Python in Physics Education

ajith@iuac.res.in

July 31, 2008

1 Introduction

Mathematics is considered as the language of physics. Physical laws are expressed as mathematical relations and they are solved under desired boundary conditions to obtain the results. To illustrate this point take the example of the relationship between electric and magnetic fields as represented by the Maxwell's equations

$$\nabla \times E = -\frac{\partial B}{\partial t} \,;\; \nabla \times B = \mu_0 J + \mu_0 \varepsilon_0 \frac{\partial E}{\partial t} \,;\; \nabla . E = \frac{\rho}{\varepsilon_0} \,; \nabla . B = 0$$

In order to find out the field distribution inside a resonator cavity, we need to solve these equations by applying suitable boundary conditions dictated by the geometry of the cavity and electromagnetic properties of the material used for making it. For simple geometries, analytic solutions are possible but in most cases numerical methods are essential and that is one reason why computers are important for physics.

With the wide spread availability of personal computers, programming has been included in almost all Physics courses. Unfortunately the learning has been in separate compartments without any attempt to apply the acquired programming skills to understand physics in a better way. Partially this has been due to, in my view, the complexities of the chosen languages like FORTRAN, C, C++ etc. Another drawback was the lack of libraries to generate visual output. This writeup is an attempt to connect computer programming with physics learning. The language chosen is Python, due to its simplicity and the huge collection of libraries, mainly the ones for doing math and graphics.[1]

2 Python Basics

The first reaction from most of the readers would be like "Oh. No. One more programming language. I already had tough time learning the one know now" . Cool down, Python is not harmful as the name sounds. A high school student with average intelligence can pick it up within two weeks. What is there in a programming language anyway. It allows you to create variables, like elementary algebra, and allows you to manipulate them using arithmetic and logical operators. A sequence of such statements make the program. But most of the program are are not meant for running from the first line to the last line. Depending on the situation some lines may execute more than once and some may be totally skipped. This is done by using control flow statements for looping and decision making (while/for loops and if-elif-else kind of stuff). The variables belong to certain types like 'integer', 'float', 'string' etc. If you are new to programming refer to the book 'Snake wrangling for Kids', included in Phoenix CD. After that, as a test, read the code fragments below and if you can guess their outputs, you know enough to proceed with the material in this writeup.

Example 1: test1.py

[1] All the Python programs listed in this article are on the Phoenix Live CD. To run them boot from the liveCD, in graphics mode open a terminal, change to directory 'phy' and type
python program_name.py

```
a = 1
b = 1.2
c = 2 + 3j
d = 'i am a string'
e = [a, b, c, 'hello']
print d, b * c, e                        #what will be the output
```

Example 2: test2.py

```
x = 1
while x <= 10:
    print x * 8
    x = x + 1
```

Example 3: test3.py

```
s = 'hello world'
m = [1, 2.3, 33, 'ha']
for k in s:
    print k
for k in m:
    print m
```

Example 4: test4.py

```
x = range(10)
print x
for i in x:
    if i > 8:
        print i
```

Example 5: test5.py

```
a = [12, 23]
a.append(45)
print a
```

Example 6: test6.py (Three different ways to import a module)

```
import time
print time.ctime()
from time import *
print time()
import time as t
print t.time()
from math import sin  #imports only one function
print sin(2.0)
```

Some modules may have several sub-packages containing functions. For example the 'Special functions' sub-package from 'scipy' module can be imported as

```
from scipy import special
print special.j0(0.2)  # print values of Bessel function
```

3 Plotting Graphs

While studying physics, we come across various mathematical functions. In order to understand a function, we need to have some idea about its derivatives, integrals, location of zeros, maxima and minima, etc. A graphical representation conveys them quickly. To do graph plotting without much coding, we need to use some library (called modules in Python). We will use Matplotlib (refer to the matplotlib user's guide on Phoenix CD for details). Let us start with a simple example (plot1.py).

```
from pylab import *
x = range(10)
plot(x)
show()
```

