VP4: Collision and pendulum [function]

I. 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

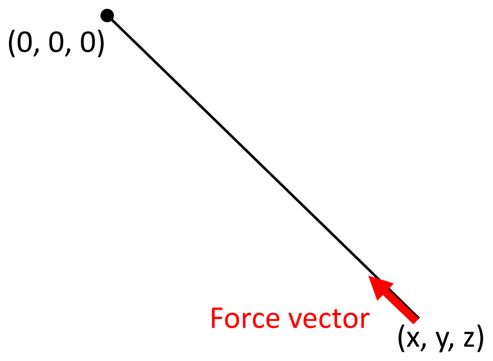
```
V1, V2 = after_col_v(v1= vec(1, 2, 3), v2= vec(4, 5, 6), m2= 0.6,  x1= vec(1, 0, 1), x2=vec(0, 1, 1), m1 = 0.5 )
```

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

II. 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 homework 3 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 spring 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.

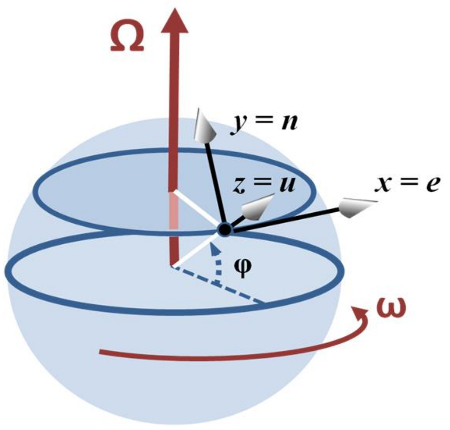
This simulation is more "fundamental" than the pendulum analysis in the high school physics. The ball is exerted by two forces, one gravitational force and the other is the tension from the string. As the ball is swinging at the bottom of the string, it is doing a circular motion that requires a centripetal 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.

III. Homework

(must) Combine I. and II., and the list in VP3 to write a program for Newton's cradle with 5 balls. The program need to have a variable N that indicates how many balls (1 - 4) are lifted at the beginning.

(optional) (continue from II. Pendulum)
We can add Coriolis force to the pendulum system. We assume the pendulum is set at latitude of φ, and the direction to the east is toward + x, the direction to the north is toward –z, and the direction to the sky is toward +y (Notice: it is different from the following figure). For a quicker and better observation, we can let the simulation run at rate(10000). If we project the swing onto the x-z plane, we will see the motion of the projection similar to a harmonic oscillation, originally oscillating in x-axis. Find the angle deviation of the swing projection from its original direction (+x axis) after 1000 periods of the pendulum if the pendulum is in Taipei and of 2m long and with a still start angle at 30 degrees.

Notice: When we have a coordinate attached on a rotating sphere as in the figure, the Coriolis acceleration is shown as the equation below the figure, where v_e, v_u, and v_n are the velocity to the east, to the sky, and to the north.



https://upload.wikimedia.org/wikipedia/commons/8/8a/Earth_coordinates.PNG

$$\boldsymbol{a}_C = -2\boldsymbol{\Omega} \times \boldsymbol{v} = 2\,\omega \begin{pmatrix} v_n \sin\varphi - v_u \cos\varphi \\ -v_e \sin\varphi \\ v_e \cos\varphi \end{pmatrix}$$