```
from numpy import *
import matplotlib.pyplot as plt
#First define the function for the first 3 questions
def f(y):
    return y
results = [] #A list containing the essential results of the experiment
printing = False #A flag for output
# ============================================================================
# Question 1
# ============================================================================
#First define the Euler function, taking a function, the initial and final
#points, the initial y value, and the number of steps used
def Euler(f,x0,x1,y0,n):
    h = (x1-x0)/float(n)
    y=y0
    for j in arange(n):
        y+=h*f(y)
    return y
#Set a tolerance to ensure a precision of 10^-5 of the result
Tol = 0.000001
#For Euler's Method, we begin with a rough underestimate as a reasonable
#starting point for iteration
N = 700000
yN = Euler(f,0.0,1.0,1.0,N)
#creating a loop to iterate the N values until the result is within the
#defined tolerance of the 'true' answer
while abs(yN-2.718281)> Tol:
    yN = Euler(f,0.0,1.0,1.0,N)
    N += 100
    if abs(yN-2.718281)> Tol:
        if printing:
            print N , yN
#just adding essential results...to our results list!
results.append(N)
results.append(yN)
if printing:
    print N , yN, '\n'
# ============================================================================
# Question 2
# ============================================================================
#Defiing the improved Euler method: very similar to forward Euler in terms of
#set-up, the only difference is in the loop obviously
def ImprovedEuler(f,x0,x1,y0,n):
    h = (x1-x0)/float(n)
    y=y0
    for j in arange(n):
        y+=0.5*h*(f(y)+f(y+h*f(y)))
    return y
Tol = 0.000001
#Testing has shown this method needs a couple orders of magnitude less N's to
#get the job done, so we begin with a much smaller N. The rest of the code
#Is exactly as above
N =10
yN = ImprovedEuler(f,0.0,1.0,1.0,N)
while abs(yN-2.718281)> Tol:
    yN = ImprovedEuler(f,0.0,1.0,1.0,N)
    N += 1
    if abs(yN-2.718281)> Tol:
        if printing:
            print N , yN
results.append(N)
results.append(yN)
```

```
if printing:
    print N , yN,'\n'
# ============================================================================
# Question 3
# ============================================================================
#Finally to the classical fourth-order Runge-Kutta method. I added the color
#parameter for the plotting in Question 4. The Xstep and Ystep lists are merely
#The obtained x and y position values at the mesh points. The k's follow from
#the definition of RK4 as does the value for y_n+1. The 'if not instance' line
#is added because I am using the code for inputs of both floats and lists,
#Thus the distinction is made to ensure there are no bugs for Question 4b.
def RK4(f,x0,x1,y0,n,color):
    h = (x1-x0)/float(n)
    y=y0
    Xstep = []
    Ystep = []
    for j in arange(n):
        k1 = f(y)
        k2 = f(y + 0.5*h*k1)
        k3 = f(y + 0.5*h*k2)
        k4 = f(y + h*k3)
        y += (1.0/6.0)*h*(k1 + 2.0*k2 + 2.0*k3 + k4)
        if not isinstance(y,float):
            Xstep.append(y[0])
            Ystep.append(y[1])
            if Ystep[j] < 0:
                results.append(Xstep[j-1])#appending the maximum x value
                for k in arange(j-1):
                    if Ystep[k+1] < Ystep[k]:#Catches the maximum y value
                        results.append(Ystep[k])#append max y value
                        if printing:
                            print '\n' , Ystep[k] , '\n'
                        break
                plt.plot(Xstep,Ystep,color,linewidth=1)
                plt.title('Trajectory of a Soccer-ball parametrized by time t')
                plt.xlabel('x(t)')
                plt.ylabel('y(t)')
                break
    return y
#The code below is exactly as the last segments for Questions 1 and 2
Tol = 0.000001
N =1
yN = RK4(f,0.0,1.0,1.0,N,None)
while abs(yN-2.718281)> Tol:
    yN = RK4(f,0.0,1.0,1.0,N,None)
    N += 1
    if abs(yN-2.718281)> Tol:
        if printing:
            print N , yN
results.append(N)
results.append(yN)
if printing:
    print N , yN , '\n'
#============================================================================
# Question 4
# ============================================================================
#Our vecorized Phi function of 4 variables. PosiVelo stands for
#'PositionVelocity'. Returns a list of u,v,u', and v' when called
def Phi(PosiVelo):
    Phi = array([0.,0.,0.,0.,])
    Phi[0] = PosiVelo[2]
    Phi[1] = PosiVelo[3]
```

```
    Phi[2] = -0.024*PosiVelo[2]*sqrt(PosiVelo[2]**2 + PosiVelo[3]**2)
    Phi[3] = -9.8-0.024*PosiVelo[3]*sqrt(PosiVelo[2]**2 + PosiVelo[3]**2)
    return Phi
#Same as Phi but now neglecting air-resistance
def PhiPrim(PosiVelo):
    PhiPrim = array([0.,0.,0.,0.,])
    PhiPrim[0] = PosiVelo[2]
    PhiPrim[1] = PosiVelo[3]
    PhiPrim[2] = 0.0
    PhiPrim[3] = -9.8
    return PhiPrim
#Below is the code for Question 4a. i.e. the code for finding the ratio R
N = 100
n1 = RK4(Phi,0.0,2.0,array([0.0,0.0,25.0*sqrt(2.0)/2.0,25.0*sqrt(2.0)/2.0]),N,
None)[0]
N = 200
n2 = RK4(Phi,0.0,2.0,array([0.0,0.0,25.0*sqrt(2.0)/2.0,25.0*sqrt(2.0)/2.0]),N,
None)[0]
N = 400
n3 = RK4(Phi,0.0,2.0,array([0.0,0.0,25.0*sqrt(2.0)/2.0,25.0*sqrt(2.0)/2.0]),N,
None)[0]
R = (n1 - n2)/(n2 - n3)
results.append(R)
if printing:
    print R
#Code for questions 4c,d. Note that h is implicitly given as 10^-3, through
#its definition via N and x_n-x_0. Here we have h = 4/4000 = 10^-3 as desired.
N = 4000
fullFlightDrag =
RK4(Phi,0.0,4.0,array([0.0,0.0,25.0*sqrt(2.0)/2.0,25.0*sqrt(2.0)/2.0]),N,'b-')
fullFlight =
RK4(PhiPrim,0.0,4.0,array([0.0,0.0,25.0*sqrt(2.0)/2.0,25.0*sqrt(2.0)/2.0]),N,
'r--')
plt.savefig('C:/Users/Parma_Shon/Desktop/Math 514/HW//HW6/figure_1.png',
bbox_inches = 'tight')
#plt.show()
print results
#Below is the list of results. In order, we have: N_ForwardEuler,
#eapprox_ForwardEuler,N_ImprovedEuler,eapprox_ImprovedEuler,N_RK4,
#eapprox_RK4,the ratio R, X_maximum with drag, Y_maximum with drag,
#X_maximum without drag, and Y_Maximum without drag
#Note the value for the ratio of R. Since RK4 is a fourth order method,
#we expect the global error to go down as a fourth power, thus taking
#four times as many N gives an decrease in the global error by 2^4 = 16.
#So we're overall pretty close to this number.
# [743500, 2.718280000184576, 499, 2.7182800044357713, 12, 2.7182803940365696,
# 16.195583212805523, 31.465361952749166m, 10.372835405913872m,
# 63.763353993493219m, 15.943877431513434m]
```