VP2: SHM, Collision, Pendulum, and Newton Cradle [tuple, list, for, function]

I. List: (See the right figure)

Variable *a* is assigned to a 'list', which is an 'ordered collection of objects'. The objects can be integers, floats, strings, lists, or any other objects. In the example here, the 0th element of *a* is integer 1, the 1st is integer 2, the 2nd is 'x', and so forth. (Notice that in python, the order starts at 0.) More importantly, the elements of list can be mixed of any different types.

```
In [38]: a = [1, 2, 'x', 4, 5]
         a[1]
Out[38]: 2

In [39]: a[2]
Out[39]: 'x'

In [40]: a[0]
Out[40]: 1

In [41]: a[-1]
Out[41]: 5

In [42]: a[1:3]
Out[42]: [2, 'x']

In [43]: b = a
         b
Out[43]: [1, 2, 'x', 4, 5]

In [44]: a[1] = 99
         a
Out[44]: [1, 99, 'x', 4, 5]

In [45]: b
Out[45]: [1, 99, 'x', 4, 5]

In [46]: a[2:5]=[6, 'y', 8]

In [47]: a
Out[47]: [1, 99, 6, 'y', 8]

In [48]: for i in a:
             print(i)
         1
         99
         6
         y
         8

In [49]: a.append(10)
         a
Out[49]: [1, 99, 6, 'y', 8, 10]
```

1. a[0] is the 0-th element of list a.
2. a[-1] is the last element of list a.
3. a[1:3] is a list, with the content of the 1st and 2nd elements (the one before 3) of list a.
4. b = a makes variable b point to the same list as variable a. (See the following Note)
5. If we assign 99 to the 1st element of the list a, then the list becomes [1, 99, 'x', 4, 5]. Since b points to the same list as a, if we show list b, we will see the same contents.
6. We can assign multiple consecutive elements at once, such as a[2:5]=[6, 'y', 8]. Then, the list becomes [1, 99, 6, 'y', 8]
7. for i in a:
   associated codes …

   The code lets i be assigned to each element in list a (in order) and then executes the associated codes until it runs out all the elements.
8. One can use a.append(10) to append a new element 10 at the end of list a.

Note:

There are two storing spaces in python to store the variables, Namespace and Valuespace. Namespace stores the variable names, such as x, ball, ball.v, and etc. and Valuespace stores the values, such as 3, 5.0, and vec(5, 0, 3). When we write x = 1, in python, it does the following. First it creates a space in Valuespace and store the value 1 there. Then it creates a variable name 'x' in Namespace. Finally, it lets 'x' to point to 1.

When we write y = x + 2, it takes out explicitly the value the variable name 'x' points to, which is 1, adds it by 2 and gets 3, creates a new space in Valuespace and stores 3 there. Finally, it creates variable 'y' and points 'y' to this space containing 3. If we write x = x + 2 the procedure is similar but only that 'y' is not created but instead 'x' is reused and pointed to value 3. What is the difference between this and x += 2? In this, it goes to the space in Valuespace where 'x' points to, and add 2 directly **on-site**. With this, you can understand, why b = a causes b to points to the same list as a and this causes the effects shown in the 5th point above.

II. Simple Harmonic Motion:

```
from vpython import *

g = 9.8
size, m = 0.05, 0.2
L, k = 0.5, 15
scene = canvas(width=500, height=500, center=vec(0, -0.2, 0), background=vec(0.5,0.5,0))
ceiling = box(length=0.8, height=0.005, width=0.8, color=color.blue)
ball = sphere(radius = size,  color=color.red)
spring = helix(radius=0.02, thickness =0.01)    # default pos = vec(0, 0, 0)
ball.v = vec(0, 0, 0)
ball.pos = vec(0, -L, 0)
```

```
dt = 0.001
while True:
   rate(1000)
   spring.axis = ball.pos - spring.pos     # new: extended from spring endpoint to ball

   spring_force = - k * (mag(spring.axis) - L) * spring.axis.norm()   # to get spring force vector
   ball.a = vector(0, - g, 0) + spring_force / m                                  # ball acceleration = - g in y + spring force /m

   ball.v +=   ball.a*dt
   ball.pos += ball.v*dt
```

1. Tuple assignment

```
size, m = 0.05, 0.2       or    (size, m) = (0.05,  0.2)
```

*** (0.05, 0.2) is a tuple data type. When two tuples are on either sides of a '=', the values are assigned term by term. This can be done for as many terms as long as both sides have the same number of variables and values.

2. Draw a spring, by `spring = helix(radius=0.02, thickness =0.01)`, as a helix object with radius = 0.02 and thickness = 0.01. Its endpoint defaults to position vector(0, 0, 0). The parameters can be set by `spring = helix(pos = vector(0,0,0), radius=0.02, thickness =0.01)` when it is constructed, or they can be changed later by assignment, such as `spring.pos = vector(0,0,0)`. The `axis` attribute, such as used in `spring.axis = ball.pos - spring.pos`, draws the spring from `spring.pos` to `ball.pos`

3. Force vector by the spring,

```
spring_force = - k * (mag(spring.axis) - r) * spring.axis.norm()
```

mag(a vector) is a function that yields the magnitude of the vector. The norm() 'method' will give the unit direction vector of the vector attached, e.g. `spring.axis.norm()` gives the unit direction vector for spring.axis. Can you figure out why this line of code yields the force generated by the spring on the ball?

