UNIVERSITY OF SUSSEX
Scientific Computing
Tutor: Dr. Ilian Iliev, Office: Pev III 4C5

Assignment 3
Deadline: 12pm on Thursday, December 11th, 2014.
Penalties will be imposed for submissions beyond this date.
Final submission date: Friday, December 12th, 2014
No submissions will be accepted beyond this date.

1. (a) Write a code to solve for the motion of the anharmonic oscillator described by the equation

$$\frac{d^2x}{dt^2} = -\omega^2 x^3 \tag{1}$$

Take $\omega = 1$ and initial conditions $x = 1$ and $dx/dt = 0$ and make a plot of the motion of the oscillator from $t = 0$ to 20. Increase the amplitude to 10. You should observe that the oscillator oscillates faster at higher amplitudes. (You can try lower amplitudes too if you like, which should be slower.)

(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. Such a plot is called a phase space plot. Make sure you say **pylab.axis('equal')** to ensure the plot axes have the same length for the phase space plots.

(c) How does the behaviour in (a) and (b) differ (quantitatively) from a normal harmonic oscillator? (Just run your code again after modifying the equation appropriately.) [30]

Solution: (a) As usual, we first import the required modules and define the equation to be solved, as follows:

```
from numpy import array,zeros
from printSoln import *
from run_kut4 import *
import pylab as pl

omega=1.0

def f(x,y):
    f=zeros(2)
    f[0]=y[1]
    f[1]=-omega**2*y[0]**3
    return f
```

The solution with the Runge-Kutta method is straightforward, as shown in class and then we plot the obtained solution:

```
x = 0.0 # Start of integration
xStop = 20.0 # End of integration
y = array([1.0, 0.0]) # Initial values of {y}
h = 0.001 # Step size
```

```
freq = 1 # Printout frequency

X,Y = integrate(f,x,y,xStop,h)
printSoln(X,Y,freq)

pl.plot(X,Y[:,0])
pl.show()
```

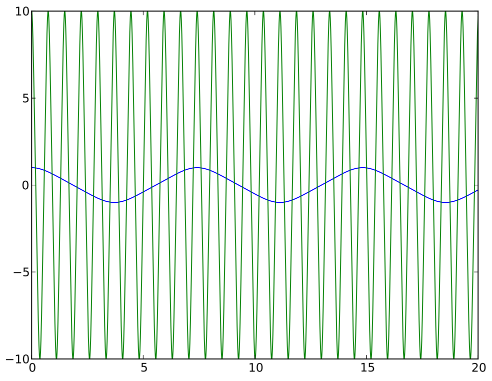
Note that we chose quite small value of h in order to obtain smooth plots later (this could be done by testing different values until plots look good). Then, we repeat all steps above but with higher amplitude, by setting

```
y1 = array([10.0, 0.0]) # Initial values of {y}

X1,Y1 = integrate(f,x,y1,xStop,h)
printSoln(X1,Y1,freq)

pl.plot(X1,Y1[:,0])
pl.show()
```

This gives the plot



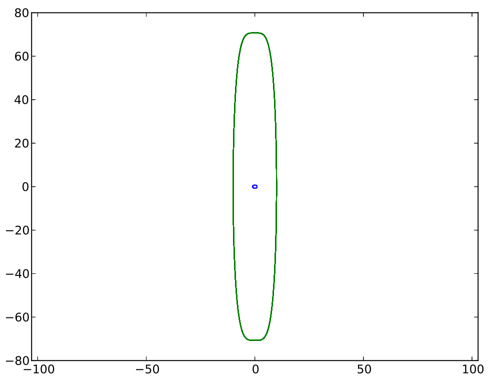
We see that indeed the higher-amplitude case exhibits faster oscillations (ω here is clearly NOT the frequency, since it does not change between the two cases).

(b) The phase space plot simply means that we plot the coordinate vs.the velocity, i.e. the first component of Y vs. the second:

```
pl.clf()

pl.axis('equal')
pl.plot(Y[:,0],Y[:,1])
pl.plot(Y1[:,0],Y1[:,1])

pl.show()
```

