```
# -*- coding: utf-8 -*-
"""
Jonathan Ryding
2nd yr Computational Physics, Project 2: Damped Harmonic Oscillator

Program that uses four step-by step iterative methods to solve an unforced damped
harmonic oscillator and compares with a solution calculated analytically.
#Data is written to files then plotted from these files.
Program shows the accuracy of the methods with varying step sizes.
The best method, Verlet, is then used to solve the differential equation  with
driving forces for the analytically unsolved.
Effects of driving forces (pulses and sinusoidals at varying frequencies) are
viewed.

@author: mclssjr6
"""

import numpy as np
import matplotlib.pyplot as plt


#variables of the system, can be edited.

k = 1.55 #spring constant of the oscillator [N / m]
m = 5.81 #mass of ball
b = 0 #damping coefficient

omega0 = (k/m)**0.5
omega  = ((k / m) - pow((b/(2*m) ),2) )**0.5 #oscillator angular frequency general
for damped oscillations
gamma = b/m #gamma constant

b_crit = 2 *np.sqrt(k*m) #critical damping coefficient, defined where omega = 0


#initial conditions of the oscillator, can be edited.
x0 = 0 #initial displacement
v0 = -1 #initial velocity

h = 0.1 #step size used per iteration of each method
#cannot choose h so small that round-off errors become significant
time_tot = 100 #length of time desired to view oscillations

def writeFile(file_name, time, X, V, E ):
#Function that writes given data to a file.
    file = open(file_name, "w")
    print("Creating file: " + file_name + " for writing data into.")

    for i in range(0, len(V)):
        file.write("%f\t%f\t%f\t%f\n" %(time[i], X[i], V[i], E[i]) )

    print("Writing completed, closing.")
    file.close()

    return(None)

def readFile(file_name):
#Function that reads data in from a file
    opened = 0
```

```
    try:
#try to open the file, change a local variable if opens
        file = open(file_name,'r')

        opened = 1

   #file opened
    except:

        print("Could not open file: " + file_name + " for reading in data.")

    #file did not open, print a failure message

    if opened is 1:

        file = np.loadtxt(file_name)

        X = np.array(file[:,1]) #position, in metres
        V = np.array(file[:,2]) #velocity, in metres per second
        E = np.array(file[:,3]) #total energy, in Joules

    return(X, V, E)

def time(time_total, h_step):
    N_points = int(time_total/h_step) #converts measuring time and step size into a
integer number of steps to do

#creating a time set that is universal across all functions
    t_time = np.zeros(N_points) #creating a zeros array with sufficient entries for
the given time desired

    for i in range(1, N_points):
        t_time[i] = i * h_step #progressing the time by steps to avoid small errors
functions such as np.linspace have for h steps that are not factors of the total
time, e.g. h = 0.3
#ensures time array is correct
    return(t_time, N_points)

def euler(file_name, x0, v0, h_step, N):
#Euler function
#Uses the Euler step-by-step iterative method to approximate numerically the damped
harmonic differential equation given initial conditions (as x0,v0)


    X, V, E = np.zeros(N), np.zeros(N), np.zeros(N-1)
    #X, V, E correspond to displacement, velocity and total energy respectively.
    #It is more memory efficient to create an array of size N than to append to an
empty array.

    X[0] = x0 #initial position and velocity are set into the arrays
    V[0] = v0

    for i in range(1, N): #calculate positions and velocities step-by-step until
the number of iterations, N, is reached.

        X[i] = X[i-1] + V[i-1] * h_step #Euler method: X_n+1 = X_n + V_n * h
        #position local truncation error of order h^2, #taylor expansion
        V[i] = V[i-1] - ((b/m)*V[i-1]   + (k /m)*X[i-1] )* h_step #V_n+1 = V_n +
```

```
a_n * h , a_n is an acceleration

        #velocity local truncation error of order h^2
    for i in range(0,N-1):
        E[i] =(m*V[i]**2/2 + k*X[i]**2/2)

    #Remove last elements of position and velocity to data can be compared to
Verlet method (function below) where last velocity element cannot be calculated.
    X = X[0:-1]
    V = V[0:-1]

    writeFile(file_name , t, X, V, E)
    return(None)

def improvedEuler(file_name, x0, v0, h_step):
#Improved Euler method function
#Similar to the Euler method above, but the position, time-derivative of velocity,
an extra term from the Taylor expansion.

    X, V, E = np.zeros(N), np.zeros(N), np.zeros(N-1)
    #X, V, E correspond to displacement, velocity and total energy respectively.

