Note: I’ve been having issues with the Spyder software on my laptop. Whenever I save a program and attempt to open it through my files, it does not allow me to view it, so I can’t access a file once I close the file window on Spyder. I have provided screenshots for my code in this file as well as text forms in a separate file so that you can copy and paste if needed. I’m really sorry if this makes it harder for marking, but I haven’t been able to fix it.

I have submitted my plots in separate files. Each file name is the corresponding question’s number.

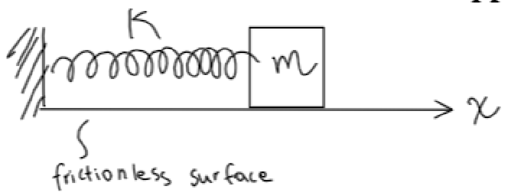
PHY254 - Computational Assignment - Idil Yaktubay - 1005679240

December 15, 2020 11:35 AM

1. **Solving oscillations numerically**

- Mass *m* attached to a spring constant *k*
- Undamped simple harmonic oscillator that conserves energy

(a) Write a program in python to solve this system by replacing the time derivatives with a Forward Euler finite difference approximation.



Eqn of motion of the mass

$$m\ddot{x} = -kx$$
$$\Rightarrow m\ddot{x} + kx = 0$$
$$\Rightarrow \ddot{x} + \frac{k}{m}x = 0$$
$$\text{let } \omega^2 = \frac{k}{m} \Rightarrow \boxed{\ddot{x} + \omega^2 x = 0}$$

- Now applying the Forward Euler method:

$$\dot{x} = \frac{dx}{dt} = \frac{x(t+\Delta t) - x(t)}{\Delta t} \Rightarrow \dot{x}\Delta t + x(t) = x(t+\Delta t)$$
$$\rightarrow x(t+\Delta t) = \dot{x}\Delta t + x(t)$$
$$\Rightarrow x(t+\Delta t) = v\Delta t + x(t)$$
$$\Rightarrow \boxed{x_{i+1} = x_i + v_i \Delta t}$$

• we know that $\frac{dv}{dt} = -\omega^2 x$

$$\Rightarrow \frac{dv}{dt} = \frac{v(t+\Delta t) - v(t)}{\Delta t} \Rightarrow a\Delta t + v(t) = v(t+\Delta t)$$
$$\Rightarrow v(t+\Delta t) = v(t) + a\Delta t$$
$$\Rightarrow \boxed{v_{i+1} = v_i - \omega^2 x_i \Delta t}$$

• We can also write these as

$$\boxed{x_i = x_{i-1} + (v_{i-1})\Delta t} \quad \boxed{v_i = v_{i-1} - (\omega^2)(x_{i-1})\Delta t}$$

which I will be using on the code.

(b) Solve this system analytically and plot the numerical and analytic solution for the position x, the velocity v = xdot and the total energy E against time, for the same time axis. Vary dt and see what happens. What are the main differences between the analytic and numerical solutions?

Analytic solution calculations:

- In the code, I assigned $x_0 = 1.0\,m$ and $v_0 = 0.0\,ms^{-1}$ for my initial conditions to keep things simple.
- Remembering that the equation of motion is $\ddot{x} + \omega^2 x = 0$, we guess a solution in the form of

$$x(t) = C_1\cos(\omega t) + C_2\sin(\omega t)$$
$$\Rightarrow \dot{x}(t) = v(t) = -C_1\omega\sin(\omega t) + C_2\omega\cos(\omega t)$$

Applying initial conditions, we solve for C_1 and C_2:

$$x(0) = x_0 = 1.0 = C_1\cos(0) + C_2\sin(0) \Rightarrow \boxed{1 = C_1}$$

$$v(0) = v_0 = 0 = -C_1\omega\sin(0) + C_2\omega\cos(0) \Rightarrow \boxed{0 = C_2}$$

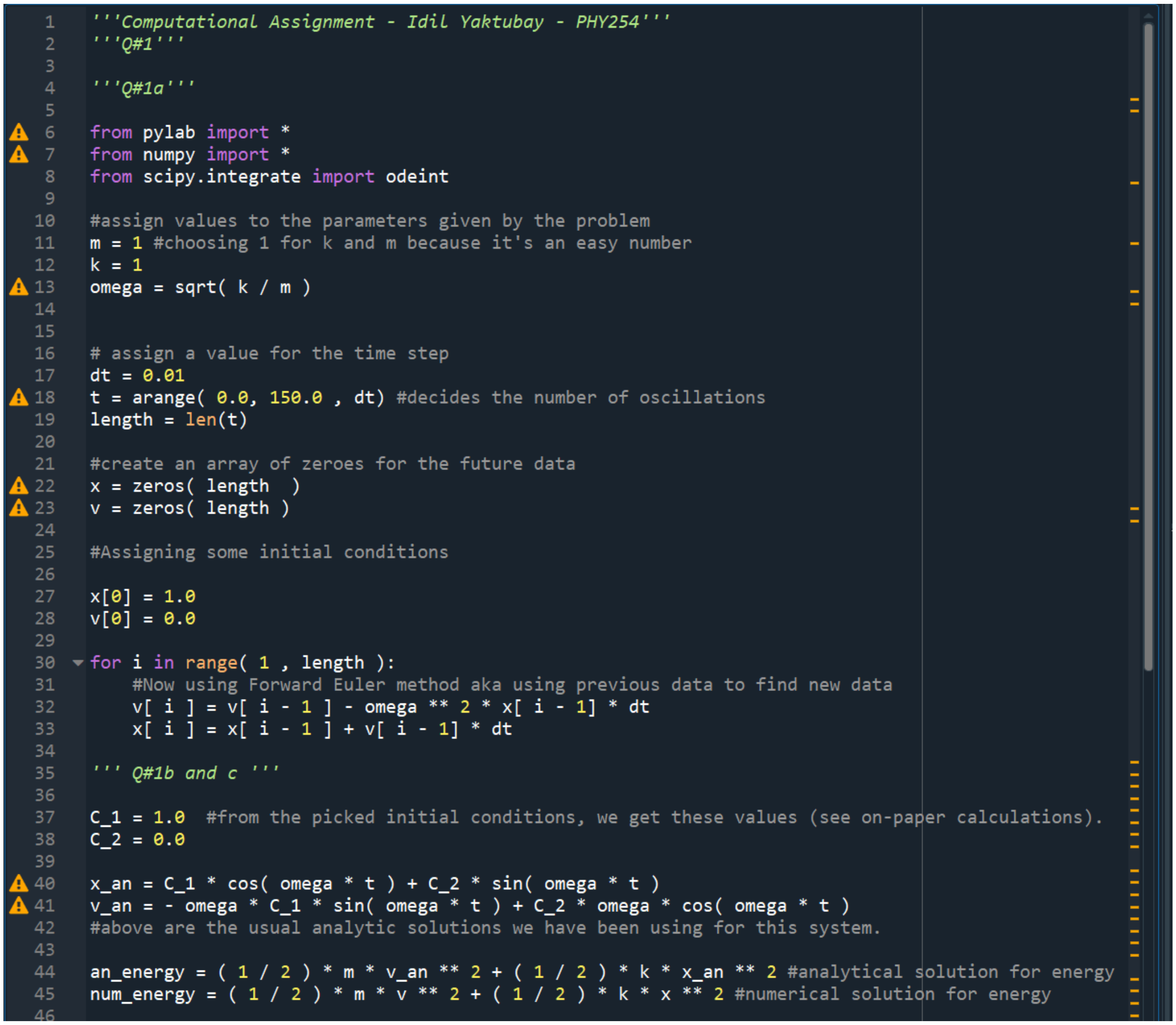
- Therefore, I assigned $C_1=1$ and $C_2=0$ in my code for the analytic solution aswell.