4. Find the acceleration

```
ball.a = vector(0, - g, 0) + spring_force / m
```

III. Multiballs:

```
from vpython import *

g = 9.8
size, m = 0.05, 0.2
L, k = 0.5, [15, 12, 17]
v = [1, 2, 2.2]
d = [-0.06, 0, -0.1]

scene = canvas(width=400, height=400, center =vec(0.4, 0.2, 0), align = 'left', background=vec(0.5,0.5,0))
floor = box(pos = vec(0.4, 0, 0), length=0.8, height=0.005, width=0.8, color=color.blue)
wall = box(pos= vec(0, 0.05, 0), length = 0.01, height = 0.1, width =0.8)

balls = []
for i in range(3):
   ball = sphere(pos = vec(L+d[i], size, (i-1)*3*size), radius = size,  color=color.red)
   ball.v = vec(v[i], 0, 0)
   balls.append(ball)

springs =[]
for i in range(3):
   spring = helix(pos = vec(0, size, (i-1)*3*size), radius=0.02, thickness =0.01)
   spring.axis = balls[i].pos-spring.pos
   spring.k = k[i]
   springs.append(spring)
```

This code generates a list of 3 ball objects on the floor, a list of 3 springs connecting the 3 balls, respectively, to the wall. Have a look how for is used in this program.

The only thing new to you is

```
for i in range(3):
   code….
```

for which i is assigned sequentially to 0, 1, and 2 and the associated code are executed. If we want to execute from 1 to 2, we can use

```
for i in range(1,3):
   code….
```

IV. Collision

```
from vpython import *

size = [0.05, 0.04]          #ball radius
mass = [0.2, 0.4]                #ball mass
colors = [color.yellow, color.green]                  #ball color
position = [vec(0, 0, 0), vec(0.2,-0.35, 0)]#ball initial position
velocity = [vec(0, 0, 0), vec(-0.2, 0.30, 0)]          #ball initial velocity

scene = canvas(width = 800, height =800, center=vec(0, -0.2, 0), background=vec(0.5,0.5,0))       # open window
ball_reference = sphere (pos = vec(0,0,0), radius = 0.02, color=color.red)

def af_col_v(m1, m2, v1, v2, x1, x2):                          # function after collision velocity
   v1_prime = v1 + 2*(m2/(m1+m2))*(x1-x2)  * dot (v2-v1, x1-x2) / dot (x1-x2, x1-x2)
   v2_prime = v2 + 2*(m1/(m1+m2))*(x2-x1)  * dot (v1-v2, x2-x1) / dot (x2-x1, x2-x1)
   return (v1_prime, v2_prime)

balls =[]
for i in [0, 1]:
   balls.append(sphere(pos = position[i], radius = size[i],  color=colors[i]))
   balls[i].v = velocity[i]
   balls[i].m = mass[i]

dt = 0.001
while True:
   rate(1000)

   for ball in balls:
      ball.pos += ball.v*dt

   if (mag(balls[0].pos - balls[1].pos) <= size[0]+size[1] and dot(balls[0].pos-balls[1].pos, balls[0].v-balls[1].v) <= 0) :
      (balls[0].v, balls[1].v) =  af_col_v (balls[0].m, balls[1].m, balls[0].v, balls[1].v, balls[0].pos, balls[1].pos)
```

This program demonstrates an elastic collision event between two balls.

1. function

There are generally two cases in which we want to write a section of codes as a function. In one, we want to reuse many times the same section of codes. In the other, we want to make the entire codes more readable. Here is the example function that yields the velocities of two spherical objects after an elastic collision.

```
def af_col_v(m1, m2, v1, v2, x1, x2):                          # function after collision velocity
   v1_prime = v1 + 2*(m2/(m1+m2))*(x1-x2)  * dot (v2-v1, x1-x2) / dot (x1-x2, x1-x2)
   v2_prime = v2 + 2*(m1/(m1+m2))*(x2-x1)  * dot (v1-v2, x2-x1) / dot (x2-x1, x2-x1)
   return (v1_prime, v2_prime)
```

The first line def af_col_v(m1, m2, v1, v2, x1, x2): declares by def the function name as af_col_v. The function name is better to have meaning, here it means ‘after collision velocities’. After the function name is the

parentheses and a colon. Inside the parentheses can be empty or any parameters which pass information from the main program to the function. They are m1, m2, v1, v2, x1, x2 for masses, velocities, and positions of the two objects, respectively.

