, and explore how dynamical interactions between planets in multi-planet systems can be detected in the data.

Introduction

The first extrasolar planet discovered orbiting a Sun-like star was 51 Pegasi b (Mayor & Queloz 1995), and this discovery led to Michel Mayor and Didier Queloz being awarded the Nobel Prize for Physics in 2019. During the 24 years since this ground-breaking discovery, the total number of confirmed extrasolar planets has increased to 4,073[1] with new discoveries being made on an almost daily basis. We now know that the population of exoplanets in our Galaxy is very diverse, and that our Solar System does not provide an example of the most typical architecture.

The launch of the Kepler spacecraft in March 2009 heralded a dramatic improvement in our knowledge about exoplanets (Borucki et al 2010). Kepler was launched into an Earth-trailing heliocentric orbit, which allowed it to stare

[1] Number taken from exoplanetarchive.ipac.caltech.edu in October 2019.

continuously at one region of the sky for a period of approximately 4 years[2]. The mission used the transit detection technique to find planets. This method looks for the periodic dimming of the light from a star as seen by an observer when a planet passes in front of the star during its orbit (see Fig. 1).

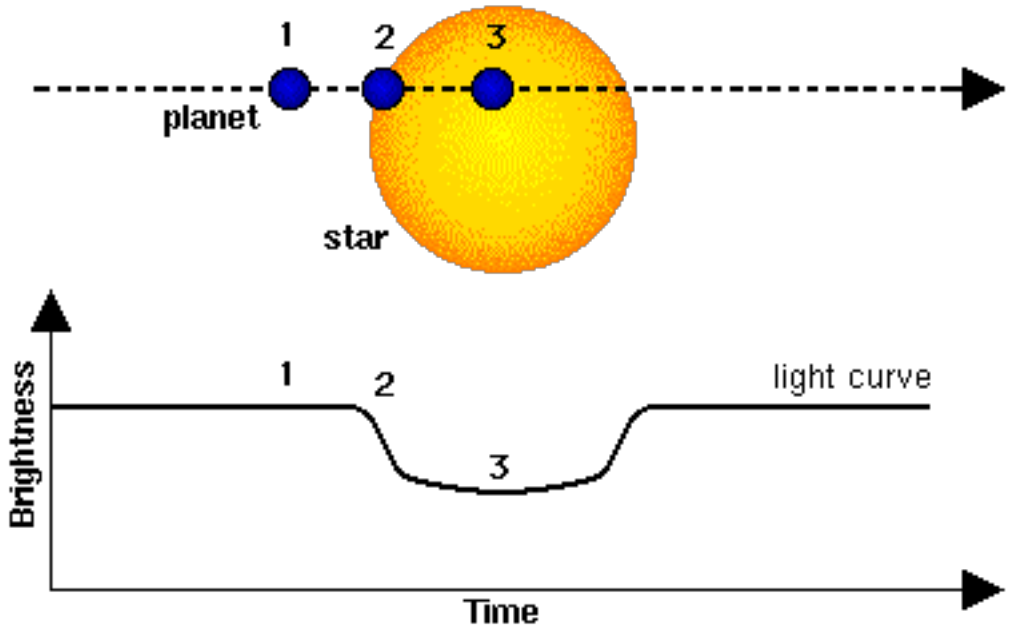


Figure 1: This diagram illustrates the physical principles behind the transit detection method. When an orbiting planet passes in front of its host star, some of the light from the star is blocked out and the star appears to dim slightly. Detecting the periodic dimming of a distant star may therefore indicate the presence of a planet.

For geometrical reasons, most planetary orbits around distant stars do not cause the planet to pass in front of the star when observed by the Kepler spacecraft. For a randomly orientated orbit the probability of observing a transit is typically about 1%, so for this reason the Kepler spacecraft monitored the brightness of approximately 150,000 stars for the duration of the mission, resulting in the discovery of 4717 planet candidates, of which 2303 have been confirmed to be genuine exoplanets. Note that there are numerous ways in which different astrophysical phenomena can mimic a transiting planet, hence the need for additional observations that are able to confirm the planetary nature of the Kepler planet candidate systems.

The Kepler spacecraft produced data in the form of light curves for each of the observed stars – data files that list the time of observation and the brightness of the star over an extended period of time. These light curves were then processed to remove artefacts and other unwanted features from the data (a process known as *detrending*) prior to the data being made available for scientists to analyse. See Fig. 2 for an example of a detrended light curve. You will use these detrended light curves in your analysis, and one of the things that you will learn is how to download the data files for individual systems from the Kepler data archive so that you can choose which systems to analyse.

[2] More information about the Kepler mission can be found at kepler.nasa.gov

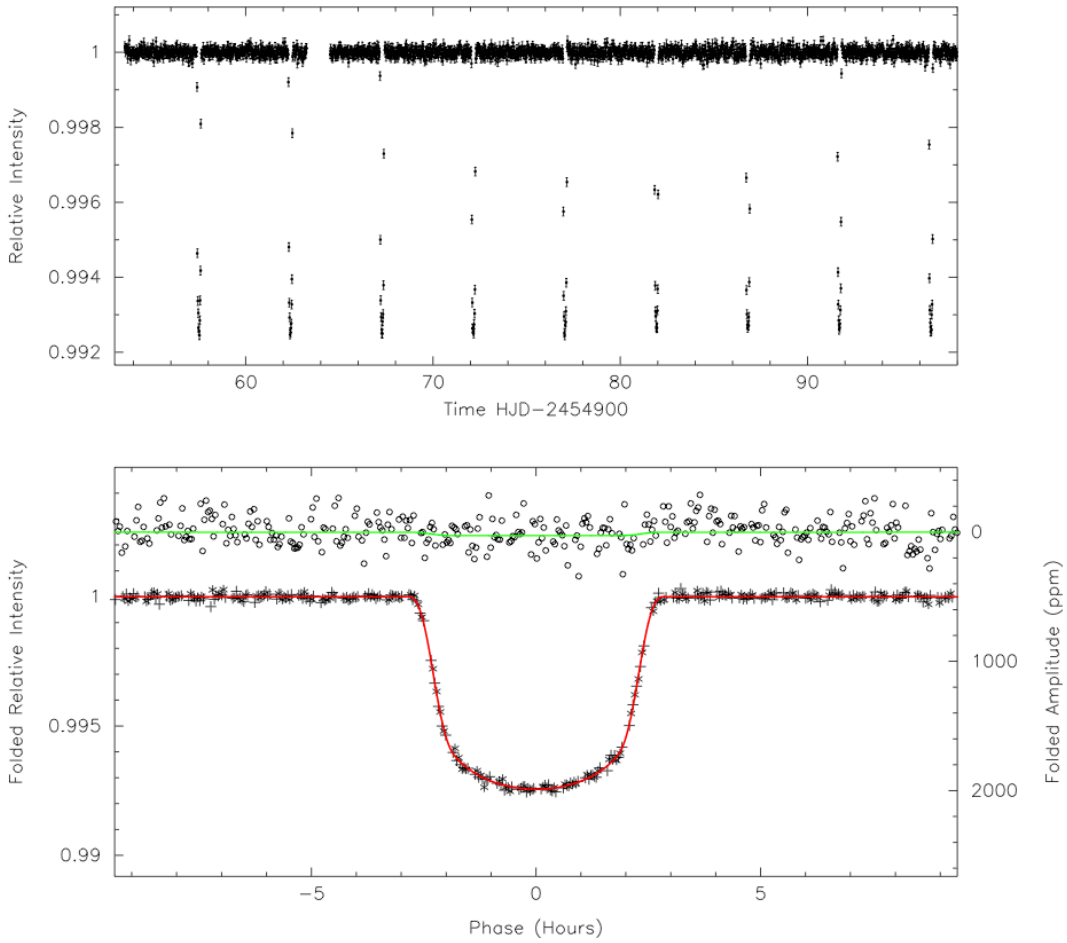


Figure 2. The top panel shows a detrended light curve for the system Kepler 7b. The y-axis shows the flux of light from the star in arbitrary units and the x-axis shows the time (in units called Julian days). The periodic dips in the light curve correspond to when the planet is passing in front of the star. The lower panel shows what is called the phase-folded light curve (black crosses) with a model fitted to it (red curve). The phase–folded curve contains all the data in the top panel, but plotted so that the mid-point of each transit occurs at time t=0.

A primary aim of this project is to provide hands-on experience of programming in Python within the context of astronomical data analysis. Python is becoming a commonly used programming language in many areas of life, including in the scientific world where it is used for data analysis and mathematical modelling, and in schools and universities where it is becoming the language of choice when teaching computer programming. Python's increasing popularity arises for a number of reasons: it is free to use; it runs under all operating systems (Windows, Mac OS X, Linux); its basic structure and syntax is similar to many other languages; it contains the ability to perform mathematical calculations and to plot data in graphical form within a self-contained programming package. Later in this document you will be provided with a sequence of programming tasks of increasing complexity to help you develop your programming skills in incremental stages.

Obtaining physical information from transit data

From Fig. 1 we can see that a reasonable model for a planetary transit is to assume that an opaque circular disc of radius R_p moves across a uniformly luminous circular disc of radius R_s, blocking some fraction of the light detected by an observer. Here, R_p and R_s are the radii of the planet and star, respectively.

Obtaining the planet radius from the light curve

Exercise:
Using the simple model described above, and shown in Fig. 1, for a planetary transit, obtain an expression for the change in the observed luminosity of a star of radius R_s when a planet of radius R_p transits in front of it. In particular, obtain an expression for $\Delta L/L_s$ in terms of R_s and R_p, where $\Delta L = L_s - L_T$ and L_s denotes the stellar luminosity when the planet is not transiting, and L_T denotes the observed luminosity when the planet is at the mid-transit point.
[Hint. Think about what fraction of the emitted light from the star is blocked out by the opaque disc (i.e. the planet) passing in front of it. The surface area of the star is πR_s^2 and the area of the planet is πR_p^2. The luminosity or brightness of a uniformly emitting disc (i.e. the star) can be written as the flux of radiation being emitted from the surface multiplied by the surface area of the disc. The flux is defined to be the amount of radiation emitted by the disc per unit area per unit time.]

Solution:

Your teacher has the solution. Try to work it out for yourself before seeking assistance.

Planet's orbital period
Obtaining the planet's orbital period from a transit light curve is easy when there is a single planet transiting a star. It is simply the time that elapses between successive dips in the light curve.

When you start to work with the Kepler data, you will notice that the times listed in the light curve data files, and on the website that hosts the data archive, are in strange units – "Barycentric Julian Day minus a constant offset" (abbreviated to either BJD or [BJD – 2454833]). For complicated historical reasons, astronomers measure time from noon on the 1st January 4714 BC. This day is known as Julian Day zero, and all subsequent days are known as Julian Day 1, 2, 3, 4, 5, … The number 2,454,833 mentioned above corresponds to noon on 1st January 2009 and is subtracted from the Julian date in order to make the numbers more manageable. Hence the times listed in Kepler light curves are the number of days since noon on 1st January 2009.