- I have solved the system analytically above, and plotted the numerical and analytical solution for x, v and E as functions of t for the same axis. The plots that I'm submitting are for the following values of dt: 0.01, 0.05, 0.1. I picked these values because we can clearly tell the differences.
- As I change dt (i.e., increase it) the numerical solution becomes less and less good of an approximation for x and v as we have expected. For example, for dt = 0.01, the numerical solution is more or less the same as the analytic solution for about 8 oscillations. For dt = 0.05, the same is true for about 3 oscillations, and for dt = 0.1, this is only true for about one oscillation. In summary, as I increase dt, the numerical solutions for x and v diverge from the analytic solutions a lot faster, because increasing dt makes the approximation less accurate. However, the numerical solution for energy gets more accurate as I increase dt.
- The main difference between the analytical and numerical solutions for x and v is that as time t increases, the amplitudes of the analytic solutions stay the same whereas the numerical solution increases in amplitude much like a driven undamped oscillator. In terms of energy, the analytic solution stays constant, which makes sense because we aren't adding energy to the system, but the numerical solution diverges to infinity.

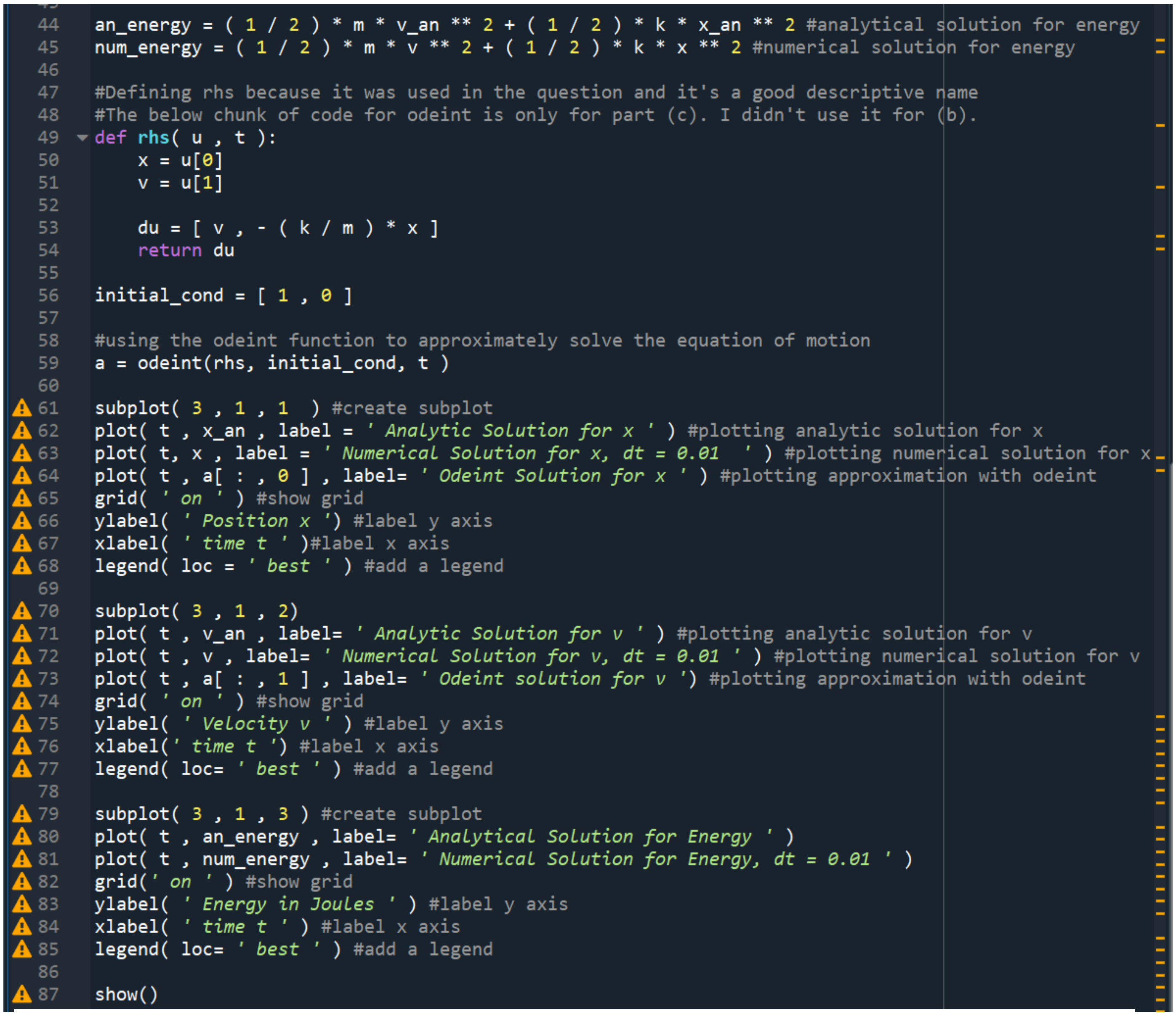
(c) Now repeat part (b), except plot the output of the odeint function in addition to the analytic solution and the Forward Euler time stepping scheme. How do the results differ? Which do you think is a better numerical approximation?

- I have included a plot of my Forward Euler, analytic, and odeint solutions all for the same axis for dt = 0.01.
- The results differ in terms of accuracy. The solution I have obtained from odeint is much more accurate than the Forward Euler solution as it seems like it is exactly on top of my analytic solution. In my plot, we can see ~23 oscillations, and the odeint solution doesn't diverge from the analytic solution for all 23 oscillations, whereas the Forward Euler solution diverges after ~8 oscillations.
- I think the odeint function's approximation is a much better numerical approximation than Forward Euler because of reasons I listed above.

Next pages include my code for the entirety of question 1.



```python
'''Computational Assignment - Idil Yaktubay - PHY254'''
'''Q#1'''

'''Q#1a'''

from pylab import *
from numpy import *
from scipy.integrate import odeint

#assign values to the parameters given by the problem
m = 1 #choosing 1 for k and m because it's an easy number
k = 1
omega = sqrt( k / m )


# assign a value for the time step
dt = 0.01
t = arange( 0.0, 150.0 , dt) #decides the number of oscillations
length = len(t)

#create an array of zeroes for the future data
x = zeros( length  )
v = zeros( length )

#Assigning some initial conditions

x[0] = 1.0
v[0] = 0.0

for i in range( 1 , length ):
    #Now using Forward Euler method aka using previous data to find new data
    v[ i ] = v[ i - 1 ] - omega ** 2 * x[ i - 1] * dt
    x[ i ] = x[ i - 1 ] + v[ i - 1] * dt

''' Q#1b and c '''

C_1 = 1.0  #from the picked initial conditions, we get these values (see on-paper calculations).
C_2 = 0.0

x_an = C_1 * cos( omega * t ) + C_2 * sin( omega * t )
v_an = - omega * C_1 * sin( omega * t ) + C_2 * omega * cos( omega * t )
#above are the usual analytic solutions we have been using for this system.

an_energy = ( 1 / 2 ) * m * v_an ** 2 + ( 1 / 2 ) * k * x_an ** 2 #analytical solution for energy
num_energy = ( 1 / 2 ) * m * v ** 2 + ( 1 / 2 ) * k * x ** 2 #numerical solution for energy
```



