VP4 Damped Oscillator, Spring-Wave, and Phonons [empty class, list representation, numpy array]

I. List Representation

1. range(). Notice the number in range should be integer (int)

```
L = range(5)                              # list(L) = [0, 1, 2, 3, 4].
L = range(4, 9)                           # list(L) = [4, 5, 6, 7, 8]
L = range(1, 6, 2)                        # list(L) = [1, 3, 5]    1 to 6 every other 2 numbers
```

2. list representation. Sometimes we want to generate a list with some conditions, e.g.

```
L = [i**2  for i  in  range(5)]           # =  [0, 1, 4, 9, 16]
L = [0.1*i*pi  for  i  in  range(-3, 3)]  # =  [-0.3*pi, -0.2*pi, -0.1*pi, 0, 0.1*pi, 0.2*pi]
L = [i**2  for  i  in  range (5)  if  i != 3]   # =  [0, 1, 4, 16]
```

3. List representation can be used in a nested structure, or for dictionary or tuple. e.g.

```
L = [i*10 + j for i in range(3) for j in  range(5) ]   # =  [0, 1, 2, 3, 4, 10, 11, 12, 13, 14, 20, 21, 22, 23, 24]
D = {i:i**2  for i  in [0, 1, 2]}                      # = {0:0,   1:1, 2:4}
```

II. Numpy Array

Python has a very powerful module called “numpy”, in which there is a type called array. Array can help us to speed up program dramatically (30 times to 300 times faster). First, let’s see some operations about array.

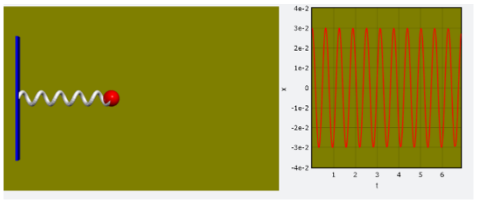
```
>>> from numpy import *
>>> a = arange(4)                 # array range
>>> a
array([ 0, 1, 2, 3])
>>> a = arange(0, 4.0, 0.5)       # array range
>>> a
array([ 0. ,  0.5,  1. ,  1.5,  2. ,  2.5,  3. ,  3.5])
>>> a[0] = 5                      # change the 0th element of a to 5
>>> a
array([ 5. ,  0.5,  1. ,  1.5,  2. ,  2.5,  3. ,  3.5])
>>> b = array(range(10))          # b is an array from a list generated by range(10)
>>> b
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a[1:-1] **2                   # from the 1st to before the last element, generate a new array with squared value
array([ 0.25,  1.  ,  2.25,  4.  ,  6.25,  9.  ])
>>> a[-1] = 100                   # change the last element (-1) to 100
>>> a
array([   5. ,    0.5,    1. ,    1.5,    2. ,    2.5,    3. ,  100. ])
>>> a[5:] *= 0.5                  # from the 5th element to the end, change each of them by multiplying 0.5
>>> a
array([  5.  ,   0.5 ,   1.  ,   1.5 ,   2.  ,   1.25,   1.5 ,  50.  ])
>>> a[:-1] + b[-7:]               # add array a from 0th to before the last element by array b from the last 7th
array([  8.  ,   4.5 ,   6.  ,   7.5 ,   9.  ,   9.25,  10.5 ])   # element to the end, and generate a new array to store the values
```

**Notice the index starts from 0. The following is to compare code execution speed for different methods.

```
from timeit import default_timer as timer
from numpy import *

start = timer()
for x in range(100):
   j = []
   for i in range(10000): j.append(i**2)
end = timer()
print(end-start)

start = timer()
for x in range(100): j=[i**2 for j in range(10000)]
end = timer()
print(end-start)

start = timer()
for x in range(100):
   j=arange(10000)**2
end = timer()
print(end-start)
```

