Joseph Perkins

Scientific Computing Assignment 3 writeup

Note:

In order to run correctly, some of these programs require `run_kut4` (along with its associated dependencies) to be placed in the directory that the program executes from; likewise, it is assumed that any external data (e.g. Dow Jones index information, `dow.txt`) is placed in the directory. They are not included with my files. It is also assumed that SciPy and its associated tools and libraries are installed and available.

Where relevant, excerpts from the programs that have been written are included in this document. These are **not** always verbatim excerpts, however, and may exclude comments or certain lines and sections.

Question 1

A) *"Write a code to solve for the motion of the anharmonic oscillator described by the equation* $\frac{d^2x}{dt^2} = -\omega^2 x^3$.*"*

We need first to rewrite this second-order differential equation into a system of two first-order equations. The process is relatively straight-forward: we can introduce the new variable $y(t)$ and assign that the value of x'. This gives us the following system:

$$y(t) = \frac{dx(t)}{dt}$$

$$\frac{dy(t)}{dx} = -\omega^2\, x(t)^3$$

Nobody likes to reinvent the wheel, and the wheel in this particular case is a solving routine. We can use `scipy.integrate.odeint()` in this case as it is extremely robust and accurate.

We'll go ahead and import the modules that we'll use:

```
from scipy import integrate
import numpy as np
import matplotlib.pyplot as plt
```

Then we need to define a function that encapsulates our set of differential equations:

```
def deriv(q,t):
      xi = q[0]
      yi = q[1]
      deriv0 = yi
      deriv1 = (-(omega**2)*(xi**3))
      return [deriv0, deriv1]
```

This returns a vector containing our equation for $\frac{dx}{dt}$, which is `deriv0`, and $\frac{dy}{dt}$, which is `deriv1`. This has to be in vector form due to the way that `odeint` works. `odeint` also needs the integration variable (in this case, t) to be specified as a variable of the function, even if it isn't used within the function.

We also need to define the initial conditions and the value of the constant ω that we're using. We also have to specify what values we want to evaluate the system at: in this case, since we're interested in the period $t = 0 \rightarrow 20$. We define this array of points using `linspace`, and use 2000 points. We use this many points so that the lines are sufficiently smooth and continuous when we go to plot the solution.

```
omega = 1.0
x0 = 1
y0 = 0
ics = [x0,y0]
t = np.linspace(0,20.0,2000) #integration range
```

Once again, due to the way that `odeint` works, we have to supply the initial conditions as a tuple containing all the values.

The next step is to call `odeint` and define the solutions.

```
sol_1 = integrate.odeint(deriv,ics,t)
```

This returns an array with a solution for both x and y. We extract both these solutions by slicing the array:

```
x_1 = sol_1[:,0]
y_1 = sol_1[:,1]
```

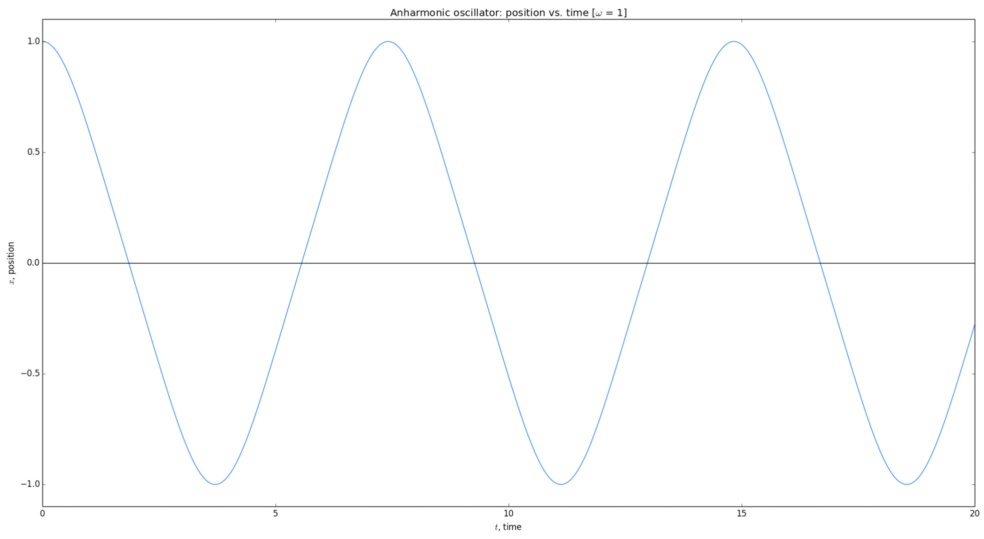
`y_1` is $y \equiv \frac{dx}{dt}$. We'll go on to use this in a phase-space plot later.

Plotting is a trivial affair. We'll plot the solution, x on the y-axis, and the time, t on the x-axis:

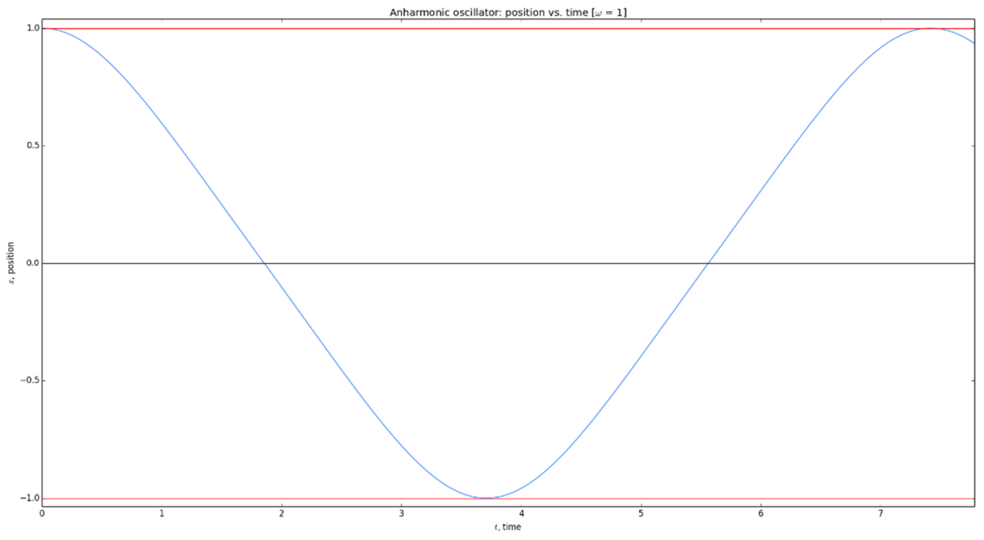
```
plt.figure(1)
plt.plot(t,x_1, color='#1C71FF')
plt.xlabel(r'$t$, time')
plt.xlim(0,20)
plt.ylabel(r'$x$, position')
plt.ylim(-1.1,1.1)
plt.axhline(y=0,color='k')
plt.title(r'Anharmonic oscillator: position vs. time [$x(0)$ = 1]')
plt.show()
```

I chose to specify the limits of the y-axis manually, since by default `pyplot` will use the range $[-1,1]$; this would stop it from being clear where the maximum value x is. Similarly, I chose to place a line at $x = 0$ for clarity.

All told, we get a graph which looks like this:



As we can see, the oscillator has a maximum amplitude, x_{max} of 1. To make this even clearer, we can add lines corresponding to $x = -1$ and $x = 1$. We can use the command `plt.axhline(y=1,color='k')` for this purpose.



The next step is to solve the equation again, but this time using $x(0) = 10$.

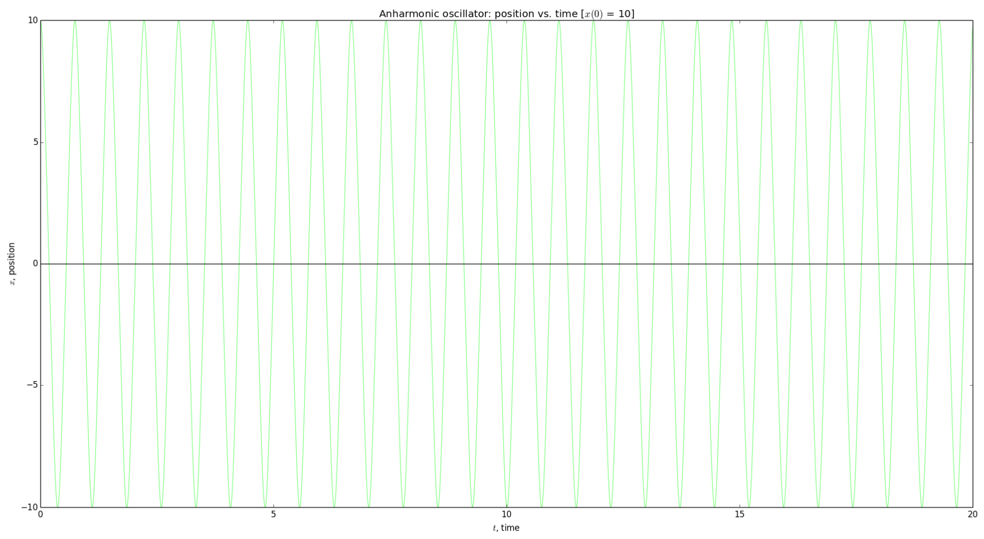
```
x0 = 10.0
ics = [x0,y0]
sol_10 = integrate.odeint(deriv,ics,t)
x_10 = sol_10[:,0]
y_10 = sol_10[:,1]
```

This is the same code that we previously used; we've just changed the value of $x(0)$ prior to running it. (We have to re-declare `ics` after changing `x0`; if we don't, the values that it contains won't change.)