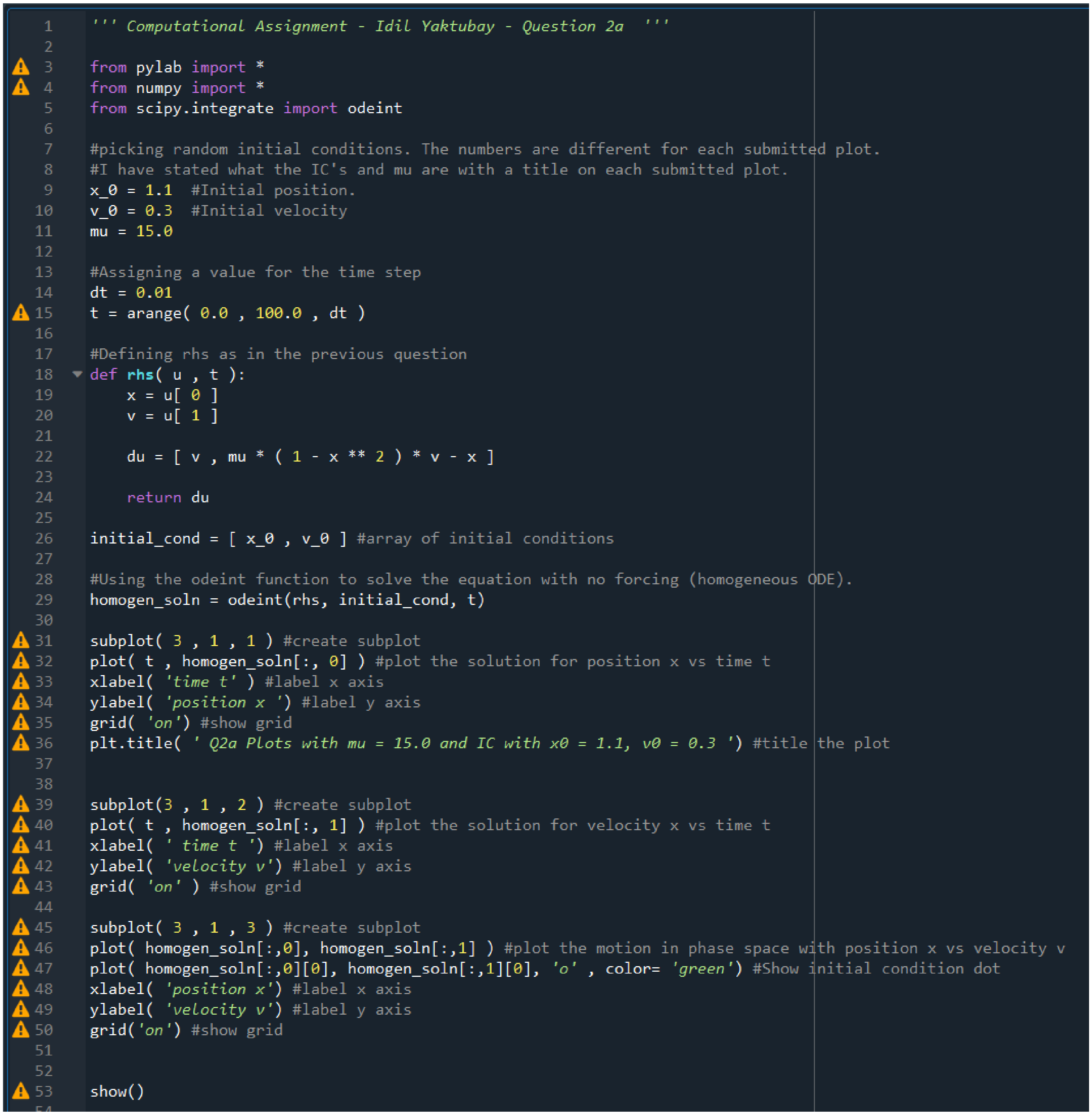
```
an_energy = ( 1 / 2 ) * m * v_an ** 2 + ( 1 / 2 ) * k * x_an ** 2 #analytical solution for energy
num_energy = ( 1 / 2 ) * m * v ** 2 + ( 1 / 2 ) * k * x ** 2 #numerical solution for energy

#Defining rhs because it was used in the question and it's a good descriptive name
#The below chunk of code for odeint is only for part (c). I didn't use it for (b).
def rhs( u , t ):
    x = u[0]
    v = u[1]

    du = [ v , - ( k / m ) * x ]
    return du

initial_cond = [ 1 , 0 ]

#using the odeint function to approximately solve the equation of motion
a = odeint(rhs, initial_cond, t )

subplot( 3 , 1 , 1  ) #create subplot
plot( t , x_an , label = ' Analytic Solution for x ' ) #plotting analytic solution for x
plot( t, x , label = ' Numerical Solution for x, dt = 0.01  ' ) #plotting numerical solution for x
plot( t , a[ : , 0 ] , label= ' Odeint Solution for x ' ) #plotting approximation with odeint
grid( ' on ' ) #show grid
ylabel( ' Position x ') #label y axis
xlabel( ' time t ' )#label x axis
legend( loc = ' best ' ) #add a legend

subplot( 3 , 1 , 2)
plot( t , v_an , label= ' Analytic Solution for v ' ) #plotting analytic solution for v
plot( t , v , label= ' Numerical Solution for v, dt = 0.01 ' ) #plotting numerical solution for v
plot( t , a[ : , 1 ] , label= ' Odeint solution for v ') #plotting approximation with odeint
grid( ' on ' ) #show grid
ylabel( ' Velocity v ' ) #label y axis
xlabel(' time t ') #label x axis
legend( loc= ' best ' ) #add a legend

subplot( 3 , 1 , 3 ) #create subplot
plot( t , an_energy , label= ' Analytical Solution for Energy ' )
plot( t , num_energy , label= ' Numerical Solution for Energy, dt = 0.01 ' )
grid(' on ' ) #show grid
ylabel( ' Energy in Joules ' ) #label y axis
xlabel( ' time t ' ) #label x axis
legend( loc= ' best ' ) #add a legend

show()
```

2. The Van Der Pol oscillator.

(a) The unforced van der Pol oscillator.

- I have submitted plots for the following values of mu: 0.1, 1.0, 4.2, 9.2, 15.0. Both my initial conditions and values of mu are listed in the title of each plot.
- For each plot, the corresponding program is exactly the same except I assigned the corresponding values of mu, x_0, v_0 for each one and changed the title.
- From my results, it seems like no matter what my initial conditions and values of mu are, the motion approaches a limit cycle for later times, which means that the motion repeats itself.
- I don't think that this system can be chaotic for any value of mu, because we know that the solution to this equation with no forcing will approach a limit cycle for every value of mu. Even though in theory, chaotic behavior is deterministic, it still does not repeat itself and hence cannot have a limit cycle. This system could only become chaotic if there was forcing involved.

