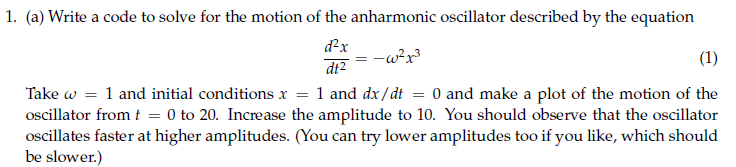
Scientific Computing

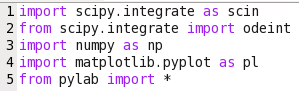
Python Assessment 3

**Question 1**

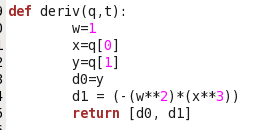


This is a second order differential equation, expressing this into a system of first order equations involves introducing a new variable. I set this variable to as. This can also be expressed as . We can substitute this expression into (1) to give since differentiating our new variable gives.

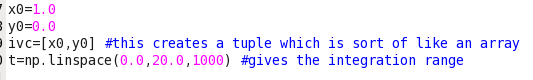
I imported the modules I needed to use in order to use the scipy,integrate.odeint function since its accurate.



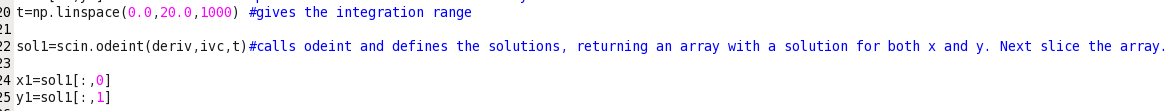
I then defined a function to express the second order differential equation given in the form of the two first order equations formed earlier. I defined my constant omega (w) here also. This function was defined such that it returns a vector, d0 is my and d1 is my .



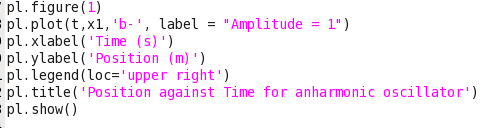
I then went on to define the initial value conditions of the equation, labelled ‘ivc’. I defined the array of points using linspace, evaluating the system from t= 0 to 20 using 1000 points because this will produce a more continuous graph which in turn is easier to analyse. I used linspace as this is a function that takes into account the last term, in this case 20, as opposed to arrange which would not. In order to meet the criteria of applying odeint, the initial conditions need to be provided in a tuple form.



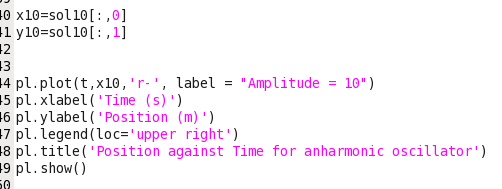
I proceeded to call odeint in order to define the solution which in turn returned an array with a result for x and y which is obtained by slicing the array, as shown below. ‘y1’ in this case is . I have done this here, as I will later use it in my phase plot.



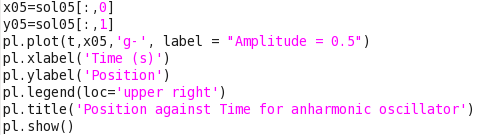
Next, I plot the solution. This is done with x on the y a-xis, and the time t on the x-axis. I assigned a colour to the line produced by ‘b-‘ giving a smooth blue line and labelled it appropriately. Having then proceeded to label my axis with their corresponding units, I assigned a title and typed ‘show()’ so that the code would show me the function as desired.

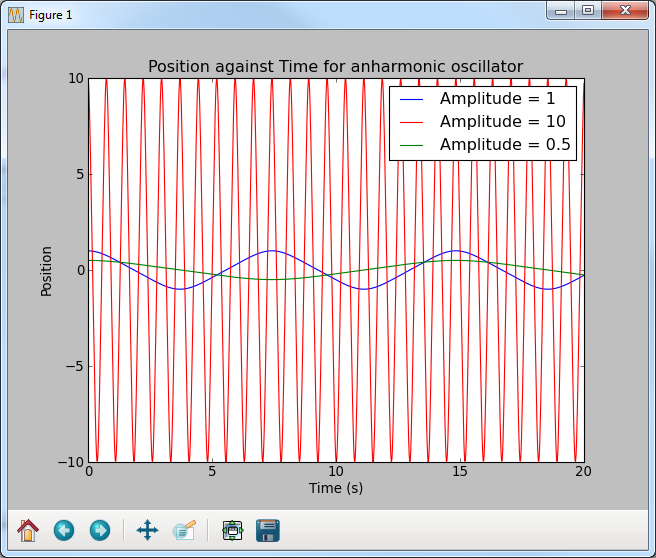


I proceeded to plot a solution with an increased amplitude to 10 in the same fashion as above.