    X[0] = x0 #initial position and velocity are set into the arrays
    V[0] = v0

    for i in range(1, N):

        X[i] = X[i-1] + V[i-1] * h_step - ((b/m)*V[i-1]   + (k /m)*X[i-1] )*
(h_step**2)/2 #extra term: a_n* h^2/2 included
        #position local truncation error of order h^3, expected to be more accurate
then previous Euler method.
        V[i] = V[i-1] - ((b/m)*V[i-1]   + (k /m)*X[i-1] )* h_step
        #velocity local truncation error of order h^2

    for i in range(0,N-1):
        E[i] =(m*V[i]**2/2 + k*X[i]**2/2)

    X = X[0:-1]
    V = V[0:-1]

    writeFile(file_name , t, X, V, E)
    return(None)

def verlet(file_name, x0, v0, h_step, b_input, F):
#Verlet method function
#Uses Verlet method (position as the centred time- derivative) to calculate
solutions step-by-step.
#Requires an extra starting point from Euler method.
#Verlet was found to be the best method. Adjusted for inclusion of damping and a
varying input forces.

    X, V, E = np.zeros(N), np.zeros(N), np.zeros(N-1) #empty arrays are created for
position, velocity, energy

    X[0] = x0 #initial position and velocity are set into the arrays
    V[0] = v0


    X[1] = X[0] + h_step * V[0] #finding the first step using Euler method, then
```

```
continuing the iteration through with the Verlet method

    D = 2*m + b_input * h_step #calculating D,A,B Verlet coefficients.

    A = 2*(2*m - k * (h_step**2) )/ D

    B = (b_input*h_step - 2*m) / D

    for i in range(2,N):

        X[i] = (A * X[i-1] + B * X[i-2] + 2*(h_step**2)*F[i-1]/D)  # x2 acts as the
next step, x1 acts as current step, x0 acts as previous step.

        V[i-1] = (X[i] - X[i-2])/(2*h_step)

    for i in range(0,N-1):
        E[i] =(m*V[i]**2/2 + k*X[i]**2/2)

    X = X[0:-1]
    V = V[0:-1]

    writeFile(file_name , t, X, V, E)

    return(None) #end of function

def eulerCromer(file_name, x0, v0, h_step):
#Euler- Cromer method function
#Uses the Euler-Cromer method to calculate solutions step-by-by step.
#The Euler-Cromer method is symplectic. Energy of the oscillator is preserved.
#Identical to the standard Euler method, except the position is calculated by using
the velocity at the next step.

    X, V, E = np.zeros(N), np.zeros(N), np.zeros(N-1) #empty arrays are created for
position, velocity, energy

    X[0] = x0 #initial position and velocity are set into the arrays
    V[0] = v0

    for i in range(1, N):
        #calculate solutions step-by-step until the time limit is reached
        V[i] = V[i-1] - ((b/m)*V[i-1]   + (k /m)*X[i-1] )* h_step
        #V_n = V_n-1 + a_n-1 * h
        X[i] = X[i-1] + V[i] * h_step
        #X_n = X_n-1 + V_n

    for i in range(0,N-1):
        #calculate total  at each point manually from kinetic energy + potential
energy
        E[i] =(m*V[i]**2/2 + k*X[i]**2/2)

    X = X[0:-1]
    V = V[0:-1]

    writeFile(file_name , t, X, V, E)

    return(None) #end of function

def analytical(file_name, x0, v0, t, h):
#analytically calculating given initial conditions. Works for light damping where
```

```
omega is real.
#Useful website:
http://www.casaxps.com/help_manual/mathematics/Mechanics3_rev12.pdf

#Coefficients A, B calculated from initial conditions x0, v0.

    A = x0/2 - 1j *  (v0 + x0*gamma/2)/( 2 * omega)

    B = x0 - A

    X, V, E = np.zeros(N-1), np.zeros(N-1), np.zeros(N-1)
    #It is more memory efficient to create an array of size N than to append to an
empty array.
    #Length N-1 so number of data points is the same as found from the Verlet
method.

    for i in range(0, N-1):

        X[i] = np.exp(-gamma*t[i]/2)*( A * np.exp(1j * omega* t[i]) + B * np.exp(-
1j*omega*t[i]) )
        #Calculating position at each point given the analytical solution X(t).
        V[i] = (1j*omega*(np.exp(-gamma*t[i]/2)) * (A * np.exp(1j * omega* t[i]) -
B * np.exp(-1j*omega*t[i]) ) - gamma*X[i]/2)
        #Calculating velocity at each point using the derivative of X(t) found
analytically.