III. Damped Oscillator (SHM)

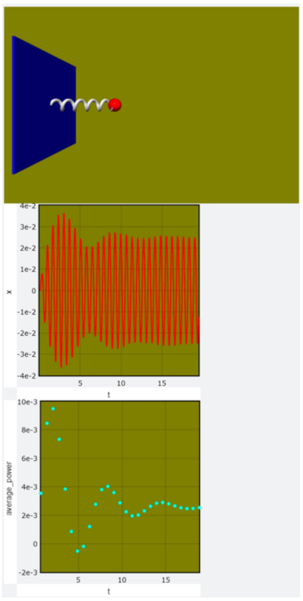
```
from vpython import *
size, m = 0.02, 0.2   # ball size =  0.02 m, ball mass = 0.2kg
L, k = 0.2, 20          # spring original length = 0.2m, force constant = 20 N/m
amplitude = 0.03
b = 0.05 * m * sqrt(k/m)

scene = canvas(width=600, height=400, range = 0.3, align = 'left', center=vec(0.3, 0, 0), background=vec(0.5,0.5,0))
wall_left = box(length=0.005, height=0.3, width=0.3, color=color.blue)          # left wall
ball = sphere(radius = size,  color=color.red)                                      # ball
spring = helix(radius=0.015, thickness =0.01)
oscillation = graph(width = 400, align = 'left', xtitle='t',ytitle='x',background=vec(0.5,0.5,0))
x = gcurve(color=color.red,graph = oscillation)

ball.pos = vector(L+amplitude, 0 , 0)                                                 # ball initial position
ball.v = vector(0, 0, 0)                          # ball initial velocity
ball.m = m
spring.pos = vector(0, 0, 0)
t, dt = 0, 0.001
while True:
   rate(1000)
   spring.axis = ball.pos - spring.pos      # spring extended from spring endpoint A to ball
   spring_force = - k * (mag(spring.axis) - L) * norm(spring.axis)         # spring force vector
   ball.a = spring_force / ball.m                # ball acceleration = spring force /m - damping
   ball.v +=   ball.a*dt
   ball.pos += ball.v*dt
   t += dt
   x.plot(pos=(t,ball.pos.x - L))
```



Modified from hw2, the above code simulates the horizontal oscillation of a given amplitude.

1. Now, in addition to the restoring force from the spring, add the air resistance force $\vec{f} = -b\vec{v}$ to the ball with **damping factor** $b = 0.05m\sqrt{k/m}$.

2. Follow 1, but instead of letting the ball to oscillate from the initial amplitude, i.e. ball.pos = vector(L+amplitude, 0, 0), now let the ball to be initially at rest at $x = L$, i.e. ball.pos = vector(L, 0, 0), and allow a sinusoidal force $\vec{F} = f_a \sin(\omega_d t)\hat{x}$ ($\hat{x}$ is the unit vector in x-axis) applied on the ball, with f_a = 0.1 and $\omega_d = \sqrt{k/m}$. We know when a force $\vec{F}$ exerts on an object of velocity $\vec{v}$, the power on the object by the force is $P = \vec{F} \cdot \vec{v}$. Find by your simulation P_T, the power averaged over a period $T = 2\pi / \omega_d$ at the end of each period. In additional to the curve graph for ball's position versus time, plot a dot graph for P_T versus time. Observe the results with different settings, such as ω_d =0.8 $\sqrt{k/m}$, 0.9 $\sqrt{k/m}$, 1.1 $\sqrt{k/m}$, or 1.2 $\sqrt{k/m}$ and/or with different *b* values. Think about the results. Before proceeding to Practice 3, change ω_d and *b* back to the original values.



3. Often, we do not want an animated simulation, which is slow due to the animation, but only the calculation results. We can modify the above computer codes easily for such purpose. We can just delete the code that creates the canvas, the plot, and the graphs (i.e., those bold blackened codes), delete rate(1000), and replace codes that generate visual objects by the following codes that generate objects from an empty class.

```
class obj:   pass

wall_left, ball, spring = obj(), obj(), obj()
```

With these, we still do the same simulation but do not animate them. This will speed up the simulation. Instead of plotting P_T, now only print P_T every period. You can see after certain number of periods, the system reaches a **steady state** and the P_T is almost a constant. Also notice that how much faster the

simulation can run without the animation. (NOTICE: for this technique to work, you need to have all the proper parameters set for every object after they have been created by the empty class obj.)
The following is example codes:

```
from vpython import *
size, m = 0.02, 0.2                   # ball size =  0.02 m, ball mass = 0.2kg
L, k = 0.2, 20           # spring original length = 0.2m, force constant = 20 N/m
fa = 0.1
b = 0.05 * m * sqrt(k/m)
omega_d = sqrt(k/m)

'''
scene = canvas(width=600, height=400, fov = 0.03, align = 'left', center=vec(0.3, 0, 0), background=vec(0.5,0.5,0))
wall_left = box(length=0.005, height=0.3, width=0.3, color=color.blue)                # left wall
ball = sphere(radius = size,  color=color.red)                                        # ball
spring = helix(radius=0.015, thickness =0.01)
oscillation1 = graph(width = 400, align = 'left', xtitle='t',ytitle='x',background=vec(0.5,0.5,0))
x=gcurve(color=color.red,graph = oscillation1)
'''
oscillation2 = graph(width = 400, align = 'left', xtitle = 't', ytitle = 'average_power', background=vec(0.5,0.5,0))
p = gdots(color=color.cyan, graph = oscillation2)

class obj: pass                              ####
ball, spring = obj(), obj()                  ####

spring.pos = vector(0, 0, 0)
ball.pos = vector(L, 0 , 0)                                        # ball initial position
ball.v = vector(0, 0, 0)                              # ball initial velocity
ball.m = m

T = 2*pi / omega_d
t, dt, n = 0, 0.001, 1
power = 0.0
while True:
   #rate(1000)
   spring.axis = ball.pos - spring.pos
   spring_force = - k * (mag(spring.axis) - L) * norm(spring.axis)
   force = vector(fa*sin(omega_d*t), 0, 0)
   ball.a =  (force + spring_force - b * ball.v ) / m
   ball.v +=   ball.a*dt
   ball.pos += ball.v*dt
   t += dt
   #x.plot(pos=(t,ball.pos.x - L))

   power += dot(force, ball.v)*dt
   if t / T >= n:
      p.plot(pos = (t, power / T))
      n += 1.0
      power = 0
```