The subordinate programs below the colon are the major part of the function that handles the job. Here, they calculate the velocities after collision from the masses, velocities, and positions of the two spherical objects and put them into two variables, v1_prime and v2_prime. In the last line, it return the values of the two variables back to where the function is called. Say if we have the following code

```
V1, V2 = after_col_v(0.5, 0.6, vec(1, 2, 3), vec(4, 5, 6), vec(1, 0, 1), vec(0, 1, 1) )
```

It will call the after_col_v function and let m1 = 0.5, m2= 0.6, v1 = vec(1, 2, 3)…, and so on. And after the function is executed, V1 and V2 will be the values of v1_prime and v2_prime, respectively. Note, that this way of calling function is 'call by position', i.e. the parameters are matched by position order. We can also do this by 'call by keywords', such as

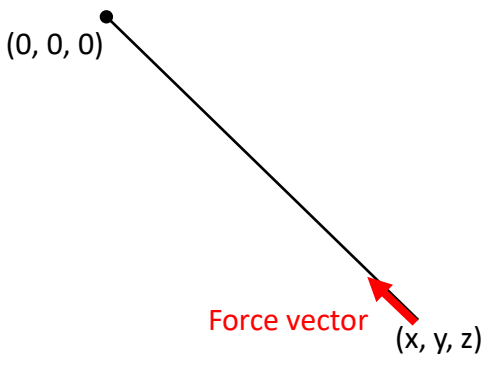
```
V1, V2 = after_col_v(v1= vec(1, 2, 3), v2= vec(4, 5, 6), m2= 0.6,  x1= vec(1, 0, 1), x2=vec(0, 1, 1), m1 = 0.5 )
```

Within the parentheses, the parameter order is not at all important, since the parameter is assigned explicitly.

V. Pendulum

```
from vpython import *

g = 9.8
size, m = 0.02, 0.5
L, k = 0.5, 15000

scene = canvas(width=500, height=500, center=vec(0, -0.2, 0), background=vec(0.5,0.5,0))
ceiling = box(length=0.8, height=0.005, width=0.8, color=color.blue)
ball = sphere(radius = size,  color=color.red)
spring = cylinder(radius=0.005)    # default pos = vec(0, 0, 0)
ball.v = vec(0.6, 0, 0)
ball.pos = vec(0, -L - m*g/k, 0)

dt = 0.001
t = 0
while True:
   rate(1000)
   t += dt
   spring.axis = ball.pos - spring.pos     #spring extended from endpoint to ball
   spring_force = - k * (mag(spring.axis) - L) * spring.axis.norm()   # to get spring force vector
   ball.a = vector(0, - g, 0) + spring_force / m                      # ball acceleration = - g in y + spring force /m

   ball.v +=   ball.a*dt
   ball.pos += ball.v*dt
```

Does this program look familiar? Yes, it is only different from the 'simple harmonic motion' program in II by 4 lines: ball.v = …, ball.pos = …, spring = …, and L, k = 0.5, 15000. But now, instead of an oscillating spring-ball system, we have a pendulum. Let's see how the difference of the program works.



Since we want to have a pendulum, i.e. a massive object attached to a rope, we need to have a rope. Spring is therefore changed from a helix to a cylinder for the shape of a rope. More, when you try use a force to stretch a rope, there is a tension on the rope, and the tension is equal to the force. Also, the extension of the rope is very tiny, this means that the rope is just like a spring but with a very large force constant. Thus, we set k to be 15000, any large number will do, but do not let it be too large or too small, which will cause the simulation to be

very unstable. Due to gravitation, the rope is stretched a bit from its original length, therefore ball.pos = vec(0, -L - m*g/k, 0). The ball is set to have an initially horizontal velocity, ball.v = vec(0.6, 0, 0). Then, we have a pendulum simulation.

This simulation is more "fundamental" than the pendulum analysis in the high school physics. The ball is exerted by two forces, one gravitational force and the other is the tension from the string. As the ball is swinging at the bottom of the string, it is doing a circular motion that requires a centripetal force. But how does the string "know" exactly how much force to exert on the string to make the ball to move in a circular motion as the ball velocity is changing all the time? A traditional way to analyze the pendulum never mentions this. By a careful examination, we will find actually the string must be stretched a little bit to provide the extra force as the centripetal force. This is what we do in this simulation.

VI. Homework

write a program for Newton's cradle with 5 balls. The dimensions of the cradle and the initial conditions, in which all balls are at rest, are shown in the figure below. The mass of each ball is 1kg, and the radius of each ball is 0.2m and the distance between any two adjacent pivots are 0.4m. The program need to have a variable N that indicates how many balls are lifted at the beginning, for example in the figure N=2. In the program, also use dt, k, g = 0.0001, 150000, vector(0,-9.8,0) for the time step of the simulation, the force constant of the rope, and the gravitation acceleration, respectively. And the simulation is set at rate(5000).

Also (1) In one graph, plot versus time both the total of the instant kinetic energies of all balls, and the total of the instant gravitational potential energy of all balls (the potential energy of the balls at the lowest positions are taken to be 0). (2) In a second graph, plot versus time the averaged total kinetic energy over the time period (from the beginning of the simulation till the time of the current plot point) and the averaged total gravitational potential energy.