Here is my code for 2(a):

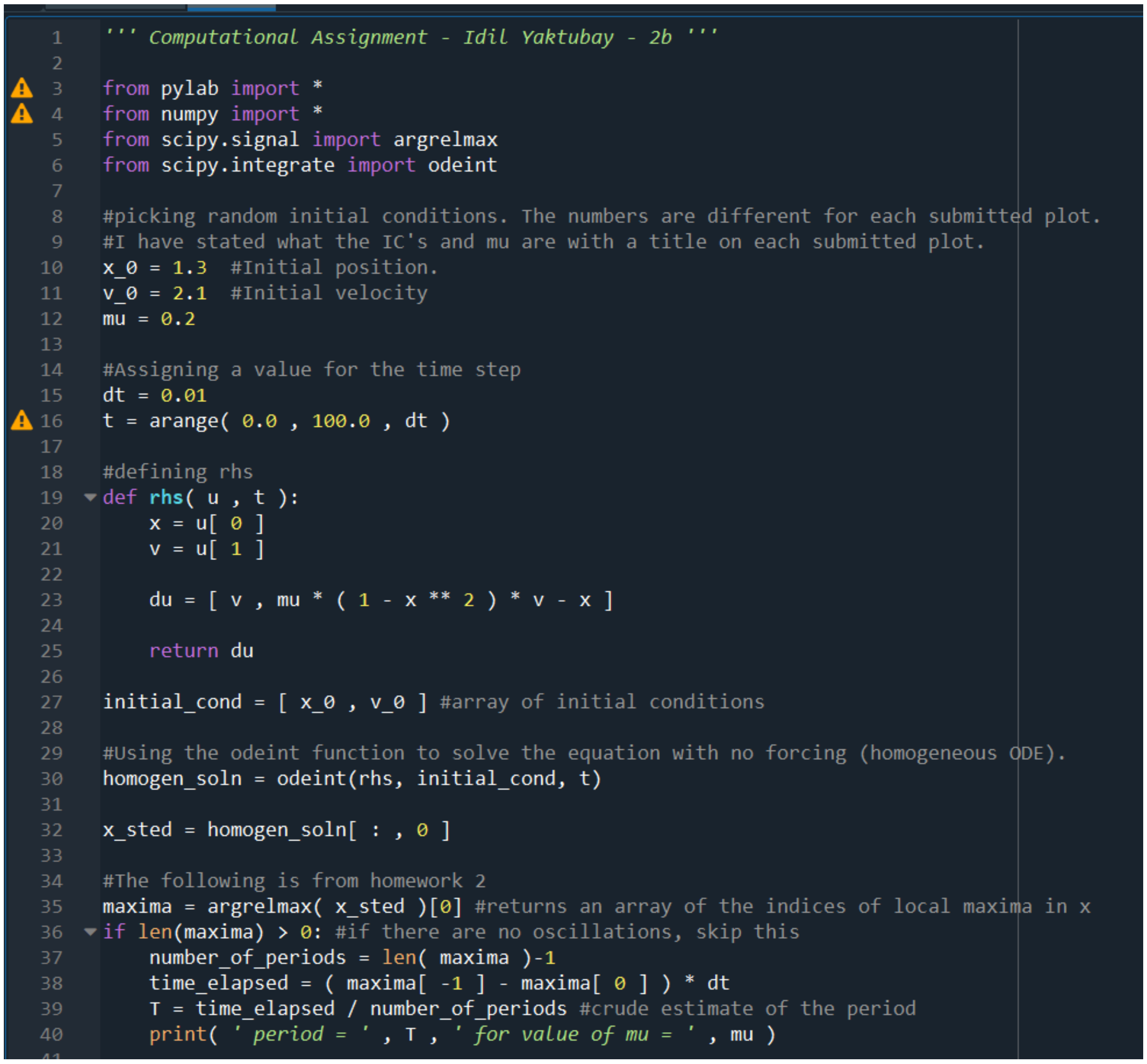


```python
''' Computational Assignment - Idil Yaktubay - Question 2a  '''

from pylab import *
from numpy import *
from scipy.integrate import odeint

#picking random initial conditions. The numbers are different for each submitted plot.
#I have stated what the IC's and mu are with a title on each submitted plot.
x_0 = 1.1  #Initial position.
v_0 = 0.3  #Initial velocity
mu = 15.0

#Assigning a value for the time step
dt = 0.01
t = arange( 0.0 , 100.0 , dt )

#Defining rhs as in the previous question
def rhs( u , t ):
    x = u[ 0 ]
    v = u[ 1 ]

    du = [ v , mu * ( 1 - x ** 2 ) * v - x ]

    return du

initial_cond = [ x_0 , v_0 ] #array of initial conditions

#Using the odeint function to solve the equation with no forcing (homogeneous ODE).
homogen_soln = odeint(rhs, initial_cond, t)

subplot( 3 , 1 , 1 ) #create subplot
plot( t , homogen_soln[:, 0] ) #plot the solution for position x vs time t
xlabel( 'time t' ) #label x axis
ylabel( 'position x ') #label y axis
grid( 'on') #show grid
plt.title( ' Q2a Plots with mu = 15.0 and IC with x0 = 1.1, v0 = 0.3 ') #title the plot


subplot(3 , 1 , 2 ) #create subplot
plot( t , homogen_soln[:, 1] ) #plot the solution for velocity x vs time t
xlabel( ' time t ') #label x axis
ylabel( 'velocity v') #label y axis
grid( 'on' ) #show grid

subplot( 3 , 1 , 3 ) #create subplot
plot( homogen_soln[:,0], homogen_soln[:,1] ) #plot the motion in phase space with position x vs velocity v
plot( homogen_soln[:,0][0], homogen_soln[:,1][0], 'o' , color= 'green') #Show initial condition dot
xlabel( 'position x') #label x axis
ylabel( 'velocity v') #label y axis
grid('on') #show grid


show()
```

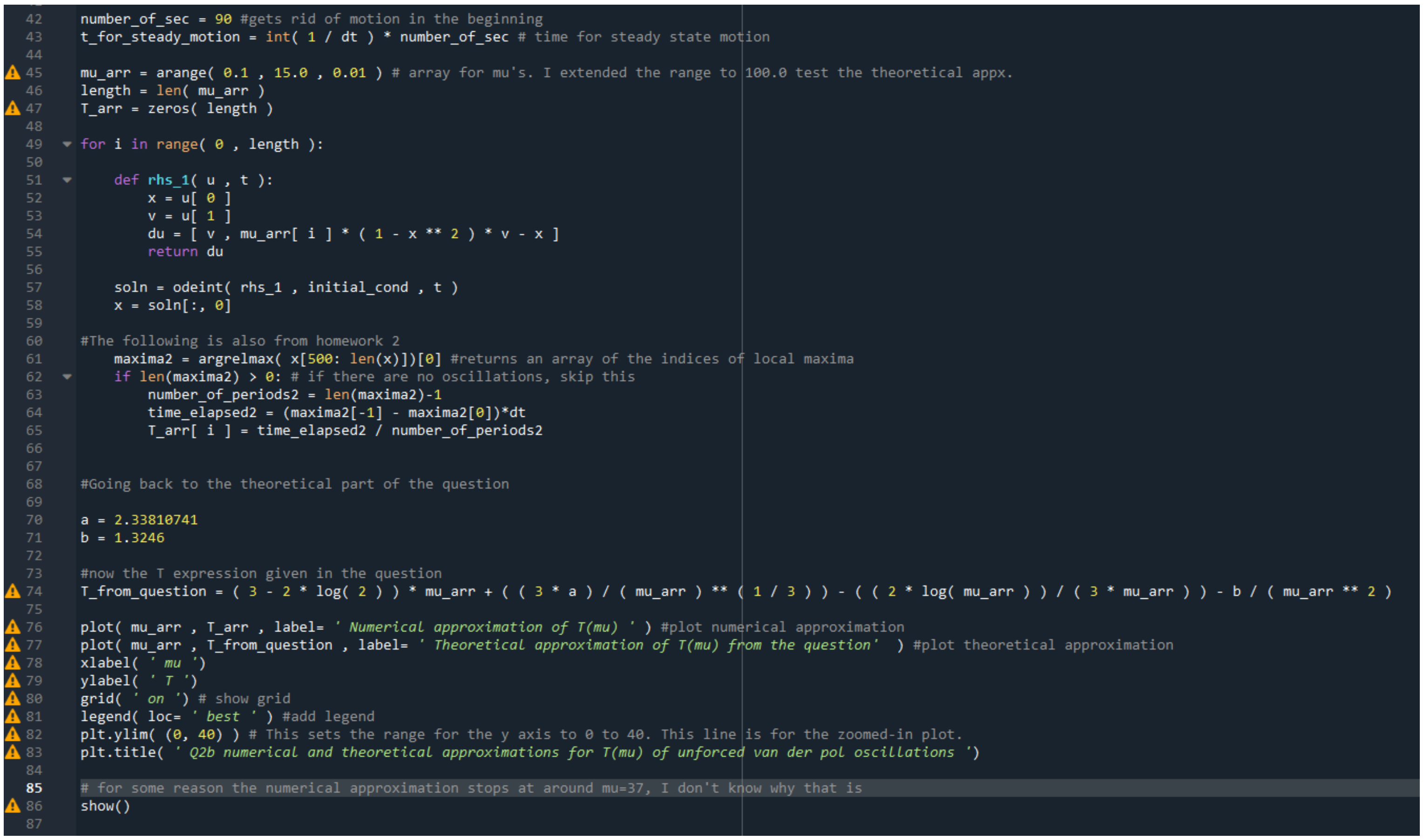
(b) Devise a numerical means of estimating the period of the unforced oscillator for each mu. Make a plot of the numerical period T vs. Mu for (0.1, 15.0) by adding a loop over mu to your code from part (a).