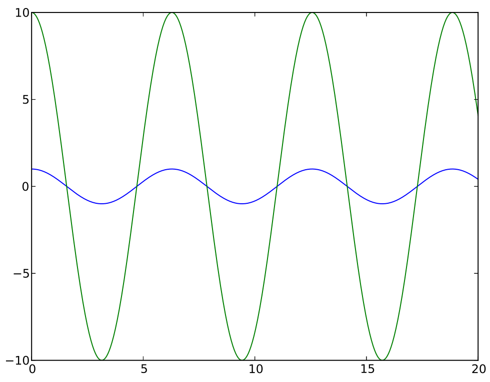
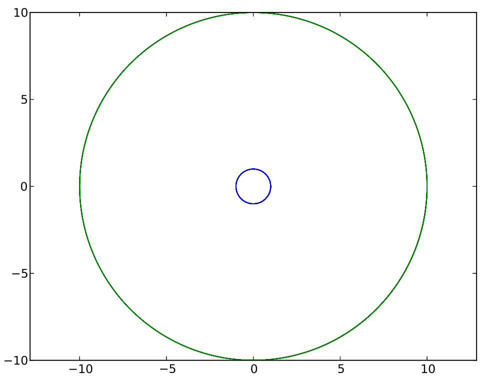


This gives:

(c) For solving the harmonic oscillator, we simply change the equation definition as follows:

```
def g(x,y):
    g=zeros(2)
    g[0]=y[1]
    g[1]=-omega**2*y[0]
    return g
```

and then we repeat all steps above. The resulting plots are shown below.

Clearly, in this case the frequency is independent of the amplitude and the phase space trajectory is always a perfect circle, unlike the anharmonic osillator case above.

2. Consider the differential equation (called van der Pol oscillator):

$$x''(t) = -x - \epsilon(x^2 - 1)x'(t), x(0) = 0.5, x'(0) = 0,$$

where ϵ is a positive constant. As the value of ϵ increases, this equation becomes increasingly stiff.

   (a) Convert this equation to two first-order equations.

   (b) Solve these equations using the Runge-Kutta method of 4th order provided in class with $\epsilon = 10$, and $[a, b] = [0, 10\pi]$ and different choices of step-size: $h = 0.02, 0.05$ and 0.1. Plot all solutions (one-by-one, in the above order), along with the solution obtained by using the in-build `scipy.integrate.odeint` using linear axes, and (in a separate figure) plot the absolute errors of all solutions using semi-log axes. What do you observe? **Note that the in-build method has a different interface to the function defining the equation(s) compared to the supplied functions (the arguments are reversed). Result is also formatted differently, check the help.**.

(c) Make a phase-space (coordinate vs. velocity) plot of the `scipy.integrate.odeint` solution for $\epsilon = 1, 4$ and 10. Make sure you use a small enough value of the time interval h to get a smooth, accurate phase space plot and again use `pl.axis('equal')`.

[40]

Solution:

(a) We convert this equation to two first-order equations by setting $z = dx/dt$:

$$\begin{aligned} x' &= z \\ z' &= -x - \epsilon(x^2 - 1)z \end{aligned}$$

(b) We first implement the system of equations as a (vector) function:

```
import numpy as np
import scipy
from scipy import integrate
from printSoln import *
from euler0 import *
import pylab as pl

epsi=10.

def f(t,y):
    f=np.zeros(2)
    f[0]=y[1]
    f[1]=-y[0]-epsi*(y[0]**2-1)*y[1]
    return f

def g(y,t):
    f=np.zeros(2)
    f[0]=y[1]
    f[1]=-y[0]-epsi*(y[0]**2-1)*y[1]
    return f
```

Note that (as discussed in class) we need two functions with different order of arguments, one to use with the supplied RK4 solver, one for the Python internal solver.

We then solve this system using the RK4 method and the in-build Python ODE integrator over the requested interval and using the given initial conditions e.g. as follows:

```
x = 0.0 # Start of integration
xStop = 10.*np.pi # End of integration
y = np.array([0.5,0]) # Initial value of {y}
freq = 1 # Printout frequency

h = 0.1 # Step size
X1,Y1 = integrate(f,x,y,xStop,h)
```

```
printSoln(X1,Y1,freq)

h = 0.05 # Step size
X2,Y2 = integrate(f,x,y,xStop,h)
printSoln(X2,Y2,freq)

h = 0.02 # Step size
X3,Y3 = integrate(f,x,y,xStop,h)
printSoln(X3,Y3,freq)

x1=np.linspace(0.0,xStop,np.size(X1))
z1=scipy.integrate.odeint(g,y,x1)
x2=np.linspace(0.0,xStop,np.size(X2))
z2=scipy.integrate.odeint(g,y,x2)
x3=np.linspace(0.0,xStop,np.size(X3))
z3=scipy.integrate.odeint(g,y,x3)

Yexact1=z1[:,0]
Yexact1=Yexact1.reshape(np.size(x1))

Yexact2=z2[:,0]
Yexact2=Yexact2.reshape(np.size(x2))

Yexact3=z3[:,0]
Yexact3=Yexact3.reshape(np.size(x3))

error1=abs((Y1[:,0]-Yexact1))
error2=abs((Y2[:,0]-Yexact2))
error3=abs((Y3[:,0]-Yexact3))
```