The 'Matplotlib' package contains much more than simple plotting, and are available in a module named *pylab*. The first line imports all the functions from 'pylab'. The *range()* function generates a 'list' of ten numbers, that is passed on to the plot routine. When only a single list is given, *plot()* takes it as the y-coordinates and x-coordinates are taken from 0 to N-1, where N is the number of y-coordinates given. The last statement *show()* tells matplotlib to display the created graphs. You can specify both x and y coordinates to the plot routine. It is also possible to specify several other properties like the marker shape and color. (plot2.py)

```
from pylab import *
x = range(10,20)
y = range(40,50)
plot(x,y, marker = '+', color = 'red')
show()
```

Well, enough of the 2D plots using 'list' of integers. How about plotting some mathematical functions. Let us plot a sine curve. (plot3.py)

```
from pylab import  *
x = []                          # empty lists
y = []
for k in range(100):
    ang = 0.1 * k
    sang = sin(ang)              # sine function from pylab
    x.append(ang)
    y.append(sang)
plot(x,y)
show()
```

First we made two empty lists x and y. Then entered a *for loop* where the integer k goes from 0 to 99. For each value of 'k' we took '0.1 * k' as the angle and added to the list 'x'. The sine of each angle is added to the list 'y'. Doing this kind of coding is fine if your objective is to learn computer programming but we are trying to use it as a tool for learning physics.

3.1 NumPy and Scipy

To get rid of loops etc. we need data types likes arrays and matrices. Fortunately Python has such libraries, for example Numpy. For learning more about Numpy, refer to the Numpy documentation on the Phoenix CD or www.scipy.org website.[2] Try out the simple examples shown below. (numpy1.py)

[2] Numpy documentation is not complete, however one can go through the online help about numpy at its subpackages. Start Python interpreter from a Terminal, import the package and ask for help as shown below.

```
from numpy import *
a = array( [ 10, 20, 30, 40 ] )  # create an array from a list
print a
print a * 3
print a / 3
b = array([1.0, 2.0, 3.0, 4.0])
c = a + b
a = array([1,2,3,4], dtype='float')
a = array([[1,2],[3,4]])
```

Note that the operations performed on an array is carried out on each element, we need not do it explicitly. Numpy has several functions for the automatic creation of array objects. Try the following lines to see how they work.

```
a = ones(10)
print a
a = zeros(10)
print a
a= arange(0,10)
print a
a = arange(0, 1, 0.1)
print a
a = arange(0,10,1,dtype = 'float')
print a
a = linspace(0,10,100)
print a
```

Both 'arange' and 'linspace' functions generate an array. The first and second arguments are the first and last elements. For 'arange' , the third argument is the increment and for 'linspace' it is the number of elements in the array. In the case of arange() the last element may not match with the second argument given. We will try to make use of them in order to plot some mathematical functions. (numpy2.)

```
from pylab import *
from numpy import *
x = linspace(0.0, 2 * pi, 100)
s = sin(x)      # sine of each element of 'x' is taken
c = cos(x)      # cosine of each element of 'x' is taken
plot(x,s)
plot(x,c)
show()
```

Note that we have not imported Numpy explicitly. Pylab requires 'numpy' and importing pylab brings numpy automatically. The 'linspace' function generates an array of 100 equi-spaced values ranging from $0\, to\, 2\pi$. Also note that the $sin()$ function ,from Numpy, is a vectorized version. It takes an array as an argument, computes the sine of each element and returns the result in an array. Such features reduce the complexity of the program and allows us to concentrate on physics rather than the computer program. Comparing this with the previous program illustrates the simplicity achieved. Try another example generating lissagous figures, (numpy3.py)

```
#python
>>>import numpy
>>>help(numpy)
>>>help(numpy.core)
```

```
from pylab import *
x = linspace(0.0, 2 * pi, 100)
plot(sin(x), cos(x), 'b')            # plot lissagous figures
plot(sin(2*x), cos(x),'r')
plot(sin(x), cos(2*x),'y')
show()
```