To plot this function we can use the same plot code as before, just altering the names of the variables we're plotting (and of course, adjusting the title):

```
plt.figure(3)
plt.plot(t,x_10,color='#84FF84')
plt.xlabel(r'$t$, time')
plt.xlim(0,20)
plt.ylabel(r'$x$, position')
plt.title(r'Anharmonic oscillator: position vs. time [$x(0)$ = 10]')
plt.axhline(y=0,color='k')
plt.show()
```

This generates this graph:



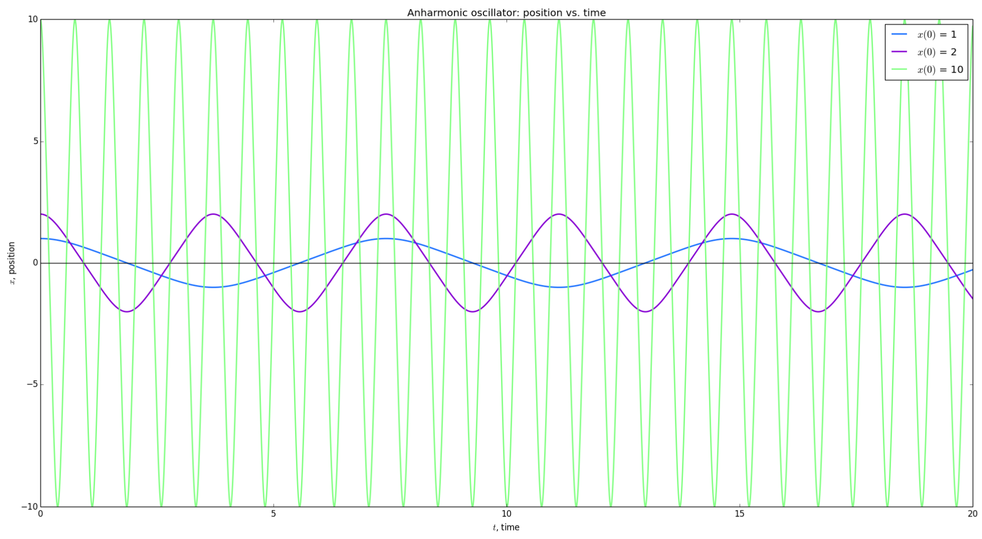
It appears that by increasing $x(0)$, we not only increase the amplitude, but we also increase the frequency of oscillation. To confirm our assertion we can calculate the solution to the differential equation with $x(0) = 2$. The process is similar to before:

```
x0 = 2.0
ics = [x0,y0]
sol_2 = integrate.odeint(deriv,ics,t)
x_2 = sol_2[:,0]
y_2 = sol_2[:,1]
```

Plotting is likewise similar. In this instance I have added a legend for the sake of clarity. I have also increased the thickness of the lines (using the `linewidth` attribute) in order to make them more distinct.

```
plt.figure(4)
plt.plot(t,x_1,label=r'$x(0)$ = 1', color='#1C71FF', linewidth=2.0)
plt.plot(t,x_2,label=r'$x(0)$ = 2', color='#8600D3', linewidth=2.0)
plt.plot(t,x_10,label=r'$x(0)$ = 10', color='#84FF84', linewidth=2.0)
plt.xlabel(r'$t$, time')
plt.xlim(0,20)
plt.ylabel(r'$x$, position')
plt.title('Anharmonic oscillator: position vs. time')
plt.axhline(y=0,color='k')
plt.legend(loc=0)
plt.show()
```

This yields the following graph:



It is starkly clear that increasing $x(0)$ increases the frequency of oscillations *in addition to* the amplitude. In fact, the solution where $x(0) = 2$ has a frequency that is **double** that of the solution where $x(0) = 1$. Similarly, the result for $x(0) = 10$ oscillates ten times faster than the $x(0) = 1$ solution. This behaviour is noteworthy because it differs to that of a regular harmonic oscillator.

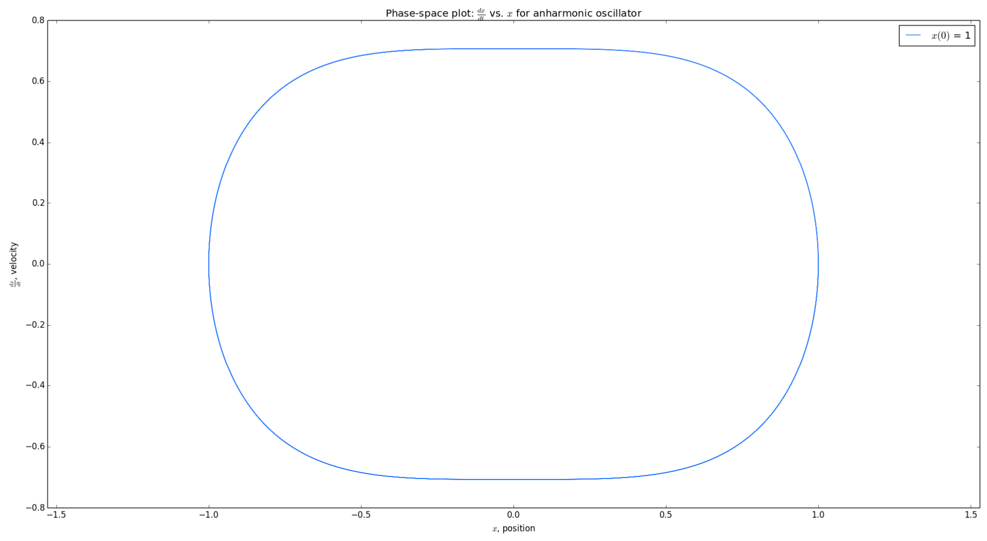
B) *"Modify your program so that instead of plotting x against t, it plots dx/dt against x, i.e., the velocity of the oscillator against its position."*

We have previously calculated $\frac{dx}{dt}$; it takes the form of y in our system of equations. All we have to do is plot it! We'll first plot y vs x just for $x(0) = 1$.

```
plt.figure(5)
plt.plot(x_1,y_1,label=r'$x(0)$ = 1', color='#1C71FF')
plt.xlabel(r'$x$, position')
plt.ylabel(r'$\frac{dx}{dt}$, position')
plt.axis('equal')
plt.title(r'Phase-space plot: $\frac{dx}{dt}$ vs. $x$ for anharmonic
oscillator')
plt.legend(loc=0)
plt.show()
```

It is important that we've specified `axis(‘equal’)` in this case: it ensures that the scales are kept equal is size. That is to say, if moving +1cm along the x axis corresponds to an increase of 2, moving +1cm along the y axis will also result correspond to an increase of 2.

The generated graph is:



The resulting shape is a sort-of hybrid between a circle and a square. Notably, at some places on the plot the gradient appears to be zero – or at least very close to it. That is to say, the shape does **not** continually vary.

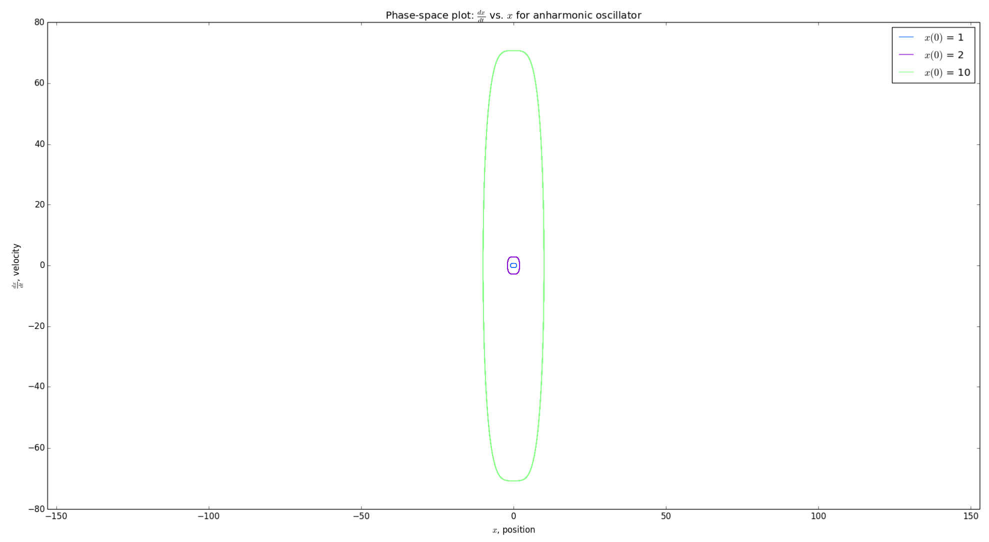
We'll go ahead and make a phase-space plot for the rest of the results. We can use the same code as before, but with the addition of the following lines:

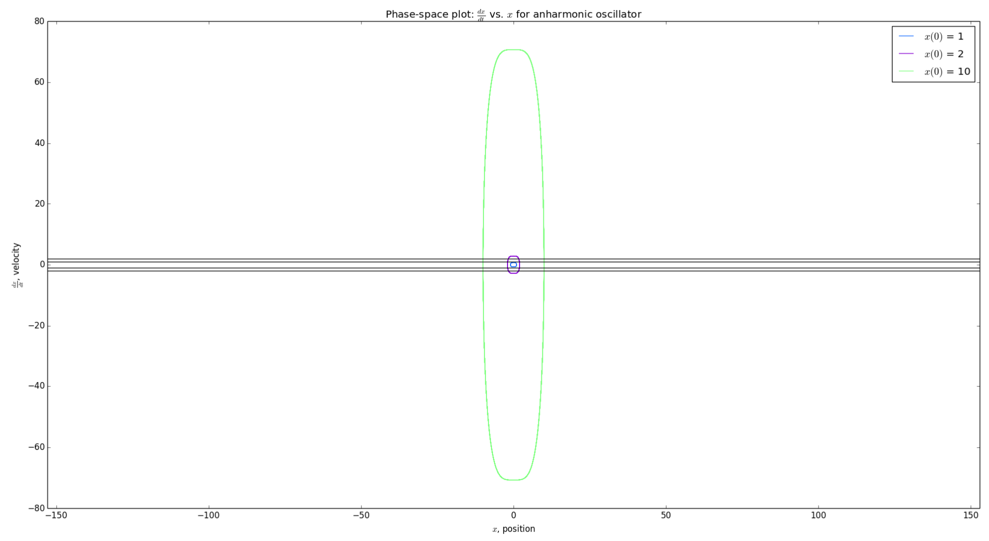
```
plt.plot(x_2,y_2,label=r'$x(0)$ = 2', color='#8600D3')
plt.plot(x_10,y_10,label=r'$x(0)$ = 10', color='#84FF84')
```

Adding lines at -1, 1, -2 and 2 can help us to visually identify where the maximum values for $\frac{dx}{dt}$ are. We can achieve this using the following code:

```
plt.axhline(y=2,color='k')
plt.axhline(y=-2,color='k')
plt.axhline(y=1,color='k')
plt.axhline(y=-1,color='k')
```

I plotted the graphs twice: one with the horizontal lines on, and again with the code for the horizontal lines commented out. This is how the graphs look:





We can see that all of the solutions have a region where the gradient is equal to zero, and all have the same shape – just that the solutions with higher ω values appear to be almost 'stretched' vertically.