    for i in range(0,N-1):
        #Calculating the total energy of the system at each point from kinetic and
potential energies summed.
        E[i] =(m*V[i]**2/2 + k*X[i]**2/2)

    writeFile(file_name , t, X, V, E)

    return(None)


def ComparisonOfMethods(Title, xAxis,yAxis, X, X1, X2, X3, X4, Y, Y1, Y2, Y3, Y4):
    #Graph plotter function
    #Plots data sets together

    if (X != "none"):
        plt.plot(X, Y, color = "g", label = "Analytical")
    plt.plot(X1, Y1, color = "b", label = "Euler")
    plt.plot(X2, Y2, color = "c", label = "Improved Euler")
    plt.plot(X3, Y3, color = "r", label = "Verlet")
    plt.plot(X4, Y4, color = "k", label = "Euler-Cromer")

    plt.legend(loc = "best")
    plt.title(Title)
    plt.xlabel(xAxis)
    plt.ylabel(yAxis)

    plt.show()

    #The methods can be compared by considering the total energy of the system. The
better methods will be closer to the real solutions calculated analytically.

def EnergyAreaDifference(Energy_method, Energy_analytical, time, h):
    #The ratio of the integrals of numerically calculated energies and the
```

```
analytical with time can be used as a measure of how accurate a calulation method
and step size are.

    Area_method = np.trapz(Energy_method, t[0:len(Energy_method)], h)
    #integral = np.trapz(y(x), x , step_size)
    #The integrals from the trapezium rule will be overestimates when the integrand
is increasing at a changing rate (concave up), and underestimates when the
integrand is concave down.
    #This error is reduced by using smaller step sizes.
    Area_analytical = np.trapz(Energy_analytical, t[0:len(Energy_analytical)], h)
    Area_diff = Area_method - Area_analytical

    return abs(Area_diff)

def energydifference(Energy_method, Energy_analytical):
    #The energy difference between solutions after some time for undamped
simulations could be used to show how well the methods work.

    delta = np.zeros(len(Energy_method))

    for i in range(0, len(Energy_method)):
        delta[i] = ((Energy_method[i]) - Energy_analytical[i] /
Energy_analytical[i] )

    return delta


#main

#calling functions, then plotting position-time graphs for comparison
#no external force

t, N = time(time_tot, h)
Fnone = np.zeros(N)

analytical("fileanal.txt", x0, v0, t, h) #
Xanal, Vanal, Eanal = readFile("fileanal.txt")
t = t[0:-1]

euler("fileeuler.txt", x0, v0, h, N)
Xeuler, Veuler, Eeuler = readFile("fileeuler.txt")

improvedEuler("fileimprovedeuler.txt", x0, v0, h)
XimprovedEuler, VimprovedEuler, EimprovedEuler = readFile("fileimprovedeuler.txt")

verlet("fileverlet.txt", x0,v0, h, b, Fnone )
Xverlet, Vverlet, Everlet = readFile("fileverlet.txt")

eulerCromer("fileeulercromer.txt", x0, v0, h)
XeulCrom, VeulCrom, EeulCrom = readFile("fileeulercromer.txt")


ComparisonOfMethods("Displacement of four models against time", "Time,
s","Displacement, m", t, t, t, t, t, Xanal, Xeuler, XimprovedEuler, Xverlet,
XeulCrom)

ComparisonOfMethods("Phase space of four models", "Displacement, m","Velocity,
m/s", Xanal, Xeuler, XimprovedEuler, Xverlet, XeulCrom, Vanal, Veuler,
VimprovedEuler, Vverlet, VeulCrom)
```

```
plt.yscale("log")
ComparisonOfMethods("Calculated energy of four models against time", "Time,
s","Energy, J", t, t, t, t, t, Eanal, Eeuler, EimprovedEuler, Everlet, EeulCrom)
"""
#Efficiency with varying step size.
