Assignment 11 - Main Program

```
import assign_11_mod as ruth #import the content from the other file, ‘
    wk11_mod.py’, use the name ‘ruth’ (as in ‘Rutherford’) to refer to it
    instead

iteration_counter = 0 #initialise a counter for the total number of
    iterations

c = 137.035999 #speed of light in a.u.
q_au = 79.0 #charge on gold nucleus
q_alpha = 2.0 #charge on alpha particle
k = 1.0 #coulomb constant
m_alpha = 7294.3 #mass of alpha particle
dt = 1E-5 #timestep size, also a.u.

r = ruth.Vector(0.0,-0.005) #initialise the position vector
v = ruth.Vector(0.0,0.0) #initialise the velocity vector

#read in the x-offset for the position
r.x = float(raw_input('Please specify the x-offset of the incident Alpha
    particle in Bohr Radii: '))

#read in the y-velocity and scale by speed of light
v.y = float(raw_input('Please specify the y-velocity of the incident Alpha
    particle as a fraction of c: '))
v.y *= c

#initialise the current distance minimum
min_distance = r.mag()

#set the radial boundary
r_limit = 1.1 * r.mag()

#open and write initial condition to output file
fout = open('assign_11.out','w')
fout.write('%e %e\n'%(r.x,r.y))

#keep looping while the position is within the radial boundary
while (r.mag() < r_limit):
  a = ((k*q_au*q_alpha)/(m_alpha * r.mag()**2)) * r.unit() #acceleration
    vector calculation
  v += a*dt #change velocity by dv (=a.dt)
  r += v*dt #change position by dr (=v.dt)
  fout.write('%e %e\n'%(r.x,r.y)) #output new coordinate
  min_distance = min(r.mag(),min_distance) #update if new minimum distance is
     found
  iteration_counter += 1 #increment counter of steps by 1

#output the number of iterations, simulated time, and distance of closest
    approach
print 'Calculation took %d iterations; simulated time %e t_A'%(
    iteration_counter,float(iteration_counter)*dt)
print 'Distance of closest approach = %e a_0'%(min_distance)

#close output file
fout.close()
```

Assignment 11 - Module

```
import numpy

#Vector object and methods, all vector forms and operations
#class Vector:
class Vector(object):
  #initialisation
  __slots__ = ('x','y') #limit properties to include only x and y components
   (otherwise random additionals unrelated to vectors can be added
   dynamically, which could break other parts of the vector operations)
  def __init__(self, x, y):
    self.x = x
    self.y = y
  #method to display contents of vector object (default separated by commas)
  def show(self, mode='comma'):
    if (mode == 'comma'):
      a = ','
    elif (mode == 'space'):
      a = ' '
    elif (mode == 'tab'):
      a = '\t'
    else:
      a = str(mode)
    return '[' + str(self.x) + a + str(self.y) + ']'
  def __repr__(self):
    return self.show()
  def __str__(self):
    return self.show()
  #method which returns the magnitude of a vector object
  def mag(self):
    a = numpy.sqrt(self.x*self.x + self.y*self.y)
    return a
  ###Unique methods within vector type
  #method which adds two vector objects together
  def add(a,b):
    x = a.x + b.x
    y = a.y + b.y
    return Vector(x, y)
  #method which subtracts two vector objects
  def sub(a,b):
    x = a.x - b.x
    y = a.y - b.y
    return Vector(x, y)
  #scale a vector by a scalar
  def scale(a,b):
    x = a.x * b
    y = a.y * b
    return Vector(x, y)
  ###Replacement to default operations when using vectors
  def __add__(self,other):
    return self.add(other)
  def __sub__(self,other):
    return self.sub(other)
  def __mul__(self,other):
    return self.scale(other)
  def __rmul__(self,other):
    return self.scale(other)
  #perform dot product between two vectors
  def dot(a,b):
    x = a.x * b.x
    y = a.y * b.y
```

```
59      return x + y
60    #perform cross product between two vectors
61    #def cross(a,b):
62    # x = (a.y * b.z) - (a.z * b.y)
63    # y = (a.z * b.x) - (a.x * b.z)
64    # z = (a.x * b.y) - (a.y * b.x)
65    # return Vector(x, y, z)
66    #return unit vector for one vector
67    def unit(a):
68      b = a.scale(1.0 / a.mag())
69      return b
70
71  Null = Vector(0.0,0.0)
```