I also produced a solution for a lower amplitude of 0.5 to see if it was slower.

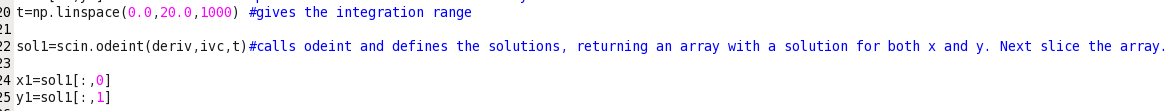


When I wrote the solutions for the graphs with amplitudes of x0=0.5 and x0=10, I did not write the ‘pl.figure()’ since I wanted all the solutions shown on the same graph so that they are easier to compare. The result was as follows:

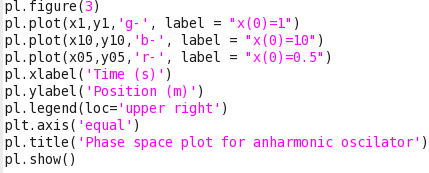
From this we can see that increasing our amplitude, increases x0 and increases the frequency of the oscillations. Comparing the result of x0=10 against x0=1, we can see that the result of x0=10 (the red line) oscillates ten times faster than that of x0=1. Also, the result of x0=0.5, displays half the speed of oscillation in comparison to x0=1.



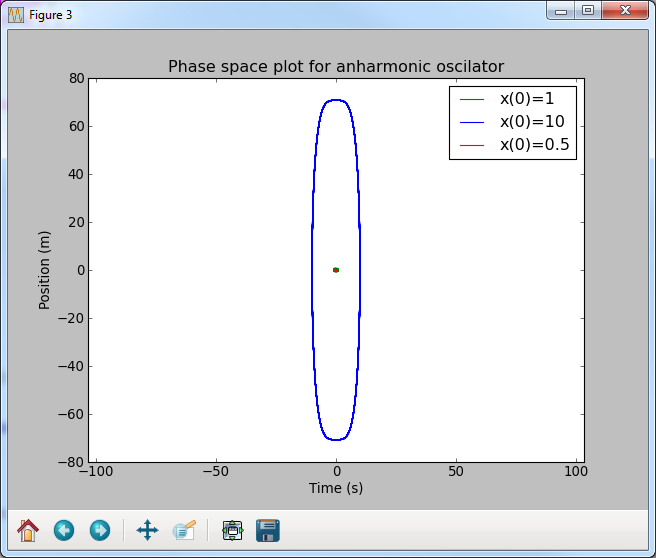
Since I earlier calculated shown below again:



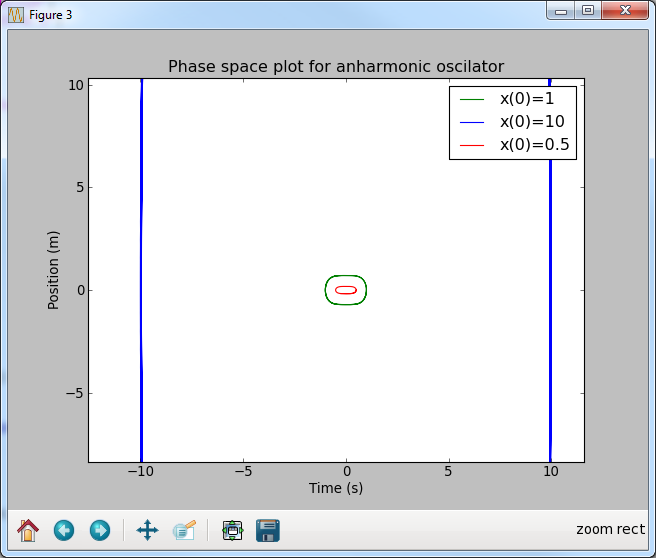
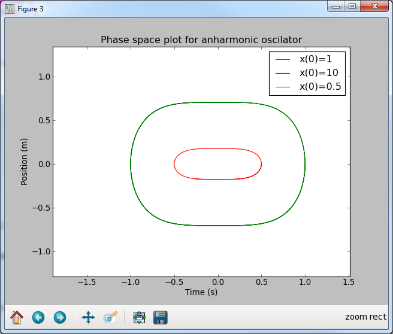
I can go on to plot this (in the same way as shown previously), plotting the result for when the amplitude is a 1, 10 and 0.5 all on the same graph. However, in this case we must specify axis(‘equal’) since it ensures that the scales are kept equal in size.