3.2 Fourier Series

The Fourier series is a mathematical tool used for analyzing periodic functions by decomposing such a function into a weighted sum of much simpler sinusoidal component functions. Fourier series serve many useful purposes, as manipulation and conceptualization of the modal coefficients are often easier than with the original function. Areas of application include electrical engineering, vibration analysis, acoustics, optics, signal and image processing, and data compression. The example below shows how to generate a Square wave using this technique. To make the output better, increase the number of terms by changing the second argument of the range function. (fourier1.py)

```
from pylab import *
x = linspace(0.0, 2 * pi, 100)
y = sin(x)
for h in range(3,10,2):
        y = y + sin(h*x) / h
plot(x,y)
show()
```

4 Power Series

Trignometric functions like sine and cosine sounds very familiar to all of us, due to our familiarity with them since high school days. However most of us would find it difficult to obtain the numerical values of sin(2.0), say, without trigonometric tables or a calculator. We know that differentiating a sine function twice will give you the original function, with a sign reversal, which implies

$$\frac{d^2y}{dx^2} + y = 0$$

which has a series solution of the form

$$y = a_0 \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n}}{(2n)!} + a_1 \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n+1}}{(2n+1)!} \tag{1}$$

These are the Maclaurin series for sine and cosine functions. The following code plots several terms of the sine series and their sum. Try varying the number of terms to see the deviation. (series_sin.py)

```
from pylab import *
from scipy import *
x = linspace(-pi,pi,40)
a = zeros(40)
plot(x,sin(x))
for n in  range(1,5):
        sign = (-1)**(n+1)
        term = x**(2*n-1) / factorial(2*n-1)
        a = a + sign * term
        print n,sign
        plot(x,term)
```

```
plot(x,a,'+')
show()
```

The program given below evaluates the cosine function using this series and compares it with the result of the cos() function from the math library (that also must be doing the same). We have written a vectored version of the cosine function that can accept an array argument. You can study the accuracy of the results obtained by varying the number of terms included. (series_cos.py)

```
import scipy, pylab
def mycos(x):
        res = 0.0
        for n in range(18):
                res = res + (-1)**n *   (x ** (2*n)) / scipy.factorial(2*n)
        return res
def vmycos(ax):
        y = []
        for x in ax:
                y.append(mycos(x))
        return y

x = scipy.linspace(0,4*scipy.pi,100)
y = vmycos(x)
pylab.plot(x,y)
pylab.plot(x,scipy.cos(x),'+')
pylab.show()
```

5 3D Plots

Matplotlib has a subpackage that supports 3D plots. Some simple examples are given below and we will come back to it later. (plot3d1.py)

```
from numpy import *
import pylab as p
import matplotlib.axes3d as p3
w = v = u = linspace(0, 4*pi, 100)
fig=p.figure()
ax = p3.Axes3D(fig)
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
ax.plot3D(u,v,sin(w))
ax.plot3D(u,sin(v),w)
ax.plot3D(sin(u),v,u)
p.show()
```

6 Special Functions

Solving certain differential equations give rise to special functions like Bessel. Legendre, Spherical harmonic etc. depending upon the symmetry properties of the coordinate systems used. Now we will start using another package called 'Scipy'.[3] Scipy has subpackages for a large number of mathematical operations inclding the computation of various special functions.

[3]Refer to the Phoenix CD or www.scipy.org for scipy documentation.