- I have added a loop over mu to my code from part a. I am submitting my plots for mu values in the range (0.1, 15.0), and also (0.1, 100.0) to test the theoretical approximation. I used the tricks from Homework 2 as well. I have also included a zoomed-in plot with the range of the y axis set to (0, 40).
- From my plot that goes till mu ~100, we can see that it is true that T is proportional to mu as mu increases because the plot looks like a straight line for large mu.
- Keeping in mind that eq4 only works for large mu, we can see that my numerical approximation and the theoretical approximation do agree for large mu. As mu increases, the two approximations seem to entirely overlap. From my zoomed-in plot, we clearly see that the theoretical approximation does not agree with the numerical approximation for small values of mu, which was expected.

Here is my code for 2b:



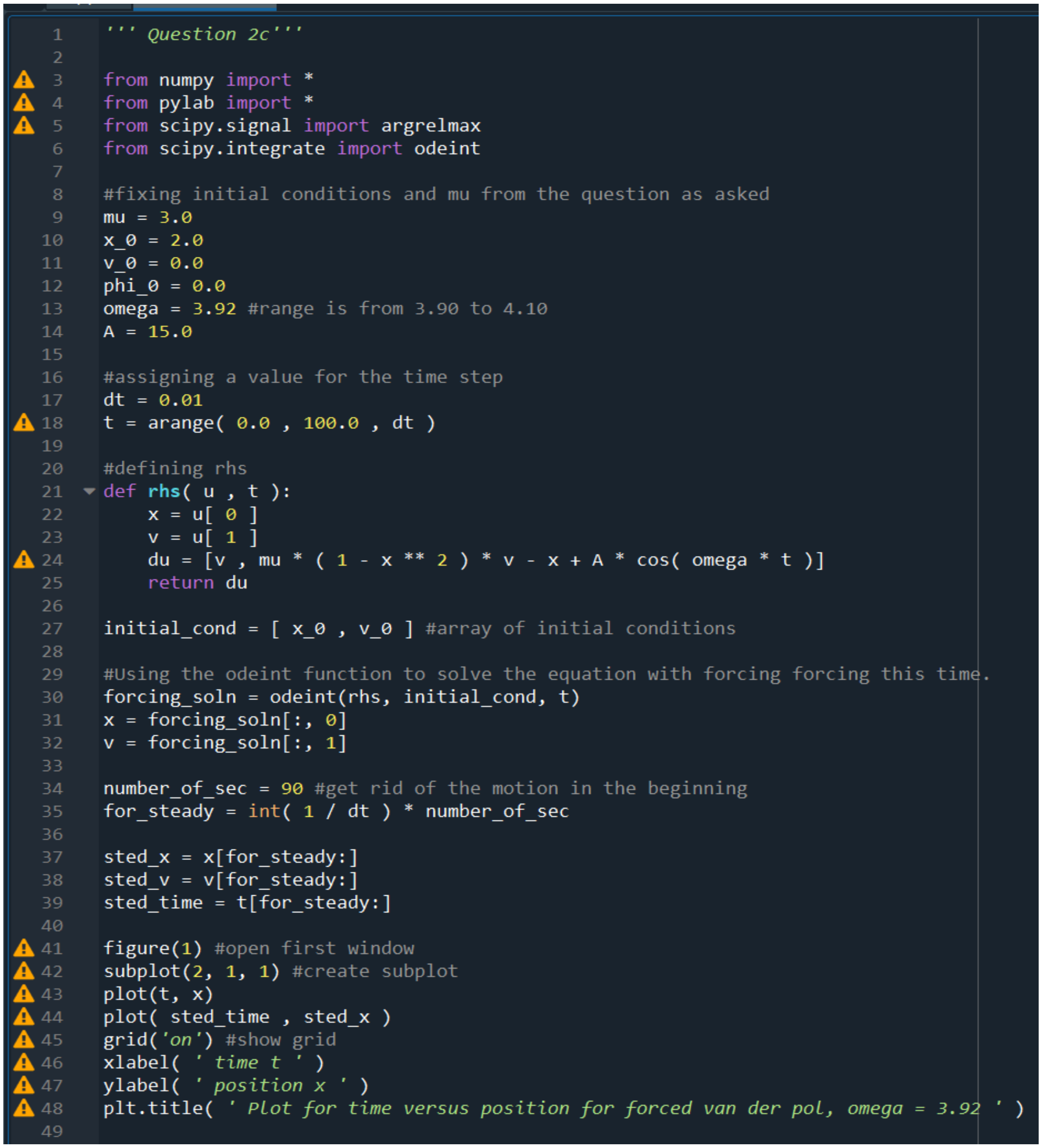
```
''' Computational Assignment - Idil Yaktubay - 2b '''

from pylab import *
from numpy import *
from scipy.signal import argrelmax
from scipy.integrate import odeint

#picking random initial conditions. The numbers are different for each submitted plot.
#I have stated what the IC's and mu are with a title on each submitted plot.
x_0 = 1.3  #Initial position.
v_0 = 2.1  #Initial velocity
mu = 0.2

#Assigning a value for the time step
dt = 0.01
t = arange( 0.0 , 100.0 , dt )

#defining rhs
def rhs( u , t ):
    x = u[ 0 ]
    v = u[ 1 ]

    du = [ v , mu * ( 1 - x ** 2 ) * v - x ]

    return du

initial_cond = [ x_0 , v_0 ] #array of initial conditions

#Using the odeint function to solve the equation with no forcing (homogeneous ODE).
homogen_soln = odeint(rhs, initial_cond, t)

x_sted = homogen_soln[ : , 0 ]

#The following is from homework 2
maxima = argrelmax( x_sted )[0] #returns an array of the indices of local maxima in x
if len(maxima) > 0: #if there are no oscillations, skip this
    number_of_periods = len( maxima )-1
    time_elapsed = ( maxima[ -1 ] - maxima[ 0 ] ) * dt
    T = time_elapsed / number_of_periods #crude estimate of the period
    print( ' period = ' , T , ' for value of mu = ' , mu )
```