Note that `odeint` results need to be 'reshaped' in order to make them in the same format as the RK4 result (otherwise one is a row vector and the other is a column one, so they cannot be subtracted to find the errors).
The results are the plotted using, e.g:

```
pl.plot(X3,Yexact3)

pl.show()

raw_input("Press return to exit")

pl.plot(X1,Y1[:,0])
pl.plot(X2,Y2[:,0])
pl.plot(X3,Y3[:,0])

pl.show()
```

```
raw_input("Press return to exit")

pl.clf()

pl.semilogy(X1,error1)
pl.semilogy(X2,error2)
pl.semilogy(X3,error3)

pl.show()

raw_input("Press return to exit")
```

which yields Figure 2.

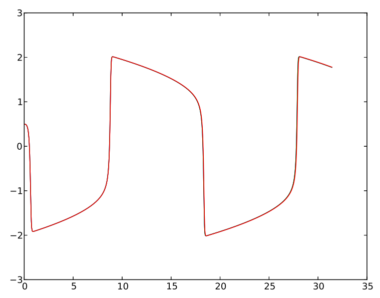


Figure 1: Solutions for Problem 2(b).

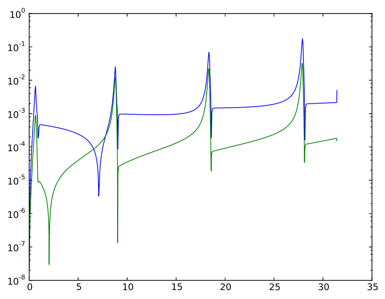


Figure 2: Errors for Problem 2(b).

The smaller stepsize solutions are relatively OK, but not very precise, with errors larger than 0.1-1% (blue/green). At certain points (where the function grows very fast) the errors are quite high, at few to tens of percent. The solution with the largest stepsize (not plotted here) fails completely, as this h value is outside the region of stability of the method for this equation. This problem is quite stiff, so normal methods like RK4 should generally not be used, to be replaced by fully implicit methods. The implicit methods used for stiff equations are generally unconditionally stable (but not necessarily very precise, that depends on the order method used).

(c) We first solve the system for $\epsilon = 1, 4$ and 10 by minor modification of the code in (b):

```
import numpy as np
import scipy
from scipy import integrate
from printSoln import *
from run_kut4 import *
import pylab as pl

xStop = 10.*np.pi # End of integration
y = np.array([0.5,0]) # Initial value of {y}

def g(y,t):
    f=np.zeros(2)
    f[0]=y[1]
    f[1]=-y[0]-epsi*(y[0]**2-1)*y[1]
    return f

epsi=1.
x1=np.linspace(0.0,xStop,10000)
z1=scipy.integrate.odeint(g,y,x1)

epsi=4.
z2=scipy.integrate.odeint(g,y,x1)

epsi=10.
z3=scipy.integrate.odeint(g,y,x1)
```

Note the large number of steps, which is required for obtaining a smooth plot.
Then the phase space plot is done by simply plotting the coordinate vs. velocity, i.e. first vs. second column of each solution z: solution are plotted as follows:

```
pl.axis('equal')

pl.plot(z1[:,0],z1[:,1])

pl.show()

raw_input("Press return to exit")

pl.plot(z2[:,0],z2[:,1])

pl.show()

raw_input("Press return to exit")

pl.plot(z3[:,0],z3[:,1])

pl.show()

raw_input("Press return to exit")
```

which yields Figure 3.

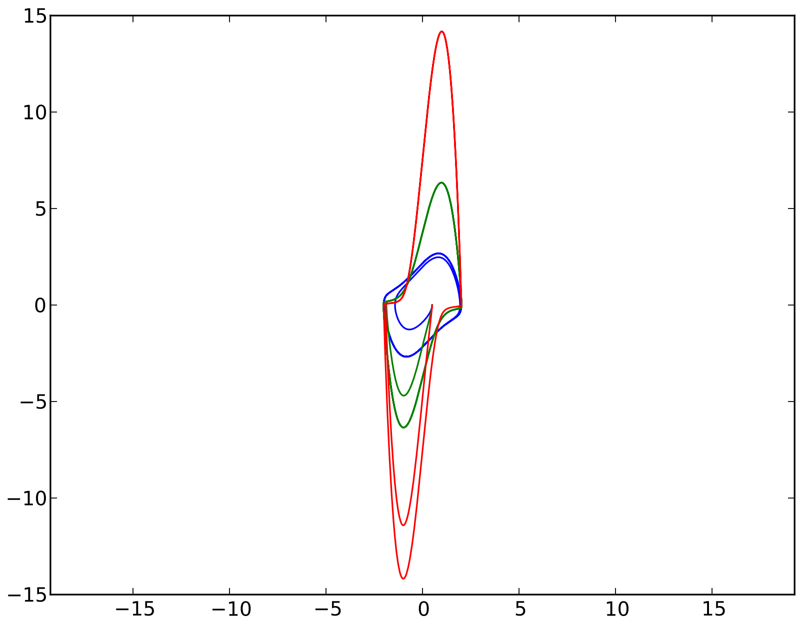


Figure 3: Results for Problem 2(c).