6.1 Bessel Functions

Let us start with plotting the Bessel functions, restricted to integral values of n, given by the expression

$$J_n = \left(\frac{x}{2}\right)^n \sum_{k=0}^{\infty} \frac{(-)^k \left(\frac{x}{2}\right)^{2k}}{(k!)\,(n+k)!} \tag{2}$$

using the following Python program. It uses the 'Scipy' module. (bes1.py)

```
from pylab import *
from scipy import *
def vj0(xarray):
        y = []
        for  x in xarray:
                val = special.j0(x)        # Compute Jo
                y.append(val)
        return y
a = linspace(0,10,100)
b = vj0(a)
plot(a,b)
show()
```

Note that function calculating Bessel function does not accept array arguments. So we had to write a vectorized version of the same. The next example generalizes the computation to higher orders as shown below. (bes2.py)

```
from pylab import *
from scipy import *
def vjn(n,xarray):
        y = []
        for  x in xarray:
                val = special.jn(n,x)        # Compute Jn(x)
                y.append(val)
        return y
a = linspace(0,10,100)
for n in range(5):
        b = vjn(n,a)
        plot(a,b)
show()
```

In the two examples above, the order in which pylab and scipy are imported matters due to the following reason. The function linspace() is there in both the packages. They do the same job but the elements of the array returned by scipy.linspace() are 'float' type but they are 'numpy.float64' in the case of numpy.linspace(). These sort of conflicts can arise when you are importing multiple libraries. The best way to avoid them is to use the following syntax.

```
import scipy
import numpy
a = scipy.linspace(0,1,10)       # from scipy
b = numpy.linspace(0,1,10)    # from numpy
c = linspace(0,1,10)                 # from numpy, last import
```

Instead of using the library function from scipy(), we can calculate the Bessel functions using the expression 2 and compare the results. Try changing the number of terms to see the deviations. (bes3.py)

```
from scipy import *
from scipy import special
import pylab
def jn(n,x):
        jn = 0.0
        for k in range(30):
                num = (-1)**k * (x/2)**(2*k)
                den = factorial(k)*factorial(n+k)
                jn = jn + num/den
        return jn * (x/2)**n
def vjn_local(n,xarray):
        y = []
        for  x in xarray:
                val = jn(n,x)        # Jn(x) using our function
                y.append(val)
        return y

def vjn(n,xarray):
        y = []
        for  x in xarray:
                val = special.jn(n,x)        # Compute Jn(x)
                y.append(val)
        return y
a = linspace(0,10,100)
for n in range(2):
        b = vjn(n,a)
        c = vjn_local(n,a)
        pylab.plot(a,b)
        pylab.plot(a,c,marker = '+')
pylab.show()
```

6.2 Polynomials

Before proceeding with other special functions, let us have a look at the one dimensional polynomial class 'poly1d' of Scipy. You can define a polynomial by supplying the coefficient as a list. For example , the statement p = poly1d([1,2,3]) constructs the polynomial x**2 + 2 x + 3. The following example describe the various operations you can do with this class. (ploy1.py)

```
import scipy
import pylab
a = scipy.poly1d([3,4,5])
print a, ' is the polynomial'
print a*a, 'is its square'
print a.deriv(), ' is its derivative'
print a.integ(), ' is its integral'
print a(0.5), 'is its value at x = 0.5'
x = scipy.linspace(0,5,100)
b = a(x)
pylab.plot(a,b)
pylab.show()
```

Note that a polynomial can take an array argument for evaluation to return the results in an array. [4]

6.3 Legendre Functions

The Legendre polynomials, sometimes called Legendre functions of the first kind, are solutions to the Legendre differential equation

$$(1-z^2)\frac{d^2w}{dz^2} - 2z\frac{dw}{dz} + n(n+1)w = 0 \tag{3}$$

If n is an integer, they are polynomials. The Legendre polynomials P_n(x) are illustrated using the Python program given below. (legen1.py)

```
from scipy import linspace, special
import pylab
x = linspace(-1,1,100)
for n in range(1,6):
        leg = special.legendre(n)
        y = leg(x)
        pylab.plot(x,y)
pylab.show()
```