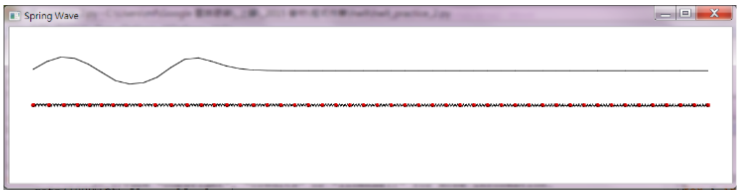
IV. Longitudinal Spring-Ball Wave

```
from vpython import *
A, N, omega = 0.10, 50, 2*pi/1.0
size, m, k, d = 0.06, 0.1, 10.0, 0.4
scene = canvas(title='Spring Wave', width=800, height=300, background=vec(0.5,0.5,0), center = vec((N-1)*d/2, 0, 0))
balls = [sphere(radius=size, color=color.red, pos=vector(i*d, 0, 0), v=vector(0,0,0)) for i in range(N)]
springs = [helix(radius = size/2.0, thickness = d/15.0, pos=vector(i*d, 0, 0), axis=vector(d,0,0)) for i in range(N-1)]
t, dt = 0, 0.001
while True:
   rate(1000)
   t += dt
   balls[0].pos = vector(A * sin(omega * t ), 0, 0)
   for i in range(N-1):
      springs[i].pos = balls[i].pos
      springs[i].axis = balls[i+1].pos - balls[i].pos
   for i in range(1, N):
      if i == N-1: balls[-1].v += - k * vector((springs[-1].axis.mag-d),0,0)/m*dt
      else: balls[i].v += k* vector((springs[i].axis.mag-d),0,0)/m*dt - k* vector((springs[i-1].axis.mag-d),0,0)/m*dt
      balls[i].pos += balls[i].v*dt
```

1. Array based simulation

Change the above codes to the following array-based codes. We also need to complete the 2 lines commented by '##'. Notice that, in the very first line, we import numpy as a shortened name 'np'. Later at the line marked #5, we use numpy's class (arange) to generate the arrays to store the positions of the balls, the original positions of the balls, the velocity of the balls, and the spring lengths. We need to use np.arange() as the proper syntax for such purpose.

```
import numpy as np
from vpython import *
A, N, omega = 0.10, 50, 2*pi/1.0
size, m, k, d = 0.06, 0.1, 10.0, 0.4
scene = canvas(title='Spring Wave', width=800, height=300, background=vec(0.5,0.5,0), center = vec((N-1)*d/2, 0, 0))
balls = [sphere(radius=size, color=color.red, pos=vector(i*d, 0, 0), v=vector(0,0,0)) for i in range(N)]            #3
springs = [helix(radius = size/2.0, thickness = d/15.0, pos=vector(i*d, 0, 0), axis=vector(d,0,0)) for i in range(N-1)]     #3
#1
ball_pos, ball_orig, ball_v, spring_len = np.arange(N)*d, np.arange(N)*d, np.zeros(N), np.ones(N)*d                  #5
t, dt = 0, 0.001
while True:
  rate(1000)
  t += dt
  ball_pos[0] = A * sin(omega * t )                              #4
  ## spring_len[:-1] =
  ## ball_v[1:] +=                                               #6
  ball_pos +=  ball_v*dt

  for i in range(N): balls[i].pos.x = ball_pos[i]                #3
  for i in range(N-1):                                           #3
    springs[i].pos = balls[i].pos                                #3
    springs[i].axis = balls[i+1].pos - balls[i].pos              #3
  #2
```



2. To get a clearer view of the wave, we plot the longitudinal displacement of the balls from their original positions by a transverse displacement.