This produces the following graph:



Here is a zoomed in version of the graph for comparison:



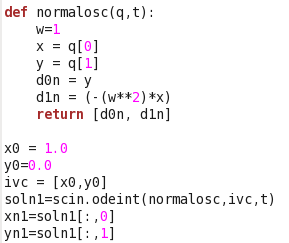
The result produces a curved square of sorts, with a varying gradient since there are areas which on sight seem to show a gradient of 0 and other areas of incline, therefore we can see from the zoomed in image that the shape does not continually vary with time and is periodic. Also we can see that all the solutions have a area at which the gradient is zero, and they all appear to follow a similar shape. However the solutions with greater values of omega (w) are stretched vertically.



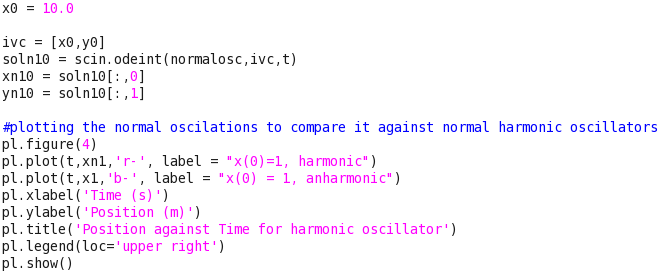
A normal harmonic oscillator can be described by the following equation:

This can be expressed as two first order equations, much like in part a where and

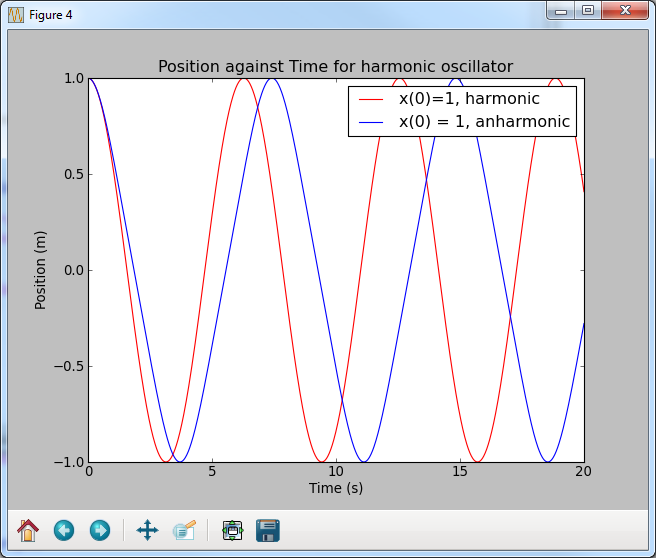
I proceeded to define the second order differential by these two first order equations in python where the initial conditions have not changed. I redefined it since the previous value for x0 was 0.5 and proceeded to get the solution much like I did previously:



I went on to obtain the solution for when omega is 10 for the new harmonic oscillator on the same graph.

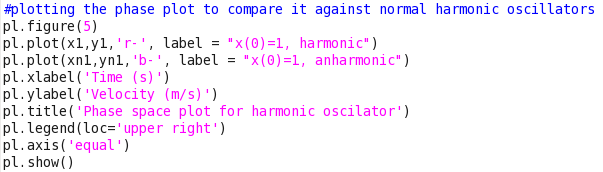


This generated a graph that was like so:

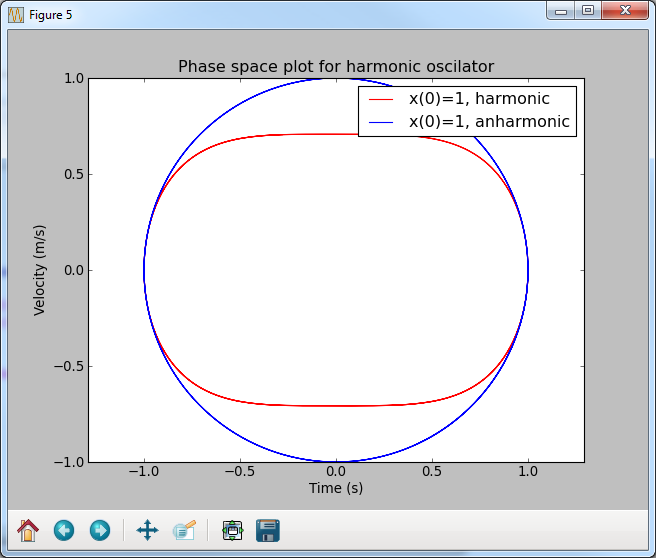


From this we can see that the anharmonic oscillation has a higher frequency in comparison to the harmonic.