6.4 Spherical Harmonics

In mathematics, the spherical harmonics are the angular portion of an orthogonal set of solutions to Laplace's equation represented in a system of spherical coordinates. Spherical harmonics are important in many theoretical and practical applications, particularly in the computation of atomic electron configurations, the representation of the gravitational field, magnetic field of planetary bodies, characterization of the cosmic microwave background radiation etc.. Laplace's equation in spherical polar coordinates can be written as

$$\nabla^2\Phi = \frac{1}{r^2}\frac{\partial}{\partial r}\left(r^2\frac{\partial\Phi}{\partial r}\right) + \frac{1}{r^2\sin\theta}\frac{\partial}{\partial\theta}\left(\sin\theta\frac{\partial\Phi}{\partial\theta}\right) + \frac{1}{r^2\sin^2\theta}\frac{\partial^2\Phi}{\partial\phi^2}$$

Separation of variables in $r, \theta and \phi$ coordinates gives the angular part of the solution as

$$Y_l^m(\theta,\phi) = \sqrt{\frac{(2\ell+1}{4\pi}\frac{(l-m)!}{(l+m)!}}P_l^m(\cos\theta)e^{im\phi} \tag{4}$$

This is a called a spherical harmonic function of degree ℓ and order m , $P_\ell^m(\theta)$ an associated Legendre function. The following program calculates the spherical harmonics for $\ell = 10, m = 0$. Change the values of $\ell and m$ to observe the changes. (sph1.py)

```
from pylab import *
from scipy import special
a_th = []       # list to store polar angle theta from -90 to + 90 deg
a_sph = []      # list to store absolute values if sperical harminics
phi = 0.0       # Fix azimuth, phi at zero
theta = -pi/2           # start theta from -90 deg
while theta < pi/2:
        h = special.sph_harm(0,10, phi, theta)  # (m, l , phi, theta)
        a_sph.append(abs(h))
```

[4] To know more type help(poly1d) at the python prompt after importing scipy;

```
>>>import scipy
>>>help(scipy.poly1d)
```

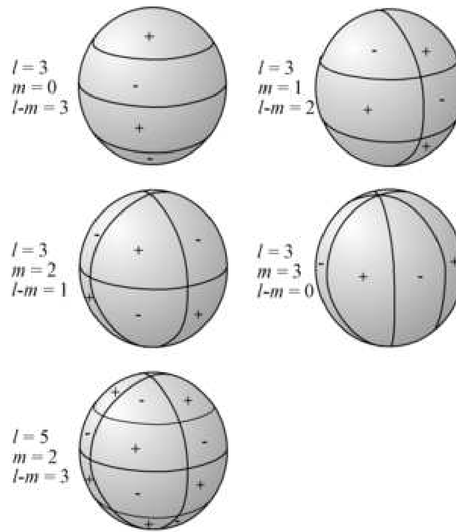

Figure 1: Schematic representation of Y_ℓ^m on the unit sphere. Y_ℓ^m is equal to 0 along m great circles passing through the poles, and along l-m circles of equal latitude. The function changes sign each time it crosses one of these lines.

```
        a_th.append(theta * 180/pi)
        theta = theta + 0.02
plot(a_th,a_sph)
show()
```

The spherical harmonics are easily visualized by counting the number of zero crossings they possess in both the latitudinal and longitudinal directions. For the latitudinal direction, the associated Legendre functions possess $\ell - |m|$ zeros, whereas for the longitudinal direction, the trigonomentric sin and cos functions possess $2\,|m|$ zeros.