We can note that the maximum values for $\frac{dx}{dt}$ do not correspond to the $x(0)$ value for any line.

C) *"How does the behaviour in (a) and (b) differ (quantitatively) from a normal harmonic oscillator?"*

A 'regular' harmonic oscillator is described by an equation of the form $x''(t) + \omega^2 x = 0$. Using the same principle as before, we can rewrite this as a system of two equations:

$$y = \frac{dx}{dt}$$

$$\frac{dy}{dx} = -\omega^2 x$$

So the system is then defined in Python using the following code:

```
def normalOsc(q,t):
      xi = q[0]
      yi = q[1]
      deriv0_n = yi
      deriv1_n = (-(omega**2)*xi)
      return [deriv0_n, deriv1_n]
```

We can assume that the initial conditions are staying the same (so there is no need to re-define them), but since the most recent value for $x(0)$ was 2, we'll have to go ahead and redefine it.

```
x0 = 1.0
ics = [x0,y0]
```

We use the same procedure as before to get the solutions to the system:

```
sol_n1 = integrate.odeint(normalOsc,ics,t)
x_n1 = sol_n1[:,0]
y_n1 = sol_n1[:,1]
```

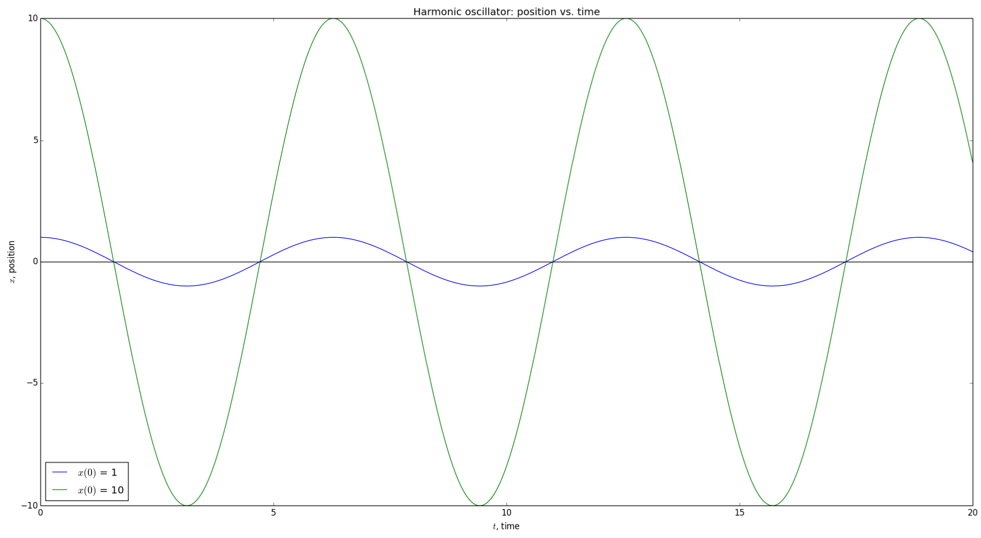
While we're at it, we may as well get a solution to the new harmonic oscillator when $\omega = 10$.

```
x0 = 10.0
ics = [x0,y0]
sol_n10 = integrate.odeint(normalOsc,ics,t)
x_n10 = sol_n10[:,0]
y_n10 = sol_n10[:,1]
```

We'll then go ahead and plot x vs t for both solutions. Again, using similar code before (but plotting different variables), we input:

```
plt.figure(7)
plt.plot(t,x_n1,label=r'$x(0)$ = 1', color='b')
plt.plot(t,x_n10,label=r'$x(0)$ = 10', color='g')
plt.xlabel(r'$t$, time')
plt.xlim(0,20)
plt.ylabel(r'$x$, position')
plt.title('Harmonic oscillator: position vs. time')
plt.axhline(y=0,color='k')
plt.legend(loc=0)
plt.show()
```

This yields the following graph:

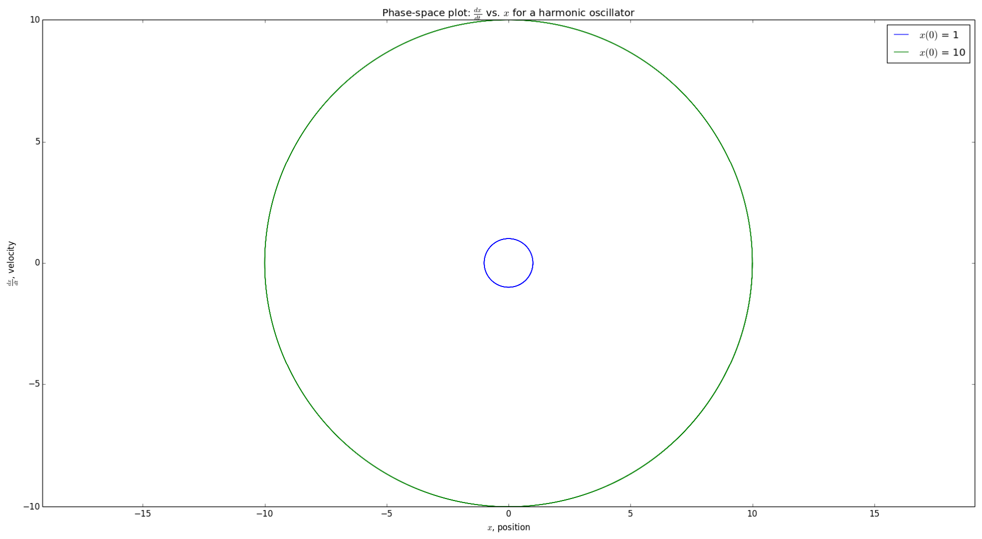


This behaviour is very different to the anharmonic oscillator: the oscillation frequency does not vary for different values of $x(0)$!

We'll go ahead and generate a phase plot:

```
plt.figure(8)
plt.plot(x_n1,y_n1,label=r'$x(0)$ = 1', color='b')
plt.plot(x_n10,y_n10,label=r'$x(0)$ = 10', color='g')
plt.xlabel(r'$x$, position')
plt.ylabel(r'$\frac{dx}{dt}$, position')
plt.axis('equal')
plt.title(r'Phase-space plot: $\frac{dx}{dt}$ vs. $x$ for a harmonic
oscillator')
plt.legend(loc=0)
```

This returns:

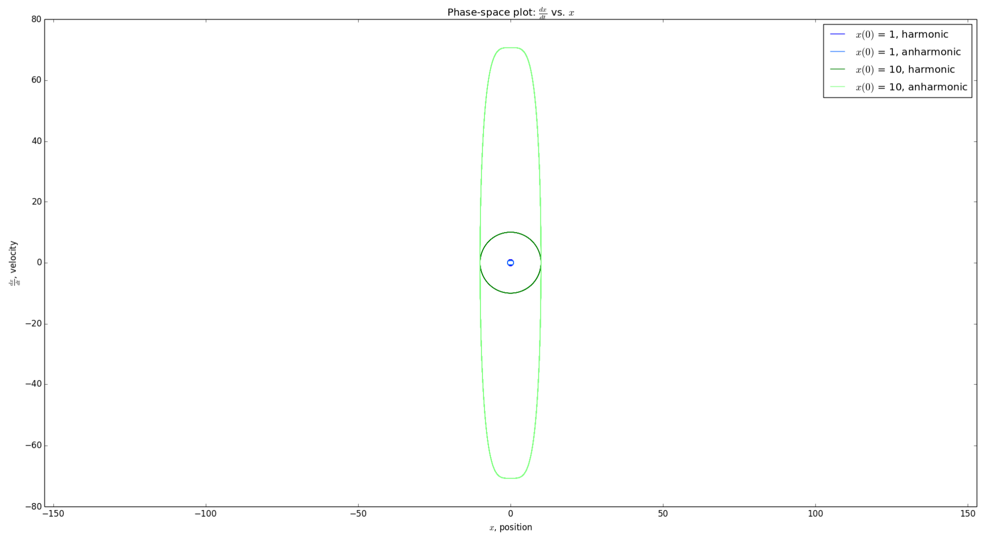


Phase plot also reveals different behaviour: there appears to be no point (for either solution) where the gradient is zero. Both plots are circular, with the maximum value for $\frac{dx}{dt}$ given by the value of $x(0)$ for each line.

We'll create another plot, this time including both the harmonic and anharmonic oscillators:

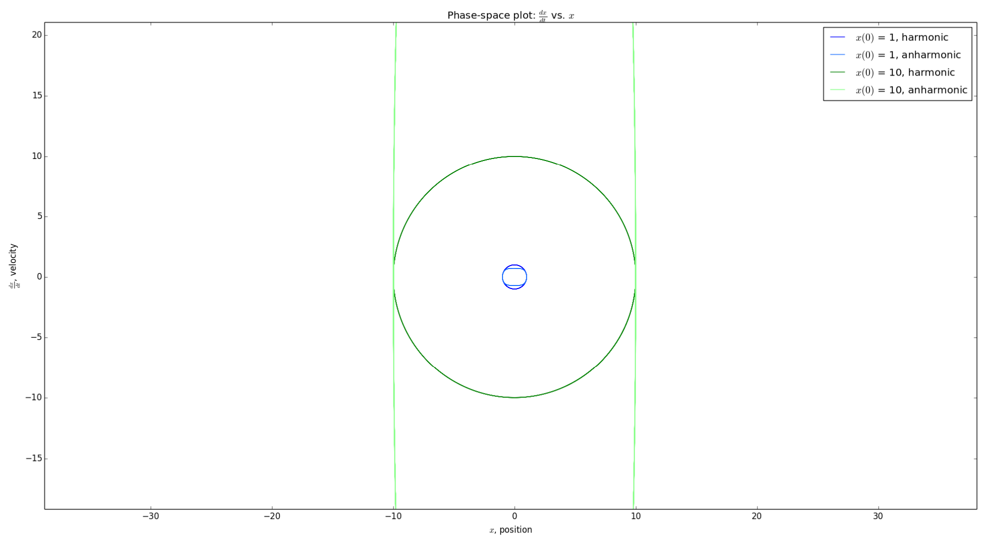
```
plt.figure(9)
plt.plot(x_n1,y_n1,label=r'$\omega$ = 1, harmonic', color='b')
plt.plot(x_1,y_1,label=r'$\omega$ = 1, anharmonic', color='#1C71FF')
plt.plot(x_n10,y_n10,label=r'$\omega$ = 10, harmonic', color='g')
plt.plot(x_10,y_10,label=r'$\omega$ = 10, anharmonic',
color='#84FF84')
plt.xlabel(r'$x$, position')
plt.ylabel(r'$\frac{dx}{dt}$, position')
plt.axis('equal')
plt.title(r'Phase-space plot: $\frac{dx}{dt}$ vs. $x$')
plt.legend(loc=0)
plt.show()
```

We get the following graph as a result:



It's now even more clear that the behaviour is very different for the harmonic and anharmonic oscillators. The maximum value for $\frac{dx}{dt}$ is higher for $x(0) = 10$ for the anharmonic oscillator than its harmonic counterpart, yet the maximum $\frac{dx}{dt}$ value is **lower** for the anharmonic $x(0) = 1$ case than its harmonic counterpart.