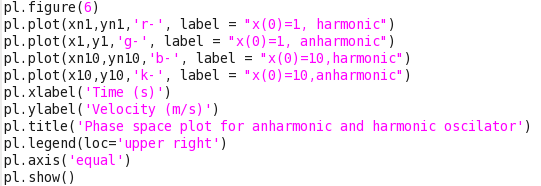
Comparing the phase space plots were done by writing the following:



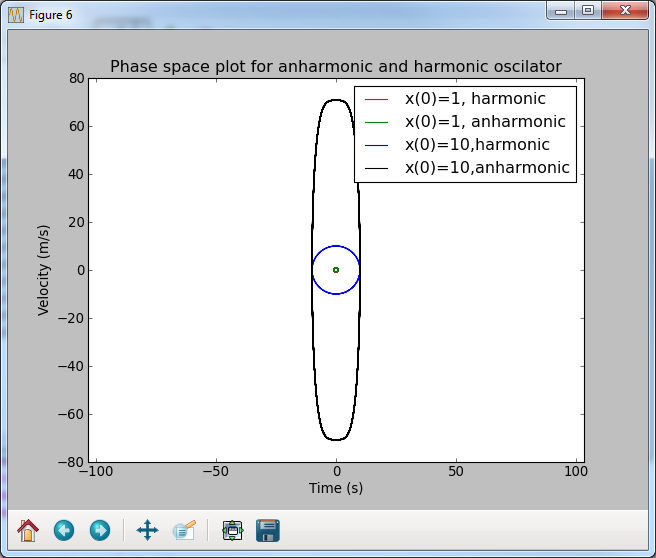
This produced:

We can see that the most quantifiable difference here between the harmonic and anharmonic oscillator is that the harmonic appears to be more circular with a greater amplitude.

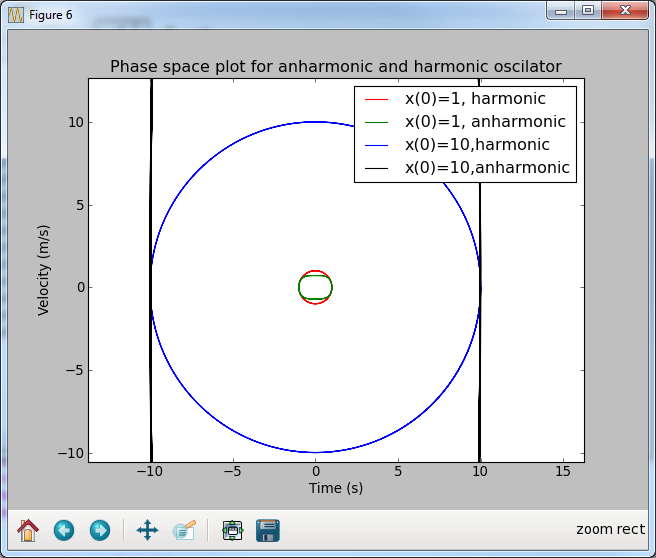
Finally, I plotted all of the graphs to compare.

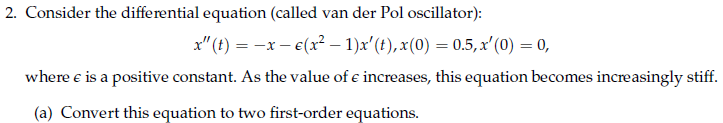


This produces:



A zoomed in version of this:

Like before, we can see the notable difference in shapes- harmonic being more circular and anharmonic being slightly more rectangular.



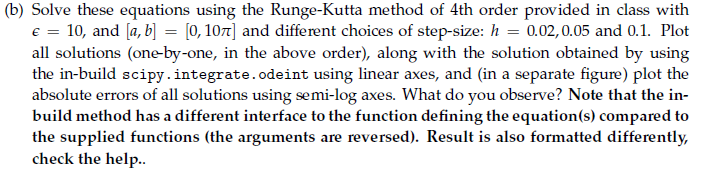
I will convert the given equation into two first order equations in order to solve it on python using odeint/integrate since this is a requirement of those modules.

Creating a new function like in question 1:

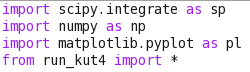
Therefore we can state that and that.

And so the two equations are as follows:

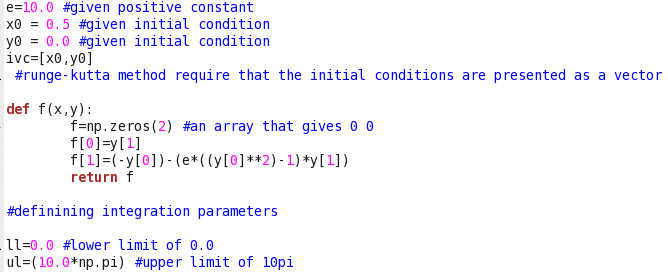
The initial conditions are that x(0)=0.5 and x’(0)=0, which when subbed into our newly formed first order differential equations gives and.