When the spherical harmonic order m is zero, the spherical harmonic functions do not depend upon longitude, and are referred to as zonal. When $\ell = |m|$, there are no zero crossings in latitude, and the functions are referred to as sectoral. For the other cases, the functions checker the sphere, and they are referred to as tesseral. The variation of a function on the surface of a sphere is shown in figure 1. The Python program given below gives a 3D plot of a sphere. The radius is modulated using the value of Y_ℓ^m so that we can visualize it. (sph2.py)

```
from numpy import *
from scipy import special
import pylab as p
import matplotlib.axes3d as p3
phi = linspace(0, 2*pi, 50)
theta = linspace(-pi/2, pi/2, 200)
ax = []
ay = []
az = []
R = 1.0
for t in theta:
        polar = float(t)
        for k in phi:
                azim = float(k)
                sph = special.sph_harm(0,5,azim, polar) # Y(m,l,phi,theta)
                modulation = 0.2 * abs(sph)
                r = R * ( 1 + modulation)
```

```
                x = r*cos(polar)*cos(azim)
                y = r*cos(polar)*sin(azim)
                z = r*sin(polar)
                ax.append(x)
                ay.append(y)
                az.append(z)
fig=p.figure()
f = p3.Axes3D(fig)
f.set_xlabel('X')
f.set_ylabel('Y')
f.set_zlabel('Z')
f.scatter3D(ax,ay,az)
p.show()
```

7 Numerical Differentiation

Simple Numerical Differentiation For a function y=f(x) , the derivative dy/dx can be obtained numerically. We take the difference between consecutive elements dy and divide it with the x increment dx. The following program create an array of *sines* and find the derivative numerically. The result is compared with the *cosines* . (diff1.py)

```
from pylab import *
dx = 0.1                # value of x increment
x = arange(0,10, dx)
y = sin(x)
yprime = []             # empty list
for k in range(99):     # from 100 points we get 99 diference values
     dy = y[k+1]-y[k]
     yprime.append(dy/dx)
x1 = x[:-1]             # A new array without the last element
x1 = x1 + dx/2          # The derivative corresponds to the middle point
plot(x1, yprime, '+')
plot(x1, cos(x1))       # Cross check with the analytic value
show()
```

(to be modified later)

8 Solving Differential Equations

If we know the value of a function $y = f(x)$ and its derivative at some particular value of 'x' , we can calculate the value of the function at a nearby point $x + dx$. $Y_{n+1} = Y_n + dx.(dy/dx)$. Integrating a function using this method is not very efficient compared to the more sophisticated methods like fourth order Runge-Kutta. However our objective is to just understand how numerical integration works. We will trace the 'sine' function, that can be

$$\frac{d(\sin(x))}{dx} = \cos(x) \tag{5}$$

8.1 Runge-Kutta Integration

In the previous section we have used the Euler method that uses the formula

$$y_{n+1} = y_n + hf(x_n, y_n) \tag{6}$$

to evaluate the value of 'y' at point 'n+1' using the known value at 'n'. The formula is asymmetrical since it uses the derivative information at the beginning of the interval. In Fourth order Runge-Kutta method, for each step the derivative is evaluated four times; once at the initial point, twice at two trial midpoints, once at trial a endpoint. The final value is evaluated from these derivatives using the equations

$$k_1 = hf(x_n, y_n)$$

$$k_2 = hf(x_n + \frac{h}{2}, y_n + \frac{k_1}{2})$$

$$k_3 = hf(x_n + \frac{h}{2}, y_n + \frac{k_2}{2})$$

$$k_4 = hf(x_n + h, y_n + k_3)$$

$$y_{n+1} = y_n + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6}$$

In numerical analysis, the Runge-Kutta methods are an important family of implicit and explicit iterative methods for the approximation of solutions of ordinary differential equations. These techniques were developed around 1900 by the German mathematicians C. Runge and M.W. Kutta. For more details on fouth order Runge-Kutta method refer to the file 'rk.pdf' on the Phoenix Live CD. The following program makes the sine curve using *fourth order RK method*. At any point the derivative of the curve is cos(t) and we use this information to estimate the next point. (rk_sin.py)

```
from pylab import *
def func(t):
        return cos(t)
def rk4(s,t):
        k1 = dt * func(t)
        k2 = dt * func(t + dt/2.0)
        k3 = dt * func(t + dt/2.0)
        k4 = dt * func(t + dt)
        return s + ( k1/6 + k2/3 + k3/3 + k4/6 )
t = 0.0         # Stating angle
dt = 0.01       # value of angle increment
val = 0.0        # value of the 'sine' function at t = 0
tm = [0.0]      # List to store theta values
res = [0.0]     # RK4 results
while t < 10:
    val = rk4(val,t)      # get the new value
    t = t + dt
    tm.append(t)
    res.append(val)
plot(tm, res,'+')
plot(tm, sin(tm))       # compare with actual curve
show()
```

9 Physics Simulations

Using the numerical methods discussed in the previous sections we will try to solve some simple problems in physics, like calculating the trajectory of an object in a known force field.