To show this difference, we can simply zoom into the graph like so:



This is the most quantifiable difference between the harmonic oscillator and the anharmonic oscillator. Otherwise, qualitatively, the shape is different between the two with the harmonic oscillator being circular and continually curved; the anharmonic oscillator is far more rectangular in shape.

Problem 2

A) *"Convert this equation to two first-order equations"*

Both `odeint` and `integrate` require that the equation (or equations) presented are first-order. The equation that we need to convert is:

$$\frac{d^2}{dt^2}\big(x(t)\big) = -x - \epsilon(x^2 - 1)\frac{d}{dt}\big(x(t)\big)$$

We can create a new function, $y(t)$, and define it such that $y(t) = \frac{d}{dt}x(t)$. Thus, we can state $\frac{d^2}{dt^2}x(t) = \frac{d}{dt}y(t) \equiv y'$, and similarly, $-x - \epsilon(x^2 - 1)\frac{d}{dt}\big(x(t)\big) = -x - \epsilon(x^2 - 1)\,y$.

And hence, we then have a system defined by two equations:

$$x' = y$$

$$y' = -x - \epsilon(x^2 - 1)\,y$$

We also need to rewrite the initial conditions. Fortunately, the process is relatively painless. We are given that $x(0) = 0.5$, which requires no further rewriting. We are additionally told that $x'(0) = 0$. Since we know $y(t) = x'(t)$, it is a straight-forward conclusion that $y(0) = 0$.

B) *"Solve these equations using the Runge-Kutta method of 4th order provided in class. Plot all solutions (one-by-one, in the above order), along with the solution obtained by using the in-build `scipy.integrate.odeint`"*

Our first step is to import the functions that we will use.

```
import scipy.integrate as sp_int
import numpy as np
import matplotlib.pyplot as plt
from run_kut4 import *
```

Then, we can then define ϵ, the initial conditions and a function containing the derivatives[1].

```
epsilon = 10.0
x0 = 0.5
y0 = 0.0
ics = [x0,y0]
```

[1] Both the Runge-Kutta (RK) method, `run_kut4` and the `odeint()` method require that the initial conditions (ics) are presented as a vector.

```
def f(x,y):
      f = np.zeros(2)
      f[0] = y[1] #f0 == dx, y[0] = x, y[1] = y
      f[1] = (-y[0])-((epsilon*((y[0]**2)-1))*y[1]) #f1 == dy
      return f
```

The `f = np.zeros(2)` line is necessary in order to create the array that we're going to define `f` against.

Next we define the integration parameters. These won't change with step size, so it makes sense to define them before performing any particular integration.

```
x_start = 0.0
x_stop = (10.0*np.pi)
```

Then we perform the actual integration using the `integrate()` function from `run_kut4`. For this first solution, we're using a step size of $h = 0.02$, which is supplied to the function at the end.

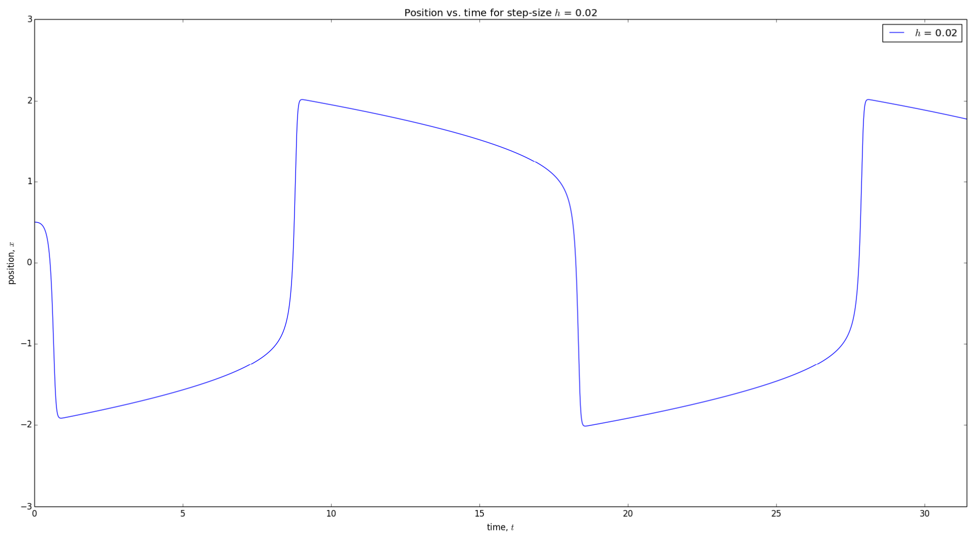
```
X_002,Y_002 = integrate(f,x_start,ics,x_stop,0.02)
```

Plotting is straight-forward. We use the following code:

```
plt.figure(1)
plt.plot(X_002,Y_002[:,0],label=r'$h$ = 0.02',color='b')
plt.xlabel(r'time, $t$')
plt.ylabel(r'position, $x$')
plt.title(r'Position vs. time for step-size $h$ = 0.02')
plt.legend(loc=0)
plt.xlim(0,(10*np.pi))
plt.show()
```

Note that it is necessary to 'unpack' the solution by taking the first column of the `Y_002` array.

I have also chosen to constrain the x-axis to $0 \rightarrow 10\pi$ in order to fill as much of the plot as possible; this is what the `plt.xlim(0,(10*np.pi))` achieves. The rendered graph is overleaf.



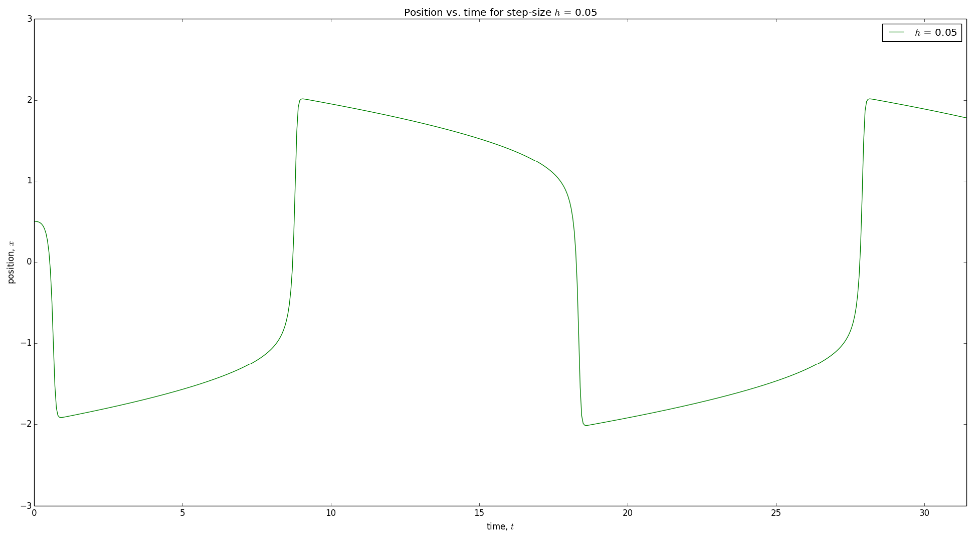
We can go ahead and perform a similar procedure for the $h = 0.05$ solution. The only difference to before is that the call to `integrate()` is slightly different.

```
X_005,Y_005 = integrate(f,x_start,ics,x_stop,0.05)
```

Plotting this is the same as before, except we change the names of the variables we're plotting (obviously) and adjust the labels. I have only included the different lines below.

```
plt.figure(2)
plt.plot(X_005,Y_005[:,0],label=r'$h$ = 0.05',color='g')
plt.title(r'Position vs. time for step-size $h$ = 0.05')
```

The resultant graph:

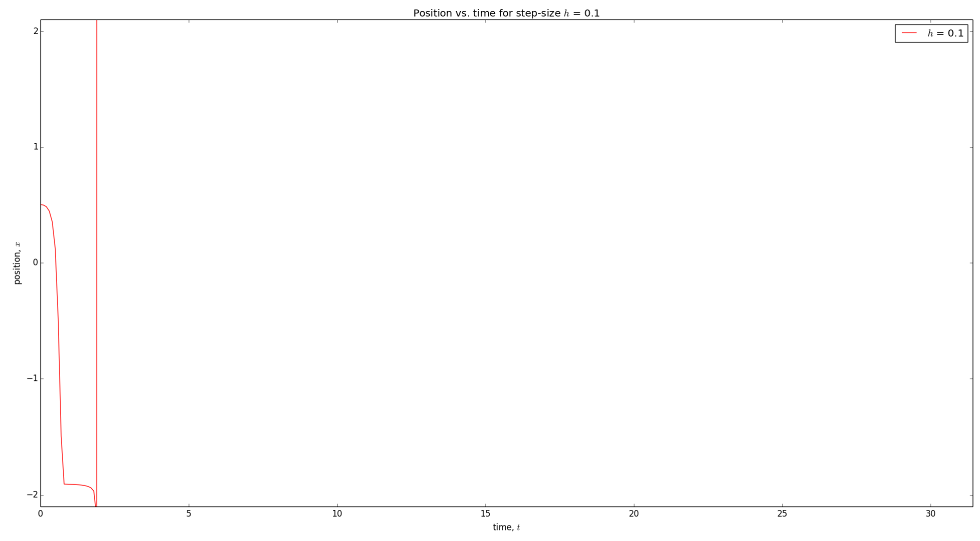


The solution's graphical shape is very similar to that of the $h = 0.02$ solution. So far, so good, then: the RK method with these step sizes converges.