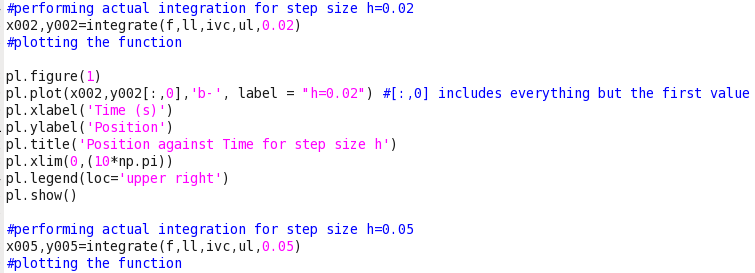
I imported all the functions I will need:



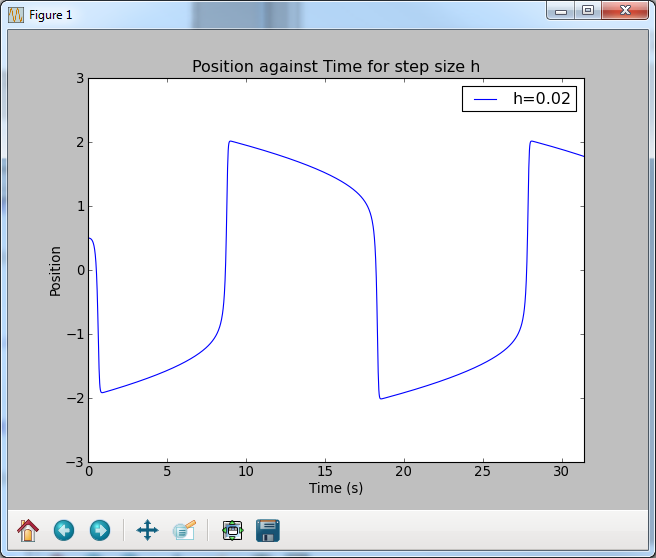
I defined the initial conditions, and a function describing the two first order differentials I formed. In order to create the array that we define f against, I wrote the line f=np.zeros(2). I also went onto define the parameters of the integration, i.e the lower and upper limits.



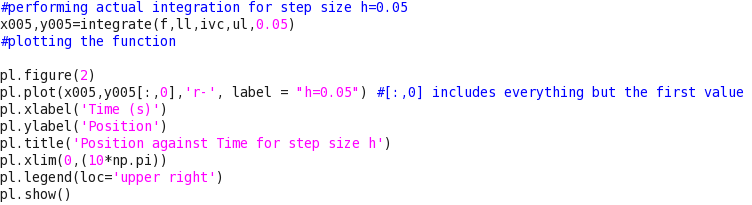
I then integrated the function from run\_kut4 to obtain the first solution with a step size of h=0.02. I went on to plot this.



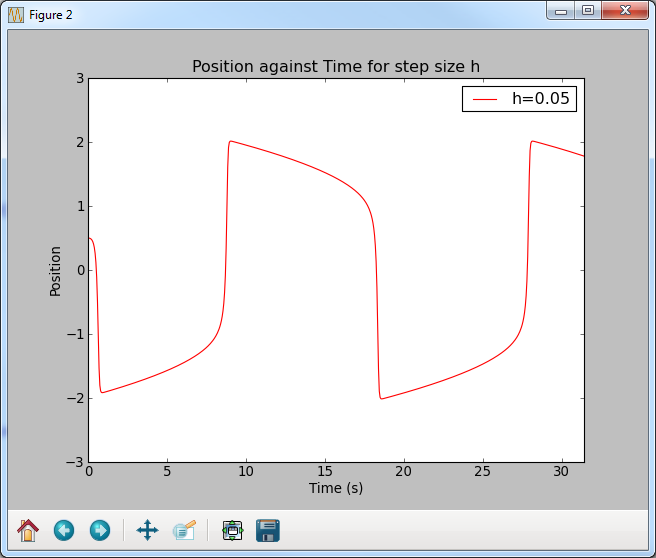
Restricting the x-axis by applying limits within the code of my plot enabled the graph to be shown more clearly. The graph produced looked like this:



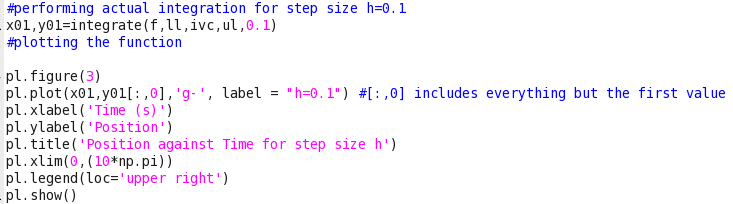
I repeated the same process in order to obtain a solution for h=0.05