9.1 Mass-Spring Problem

As a simple example let us take the "Mass-Spring" problem. A mass is attached to one end of a spring whose other end is fixed. If we displace the mass from the equilibrium position by x , the force is given by the relation $F = -kx$, where k is the spring constant. If we release the mass at this point, it will start oscillating. The objective is to plot the displacement of the mass as a function of time. The force is known at the initial point $-kx$, the acceleration of the mass when it is released is given by Newton's equation $F = ma$, where m is the mass of the object. The initial position is Xo and initial velocity Vo is zero. We can integrate the acceleration to find out the velocity after a small time interval 'dt'. Similarly, integrating the velocity will give the displacement. (spring1.py)

```
from pylab import *
t = 0.0          # Stating time
dt = 0.01        # value of time increment
x = 10.0          # initial position
v = 0.0          # initial velocity
k = 10.0        # spring constant
m = 2.0          # mass of the oscillating object
tm = []          # Empty lists to store time, velocity and displacement
vel = []
dis = []
while t < 5:
  f = -k * x                      # Try (-k*x - 0.5 * v) to add damping
  v = v +  (f/m ) * dt            # dv = a.dt
  x = x + v * dt                  # dS = v.dt
  t = t + dt
  tm.append(t)
  vel.append(v)
  dis.append(x)
plot(tm,vel)
plot(tm,dis)
show()
```

From the result we can see the phase relationship between the velocity and the displacement. If you want to see the effect of damping add that to the expression for force. Try changing parameters like spring constant, mass etc. to study the effect of them on the result.

The above problem is solved using the Fourth order Runge-Kutta method in the program listed below. The displacement, as a function of time, calculated using RK method is compared with the result of the analytical expression. (spring2.py)

```
"""
The rk4_two() routine in this program does a two step integration using
an array method. The current x and xprime values are kept in a global
list named 'val'.
val[0] = current position; val[1] = current velocity
The results are compared with analytically calculated values.
"""
from pylab import *
```

```
def accn(t, val):
        force = -spring_const * val[0] - damping * val[1]
        return force/mass

def vel(t, val):
        return val[1]
def rk4_two(t, h):      # Time and Step value
        global xxp      # x and xprime values in a 'xxp'
        k1 = [0,0]      # initialize 5 empty lists.
        k2 = [0,0]
        k3 = [0,0]
        k4 = [0,0]
        tmp= [0,0]
        k1[0] = vel(t,xxp)
        k1[1] = accn(t,xxp)
        for i in range(2):      # value of functions at t + h/2
                tmp[i] = xxp[i] + k1[i] * h/2
        k2[0] = vel(t + h/2, tmp)
        k2[1] = accn(t + h/2, tmp)
        for i in range(2):      # value of functions at t + h/2
                tmp[i] = xxp[i] + k2[i] * h/2
        k3[0] = vel(t + h/2, tmp)
        k3[1] = accn(t + h/2, tmp)
        for i in range(2):      # value of functions at t + h
                tmp[i] = xxp[i] + k3[i] * h
        k4[0] = vel(t+h, tmp)
        k4[1] = accn(t+h, tmp)
        for i in range(2):      # value of functions at t + h
                xxp[i] = xxp[i] + ( k1[i] + \
                2.0*k2[i] + 2.0*k3[i] + k4[i]) * h/ 6.0
t = 0.0                 # Stating time
h = 0.01                # Runge-Kutta step size, time increment
xxp = [2.0, 0.0]        # initial position & velocity
spring_const = 100.0    # spring constant
mass = 2.0              # mass of the oscillating object
damping = 0.0
tm = [0.0]              # Lists to store time, position & velocity
x = [xxp[0]]
xp = [xxp[1]]
xth = [xxp[0]]
while t < 5:
    rk4_two(t,h)                # Do one step RK integration
    t = t + h
    tm.append(t)
    xp.append(xxp[1])
    x.append(xxp[0])
    th = 2.0 * cos(sqrt(spring_const/mass)* (t))
    xth.append(th)
plot(tm,x)
plot(tm,xth,'+')
show()
```