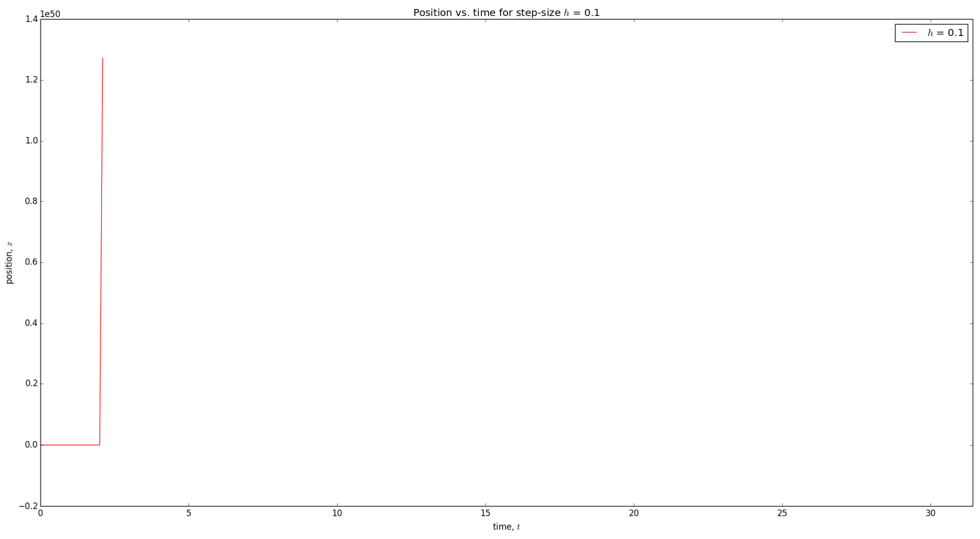
Finally, we can repeat the process for the step size of $h = 0.1$:

```
X_01,Y_01 = integrate(f,x_start,ics,x_stop,0.1)
plt.figure(3)
plt.plot(X_01,Y_01[:,0],label=r'$h$ = 0.1', color='r')
plt.xlabel(r'time, $t$')
plt.ylabel(r'position, $x$')
plt.title(r'Position vs. time for step-size $h$ = 0.1')
plt.legend(loc=0)
plt.ylim(-2.1,2.1)
plt.xlim(0,(10*np.pi))
plt.show()
```

This solution, however, blows up in our face. Its graph is below.



Commenting out the `ylim` command, we get a graph that looks like this:



Evidently, the RK method for a step size of $h = 0.1$ isn't stable and fails to converge on the function. The step size is simply too large[2]. Somewhat helpfully, the Python interpreter throws an error so that we know what's happened:

```
RuntimeWarning: overflow encountered in double_scalars
  f[1] = (-y[0])-((epsilon*((y[0]**2)-1))*y[1]) #f1 == dy
RuntimeWarning: invalid value encountered in double_scalars
  f[1] = (-y[0])-((epsilon*((y[0]**2)-1))*y[1]) #f1 == dy
run_kut4.py:20: RuntimeWarning: invalid value encountered in add
  return (K0 + 2.0*K1 + 2.0*K2 + K3)/6.0
```

I say *somewhat* helpfully because we at least know why Python stops calculating before reaching the stop value of 10π: the `f[1]` value overflows; it is just too large. We could have guessed that from the graph, however: after all, the scale for the position values is 10^{50}!

The function containing the derivatives that `odeint()` takes is subtly different to the function that `run_kut4()` takes, in that the arguments are reversed. Therefore we have to define a new function, which I have called `odeint_f`. I have also changed the names of the variables for additional clarity and distinction: this time, `t` is what we're integrating over, and `q` contains the variables in the derivatives.

```
def odeint_f(q,t):
      xi = q[0]
      yi = q[1]
      f0 = yi
      f1 = (-xi)-((epsilon*((xi**2)-1))*yi)
      return [f0,f1]
```

We can then define the integration range ($0 \rightarrow 10\pi$) and call `odeint()`.

```
t = np.linspace(0,(10*np.pi), 1000)
odeint_sol = sp_int.odeint(odeint_f,ics,t)
```

The next step is to unpack the result into values for x and y, just as we did for Q1.

```
x_ois = odeint_sol[:,0]
y_ois = odeint_sol[:,1]
```

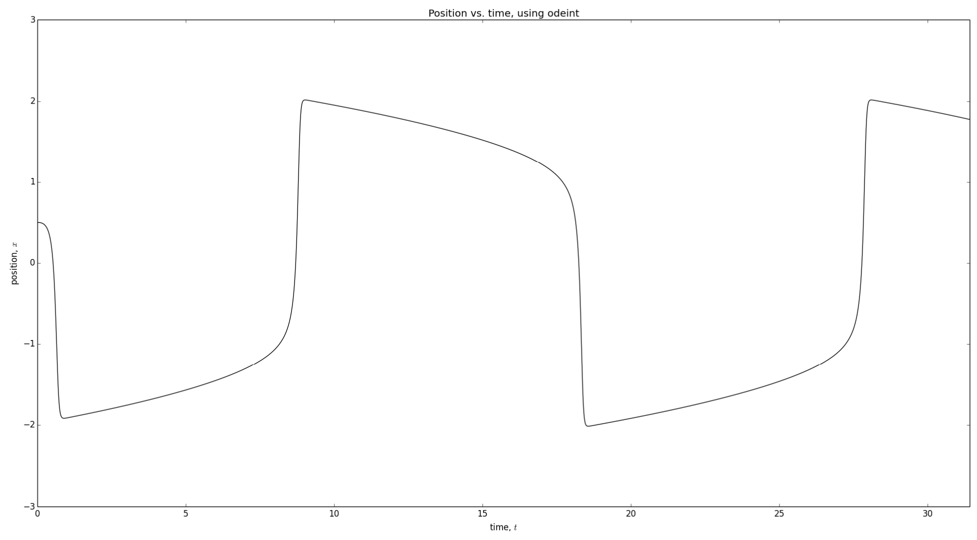
We're then ready to plot the solution. We set the x-axis' limit using `xlim` in order to maximise how much of the graph is visible.

```
plt.plot(t,x_ois_10,'k')
plt.xlabel(r'time, $t$')
```

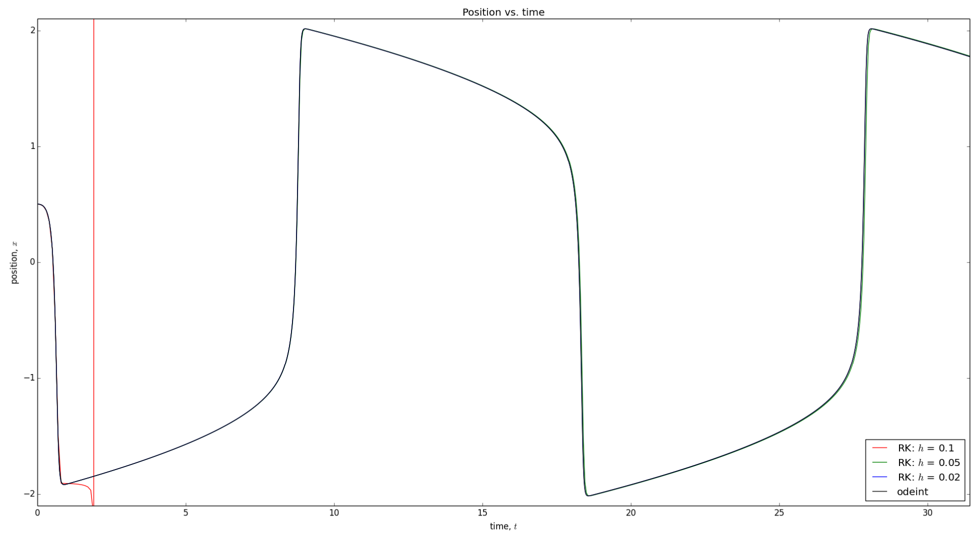
[2] The largest step size that can be used (to 2 decimal places) is $h = 0.08$. With $h = 0.09$ and above, the function overflows and fails to converge.

```
    plt.ylabel(r'position, $x$')
    plt.title(r'Position vs. time, using odeint')
    plt.xlim(0,(10*np.pi))
    plt.show()
```

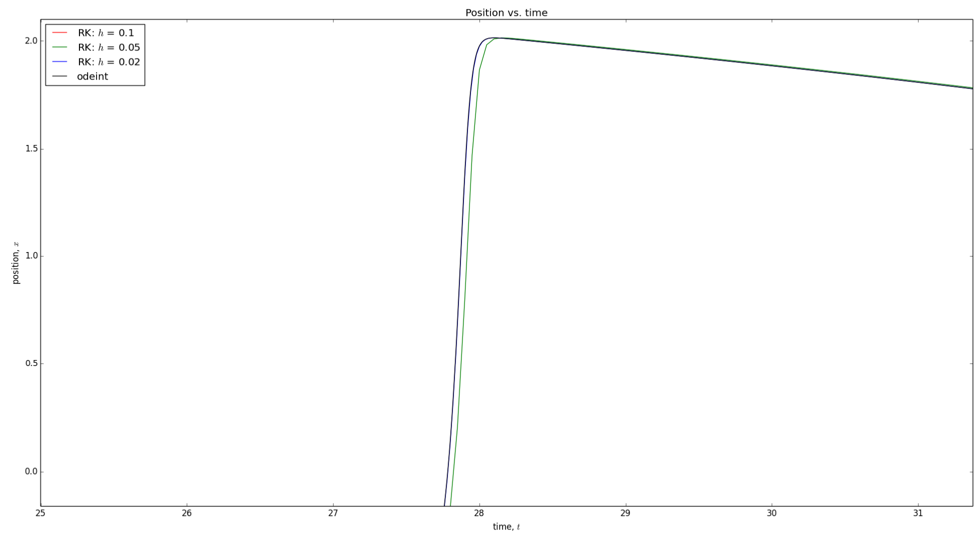
This returns this graph:



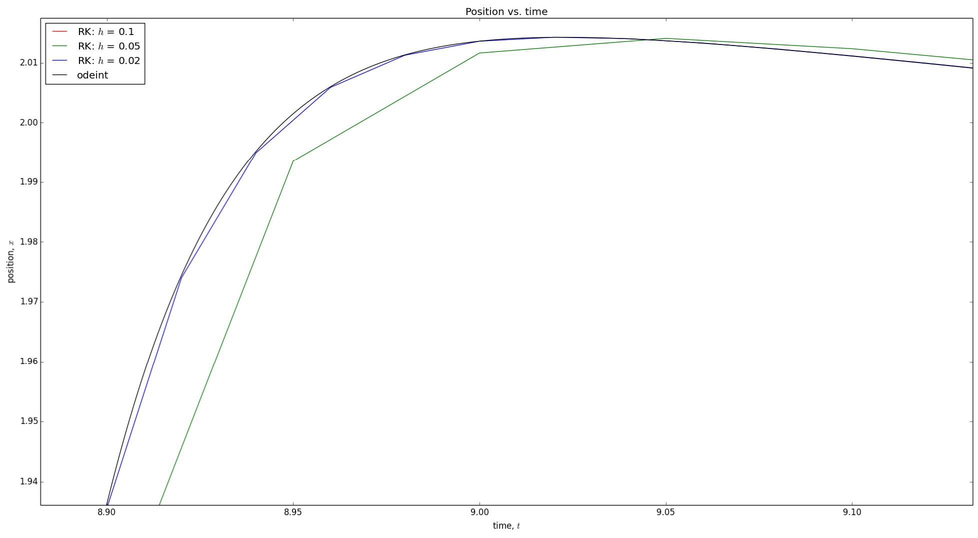
In order to help us establish how the solutions vary, we can plot all of the solutions on the same graph. All of the solutions (with the exception of the RK solution with $h = 0.1$, of course) look similar at first glance.



We see that the solutions are not entirely equivalent: zooming into the left hand-side region, we can see divergence between $h = 0.05$ and $h = 0.02$ lines. (The RK method with $h = 0.02$ is still reasonably close to the `odeint` solution.)



We can also zoom in to one of the turning points; this accentuates the differences between the two solutions.



The situation is as one may expect: the larger the step size, the less smooth the graph is. The RK solution with $h = 0.02$ is reasonably close to the `odeint` line, but the function is not as smooth. That said, the RK methods with $h = 0.02$ and $h = 0.05$ are reasonably good and are at least stable: they converge on the solution.

One of the most useful features of `odeint()` is that we are able to specify a tolerance – i.e. a maximum error. The default value is a *very* small number (1.49012×10^{-8} – such that the error is entirely negligible) and so `odeint`'s result becomes our 'true value.' This allows us to estimate the error associated with using the `run_kut4()` module.

We'll plot the absolute error, σ. We define this as $\sigma = |v_{\text{true}} - v_{\text{approx}}|$, where v_{true} is the 'true' value for a given time and v_{approx} is the approximated value (i.e. the RK solution).

In order to calculate σ we will have to calculate new arrays using `odeint()`. This is because the size of the arrays have to exactly match for us to be able to subtract the two. We also need the indexes to match for the same value of t. For convenience, we can write a function to calculate the array of points required.

```
def trange(x):
      z = np.linspace(0,10*np.pi,len(x[:,0]))
      return z
```

On each occasion that we need to get an array of points that match the distribution of points in another array, we can just call `trange()`.

The true values are then generated like so:

```
truev_002 = sp_int.odeint(odeint_f,ics,trange(Y_002))
truev_005 = sp_int.odeint(odeint_f,ics,trange(Y_005))
truev_01 = sp_int.odeint(odeint_f,ics,trange(Y_01))
```

Now that we have two arrays, subtracting them is a trivial affair:

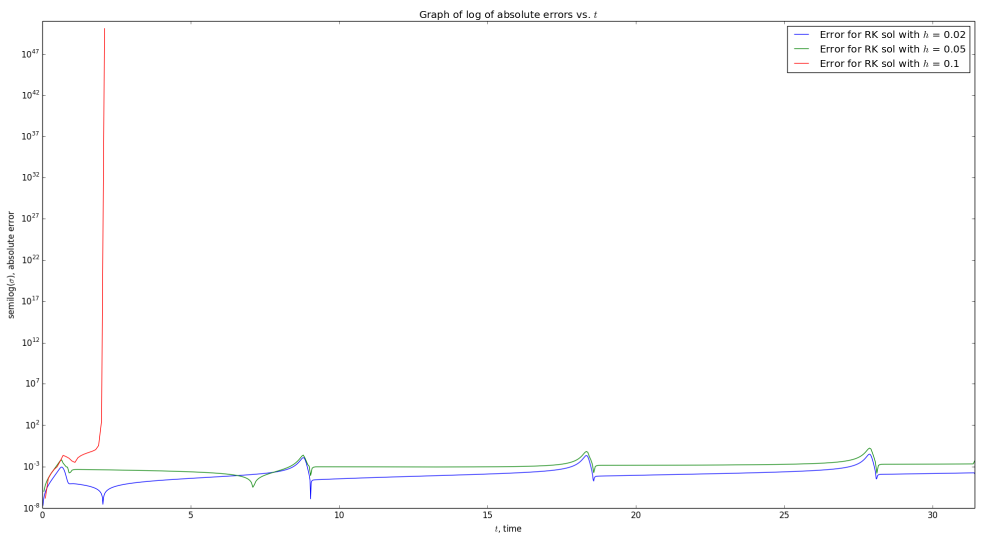
```
er_rk_002 = np.abs((truev_002[:,0]) - (Y_002[:,0]))
er_rk_005 = np.abs((truev_005[:,0]) - (Y_005[:,0]))
er_rk_01 = np.abs((truev_01[:,0]) - (Y_01[:,0]))
```

This gives us three arrays with the absolute errors related to each Runge-Kutta solution. We can then go ahead and plot these[3]:

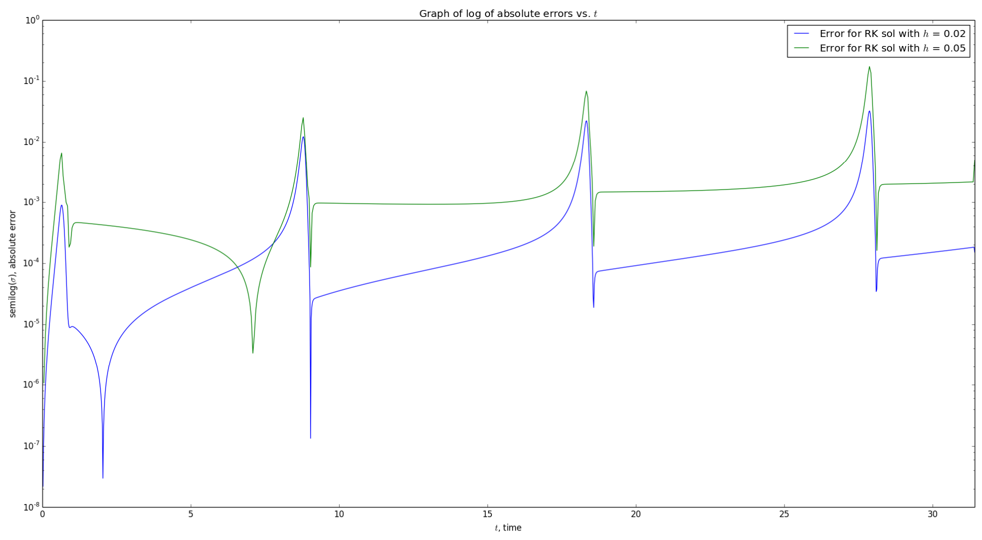
```
plt.figure(6)
plt.semilogy(trange(Y_002),er_rk_002, label=r'Error for RK sol with
$h$ = 0.02')
plt.semilogy(trange(Y_005),er_rk_005, label=r'Error for RK sol with
$h$ = 0.05')
plt.semilogy(trange(Y_01),er_rk_01, label=r'Error for RK sol with $h$
= 0.1')
plt.xlim(0,10*np.pi)
plt.title(r'Graph of log of absolute errors vs. $t$')
plt.xlabel(r'$t$, time')
plt.ylabel(r'semilog($\sigma$), absolute error')
plt.legend(loc=0)
plt.show()
```

[3] Note that the `trange()` function continues to make life easier for us!

By using `semilogy()`, we make the y-axis scale logarithmic. This allows us to better see the way the errors change[4], and if there is any pattern – for example, whether the errors are in any way cyclic. The graph is plotted below:



The RK solution with $h = 0.1$ behaves as you'd expect: the error suddenly becomes *enormous*. Since the inclusion of that data hampers the detail of the other plots, we can plot again – this time, removing the line of code that plots the $h = 0.1$ errors.



The resultant plot makes it easier to compare the $h = 0.02$ and $h = 0.05$. The error is smaller with $h = 0.02$ *at all times*. We can estimate the average error visually, and at somewhere in the $10^{-4} \rightarrow 10^{-5}$ range it's again smaller than the error for $h = 0.05$ (which seems to be mostly in the 10^{-3} region). The $h = 0.02$ solution's smallest error is smaller than $h = 0.05$'s smallest

[4] The reason behind doing this? The size of the peaks is so large compared to the rest of the curve that *all we can see* with a linear axes is just a huge peak – none of the more subtle variations. By making the scale logarithmic we can get deeper insight at a glance as to the overall trend of the data.

error; likewise, $h = 0.02$'s *largest* error is smaller than $h = 0.05$'s. All told, it is a simple conclusion that the smaller the step size, the more accurate the solution. This matches up nicely with our intuition.