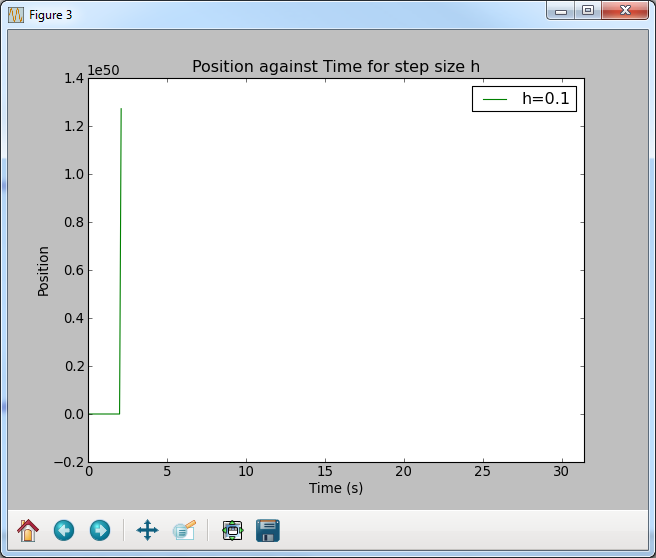
This produced the following graph:

This graph looks very similar to the result of h=0.02. On close inspection of the graphs, one can see that they aren’t exactly the same as there is a divergence between h=0.05 and 0.02.

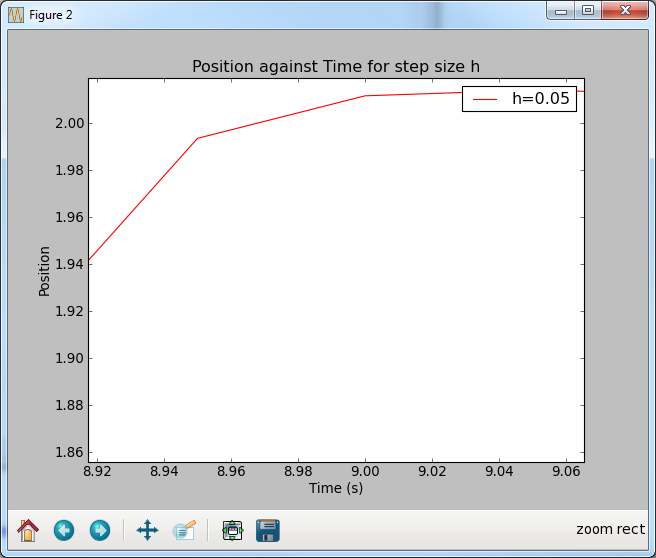
I followed the same process to obtain a solution for h=0.1

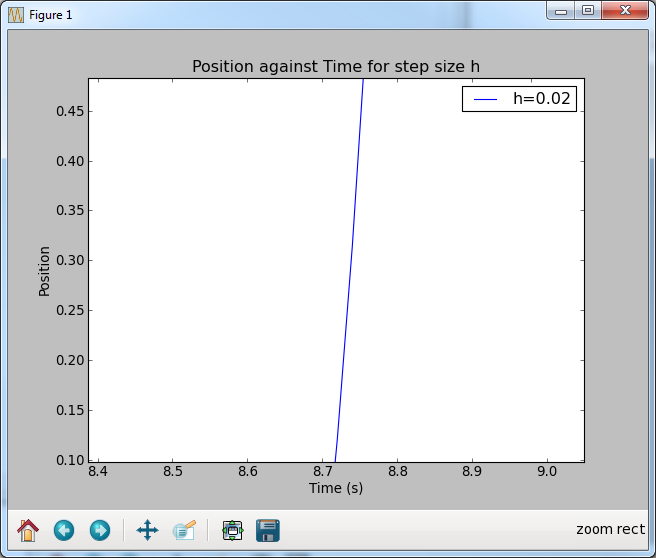


This produced the following graph:



Comparing all three graphs, we can see that the larger the step size h, the less smooth the graph is:

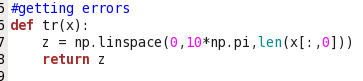




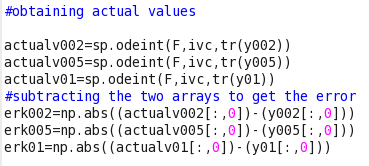
At h=0.02 its significantly smoother than at h=0.05 above

Lets define the error where the accurate value is the ‘true’ value and the approximated value is our RK solution.

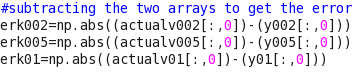
In order to subtract the two, the sizes of the array have to be equal. So I calculated new arrays.