Orbital semi-major axis

If we assume that the transiting planets are in circular orbit around their stars, then we can obtain an expression for the semi-major axis of the orbit (the distance between the star and planet during its orbit) in terms of the mass of the star (which can be obtained from the Kepler data archive) and the orbital period (which can be obtained from the transit light curve).

This is given as an exercise below with step by step hints.

Exercise:

When a planet is in circular orbit around a star, we can say there is a force balance between gravity and the centrifugal force. Hence, we start by equating the gravitational force acting between the star and planet to the centrifugal force associated with the circular orbital motion of the planet

$$\frac{G\ M_s M_p}{a^2} = \frac{M_p\ v^2}{a}$$

where v denotes the velocity of the circular orbit and a is the semi-major axis. M_s is the mass of the star and M_p is the mass of the planet, and G is Newton's gravitational constant. *Manipulate this equation so that the semi-major axis, a, is expressed in terms of everything else.* Your teacher has the solution so ask if you need assistance.

The velocity, v, at which the planet moves around its circular orbit is given by the distance around the circle (or its circumference), C, divided by the time taken, P, where in our case P is the orbital period. (Remember: Velocity is distance travelled divided by the time taken). *Obtain an expression for v in terms of the semi-major axis, a, and the orbital period P*. Your teacher has the solution so ask if you need assistance.

Now combine your expressions for the semi-major axis, a, and the velocity, v, to get an expression for a in terms of M_s, P, G and π. Your teacher has the solution so ask if you need assistance.

To work out the values of the semi-major axis, a, for each exoplanet you will study in this project, you will need the mass of the star, M_s. We have provided this in a pdf file (Stellar_Mass_Radius.pdf) on your memory stick for each of the systems you will analyse. This file also contains the radius of the star. Please note that the values of M_s and R_s are expressed in units of the Solar mass and Solar radius, where these are $M_{Sun} = 1.989 \times 10^{30}$ kg and $R_{Sun} = 7 \times 10^8$ m. Note that $G = 6.67 \times 10^{-11}$ m^3 kg^{-1} s^{-2} in these units, and the semi-major axis you will obtain will be in metres. You may want to convert this into Astronomical Units (the mean distance between the Earth and Sun), where 1 AU = 1.5×10^{11} m.

Determining the transit impact parameter

Figure 3[3] shows that the transit impact parameter, denoted by b, is a measure of how close to the centre of the stellar disc the planet passes at the midpoint of the transit. b is measured in units of the stellar radius, R_*. Note that we refer to the angle i as the inclination of the orbit, such that an angle i=90° corresponds to a planet that passes across the centre of the stellar disc. Figure 3 shows that the impact parameter, b, the semi-major axis, a, and the inclination of the orbit, i, are related by simple trigonometry.

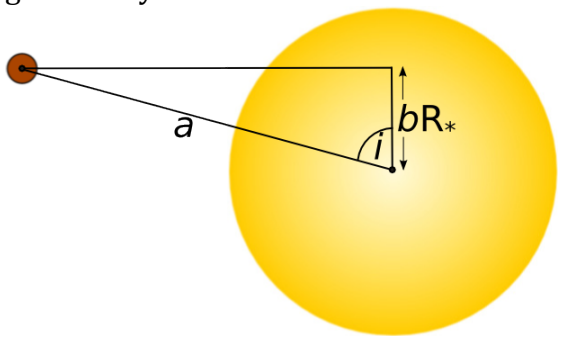


Figure 3: The observer is located on the left of the diagram and is looking right towards the star. The planet is assumed to be at the midpoint of the transit, and its projected distance above the centre of the star is given by $b \times R_*$.

Figure 4 shows a face on view of the path that the planet takes across the stellar disc during a transit, and demonstrates the trigonometric relation described above. The length of the chord that corresponds to the path taken by the planet across the stellar disc can be determined from Pythagoras' theorem.

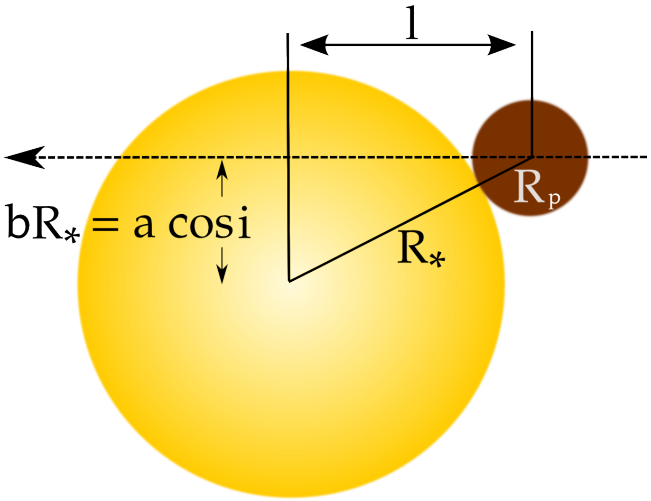


Figure 4: Face-on view of the path taken by a planet as it crosses the stellar disc during a transit.

The equation for the length of the chord is

$$2l = 2\sqrt{(R_* + R_p)^2 - (bR_*)^2}$$

[3] The diagrams used in Figures 3, 4, and 5 were obtained from https://www.paulanthonywilson.com/exoplanets/exoplanet-detection-techniques/the-exoplanet-transit-method/

The diagram below gives a 3-dimensional view of the geometry of the orbit of a transiting planet.

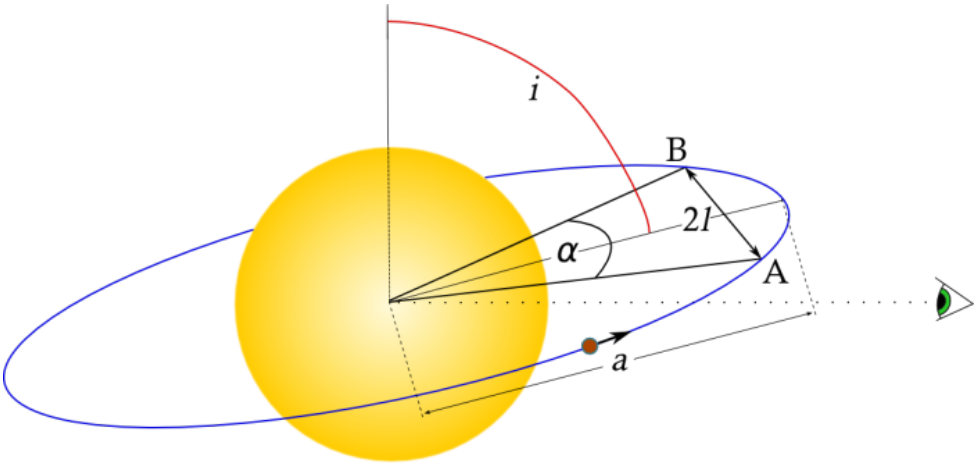


Figure 5: 3-dimensional view of the path taken by a transiting planet on its orbit.

From the triangle drawn in the orbital plane, we can see that $\sin\left(\frac{\alpha}{2}\right) = \frac{l}{a}$ (note that the diagram exaggerates how close the orbit is to the star. In almost all cases, the distance between the centre of the star and the midpoint between A and B is well approximated by the semi-major axis *a)*. Assuming that the planet is on a circular orbit, the duration of the transit is given by the expression

$$T_{dur} = P\frac{\alpha}{2\pi}$$

When combining this with the equations above, we obtain the expression

$$T_{dur} = \frac{P}{\pi}\sin^{-1}\left(\frac{\sqrt{\left(R_* + R_p\right)^2 - (bR_*)^2}}{a}\right)$$

From this you can rearrange for bR_* and determine the transit impact parameter based on measuring the duration of the transit, T_{dur}, from the light curve.

Exercise

Obtain an expression for bR_* from the above equation for T_{dur}. Using Figure 4, obtain an expression for the orbital inclination angle *i*.

Solution

Your teacher has the solution. Try to work it out for yourself before seeking assistance.

Summary: We now have sufficient information to be able to obtain the following information from the light curves and stellar parameters (stellar mass and stellar radius) provided by the Kepler data archive for each planet-hosting star:

- the planet's radius
- the planet's orbital period
- the orbital semi-major axis
- the impact parameter
- the orbital inclination

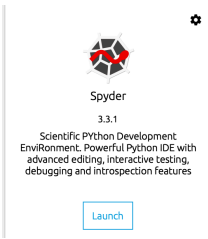
Writing programmes in Python

We now provide an introduction to writing computer code in Python, with a focus on what you will need to analyse Kepler data. First, you will be introduced to various programming concepts and Python commands, and then you will be provided with a sequence of exercises that allow you to put this new information into practice. In addition to following what is written below, you may find it useful to look at some on-line Python tutorials. If you type "python tutorial" into google, then you'll find literally hundreds of websites that offer information about programming in python. One that we found useful in getting an overview when developing this project is http://www.python-course.eu/course.php

A site that provides documentation on many aspects of python is http://scipy.org

If you get stuck and want to know how to do *something* in Python, then google "How to do *something* in Python", where *something* could be "plot a graph", "read data into an array", or any other issue you're having problems with.

There are many versions of Python available to download and install on your computer. We strongly recommend using Anaconda, which may be downloaded from this site: https://www.anaconda.com/download/ - windows
Once you have Anaconda python installed, we recommend using the Spyder developer environment which will appear as follows when you launch the Anaconda-Navigator. Click on this to launch Spyder.



PLEASE NOTE THAT PYTHON COMES IN TWO DISTINCT FLAVOURS: PYTHON 2 and PYTHON 3. WE WILL ONLY USE THE VERSION PYTHON 2.7.

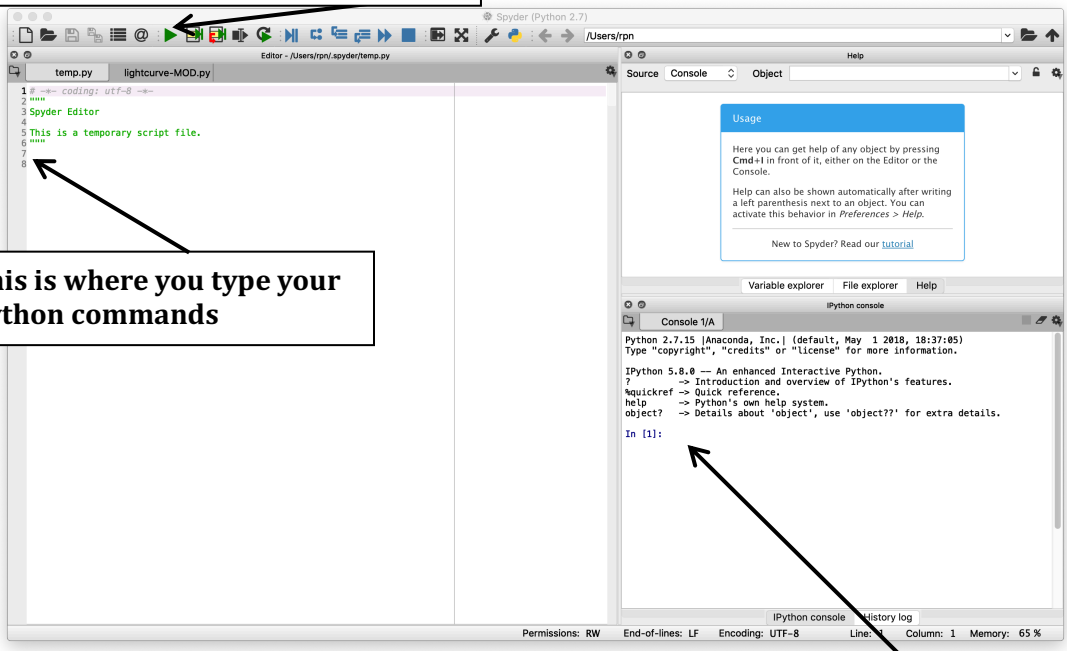
Getting started

You have been given a USB memory stick containing a folder PlanetHuntingWithPython2019. Before getting started with the programming examples below, you are advised to copy this PlanetHuntingWithPython2019 folder and all of its contents onto your computer. This folder will be where you should save all of the Python scripts that you create.