Interestingly, the behaviour of the errors is somewhat cyclic. The graph will hit a local minimum, rise gradually, then rise swiftly, hit a local maximum, then plunge down almost asymptotically to another local minimum. This cycle is reasonably regular, and all minima and maxima (with the exception of the first minima) are at the same value of t for both RK solutions – i.e., the behaviour isn't specific to the step size, provided of course that h is sufficiently small that the RK method converges. We can conclude this cyclic behaviour must be inherent to the Runge-Kutta method when solving this ODE.

C) *"Make a phase-space (coordinate vs. velocity) plot of the `scipy.integrate.odeint` solution for ϵ = 1,4 and 10"*

We have already defined a function that contains all the relevant derivatives. To get solutions for different values of ϵ (namely for $\epsilon = 4$ and $\epsilon = 1$), all we have to do is state the value we want to use, then call `odeint()` and 'unpack' its results. We'll deal with $\epsilon = 4$ first.

```
epsilon = 4.0
odeint_sol_4 = sp_int.odeint(odeint_f,ics,t)
x_ois_4 = odeint_sol_4[:,0]
y_ois_4 = odeint_sol_4[:,1]
```

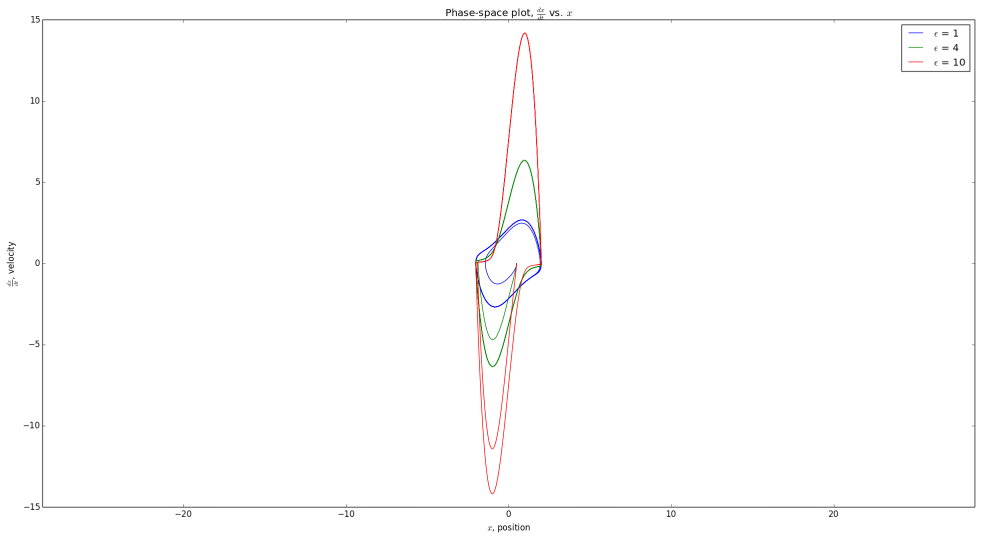
The process for $\epsilon = 1$ is virtually identical:

```
epsilon = 1.0
odeint_sol_1 = sp_int.odeint(odeint_f,ics,t)
x_ois_1 = odeint_sol_1[:,0]
y_ois_1 = odeint_sol_1[:,1]
```

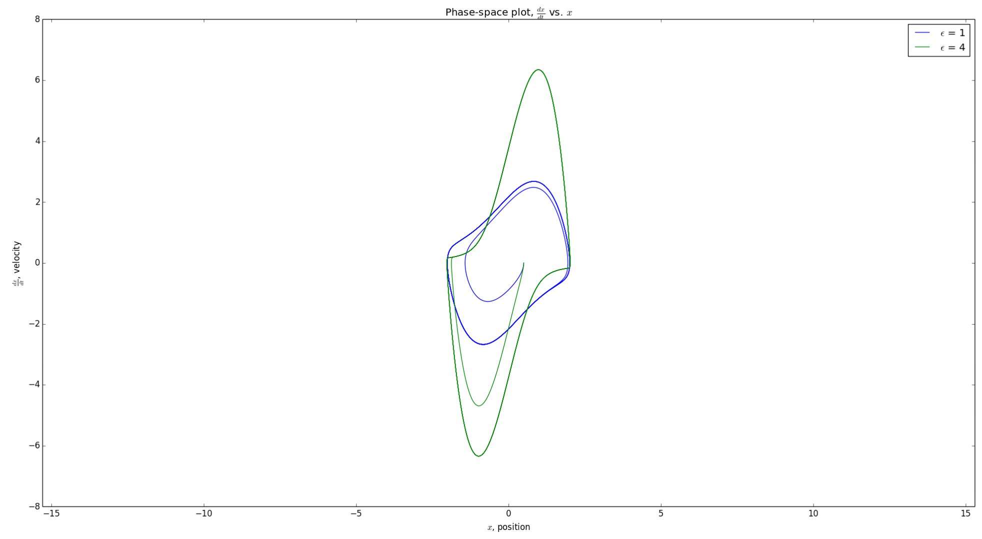
Now that we have results for all the values of ϵ, we can then plot them all on the same graph. (Obviously, to obtain a graph containing just one or two of the solutions, we can simply comment/uncomment a line or two.)

```
plt.plot(x_ois_1,y_ois_1, label=r'$\epsilon$ = 1', color='b')
plt.plot(x_ois_4,y_ois_4, label=r'$\epsilon$ = 4', color='g')
plt.plot(x_ois_10,y_ois_10, label=r'$\epsilon$ = 10', color='r')
plt.axis('equal')
plt.title(r'Phase-space plot, $\frac{dx}{dt}$ vs. $x$')
plt.xlabel(r'$x$, position')
plt.ylabel(r'$\frac{dx}{dt}$, velocity')
plt.legend(loc=0)
plt.show()
```

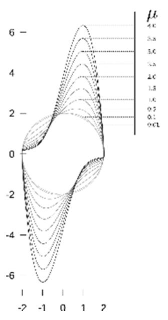
This plots the following graph:



A graph with just the $\epsilon = 1$ and $\epsilon = 4$ plots looks like so:



This phase-space behaviour matches what is expected of a Van der Pol oscillator: the oscillations **eventually** become stable and enter what is known as a *limit cycle*. (A copy of the expected behaviour is on the left[5].) This is why we see two lines for the same position: the line with the greater phase ($\frac{dx}{dt}$) is the limit cycle; the other line is the system displaying what can be ascribed to 'warm up' behaviour. We also note that the maximum phase increases substantially as ϵ increases.



[5] Source: http://en.wikipedia.org/wiki/File:VanderPol-lc.svg

Problem 3

A) *"Read in the data from `dow.txt` and plot them on a graph."*

We first place the external data dow.txt in the folder that we're executing from. We can then use NumPy's `loadtxt` function. Since the data in the file is just of a single column (i.e. doesn't have another column indicating the date in which a value is from), we can supply the function with a very simple set of arguments. This is because the options aren't necessary: the function defaults to interpreting whitespace as a delimiter between values. The only argument we supply is the `type`, i.e. that the data should be stored as floats: this is so that we don't lose any precision.

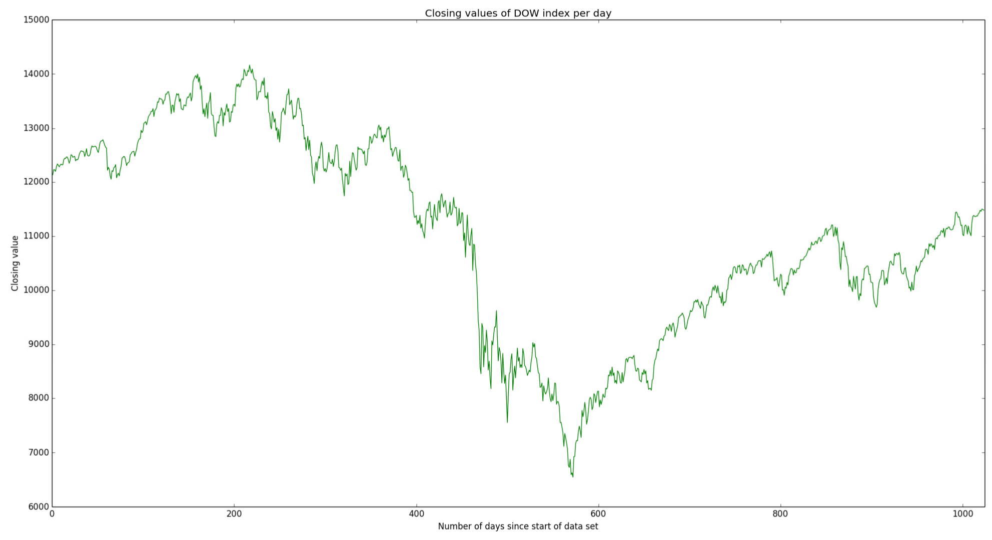
```
import numpy as np
dow = np.loadtxt("dow.txt", float)
```

In this instance we assign the data to the variable `dow`, which takes the form of an array.

We can then plot this with reasonable ease using PyPlot. We'll make the line green using `'g'`.

```
dow_l = np.arange(len(dow))
import matplotlib.pyplot as plt
plt.title('Closing values of DOW index per day')
plt.xlabel('Number of days since start of data set')
plt.ylabel('Closing value')
plt.xlim(0,(len(dow)))
plt.plot(dow_l, dow, 'g', label='Unprocessed data')
```

If we wanted to see this graph, we would call `plt.show()`. It would appear like this:



But, at this stage we won't call `plt.show()`, since later questions as us to plot "on the same graph as the original data." Since there isn't a way to modify a graph after it has been rendered, we'll leave calling `show` until later.