Clearly, the phase space trajectories are very different from the harmonic oscillator we did last time, where all were perfect circles. The larger the nonlinearity parameter the more different (elongated) they are. Initially they are no even closed.

3. **Fourier filtering and smoothing**

On Study Direct you'll find a file called `dow.txt`. It contains the daily closing value for each business day from late 2006 until the end of 2010 of the Dow Jones Industrial Average, which is a measure of average prices of the largest companies of the US stock market.

Write a program to do the following:

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

(b) 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.

(c) Now 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%).

(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. You may need to vary the colors of the two curves to make sure they both show up on the graph. Comment on what you see. What is happening when you set the Fourier coefficients to zero?

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

Solution:

We start by importing the required modules, setting the cutoffs as suggested and then loading and plotting the given data:

```
from numpy import loadtxt
from numpy.fft import rfft,irfft
from pylab import *

cutoff1=0.1
cutoff2=0.02

#Load and plot the data
y=loadtxt("dow.txt",float)
plot(y,"c-")
```

Then, we calculate the Fourier transform of the data using the internal `rfft` module and apply the first smoothing, as suggested, the calculate the inverse Fourier transform and plot the resulting smoothed curve on the same plot as above (using different colour):

```
#Calculate the Fourier transform
c=rfft(y)

#First smoothing
n=int(cutoff1*len(c))
c[n:]=0.0
z=irfft(c)
plot(z,"k-")
```

Similarly, we can do the second level of smoothing using the same method as above, by just using the higher cutoff for the high-frequency Fourier components, with results again plotted togehter with the other data:

```
#Second smoothing
n=int(cutoff2*len(c))
c[n:]=0.0
z=irfft(c)
plot(z,"g-")

show()
```

The final resulting plot is shown in the Figure below. Clearly, all curves agree well in the overall behaviour, but when the low (i.e. high-frequency) components of the signal are filtered out the curve becomes considerably smoother. The more components are removed, the smoother the curve. This does not affect the overall behaviour (as long as sufficient low-frequency, long wavelength data remains intact). Of course, it would be impossible to recover any details that have been removed, if we wish to do so later.

Similar methods for data smoothing and data reduction are used in radio astronomy and other fields.

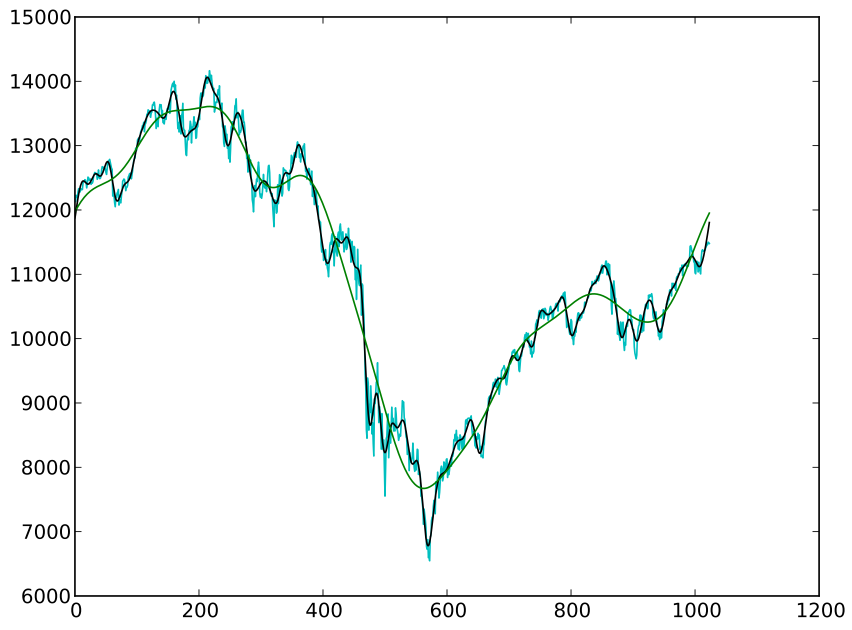


Figure 4: Results for Problem 3.