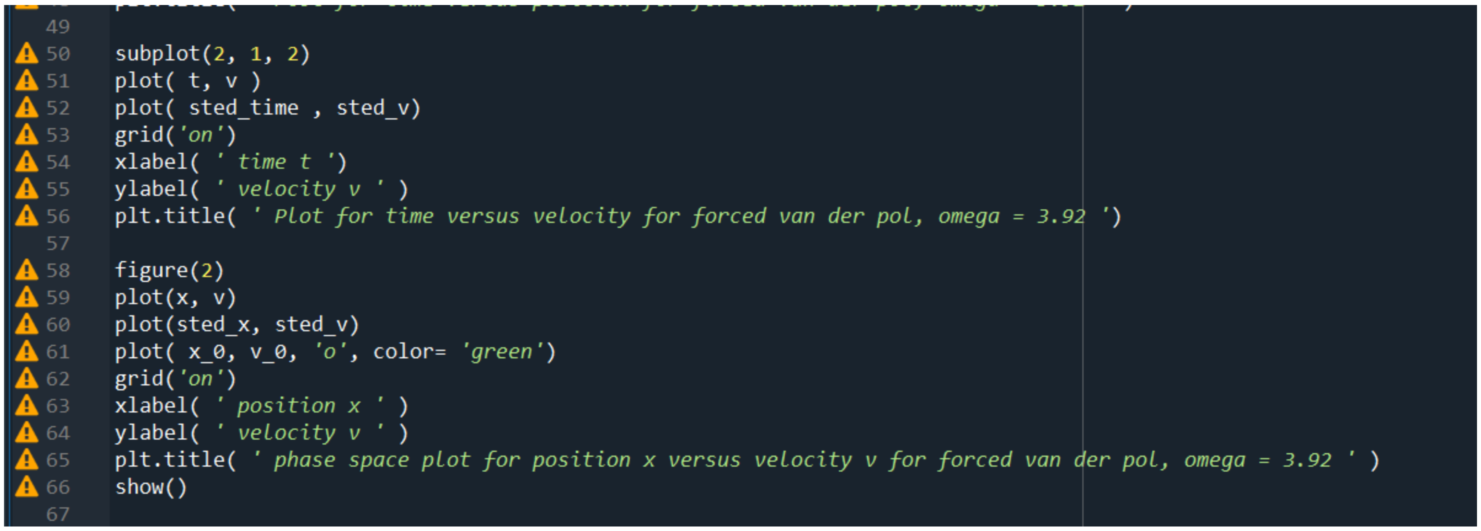
```python
number_of_sec = 90 #gets rid of motion in the beginning
t_for_steady_motion = int( 1 / dt ) * number_of_sec # time for steady state motion

mu_arr = arange( 0.1 , 15.0 , 0.01 ) # array for mu's. I extended the range to 100.0 test the theoretical appx.
length = len( mu_arr )
T_arr = zeros( length )

for i in range( 0 , length ):

    def rhs_1( u , t ):
        x = u[ 0 ]
        v = u[ 1 ]
        du = [ v , mu_arr[ i ] * ( 1 - x ** 2 ) * v - x ]
        return du

    soln = odeint( rhs_1 , initial_cond , t )
    x = soln[:, 0]

#The following is also from homework 2
    maxima2 = argrelmax( x[500: len(x)])[0] #returns an array of the indices of local maxima
    if len(maxima2) > 0: # if there are no oscillations, skip this
        number_of_periods2 = len(maxima2)-1
        time_elapsed2 = (maxima2[-1] - maxima2[0])*dt
        T_arr[ i ] = time_elapsed2 / number_of_periods2


#Going back to the theoretical part of the question

a = 2.33810741
b = 1.3246

#now the T expression given in the question
T_from_question = ( 3 - 2 * log( 2 ) ) * mu_arr + ( ( 3 * a ) / ( mu_arr ) ** ( 1 / 3 ) ) - ( ( 2 * log( mu_arr ) ) / ( 3 * mu_arr ) ) - b / ( mu_arr ** 2 )

plot( mu_arr , T_arr , label= ' Numerical approximation of T(mu) ' ) #plot numerical approximation
plot( mu_arr , T_from_question , label= ' Theoretical approximation of T(mu) from the question'  ) #plot theoretical approximation
xlabel( ' mu ')
ylabel( ' T ')
grid( ' on ') # show grid
legend( loc= ' best ' ) #add legend
plt.ylim( (0, 40) ) # This sets the range for the y axis to 0 to 40. This line is for the zoomed-in plot.
plt.title( ' Q2b numerical and theoretical approximations for T(mu) of unforced van der pol oscillations ')

# for some reason the numerical approximation stops at around mu=37, I don't know why that is
show()

```

(c) Use your generalized code to visually hunt for periodic or chaotic motions by plotting v and x vs t, and the (x, v) projection of the 3D phase space, varying only omega in the above range.

- I am submitting plots for position x and velocity v versus time t, and the projection of the 3D space with the following values of omega: 3.92, 3.96, 4.09. My plot for omega = 3.96 looks the same as the one shown in figure 1. I have labelled each one of my plots with these values in their title.
- My code for part c is the following.