9.1.1 Mass Spring problem in 3D

The following program displays the movement of the mass attached toa spring in 3D graphics. (spring3d.py)

```
from visual import *
base = box (pos=(0,-1,0), length=16, height=0.1, width=4, color=color.blue)
wall = box (pos=(0,0,0), length=0.1, height=2, width=4, color=color.white)
ball = sphere (pos=(4,0,0), radius=1, color=color.red)
spring = helix(pos=(0,0,0), axis=(4,0,0), radius=0.5, color=color.red)
ball2 = sphere (pos=(-4,0,0), radius=1, color=color.green)
spring2 = helix(pos=(0,0,0), axis=(-4,0,0), radius=0.5, color=color.green)
t = 0.0
dt = 0.01
x1 = 2.0
x2 = -2.0
v1 = 0.0
v2 = 0.0
k = 1000.0
m = 1.0
while 1:
   rate(20)
   f1 = -k * x1
   v1 = v1 +  (f1/m ) * dt        # Acceleration = Force / mass ; dv = a.dt
   f2 = -k * x2 - v2                # damping proportional to velocity
   v2 = v2 +  (f2/m ) * dt       # Acceleration = Force / mass ; dv = a.dt
   x1 = x1 + v1 * dt
   x2 = x2 + v2 * dt
   t = t + dt
   spring.length =  4 + x1
   ball.x =  x1 + 4
   spring2.length = 4 - x2
   ball2.x =  x2 - 4
```

9.2 Charged particle in Electric and Magnetic Fields

The following example traces the movement of a charged particle under the influence of electric and magnetic fields. You can edit the program to change the components of the feilds to see their effect on the trajectory. For better results we should rewrite it using RK4 method. (cp_em.py)

```
from numpy import *
import pylab as p
import matplotlib.axes3d as p3
Ex = 0.0        # Components of Applied Electric Field
Ey = 2.0
Ez = 0.0
Bx = 0.0        # Magnetic field
By = 0.0
Bz = 2.0
m = 2.0         # Mass of the particle
q = 5.0         # Charge
x = 0.0         # Components of initial position and velocity
y = 0.0
z = 0.0
```

```
vx = 20.0
vy = 0.0
vz = 0.0
a = []
b = []
c = []
t = 0.0
dt = 0.01
while t < 6:    # trace until time reaches 1
  Fx = q * (Ex + (vy * Bz) - (vz * By) )
  vx = vx + Fx/m * dt           # Acceleration = F/m; dv = a.dt
  Fy = q * (Ey - (vx * Bz) + (vz * Bx) )
  vy = vy + Fy/m * dt
  Fz = q * (Ez + (vx * By) - (vy * Bx) )
  vz = vz + Fz/m * dt
  x = x + vx * dt
  y = y + vy * dt
  z = z + vz * dt
  a.append(x)
  b.append(y)
  c.append(z)
  t = t + dt

p.plot(c,b)
fig=p.figure()
ax = p3.Axes3D(fig)
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
ax.plot3D(a,b,c)
p.show()
```

9.3 Motion in a central field

Run the program cf3d.py from the CD to simulate motion of a mass in a gravitational field.

10 Matrix manipulation

(todo)

11 Random Distributions

(todo)