You will create and run the python scripts using Spyder. When this has been launched you should see the window shown below.

Use the File menu at the top of the screen to save your scripts to the PlanetHuntingWithPython2019 directory. Choose Save as... and navigate to the required directory

Click the green play button to run a Python script



This is where you type your Python commands

This is the console where the outputs from your Python scripts are printed. If your script requires input from the user, then this is where that input should be typed.

Creating variables and arrays

Variables are names (always beginning with an upper- or lower-case letter) that you can assign a value to. You should think of a variable as a location in the computer's memory where a value is stored (note that the value can be a number (real or integer) or a string (i.e. a letter or a word)). Once a variable has been assigned a particular value, it will store that value until another value is assigned to it. An example of assigning the value 10 to a variable so that it is stored as decimal number is shown below (note the decimal point):

```
var = 10.
```

Arrays are data structures that store a collection of values that are normally of the same type (i.e. real numbers, integers or strings). This collection of numbers will often have some relation to each other. An example relevant to this project is that an array could store all of the times associated with a Kepler light curve, and another array could store the corresponding observed luminosities/fluxes of the star. Assigning variables to an array can be done as follows:

```
x = [1.,2.,3.,4.,5.]
y=[1.,4.,9.,16.,25.]
```

The previous examples were of one-dimensional arrays. Python also includes two- or three-dimensional arrays. Although we won't use these higher-dimensional arrays in this project, an example of assignment to a two-dimensional array would look like this.

```
z = [[1,2],[3,4]]
```

Here the first two elements of the array form the first column and the second two elements form the second column, as shown below. We can think of arrays as having similar structures to matrices in mathematics.

1	3
2	4

To access a specific element of a one-dimensional array, you would type

```
x[0]
```

which would return the value "1." in the example above, since the indexing of arrays in python starts at zero. Similarly, accessing an element of a two-dimensional array can be achieved as follows:

```
z[0,0]
```

which would access the element in the first column and first row, giving a value of 1 in the above example. As we will see later, arrays can be created automatically in Python by reading multiple values into a variable name from a data file, and you will do this frequently as part of this project.

Adding comments to programmes

If you want to add a comment to your scripts so that you or other people can understand them later, you can use the # symbol. This is something that we would strongly encourage you to do as you write your programmes. Python ignores everything on a line past the # symbol. The comment can be on the same line as a Python command:

```
var = 10  # This is a comment
```

Here, Python only executes the var = 10 command, and ignores 'This is a comment'. Alternatively, the comment can appear on a separate line:

```
# This is a comment
var = 10
```

Printing to the terminal

In Python you can print text to the terminal. This is done using the print command as follows:

```
print 'Hello'
```

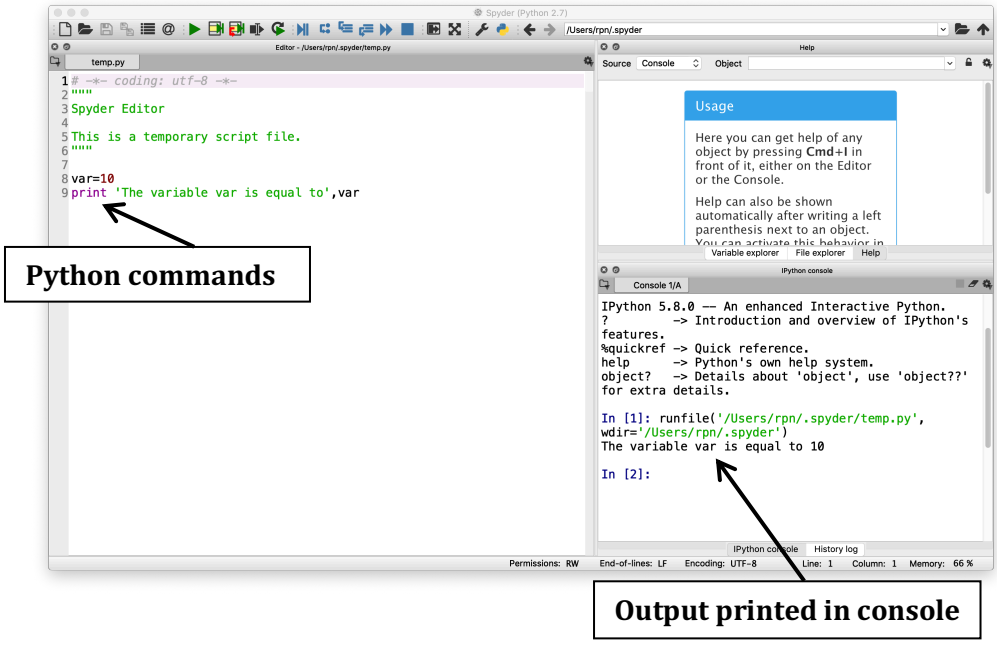
The print command can also print the values of a variable or an array. If we have a variable called 'var' which is assigned the value 10, then the following Python code command will initialise the variable and print its value to the terminal:

```
var = 10
print var
```

Note that for variables and arrays no quotation marks are required. Quotation marks are only required for printing strings. If we want to print multiple variables or pieces of information on the same line, we use the following print command:

```
var = 10
print 'The variable var is equal to: ', var
```

The 'comma' allows multiple pieces of information to be printed using a single print command as shown in the figure below



For loops

For loops are useful for running the same block of code repeatedly. They have a specific syntax (written format). Below is an example of a for loop:

```
count = 0
for i in range(0, 50):
        count = count + 1
        print 'count = ',count
```

Check for the colon here

Check the indentations here. Spyder will normally do the indentation for you

Note that the indentation (either a space or tab) is required for Python to know which commands form part of the loop. We call the commands that are executed repeatedly *the body* of the loop. All commands within the body of the loop must have the same indentation. The above loop would iterate 50 times (Python starts with the loop counter, *i,* having the value zero, so the code will iterate 50 times from zero), adding 1 each time to the variable *count* and then printing the value of *count* to the terminal.

if statements for conditional execution of code

if statements are very useful. They use conditional statements that evaluate to either *true* or *false* to determine which pieces of code to run. They have a specific syntax that must be followed for the code to execute properly. Below is an example of an *if* statement:

```
var = 5
if var == 10:
        print 'true'
else:
        print 'false'
```

CHECK: Check for the colon

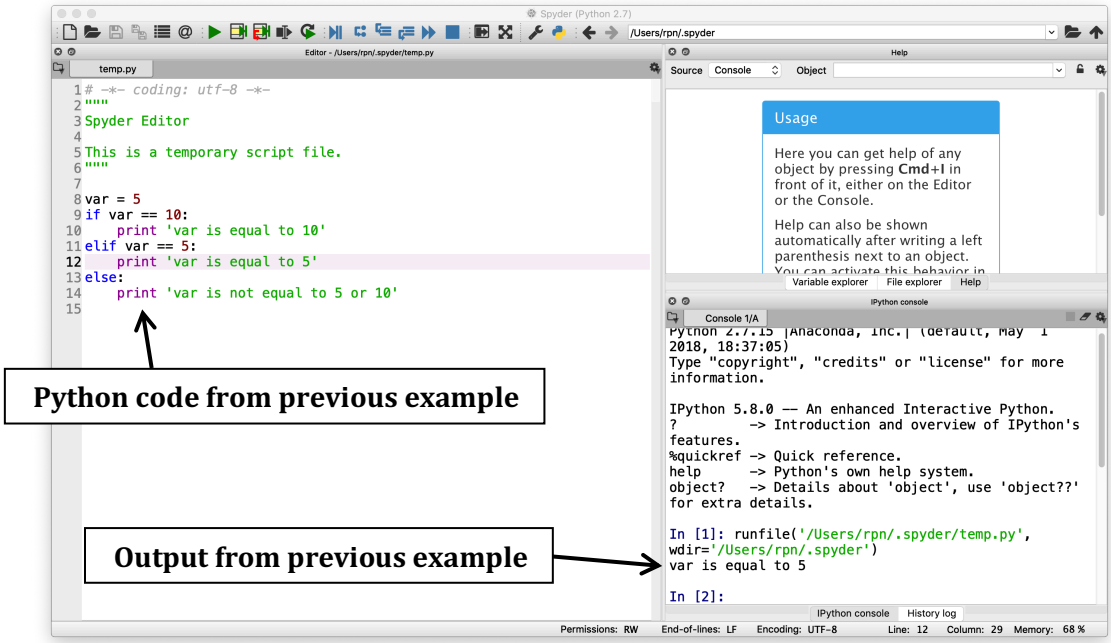
Since 5 does not equal 10, python will skip the *print 'true'* statement and execute the code associated with the *else* condition, which in this case will cause it to print *'false'* to the terminal.

It is possible to insert '*else if*' conditions, which cause the script to only run the block of code associated with the first conditional statement that it determines to be true. See the example below:

```
var = 5
if  var == 10:
        print 'var is equal to 10'
elif  var == 5:   # "elif" represents "else if" here
        print 'var is equal to 5'
else:
        print 'var is not equal to 5 or 10'
```

CHECK: Check for the indentations

With var being equal to 5, the code will now print '*var is equal to 5*' because the second *if* condition is satisfied. Within the *if* statement, you can add as many '*elif*' statements as you like, and the *else* statement should always come at the end.