<u>B</u>) *"Calculate the coefficients of the discrete Fourier transform of the data using the function `rfft` from `numpy.fft`, which produces an array of $\frac{1}{2}N+1$ complex numbers."*

To calculate the Fourier transform coefficients, we can use NumPy's built-in function for a fast Fourier transform in one dimension: this is `fft.rfft()`. We use `rfft()` rather than `fft()` since we know that we won't have to deal with any complex numbers. (`fft()` is for situations where some input might be complex; `rfft()` works with real numbers only. Because of this, `rfft` is faster: it only calculates the non-negative frequency terms.)

```
dowft = np.fft.rfft(dow)
```

Having assigned the array of coefficients to a variable, we can then `print dowft` to have a look at the coefficients being output. Below is a sample:

```
>>> print dowft
[  1.12839039e+07 +0.00000000e+00j   6.84127740e+05 -9.55087519e+05j
  -4.19226205e+05 +9.29089562e+04j   7.41901498e+04 -2.73812250e+05j
  -3.51856811e+04 +7.23658613e+04j   1.11482643e+05 -3.14779807e+04j
   2.16752824e+04 -2.45751549e+04j   1.11224416e+04 +2.47975704e+04j
   3.11884665e+04 +2.27868210e+04j   7.38523102e+03 -3.21233427e+04j
```

As we can see, the output is complex, representing sines and cosines. (Python uses *j* as the complex number, rather than the more commonly encountered *i*.)

The array is has 513 entries[6]. This is inline with expectations since we would expect `rfft()` to return ½*N* + 1 coefficients and in this case *N* = 1024. [[7]] The reason why we have this many coefficients is because `rfft()` exploits the symmetry in the coefficients and only computes non-negative terms. To illustrate this symmetry,

```
>>> np.fft.fft([0, 1, 0, 0])
array([ 1.+0.j,  0.-1.j, -1.+0.j,  0.+1.j])
```

The fourth element of the output is the *complex conjugate* of the second element. `rfft()` doesn't compute that final term.

[6]>>> len(dowft)
513

[7]>>> len(dow)
1024

Note that the expression for the number of coefficients calculated utilises integer division. Accordingly, if `dow`'s length was an odd number, the number of Fourier coefficients returned would be ½ (*N+1*).

C) *"Set all but the first 10% of the elements of this array to zero (i.e., set the last 90% to zero but keep the values of the first 10%)."*

First, we need to decide where our 'cut-off' point is: i.e. the element after which we'll set all remaining elements to zero.

```
dowft_l = len(dowft)
dowft_10pc_l = (((dowft_l)//10))
```

Although we're setting the coefficients **after** a certain cut-off point to zero, because Python's indexes start at zero there is no need for us to add 1 to the element number.

We can then go ahead and create a copy of the array of Fourier coefficients. (We maintain both sets of data – the truncated and the full set – so that we are able to compare them later.) We set the elements after the relevant point to zero.

```
dowft_f10 = np.copy(dowft)
dowft_f10[(dowft_10pc_l):(dowft_l)] = 0.0
```

D) *"Calculate the inverse Fourier transform of the resulting array, zeros and all, using the function `irfft`, and plot it on the same graph as the original data."*

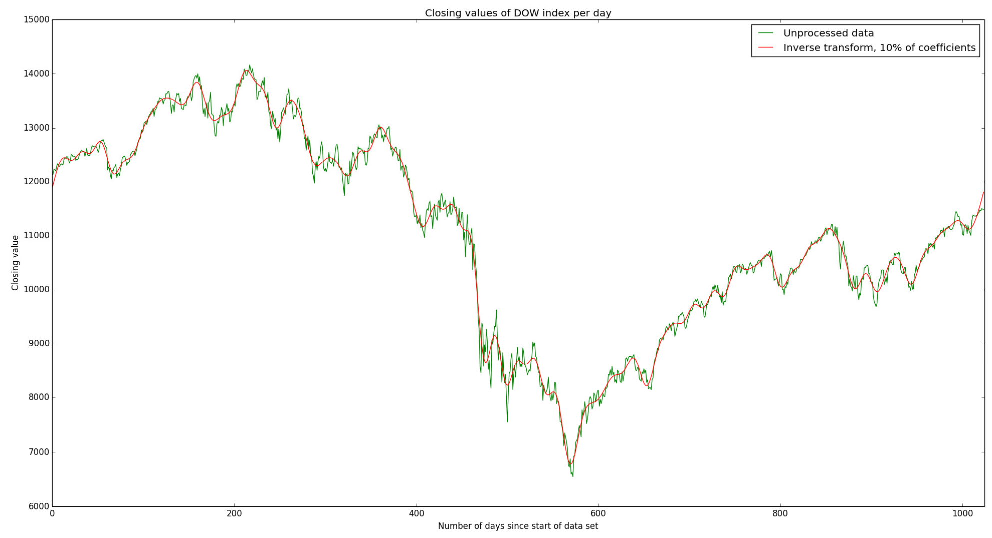
NumPy has a built-in function for the inverse Fourier transform. Similar to before, we call `irfft()` (rather than `rfft()`) since the original data contained no complex numbers.

```
inv_dowft_f10 = np.fft.irfft(dowft_f10)
```

Plotting is then trivial. We specify this new line to be red using the `'r'` argument. For clarity, we'll also add a legend to the data.

```
plt.plot(dow_l,inv_dowft_f10, 'r', label='Inverse transform, 10% of
coefficients')
plt.legend(loc=0)
plt.show()
```

Since we didn't previously call `plt.show()`, the new set of data will be plotted on the same graph as before. The end result is this:



The plot shows that new data set – missing the last 90% of the Fourier coefficients – is of similar shape to the original data, but is smoother. Since a Fourier series is a sum of sine and cosine waves, this result makes perfect sense: since the approximation gets closer with each term added, by setting the last 90% to zero we have essentially made the approximation 'worse.' As this new approximation is worse, it does not include the more subtle peaks and troughs in the data. The resulting data is hence more smooth.

E) *"Modify your program so that it sets all but the first 2% of the coefficients to zero and run it again.*

We then repeat the same procedure as before, first by finding out what our cut-off point is, then setting all elements past that point to zero. Again, we use a copy the original array of Fourier transformed data, `dowft`, to preserve the original data.

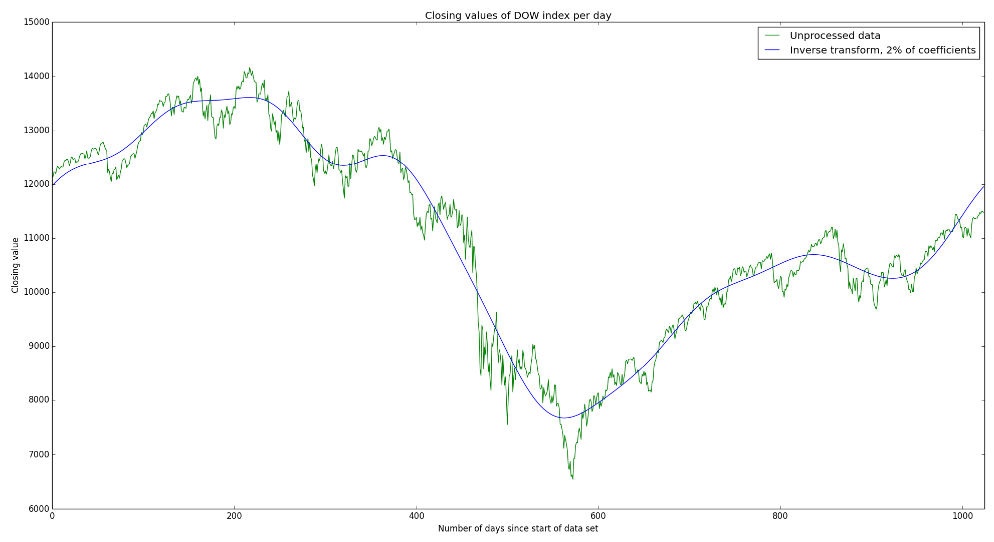
```
dowft_2pc_l = (((dowft_l)//50))

dowft_f2 = np.copy(dowft)
dowft_f2[(dowft_2pc_l):(dowft_l)] = 0.0
```

As before, we take the inverse transform of this new data set:

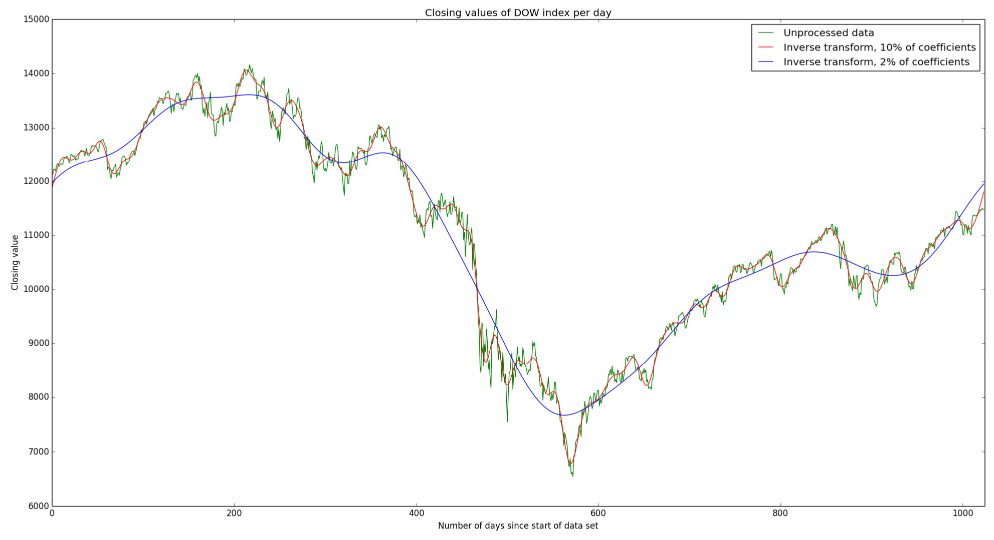
```
inv_dowft_f2 = np.fft.irfft(dowft_f2)
```

We can then go ahead and plot this new data using the same procedure as before. The graph looks like this:



The graph looks as one would expect: by setting even more of the coefficients to zero, the resulting Fourier series approximation is even smoother, omitting yet more of the details of the graph.

To visualise how much of a difference this is, we can plot all three sets of data for easy comparison. (This is why we made copies were made of the data sets, instead of overwriting them!)



As we can see, the series which omits 90% of the coefficients is a far better approximation – as we would expect. This series stays faithful to the overall shape of the data – just removing some 'noise' around the edges. Only the very sharpest and most temporary changes are omitted. By comparison, the series which omits 98% of the coefficients is far, far worse. Many significant events are smoothed over, only the **very** basic trend of the data remains.