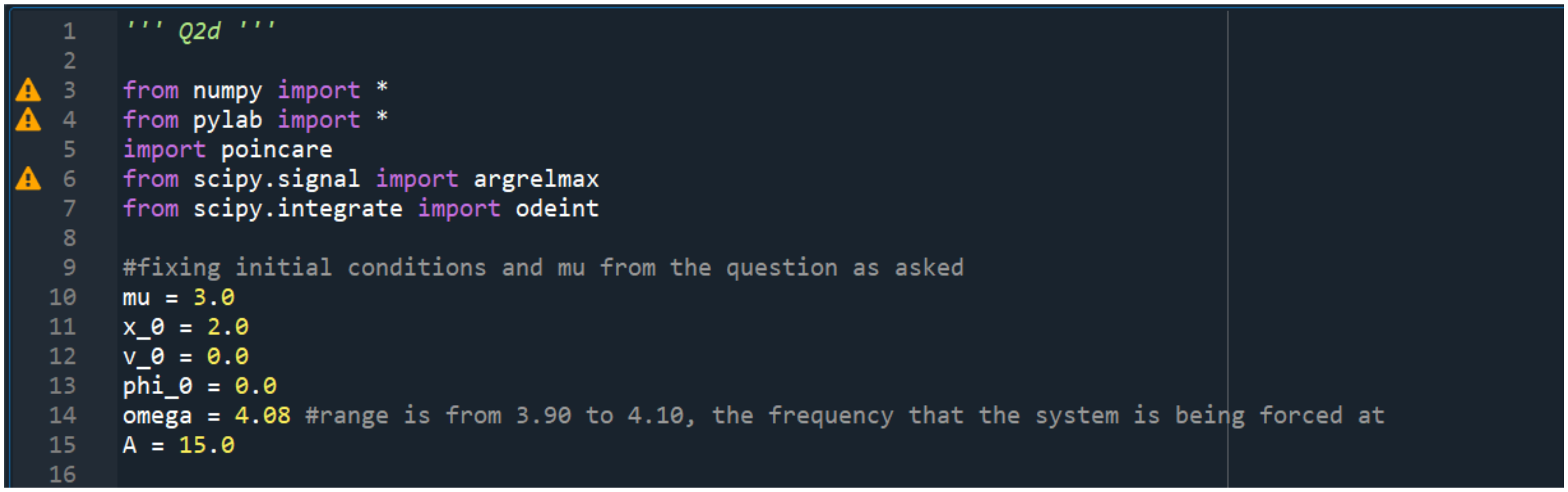
```python
''' Question 2c'''

from numpy import *
from pylab import *
from scipy.signal import argrelmax
from scipy.integrate import odeint

#fixing initial conditions and mu from the question as asked
mu = 3.0
x_0 = 2.0
v_0 = 0.0
phi_0 = 0.0
omega = 3.92 #range is from 3.90 to 4.10
A = 15.0

#assigning a value for the time step
dt = 0.01
t = arange( 0.0 , 100.0 , dt )

#defining rhs
def rhs( u , t ):
    x = u[ 0 ]
    v = u[ 1 ]
    du = [v , mu * ( 1 - x ** 2 ) * v - x + A * cos( omega * t )]
    return du

initial_cond = [ x_0 , v_0 ] #array of initial conditions

#Using the odeint function to solve the equation with forcing forcing this time.
forcing_soln = odeint(rhs, initial_cond, t)
x = forcing_soln[:, 0]
v = forcing_soln[:, 1]

number_of_sec = 90 #get rid of the motion in the beginning
for_steady = int( 1 / dt ) * number_of_sec

sted_x = x[for_steady:]
sted_v = v[for_steady:]
sted_time = t[for_steady:]

figure(1) #open first window
subplot(2, 1, 1) #create subplot
plot(t, x)
plot( sted_time , sted_x )
grid('on') #show grid
xlabel( ' time t ' )
ylabel( ' position x ' )
plt.title( ' Plot for time versus position for forced van der pol, omega = 3.92 ' )

```



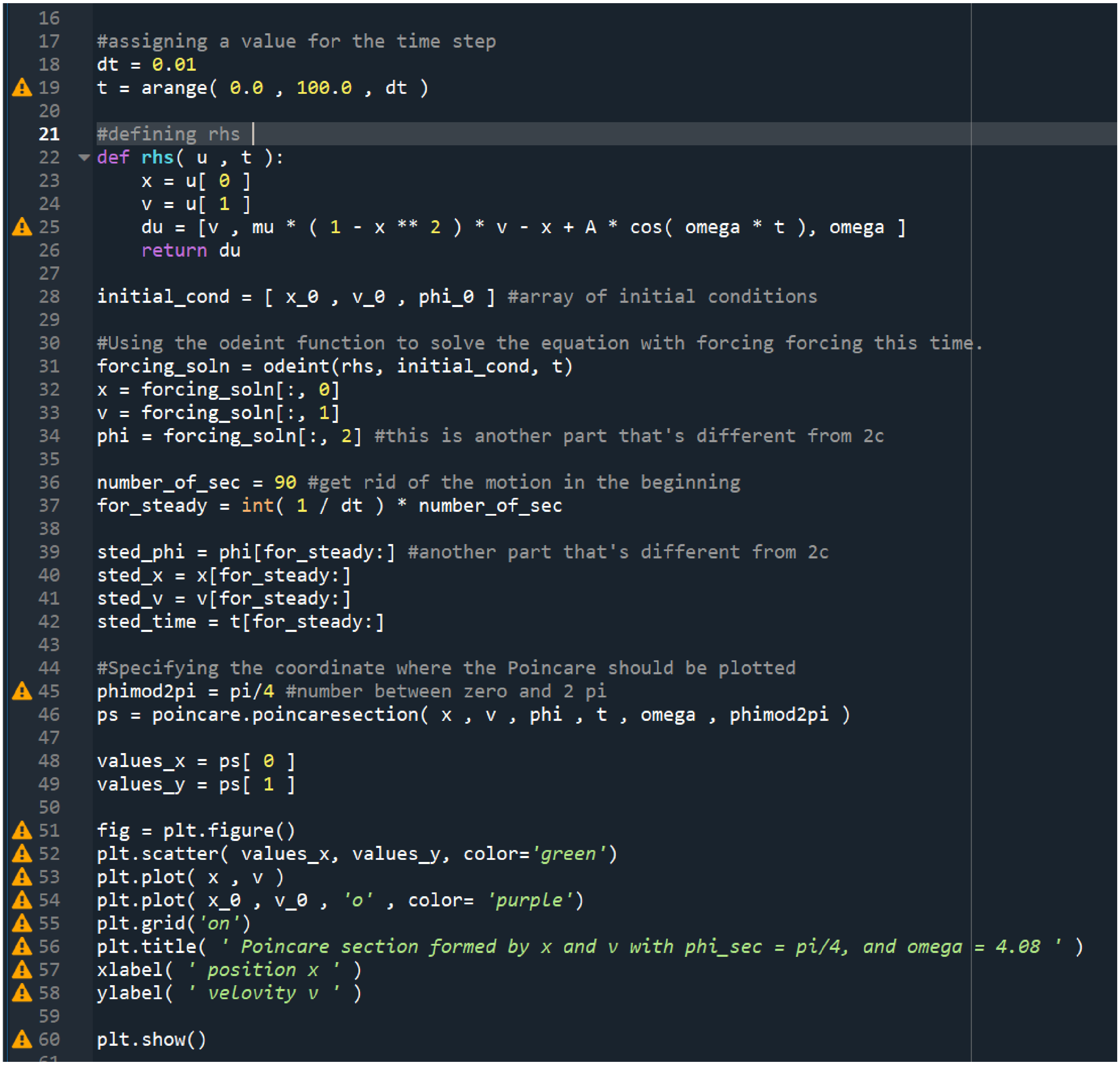
```python

subplot(2, 1, 2)
plot( t, v )
plot( sted_time , sted_v)
grid('on')
xlabel( ' time t ')
ylabel( ' velocity v ' )
plt.title( ' Plot for time versus velocity for forced van der pol, omega = 3.92 ')

figure(2)
plot(x, v)
plot(sted_x, sted_v)
plot( x_0, v_0, 'o', color= 'green')
grid('on')
xlabel( ' position x ' )
ylabel( ' velocity v ' )
plt.title( ' phase space plot for position x versus velocity v for forced van der pol, omega = 3.92 ' )
show()

```

(d) Now modify the code for plotting the phase space to call poincaresection function. Plot some Poincare sections at interesting values of omega and phi_sec. Try varying phi_sec to see how the section simplifies the messy trajectory.

- I first plotted the Poincare section for omega = 4.07, and phi_sec = 0 to ensure I did it correctly, and my plot does look like a duck. I have submitted this plot.
- The values of omega and phi_sec I have used for each plot I submitted is included in the title of each plot.
- I have also kept omega constant at 4.08, and only varied phi_sec to see what would happen at a constant value of omega. I have submitted the plots for that as well.
- Here is my code for part d.