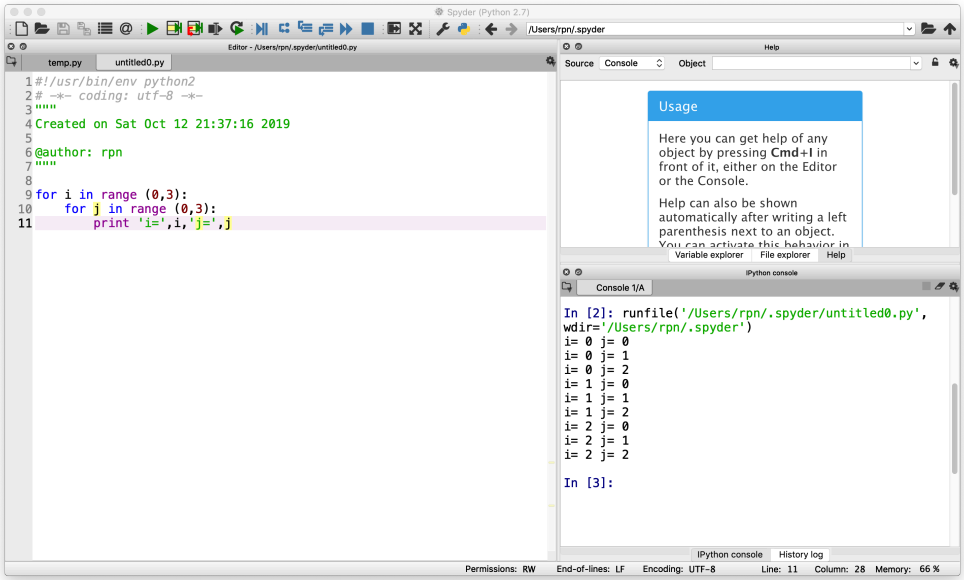


Nested loops

Nested loops are used when repeatedly executing a block of Python code that depends on the values of two independent variables (these variables can be the values of the loop counters themselves, or other variables that are defined and modified within the bodies of the loops). Note that one needs to take care of the indentations when implementing nested for loops. An example of multiple for loops is given below:

```
for i in range(0,3):
        for j in range(0,3):
                print 'i=', i, 'j=', j
```

The above code will run through the whole sequence of the 'j' loop during a single iteration of the 'i' loop, printing the values of i and j to the terminal, as shown in the diagram below:



Combining for loops and if statements

One needs to be careful about the indentation when combining loops and if statements. An example is shown below:

```
for i in range(0,10):
        if i == 4:
                print i
        elif i == 8:
                print i
```

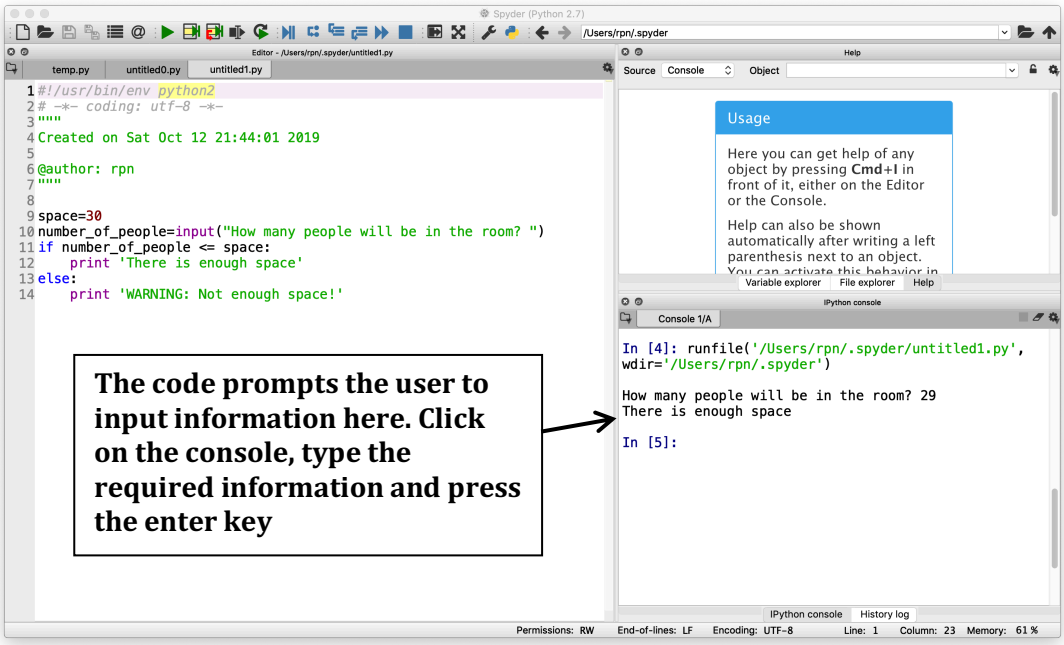
The code here would start running the *for* loop starting at i = 0, before checking the conditional *if* and *elif* statements and running the appropriate piece of code. It would then return to the *for* loop and continue to the next iteration of i, repeating this sequence 10 times.

Asking the user to input information at the terminal

It is sometimes necessary for a script to ask the user to supply information that is required for its continued execution. For example, here is a script for determining if there is enough space in a room for the number of people required.

```
space = 30
number_of_people = input('How many people will be in the room?')
if number_of_people <= space:
        print 'There is enough space'
else:
        print 'WARNING: Not enough space'
```

Here Python will stop at the line containing the *input* command until the user responds by typing in the relevant value at the console, as shown in the screenshot below. This value is then stored in the variable *number_of_people*.



Reading data into arrays from a text file – introducing *numpy*

The Kepler data that you will use in your project will be contained in text files, and your programmes will need to read this data before analysing it. One feature of Python that we have not yet discussed is the fact that many of the functions that Python offers are contained in libraries/packages that need to be loaded during a Python session before they can be used. When writing a script, it is a good idea to load the required libraries at the beginning of the script. One library that deals with mathematical functions and numerical data is called *numpy*. The useful feature that we will introduce here is its load text capability, but it also provides a large number of mathematical functions such as sine, cosine, sqrt, etc. Below is a script that will read in data from the file 'xy.txt' located in the directory DATA which is in the PlanetHuntingWithPython2019 folder, and then print the data out to the terminal.

```
import numpy as np  # We could just use 'import numpy', but assigning it as np
                    # allows for easier typing later in the script.
x, y = np.loadtxt('DATA/xy.txt', unpack=True) # Specify that loadtxt is part of
                                              # the numpy library -> np.loadtxt
for i in range(0,len(x)): # we use len(x) here as we do not know the length of x.
                          #  The command len(x) determines the number of
                          #  elements in the array x which has been created when
                          # reading in the data from the file DATA/xy.txt
        print x[i], y[i]  # Since x and y are arrays, the [i] are required to access the
                          # specific elements of that array.
```

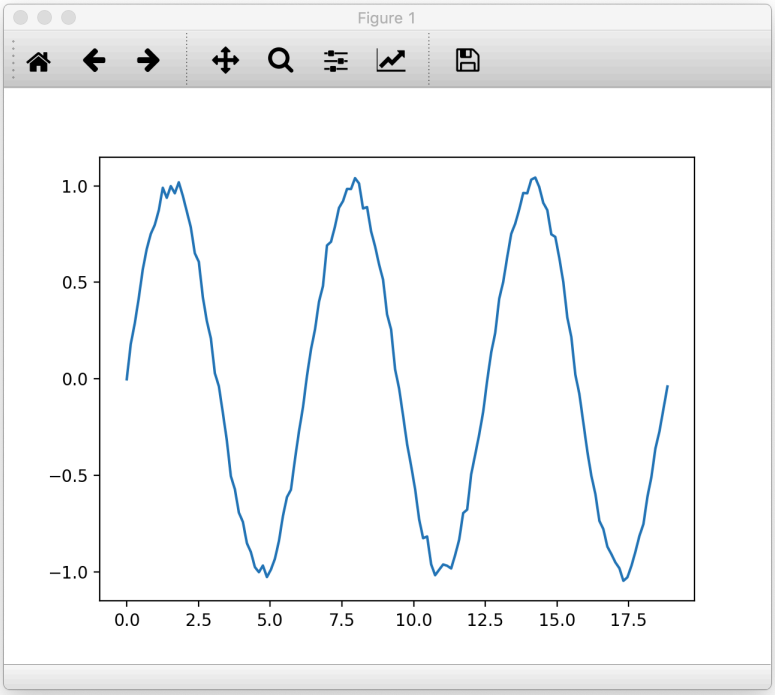
In the above example the data is read in and printed out line by line. Now that we have the x and y arrays loaded in, let's consider what we would do if we wanted to create a new array z consisting of the elements of the two existing arrays multiplied together (i.e. x[i]*y[i] for all values of i). An example is shown below:

```
z  = []  # initialise the array z which is empty
for i in range(0,len(x)):       # iterate for loop over all elements of x
        z.append(x[i]*y[i])     # This will calculate the value of x[i]*y[i] and append
                                # it to the list of values already contained in the
                                # array z
         print z[i]
```

Plotting data

It is almost always useful to plot your data, and for the task of analysing Kepler light curves it is essential. In order to plot data in Python we need to load the library *matplotlib.pyplot*, which provides the plotting functions that we need. Below is a script that will load in the data from 'xy.txt' and then plot it, as shown by the figure.

```
import numpy as np
import matplotlib.pyplot as plt   # Similar to the numpy example above,
                                  # use plt to make things easier
x, y = np.loadtxt('DATA/xy.txt', unpack=True)
plt.plot(x,y)    # This plots the data to the current figure. More options can be
                 # included in this command to define the line-width, colour,
                 # marker symbol, etc. For example, plt.plot(x,y, 'r.')  would plot the
                 # data using red dots as markers.
plt.show()       # This draws the figure on the screen, as shown below
```



Important note: When generating a plot in Spyder, by default it will embed the plot in the small console window. This is not what we want because we need to be able to manipulate the plots we generate in this project, and that is not possible when the plot is embedded in the console window. To allow Spyder to generate a separate plot window that we can manipulate please do the following steps (note it should only be necessary to do this once as Spyder will remember you've changed the settings):

1). Click on the spanner symbol that opens the Spyder preferences menu
2). Click on IPython console
3). Click on the Graphics tab
4). Under Graphics backend select Automatic, and the press Apply and OK
5). Now restart Spyder and run the plotting script

Other useful commands

```
np.sum(x) # gives the sum of all array elements
np.mean(x) # gives the mean of all array elements
if count % 10 == 0 # the % symbol gives the remainder when count is divided by
                   #10 in this example. Sometimes we call this the modulus of
                   # dividing 10 into count. The remainder value is then
                   # compared with 0 to determine if the condition in the if
                   # statement is true
plt.xlabel('String')  # puts an axis label on the x axis. Change xlabel to ylabel to
                      # label the y axis
len(x) # gives the number of elements in array x
INT(x) # gives just the integer part of the number stored in x. Could also replace
       # x with an expression such as INT(x/y) where x and y are two numbers.
np.sin(x) # Calculate the sine of x (assumed to be in radians).
np.arcsin(x) # Calculate the arcsine of x.
x**2 # Calculates x squared. Note that x**y calculates x to the power of y, where
     # y is any number
np.sqrt(x) # Calculates square root of x
```

Python programming exercises

Below we have provided 11 exercises that put into practice the programming concepts and commands discussed earlier in this document. Your teacher and visiting project teaching assistant (if present) have been provided with sample codes that solve each task, so feel free to ask for hints if you get stuck.