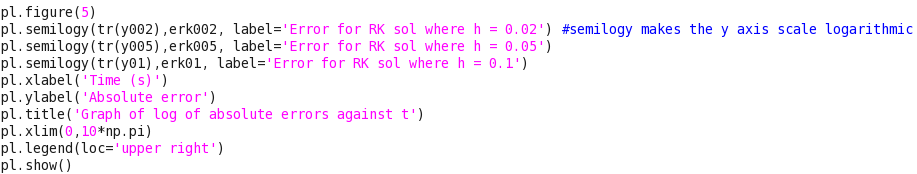
The true value were formed by doing the following:



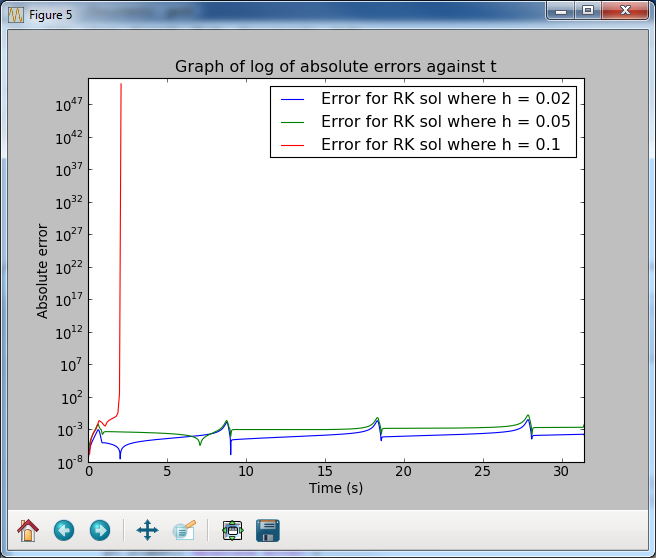
Since I now have two arrays of equal length, I subtracted them to get the absolute error to each Runge-Kutta solution.



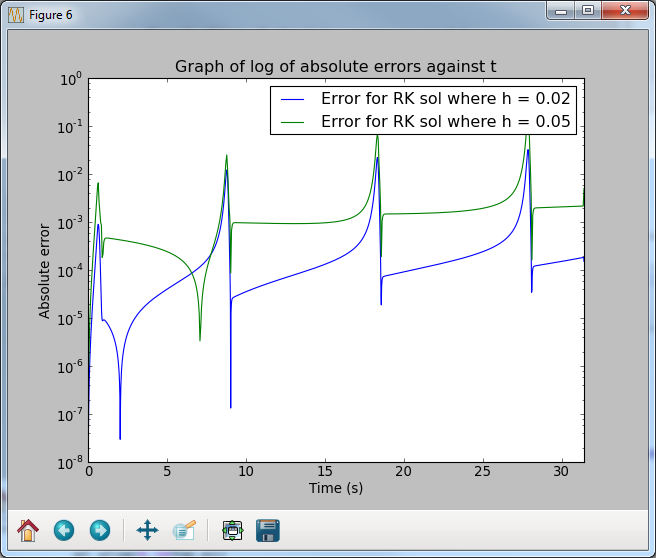
I plotted the three arrays in the same way as usual but this time, I use semilogy in order to make the y-axis scale logarithmic because it shows the change in errors better.



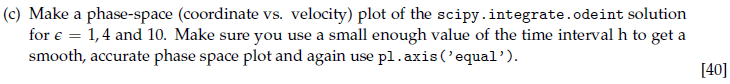
The graph produced:



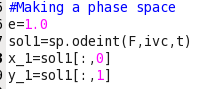
The graph shows that at h=0.1, the error is extremely large as expected. Because of this, I plotted the graph again without h=0.1 in order to compare the errors of h=0.02 and h=0.05 more accurately which produced the following graph:



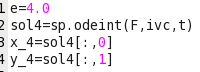
From this we can see that when the step size is h=0.02 the error is smaller, i.e the smaller the step size, the smaller the error. H=0.02’s smallest error is smaller the h=0.05’s smallest error and h=0.05’s largest error is larger than h=0.02’s largest error. So the smaller the step size, the smaller the error, the more accurate the solution. Given the cyclical nature of the graph, one can assume that this may be an inherent part of the runge-kutta method when solving differential equations.



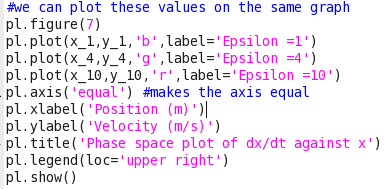
Obtaining a solution for first, I can apply this to the function I have previously formed.