at #1, add code

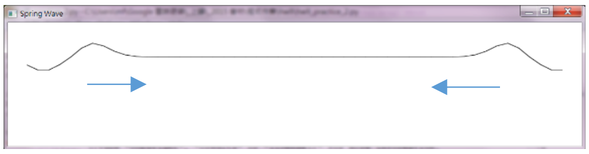
```
c = curve([vector(i*d, 1.0, 0) for i in range(N)], color=color.black)
```

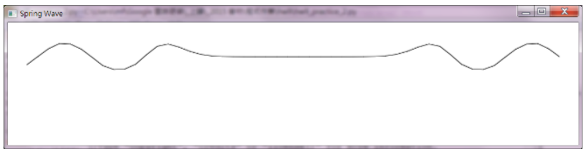
at #2, add code

```
  ball_disp = ball_pos - ball_orig
  for i in range(N):
    c.modify(i, y = ball_disp[i]*4+1)
```

3. Since we already have the clear representation of the wave, we can remove the ball-spring plotting, marked by #3. Notice that although the wave shown is with transverse motion, but it is indeed a longitudinal wave with a transverse wave representation.

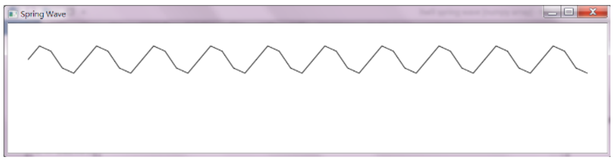
4. You may have seen such things in a video game that an object moves out of the right end of the screen and comes back from the left end of the screen. This is called "periodically boundary condition", a very often used assumption in physics and engineering. In our system here, it means that the last ball and the zeroth ball is also connected by a spring in a fashion that looks like the zeroth ball is to the right of the last ball, or the last ball is to the left of the zeroth ball.





Now, modify the codes marked by '##', and add one or two more lines to achieve such "periodically boundary condition". You will see that the wave is generated from both ends towards the center because now the last one is also connected to the zeroth one, which is the source of the wave.

5. As you saw in Practice 4, the wave amplitude at times gets larger and at times gets smaller. This means that this wave we try to create is not a "normal mode" of the system. The normal modes of the system are those waves whose amplitude can keep constant. This means that after going a full circle (Notice that we have the tail connected to the head, so we can view this system as a circle), the wave repeats itself, or we can say that wave satisfies the periodically boundary condition. These are waves with wavelength $\lambda = Nd/n$, where



$n = 1, 2, 3, \ldots$ (i.e. wavenumber $K = 2\pi/\lambda = 2\pi n/(Nd)$, which is also called wavevector). We can observe such wave by initially assigning balls to their proper positions, removing the external source, and letting the system to go by itself. This means changing #5 to

```
Unit_K, n = 2 * pi/(N*d), 10
Wavevector = n * Unit_K
phase = Wavevector * arange(N) * d
ball_pos, ball_orig, ball_v, spring_len = np.arange(N)*d + A*np.sin(phase), np.arange(N)*d, np.zeros(N), np.ones(N)*d
```

Notice in the last line, ball_pos is set to **np.arange(N)*d + A*np.sin(phase)**, in which **np.arange(N)*d** is the array for the balls' original positions at balance points. A in **A*np.sin(phase)** is the amplitude of the longitudinal wave, and each ball is originally displaced by A*sin(phase) before the simulation begins. phase in **phase = Wavevector * arange(N) * d** is the array for the phase of each ball's position in the longitudinal wave. We use np.sin(phase) because we want an array, elements of which are the sin function of phase, which is also an array. Then this array can be added to another array, np.arange(N)*d.
You also need to remove #4 and add **after #6** a line to handle zeroth ball's velocity ball_v[0].

In the simulation, you will see clearly the oscillating standing wave with n = 10. Try to obtain the period (T) of the standing wave and the corresponding angular frequency (omega = 2*pi / T). For higher accuracy, please change dt = 0.001 to dt = 0.0003.

V. Homework

In a crystal, only certain normal modes of waves can exist. These modes are called "Phonons". Practice 5. shows one of such modes in a simplified 1-dimensional crystal consisted of only 50 balls (atoms). As you already see, when the wavevector is given, the angular frequency (omega =2 pi/T) is decided by the system. Now we want to know the relationship between the angular frequency and the wavevector, which is called the dispersion relationship. Modify you program from Practice 5. such that you can obtain the angular frequency (omega) for n from 1 to N/2-1. Use the graph plotting similar to lesson 5 homework (must) to plot the the angular frequency versus the wavevector.