1. Write a program that outputs 'Hello World' to the terminal
2. Write a program that outputs 'Hello World' 50 times using a for loop
3. Write a program with a for loop that iterates 50 times and outputs 'Hello World' after every 5 iterations.
4. Write a program with a for loop with 50 iterations and outputs 'Hello World' every odd value of the loop counter and, and 'Goodbye World' every even value.
5. Read in the text file 'xy.txt' and print the data to the terminal in 2 columns
6. Plot the data in 'xy.txt' with blue dots
7. Plot the data in 'xy.txt' with read crosses
8. Plot the data in 'xy.txt' as a line plot
9. Find the individual sums of the 'x' data and the 'y' data and print them to the terminal
10. Determine the individual means for the 'x' and 'y' data and print them to the terminal

11. Write a code that creates 3 variables called ‘day’, ‘month’, ‘year’. Get the code to ask the user to input values for each variable, and then output the values is the form “Todays date is:” dd/mm/yyyy. Now run the code using todays date as the input.

Kepler data analysis exercises

Light curves from the Kepler mission are stored in files whose names indicate the reference number of the star in the Kepler Input Catalogue (KIC). This catalogue was created before the mission launch, so that astronomers had a source of information about the stars that Kepler was going to survey. The light curves are presented in a form in which the change in flux of the star, relative to the flux received when the planet is not transiting, is listed against time measured in Julian Days minus 2454833, such that time zero corresponds to 12 noon on 1st January 2009. Below, we have provided a sequence of tasks that will allow you to determine the key parameters of some transiting planets.

1. Write a programme to plot the light curve (flux against time) for the system KIC 006922244. Note that the data set is stored in the file KIC006922244.tbl that has been downloaded from the Kepler data archive (stored in the DATA directory that you will find in the PlanetHuntingWithPython2019 folder). This contains 3 columns of numbers that must all be read in by your programme. The first column contains non-useful data and should be ignored after it has been read in. Later you will learn how to download your own data files from the Kepler archive, and these will have the same format as KIC006922244.tbl. Note that the first 3 lines of this file also need to be ignored, so you need to use the *skiprows=3* option in the *np.loadtxt* command that you use to read the data file.

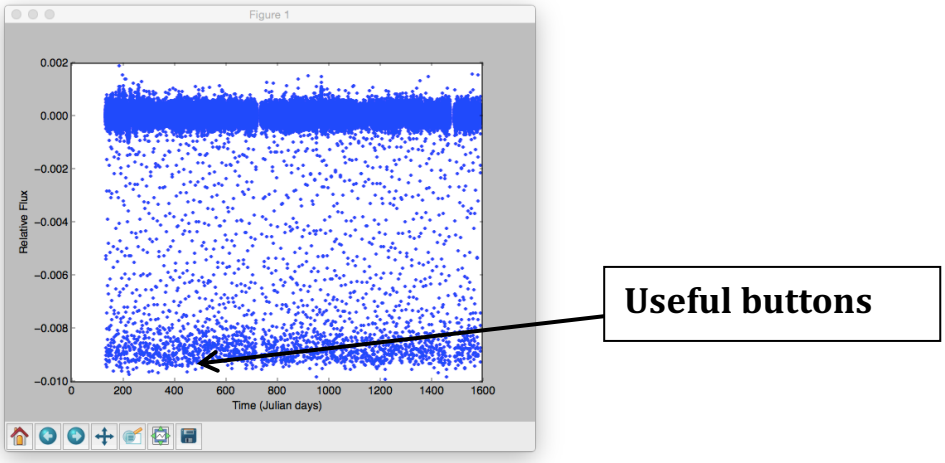


Figure 6: This figure shows the light curve contained in the data file KIC006922244.tbl

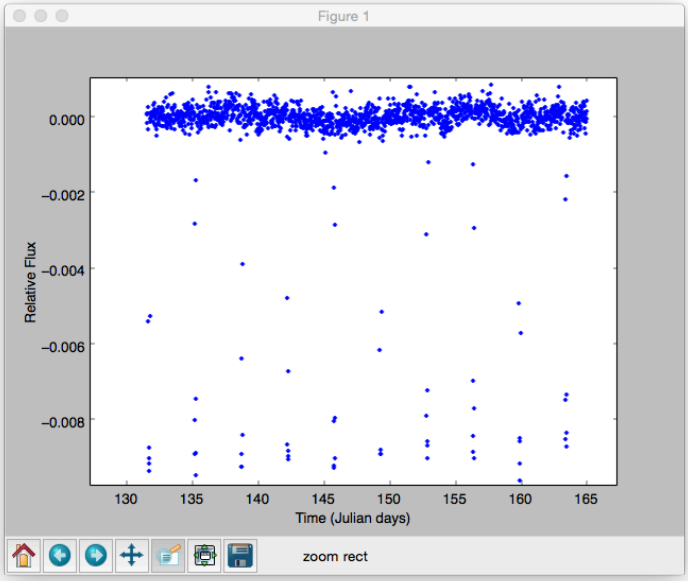


Figure 7: A zoom in on the first 10 transits

2. Find the transit signal for KIC 006922244 and estimate its period. Note that the plotting window that pops up when performing task 1 above has buttons to allow you to zoom in on different regions of the plot (i.e. press the magnifying glass – and expand the plot by dragging the cursor across the required area). Furthermore, when you move the cursor in the plot window the position of the cursor arrow should be indicated in the lower part of the window frame, and this is a useful feature for determining the time of a transit mid-point and other parameters. It is worth exploring what these buttons do as you will use them frequently in this project. To find the period, we advise finding a region of the light curve without any breaks containing 10 transits (note that when you zoom in on the light curve you will see many small breaks in the data corresponding to times when the space craft was downloading data or was firing its thrusters). Find the time of mid-transit for the first and last of these transits (make sure you zoom in on each of these and use the cursor values to get an accurate estimate), and then divide the time interval between the first and tenth transit by the number of transits minus 1 (i.e. 9 in this example). You should now have an estimate of the orbital period.
3. Write a programme to phase-fold the light curve so that all transits in the data lie on top of one another, and plot the flux with respect to the phase

(**Note: Hints about generating a phase folded light curve are given in the appendix at the end of this document! We strongly recommend that you read this. If you get really stuck then your teacher has the Python code needed to do the phase folding**). Phase folding corresponds to just shifting the time coordinate of the data by the correct amount so that the transits all occur at precisely the same time. You should arrange your data so that the phase-folded light curve has the mid-point of the transits at phase 0.0, with the light curve before and during ingress occurring with a negative phase (with minimum value = -(orbital period)/2) and egress occurring with positive phase (where the maximum time shown is +(orbital period)/2 - see the figure below). The phase should be represented in units of Julian days either side of zero. You can add the following line to your script to sort the data before plotting it, where *t2* is an array that contains the new shifted time coordinate for all of the original values of the relative flux:

```
t2, flux = zip(*sorted(zip(t2, flux))) # This uses the t2 array to sort both arrays t2 and flux. i.e.
                                       #  it sorts them with respect to time
```

Note: The quality of your phase-folded curve will depend *very sensitively* on your estimate of the orbital period. You also need to know the time of the first transit mid-point to perform the phase folding. In the example shown below, to get the phase-folded data in the left panel we estimated the period to be 3.522X where the X represents a digit that you will need to determine for yourself. i.e. You will need to have an estimate accurate to four decimal digits to get a reasonable phase folded curve. The right panel shows what happens if the orbital period estimate changes by just -0.0005 Julian Days.

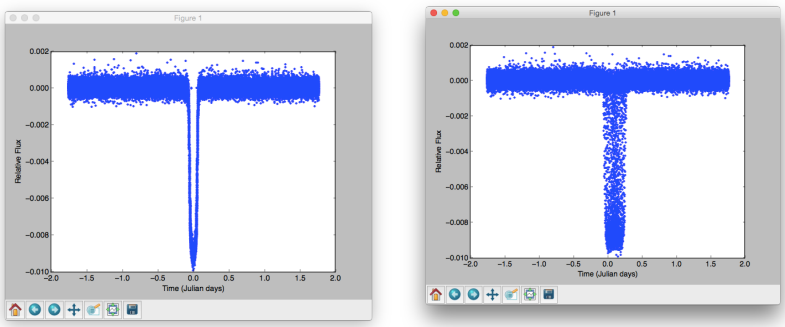


Figure 8:The left panel shows the phase-folded light curve with an accurate estimate for the orbital period. The right panel shows what happens with a slightly inaccurate estimate of the period.

4. When satisfied that the period is accurate, obtain the following parameters from the phase folded plot (using the cursor and printed values in the plot window will be very useful here!):
    a. Maximum depth of the transit (noting that this should contain an approximate average of the noise in the data).
    b. Time (or phase) that the transit begins, when the planet just begins to pass in front of the star (ingress start time)
    c. Time that the transit first reaches maximum depth, when the whole planet is in front of the star (ingress end time)
    d. Time that the transit reaches its midpoint.
5. Now construct a piece-wise function to act as a model that you will fit to the data using the parameters determined in task 4. You can assume the transit is symmetric about the midpoint. The model should include a horizontal line y=0 outside of the transit, since that is the mean value of the data there. The model function between the start of ingress and the end of ingress should be represented as a straight line with an appropriate slope. The time interval between the end of ingress and the beginning of egress, corresponding to maximum transit depth, should be represented as a horizontal line with y=minimum flux. Plot the phase-folded Kepler data and then the model on the same plot (you need to do the plotting in this order so that the data plot does not obscure the model). Plotting the model on top of the data is achieved by issuing consecutive plt.plot commands. See Figure 10 on the next page for an example how your model and data should look when plotted. As in task 3, you should add the following line to your script to sort the data

```
time3, flux = zip(*sorted(zip(time3, flux))) # This uses the time3 array to
                                             # sort both arrays time3 and
                                             # flux. i.e. it sorts them with
                                             # respect to time
```

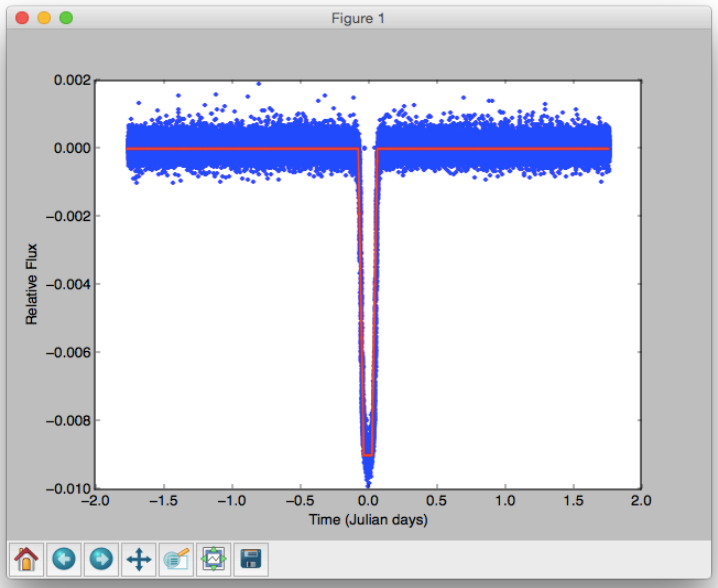


Figure 9: Plot showing the model (red line) overplotted on the data (blue dots)

6. Now determine an error estimate for your model using the following equation for the Chi-squared test:

$$\chi^2 = \frac{1}{N}\sum_{i=1}^{N}\frac{(F_i - F_m)^2}{\sigma^2}$$

where F_i is the data, F_m is the flux predicted by your piece-wise linear function (the model) corresponding to the time/phase associated with F_i, and σ^2 is the variance of the data about its mean. Although not completely accurate, when calculating the mean and σ^2 for your data you should use all data points, including those corresponding to the transits, since in practice this makes only modest difference compared with removing the contribution for the transits. Remember that the variance is really just estimating the noise in your data, or the level of scatter about zero. A model fit that is consistent with being within the noise of the data should have a χ^2 value less than 1.