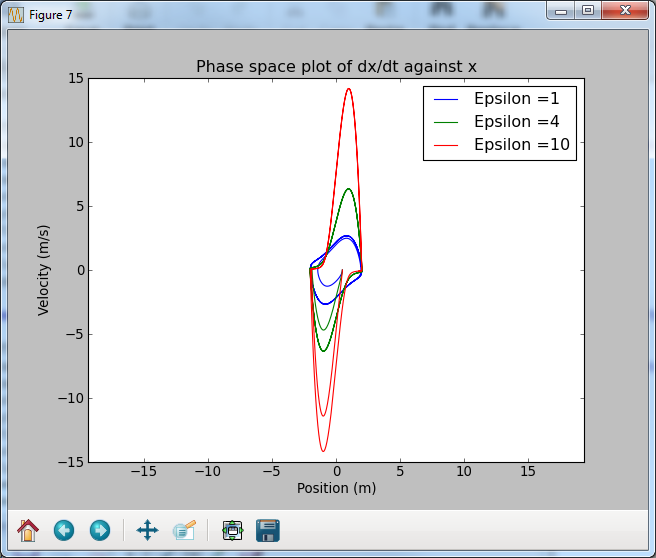
I repeat the process for



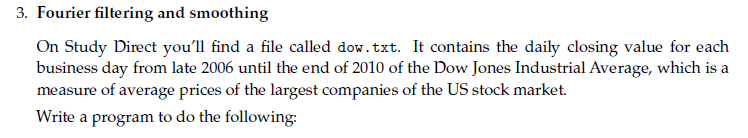
I already worked out a solution for when earlier on in the question. So I proceeded to plot these result together on one graph.



This produces the following graph:



This resultant graph matches our expectations since it is that of a Van der Pol oscillator since the oscillations gradually become stable and enter a ‘limit cycle’. We can see as increases, the maximum phase increases.





I downloaded the dow.txt file from study direct and placed it in a folder I’m running my code from. I then imported the necessary modules in order to proceed:



In order to load the text document I used numpy.loadtxt as it will convert the content into an array which was defined as prices. (I show my graph at the end of the question).





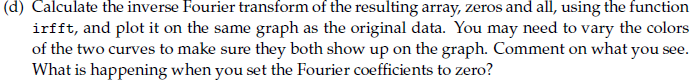
Next, I created a new variable called ‘coeff’ that assigned to be the coefficients of the new discrete Fourier transform of the array prices. This is numpy’s in-built function for a fast fourier transform in one direction. We userfft () rather than fft() since we are dealing with real numbers and not complex numbers.





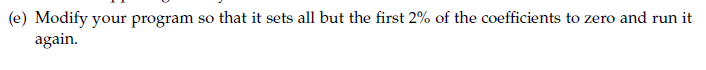
The len(coeff) gives the number of components in the array, its then divided through by 10 (10%), since python automatically rounds (if we have integer/integer) we don't need to worry about rounding manually. The ‘:’ acts as an 'onwards', so it will set every element after the first 10% to 0.





Setting the variable inverse1 to be the inverse Fourier transform of the array (data) prices by using the function irfft.





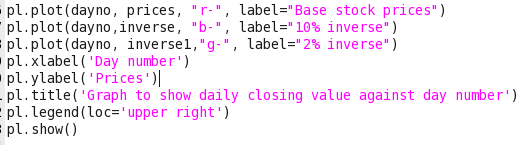
I repeated the process, having created a new variable called ‘coeff1’. I divided the length by 50 since it’s the same as multiplying it by 0.02 in order to get the first 2%.



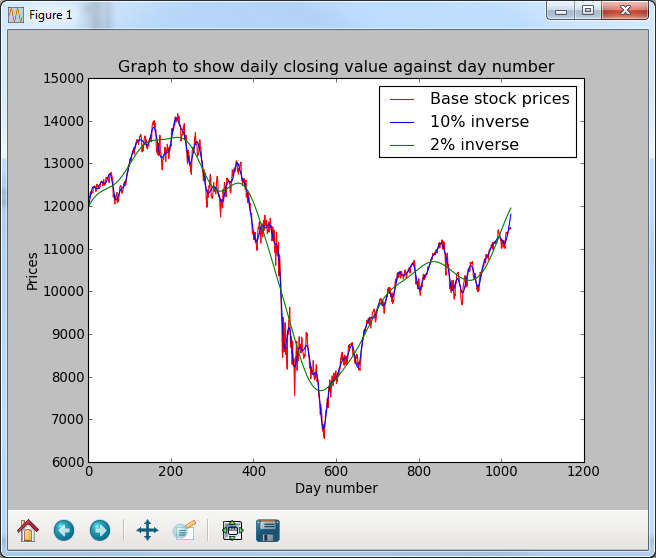
Np.arange will create an array filled with consecutive numbers up to our chosen argument which in my case was 1024. I labelled this array as ‘dayno’.



Finally, I plotted the raw data, the inverse fourier at 2% and the inverse fourier at 10% all against the days (dayno array):



This produced the following graph (please refer to next page):



Since a Fourier series is the combination of a series of sine and cosine waves, our result is as expected for the base stock prices. When we look at the blue line (the 10% inverse), the approximation gets closer with every additional term, by setting the last 90% to 0 we have made the approximation poorer. However, the series which overlooks the 90% of the coefficients is the best approximation. Overall, the series which overlooks 98% of the coefficients is the worst.