M = 3
H, effEuler, effimprovedEuler, effverlet, effeulCrom = np.zeros(M), np.zeros(M),
np.zeros(M), np.zeros(M) , np.zeros(M)

for i in range(0, (M-1)):
    H[i] = (h* 0.1**(i/4))
    t, N = time(time_tot, H[i])

    euler("fileeuler_%d.txt"%(i,), x0, v0, H[i], N)
    improvedEuler("fileeuler_%d.txt"%(i,), x0, v0, H[i])
    verlet("fileeuler_%d.txt"%(i,), x0,v0, H[i], b, Fnone)
    eulerCromer("fileeuler_%d.txt"%(i,), x0, v0, H[i])

    Xeuler, Veuler, Eeuler = readFile("fileeuler_%d.txt"%(i,))
    XimprovedEuler, VimprovedEuler, EimprovedEuler = readFile("fileimprovedeuler_
%d.txt"%(i,))
    Xverlet, Vverlet, Everlet = readFile("fileverlet_%d.txt"%(i,))
    XeulCrom, VeulCrom, EeulCrom = readFile("fileeulercromer_%d.txt"%(i,))

    Deuler =energydifference(Eeuler, Eanal)
    DimprovedEuler =energydifference(EimprovedEuler, Eanal)
    Dverlet =energydifference(Everlet, Eanal)
    DeulCrom =energydifference(EeulCrom, Eanal)

    effEuler[i] = max(Deuler)
    effimprovedEuler[i] = max(DimprovedEuler)
    effverlet[i] = max(Dverlet)
    effeulCrom[i] = max(DeulCrom)


plt.scatter(H, effEuler, color = "b", label = "Euler")
plt.scatter(H, effimprovedEuler, color = "c", label = "Improved Euler")
plt.scatter(H, effverlet, color = "r", label = "Verlet")
plt.scatter(H, effeulCrom, color = "k", label = "Euler-Cromer")

plt.title("Efficiency of models against step size")
plt.xlabel("Step size")
plt.ylabel("Efficiency")
plt.legend(loc = "best")
#energy accuracy
#calculations are more accurate if they are close to the analytical

"""
h = 0.01
t, N = time(time_tot, h)

analytical("fileanal.txt", x0, v0, t, h) #
Xanal, Vanal, Eanal = readFile("fileanal.txt")

t = t[0:-1]

#Using Verlet method for b = b_crit/2 (1), b_crit (2) and 2*b_crit (3)
Fnone = np.zeros(N)
```

```
#Plotting verlet for different damping coefficients

verlet("verletdamped1.txt", x0, v0, h, b_crit/2, Fnone)
Xverlet1, Vverlet1, Everlet1 = readFile("verletdamped1.txt")


verlet("verletdamped2.txt", x0,v0, h, b_crit, Fnone)
Xverlet2, Vverlet2, Everlet2 = readFile("verletdamped2.txt")
#


verlet("verletdamped3.txt", x0, v0, h, 2*b_crit, Fnone)
Xverlet3, Vverlet3, Everlet3 = readFile("verletdamped3.txt")

plt.plot(t, Xverlet1, label = "Verlet")

plt.title("Verlet damping, with b_crit/2")
plt.xlabel("Time, s")
plt.ylabel("Position, m")
plt.show()
#More than one complete oscillation.

plt.plot( t, Xverlet2, label = "Verlet")
plt.title("Verlet critical damping")
plt.xlabel("Time, s")
plt.ylabel("Position, m")
plt.show()

#One half complete oscillation, with the minimum disturbance time.

plt.plot( t, Xverlet3, label = "Verlet")
plt.title("Verlet damping, with 2*b_crit")
plt.xlabel("Time, s")
plt.ylabel("Position, m")
plt.show()
#One half complete oscillation, but the disturbance time is longer than at the
critical damping.
"""

#sudden application of external force for damping coefficient b

Fext = np.zeros(N)
for i in range((int(N/4)), int(3*N/4)):
    Fext[i] = 70


verlet("verletforced1.txt", x0, v0, h, b, Fext)
Xverlet4, Vverlet4, Everlet4 = readFile("verletforced1.txt")


plt.plot(t, Xverlet4)

plt.xlabel("t, s")
plt.ylabel("Position, m")
plt.show()

Fext = np.zeros(N)

for i in range((int(N/4)), int(N-1)):
```

```
    Fext[i] = np.sin(omega * t[i]) #resonant frequency

verlet("verletforced1.txt", x0, v0, h, b, Fext)
Xverlet5, Vverlet5, Everlet5 = readFile("verletforced2.txt")


plt.plot(t, Xverlet5)

plt.xlabel("t, s")
plt.ylabel("Position, m")
plt.show()

for i in range((int(N/4)), int(N-1)):
    Fext[i] = np.sin(2* omega * t[i]) #double the resonant frequency

Xverlet4, Vverlet4, Everlet4 = verlet(x0,v0, b, Fext)


plt.plot(t, Xverlet4)
plt.xlabel("t, s")
plt.ylabel("Position, m")
plt.show()

for i in range((int(N/4)), int(N-1)):
    Fext[i] = np.sin(t[i]) #some rother frequency


plt.plot(t, Xverlet4)

plt.xlabel("t, s")
plt.ylabel("Position, m")
plt.show()

"""
```