7. Repeat steps 4-6 for a maximum of 10 times to obtain an improved model for the data, where the best fit is obtained when χ^2 has its minimum value.
8. Using the your best fit parameters, write a programme (or extend your existing programme) to determine the following planet parameters (you

can use the stellar parameters located in the file 'DATA/KIC006922244_STELLAR.txt'):

a. Planet radius (in units of R_Earth)
b. Orbital Period (days)
c. Semi-major axis (AU)
d. Transit impact parameter
e. Planet orbital inclination

The values you obtain should be similar to these:

Planet radius = 15.038 Earth radii (Earth's radius is approximately 6.37 x 10^6 m)

Period = 3.5224

Semi-major axis = 0.04712 AU

Impact parameter b = 0.66256

Inclination = 84.5677 degrees

9. Now perform the same analysis for the other Kepler systems for which data files are present in the directory DATA: KICXXX... 8359498, 11853905, 6922244, 2571238, 10418224, 9631995, 7950644, 5881688

Suggestions for further independent research

We have tried to provide a structured set of exercises to guide you through the process of obtaining physical information about planetary systems from Kepler light curves. Here are some suggestions for how you might extend your work.

1. Download data from the Kepler archive and analyse a much larger number of systems (see below for detailed instructions on how to download data from the Kepler archive). Use the planet parameters that you obtain to determine information about the planet population that is orbiting the Kepler target stars. For example, determine the frequency distribution of planets with different radii. How many giant planets are there (e.g. Jupiters) versus smaller Neptune-sized bodies? Which are more common – larger or smaller planets? How are the planets distributed as a function of their orbital periods?

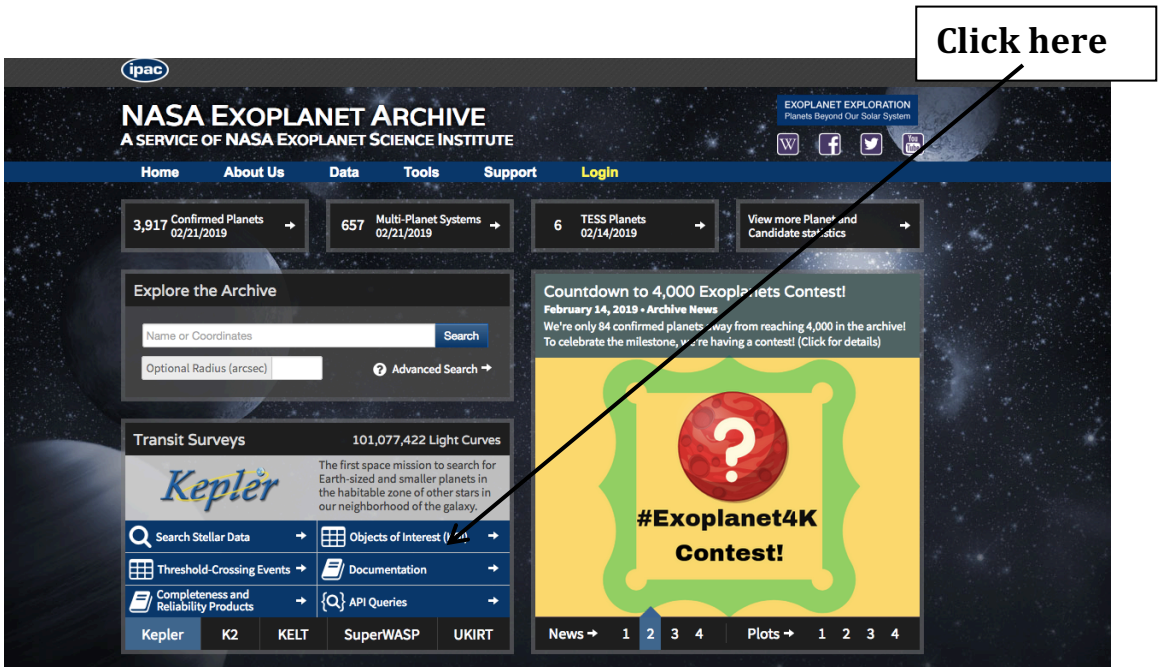
2. For each system you have examined, implement an automated search through the different model-fit parameter values in order to obtain a best-fit solution (we suggest that you use a single value for the orbital period and vary the other model parameters due to the sensitivity to changing the period). Here, you will need to implement a sequence of nested for loops that scan through a range of parameter values. For each independent set of parameter values you should apply the Chi-squared test described earlier, and select the model with the lowest Chi-squared value.

3. Download light curves for some systems known to have two planets. Use your previously developed codes to find the period and parameters of the most obvious transit signal. Write a routine to manipulate the light curve data and remove this dominant transit signal (you can do this by simply adding the negative of the model-fit to the data which then removes the transit signal, or you can remove the dominant transits by replacing the data during each transit with data copied from the time between transits), and now apply your programmes to find the orbital period and system parameters of the 2nd planet.

4. For a single planet orbiting a star, the orbital period should not change. Write a programme to examine whether or not the orbital period is changing for the systems that you have examined (if changes are occurring then these are known as Transit Timing Variations – or TTVs for short). One way of doing this is to create a plot of the original detrended light curve data and to superimpose on it a plot of the model fit applied to this whole time series rather than to the phase-folded light curve. By eye inspection should then tell you if the period remains constant since the model should fit all transits. See if you can find a way of automating the search for TTVs, and plot the TTVs versus time to see if there is any statistically significant evidence for real TTVs. Choose some 2-planet systems where there may be gravitational interactions between the planets that could perturb their orbits and induce TTVs. Can you find evidence of planet-planet dynamical interactions which are systematically changing the orbital periods?

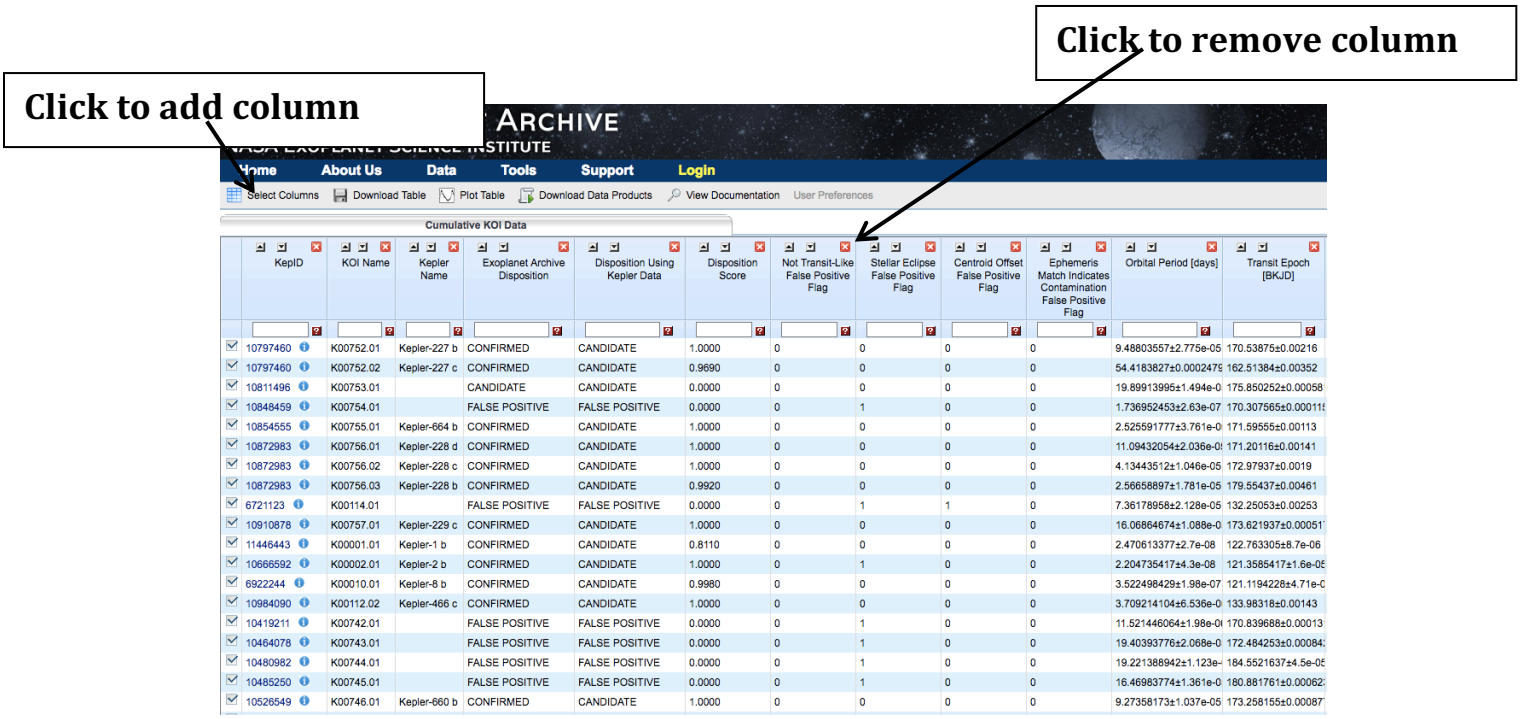
To download light curve data from the NASA exoplanet archive go to

http://exoplanetarchive.ipac.caltech.edu



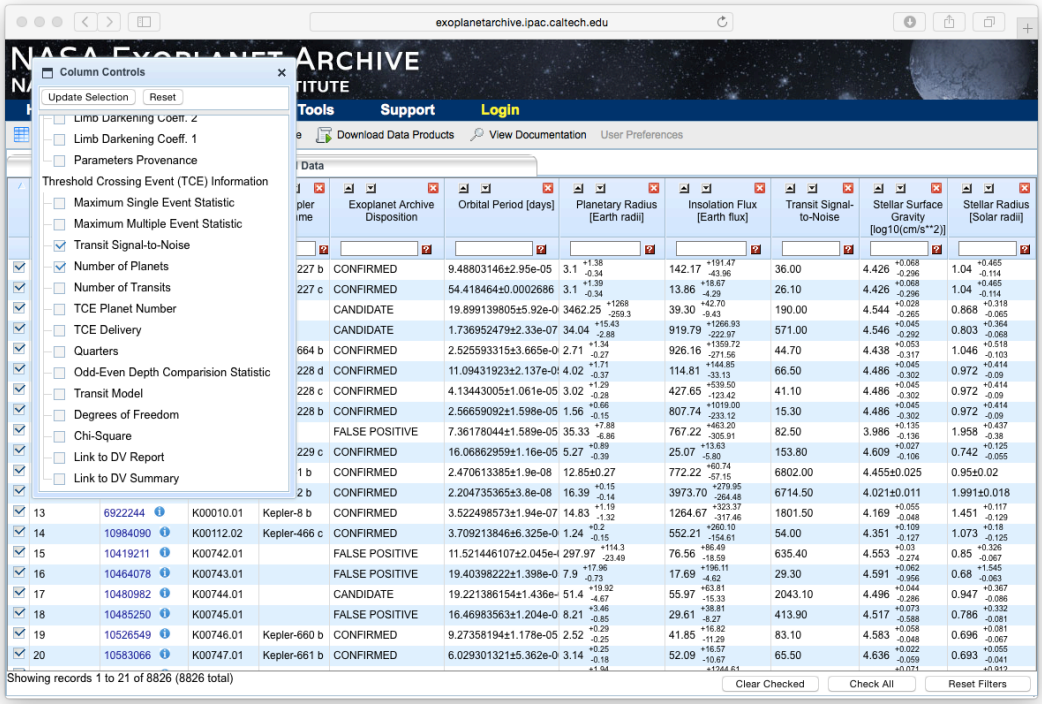
Undertake the following steps to select the systems that you wish to download data for. Our advice is to only download systems for which there are confirmed planets, and for which the planet transit signal is much bigger than the noise. This normally means that you will select planets with fairly large radii so that the transits can easily be seen by eye in plots of the light curves.