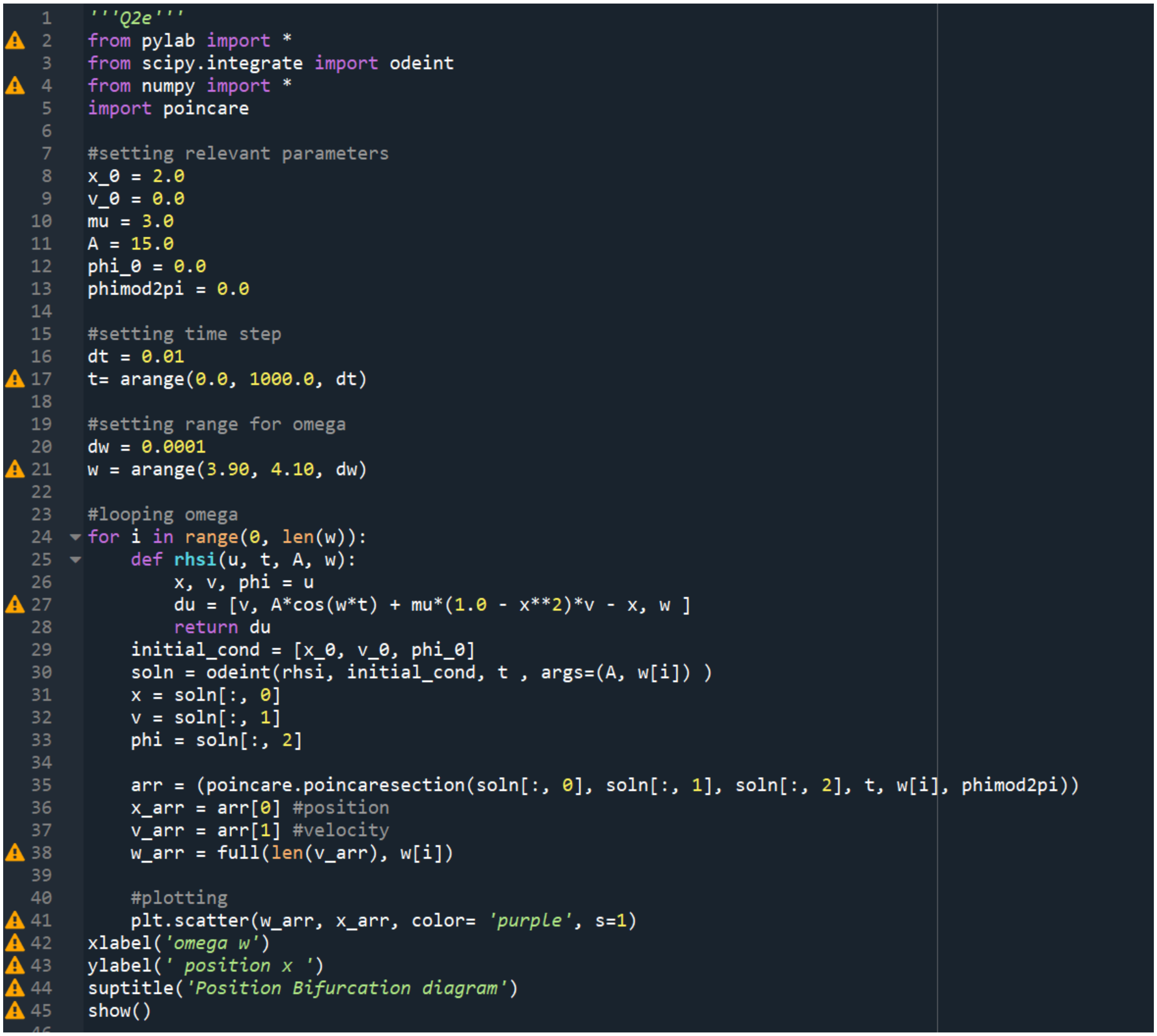
```python
''' Q2d '''

from numpy import *
from pylab import *
import poincare
from scipy.signal import argrelmax
from scipy.integrate import odeint

#fixing initial conditions and mu from the question as asked
mu = 3.0
x_0 = 2.0
v_0 = 0.0
phi_0 = 0.0
omega = 4.08 #range is from 3.90 to 4.10, the frequency that the system is being forced at
A = 15.0

```



```python

#assigning a value for the time step
dt = 0.01
t = arange( 0.0 , 100.0 , dt )

#defining rhs 
def rhs( u , t ):
    x = u[ 0 ]
    v = u[ 1 ]
    du = [v , mu * ( 1 - x ** 2 ) * v - x + A * cos( omega * t ), omega ]
    return du

initial_cond = [ x_0 , v_0 , phi_0 ] #array of initial conditions

#Using the odeint function to solve the equation with forcing forcing this time.
forcing_soln = odeint(rhs, initial_cond, t)
x = forcing_soln[:, 0]
v = forcing_soln[:, 1]
phi = forcing_soln[:, 2] #this is another part that's different from 2c

number_of_sec = 90 #get rid of the motion in the beginning
for_steady = int( 1 / dt ) * number_of_sec

sted_phi = phi[for_steady:] #another part that's different from 2c
sted_x = x[for_steady:]
sted_v = v[for_steady:]
sted_time = t[for_steady:]

#Specifying the coordinate where the Poincare should be plotted
phimod2pi = pi/4 #number between zero and 2 pi
ps = poincare.poincaresection( x , v , phi , t , omega , phimod2pi )

values_x = ps[ 0 ]
values_y = ps[ 1 ]

fig = plt.figure()
plt.scatter( values_x, values_y, color='green')
plt.plot( x , v )
plt.plot( x_0 , v_0 , 'o' , color= 'purple')
plt.grid('on')
plt.title( ' Poincare section formed by x and v with phi_sec = pi/4, and omega = 4.08 ' )
xlabel( ' position x ' )
ylabel( ' velovity v ' )

plt.show()
```

(e) What does this plot suggest about how the chaos arises in this system?

- From the bifurcation diagram, we see regions that are very messy, but we also see regions that repeat themselves when we zoom into the diagram. The regions that repeat themselves correspond to values of omega where the motion is periodic (has a limit cycle), and the distorted regions are chaotic. So we can say that this system will be chaotic or periodic depending on the driving frequency omega. If omega is just right, it'll have chaotic behavior.
- My code for part 2e is below.



```python
'''Q2e'''
from pylab import *
from scipy.integrate import odeint
from numpy import *
import poincare

#setting relevant parameters
x_0 = 2.0
v_0 = 0.0
mu = 3.0
A = 15.0
phi_0 = 0.0
phimod2pi = 0.0

#setting time step
dt = 0.01
t= arange(0.0, 1000.0, dt)

#setting range for omega
dw = 0.0001
w = arange(3.90, 4.10, dw)

#looping omega
for i in range(0, len(w)):
    def rhsi(u, t, A, w):
        x, v, phi = u
        du = [v, A*cos(w*t) + mu*(1.0 - x**2)*v - x, w ]
        return du
    initial_cond = [x_0, v_0, phi_0]
    soln = odeint(rhsi, initial_cond, t , args=(A, w[i]) )
    x = soln[:, 0]
    v = soln[:, 1]
    phi = soln[:, 2]

    arr = (poincare.poincaresection(soln[:, 0], soln[:, 1], soln[:, 2], t, w[i], phimod2pi))
    x_arr = arr[0] #position
    v_arr = arr[1] #velocity
    w_arr = full(len(v_arr), w[i])

    #plotting
    plt.scatter(w_arr, x_arr, color= 'purple', s=1)
xlabel('omega w')
ylabel(' position x ')
suptitle('Position Bifurcation diagram')
show()
```