The first step is to create a list of Kepler systems that you wish to download the data for. Click on the button indicated on the previous figure showing the NASA archive web page. This will load an interactive table:



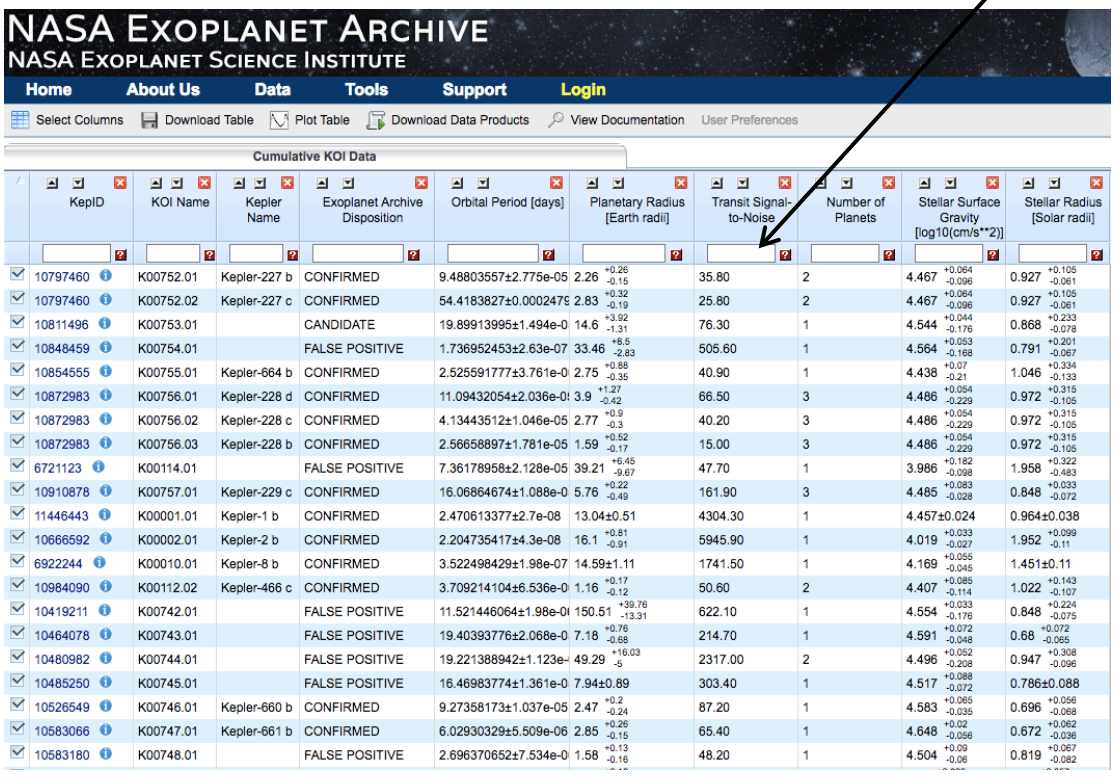
You will notice that the table contains information on many systems, and also contains information that is not relevant when selecting a list of planets (it scrolls rightwards a long way!). The red boxes containing white crosses at the top of each column can be used to remove those columns. We suggest culling the data by removing the following columns (moving from left to right): Disposition Using Kepler Data; Disposition Score; Not Transit-Like False Positive Flag; Stellar Eclipse False Positive Flag; Centroid Offset False Positive Flag; Ephemeris Match…; Transit Epoch; Impact Parameter; Transit Duration; Transit Depth; Equilibrium Temperature; Insolation Flux; TCE Planet Number; TCE Delivery; Stellar Effective Temperature; RA; Dec; Kepler-band.

Now we need to add two columns by clicking on the *Select Columns* button at the top left (see previous figure). Scroll down and select the Number of Planets option and the Stellar Mass option and click on the Update Selection button in the drop down menu (see figure below).



Type in here to select data

You should now have a table that looks like this



Now we need to cull the data further to obtain a set of planets for which your search and parameter fitting algorithms will work. Systems for which the transit signal is not much larger than the noise need much more sophisticated routines than the ones we have discussed already. Text boxes are provided at the top of

each column to select data according to user defined criteria. First type “>100” into the Transit Signal-to-Noise text box, as shown in the figure above. The table will automatically update. Now make the following selections (after which your table should have about 97 systems listed as shown in the figure below).

Number of Planets: 1

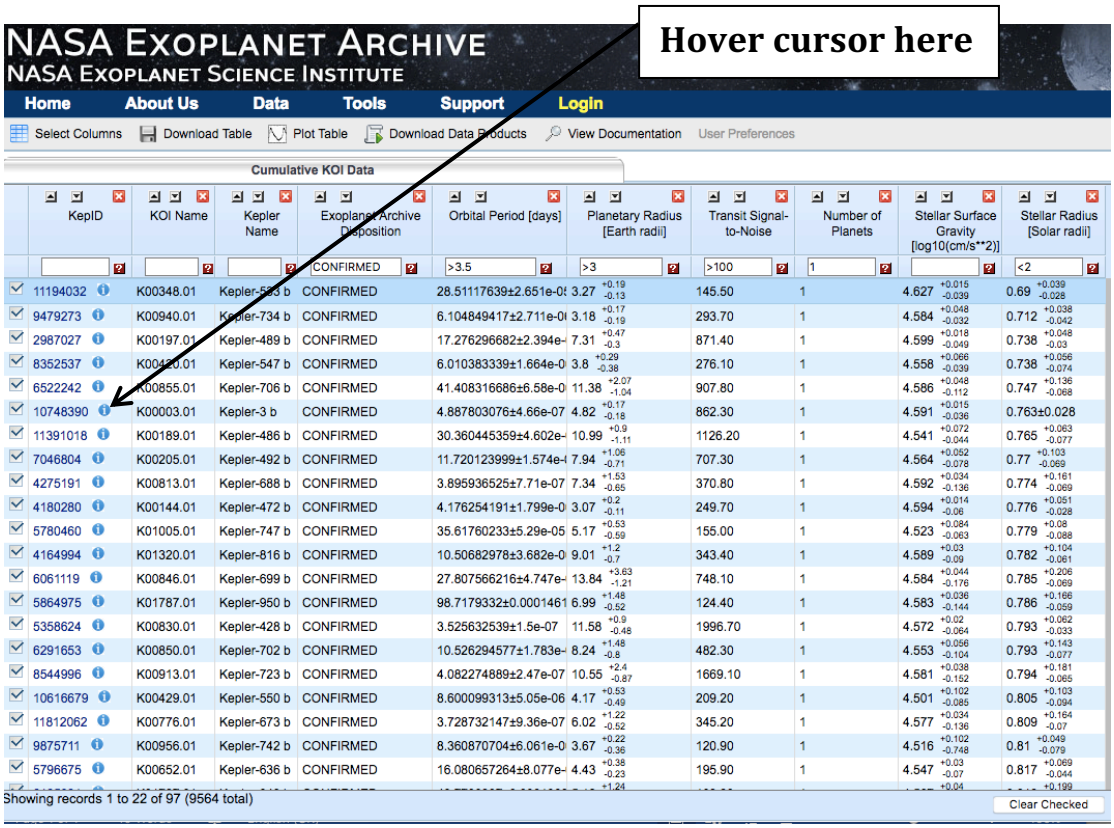
Exoplanet Archive Disposition: CONFIRMED

Orbital Period: >3.5

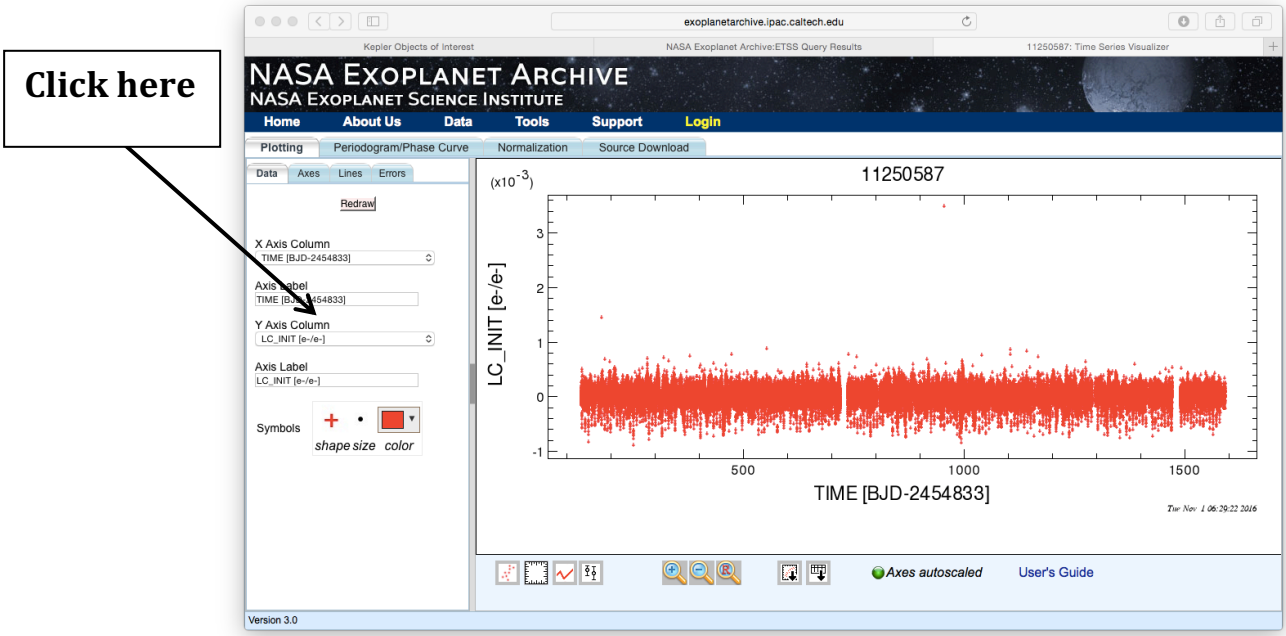
Planetary Radius: >3

Stellar Radius: <2

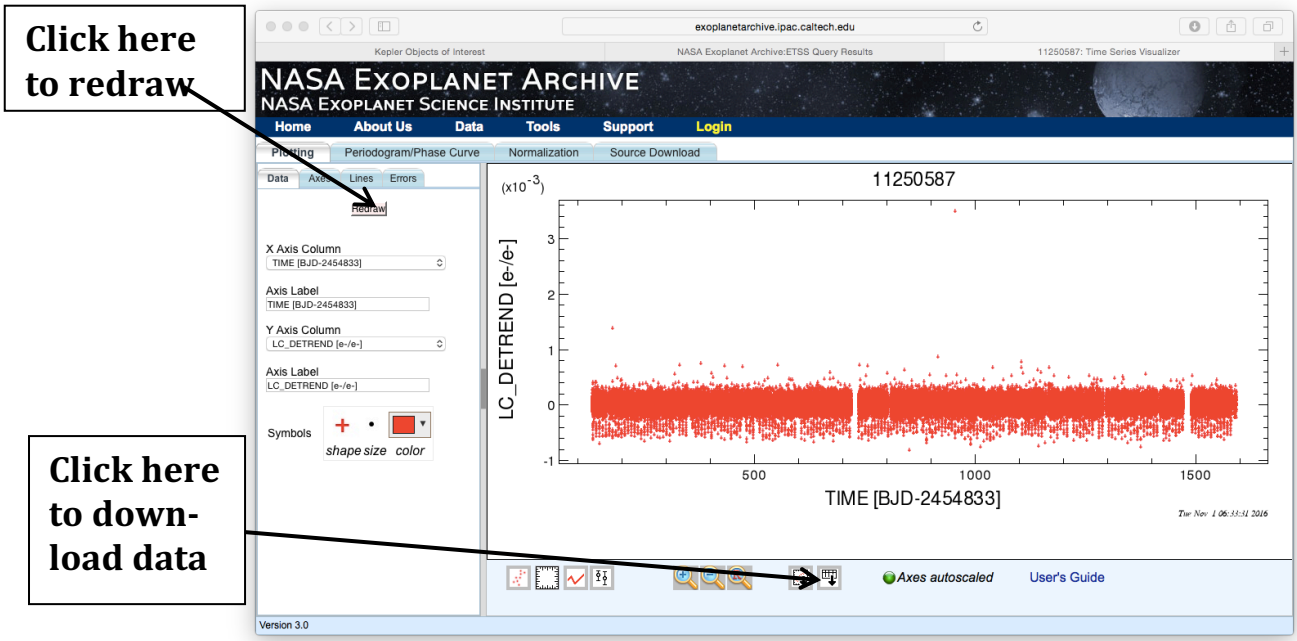
Now choose which system you want to download the data for. *Make a note of the Stellar Radius and the Stellar Mass as these will be required when determining planet parameters from the light curves.*



Hover your cursor over the “i” symbol next to the KepID number that you want to download data for. A pop-up menu should appear when you hover over the “i” symbol. Scroll down and click on the option “Kepler DV Time Series and Periodogram”. A plot should now appear (as shown below). The example below is for KepID 11250587.



Now we want to plot the detrended light curve (this is a cleaned-up version of the light curve provided by the Kepler mission) which can be obtained by clicking on the Y Axis Column option, as shown above, choosing LC_DETREND from the menu, and then pressing the redraw button, giving the following plot.



To download the detrended light curve in the form of a data file, with the same format that was used in the Kepler programming exercises provided earlier in this document, click on the button indicated in the above diagram. This will download a file called plot.tbl that contains the light curve in a text file with three

columns and a three-line header. You now have a data file to analyse using your previously developed algorithms and programmes. Change its name to match the KIC number and you're ready to go.

To download data on systems containing more than one planet, repeat the above steps, except choose Number of Planets = 2 to extract data on 2-planet systems. A recommended 2-planet system to download is Kepler 117.

References

Mayor, M., Queloz, D., A Jupiter-mass companion to a solar-type star, 1995, Nature, 378, 355-359

Borucki, W.J. et al, Keper Planet-Detection Mission: Introduction and First Results, 2010, Science, 327, 977

Information about the Kepler mission: https://kepler.nasa.gov/index.cfm

NASA Exoplanet Archive: http://exoplanetarchive.ipac.caltech.edu

Exoplanet Data Explorer (useful for generating plots of exoplanet properties and exploring the statistics of the exoplanet population): http://exoplanets.org

Extrasolar Planet Encyclopedia (similar to the Exoplanet Data Explorer): http://exoplanet.eu

A reference site for documentation on python commands and libraries: http://scipy.org
(Click on the documentation button at top of the page)

Appendix – Hints on how to phase fold the light curves

Below we provide a sequence of steps for phase folding the Kepler light curves so that we obtain data in the same format as that plotted in the left panel of Fig. 9. Figure 12 shows a zoom-in of the light curve for KIC006922244, the example used earlier in this document. It shows the time of the first transit, indicated by the black arrow.

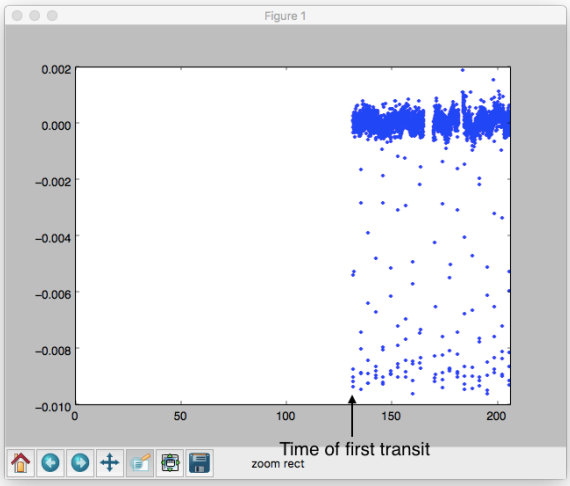


Figure 10 This image shows a zoom in of the KIC006922244 data showing the time of the first transit

Step 1: Determine the time of the first transit. Let us refer to this as t_transit1. Also determine the period with which the transits occur. We use the symbol P to denote the period.

Step 2: We now want to shift the light curve data to the left in such a way that the first transit occurs at a time t=P/2. i.e The time of the first transit should now occur at a time that corresponds to half the orbital period. This step can be implemented in python by simply subtracting the appropriate number from all of the time values contained in the light curve. The result of doing this is shown in Figure 13.

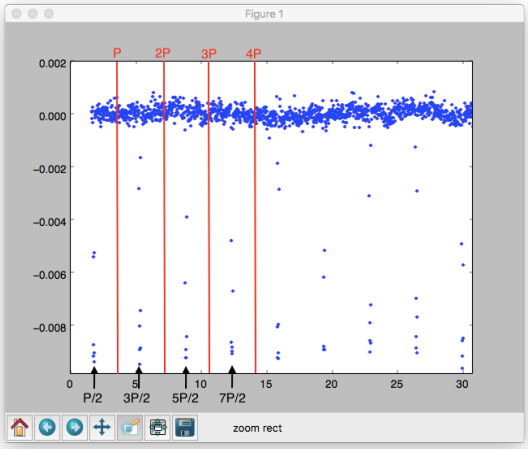


Figure 11 This figure shows the KIC006922244 data after the time coordinate has been shifted so that the first transit occurs at t=P/2, where P is the period associated with the transits. The black arrows indicate that successive transits occur at t=P/2, 3P/2, 5P/2, etc... The red vertical lines indicate the positions in time that correspond to one period, two periods, three periods etc.

Step 3: We now need to use modular arithmetic to shift sections of the light curve so that they are superimposed on one another. This is the essence of phase folding the light curve. Consider two decimal numbers, A and B. In many computing languages the command MOD(A,B) will give the remainder of dividing A by B. In other words, if we write A = N x B + R, where N is an integer, then MOD(A,B)=R. It should be clear from looking at Figure 13 that if t[i] contains the times associated with each of the points on the shifted light curve, then applying MOD(t[i],P) to all points will create a phase folded light curve with the mid-point of the transit occurring at a time t=P/2. The result of implementing this in python is shown in Figure 14. **Important note: In python, we type (A%B) and not MOD(A,B) to obtain the remainder of dividing A by B.**

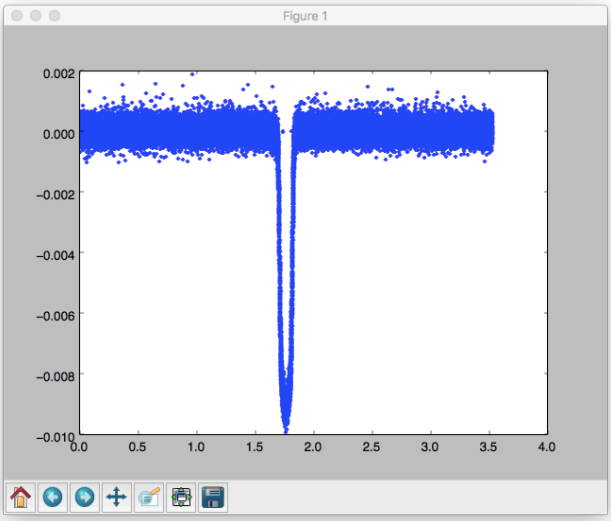


Figure 12 This image shows the KIC006922244 data after it has been phase folded, with the transits occurring at time t=P/2, where P is the period

Step 4: Now we simply need to shift the light curve by an amount P/2 to centre the transit at time t=0. We do this by subtracting P/2 from all of the time values in the data. The result of doing this is shown in Figure 15. **Remember that phase folding only works if you have an accurate figure for the period!**

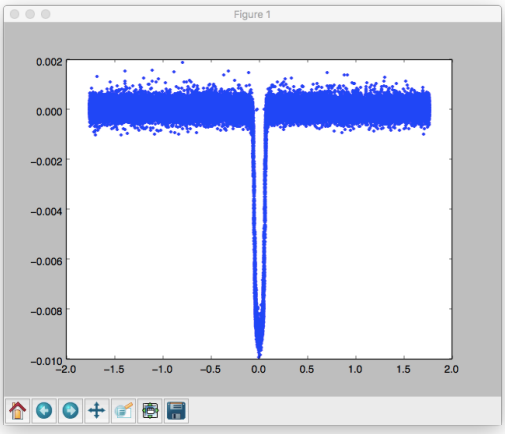


Figure 13 This figure shows the data for KIC006922244 after phase folding and shifting so that the midpoints of all the transits occur at time t=0.

Python code for phase folding

Your teacher has the solution. Try to work it out for yourself before seeking assistance.