1. A ball in a box

The goal of the following activity is to give you experience in using the momentum to update the position of an object, to create a 3D animation. You will write a program to make a ball bounce around in a box, in 3D. To position objects in the display window we use their 3D coordinates. The origin of the coordinate system is at the center of the display window. The positive \( x \) axis runs to the right, the positive \( y \) axis runs up, and the positive \( z \) axis comes out of the screen, toward you.

Start a new program like the following, which displays a ball and a wall (we’ve called it wallR because it is on the right side of the scene). Read the program carefully and make sure you understand the relationship between these statements, the coordinate system, and the display generated by the program.

```python
from visual import *
ball = sphere(pos=(-5,0,0), radius=0.5, color=color.cyan)
wallR = box(pos=(6,0,0), size=(0.2,12,12), color=color.green)
ball.momentum = vector(25,0,0)
deltat = 0.005
mass = 1
t = 0
while t < 3:
    ball.pos = ball.pos + (ball.momentum/mass)*deltat
    t = t + deltat
```

We use the “position update” relationship among position, momentum, and time interval, that the final (vector) position is the initial position plus the velocity times the time interval. \( \vec{r}_f = \vec{r}_i + (\vec{p}/m)\Delta t \). In Python this assignment statement reads: `ball.pos = ball.pos + (ball.momentum/mass)*deltat`. Since we are using the `while t < 3` condition to check for completion of the loop we also need to keep track of the time. To do so we create the variable \( t \) initialized to zero. We updated the time at each step in the loop. You can think of \( t = t + \text{deltat} \) as “time update.”

> Run your program.

Depending on the speed of your computer, the ball may have moved so fast that you saw only a flash!

> To slow down the animation, insert the following statement inside the loop (just after the `while` statement, indented as usual):

```python
rate(100)
```

This specifies that the `while` loop will not be executed more than 100 times per second. You should see the ball move to the right more slowly. However, it keeps on going right through the wall, off into empty space, because this is what it was told to do. VPython doesn’t know any physics! You have to tell it what to do.

A. Making the ball bounce: Logical tests
To make the ball bounce off the wall, we need to detect a collision between the ball and the wall. A simple approach is to compare the $x$ coordinate of the ball to the $x$ coordinate of the wall, and reverse the $x$ component of the ball’s momentum if the ball has moved too far to the right. In VPython you can access the $x$, $y$, or $z$ component of any vector by referring to the $x$, $y$, or $z$ attribute of that vector. In the statements below, `ball.pos` is a vector, and `ball.pos.x` is the $x$ component of that vector. Similarly, `ball.momentum` is a vector, and `ball.momentum.x` is the $x$ component of that vector.

> Insert these statements into your while loop, just before your position update statement

```python
if ball.pos.x > wallR.pos.x:
    ball.momentum.x = -ball.momentum.x
```

The indented statement after the `if` statement will be executed (and reverse the $x$ component of momentum) only if the logical test in the previous line gives `True` for the comparison. If the result of the logical test is `False` (that is, if the $x$ coordinate of the ball is not greater than the $x$ coordinate of the wall), the indented line will be skipped. We want this logical test to be performed every time the ball is moved, so we need to indent both of these lines, so they are inside the `while` loop.

Your program should now look like this:

```python
from visual import *
ball = sphere(pos=(-5,0,0), radius=0.5, color=color.cyan)
wallR = box(pos=(6,0,0), size=(0.2,12,12), color=color.green)
ball.momentum = vector(25,0,0)
deltat = 0.005
mass = 1
t = 0
while t < 3:
    rate(100)
    if ball.pos.x > wallR.pos.x:
        ball.momentum.x = -ball.momentum.x
    ball.pos = ball.pos + (ball.momentum/mass)*deltat
    t = t + deltat
```

> Run your program.

You should observe that the ball moves to the right, bounces off the wall, and then moves to the left, continuing off into space. Note that our test is not very sophisticated; because `ball.pos.x` is at the center of the ball and `wallR.pos.x` is at the center of the wall, if you look closely you can see that the ball appears to penetrate the wall slightly. You could if you wish make the test more precise by using the radius of the ball and the thickness of the wall.

> Add another wall at the left side of the display, and give it the name `wallL`. Make the ball
bounce off that wall also.

> Next, before the while loop, change the initial momentum to have a nonzero y component:

\[ \text{ball.momentum} = \text{vector}(25, 5, 0) \]

When you run the program, the ball bounces even where there is no wall! Again, the issue is that VPython doesn’t know any physics.

**B. Visualizing momentum**

We will often want to visualize vector quantities, such as the ball’s momentum. We can use an `arrow` object to visualize the momentum of the ball.

> Before the `while` loop, but after the program statement that sets the ball’s momentum, create an arrow, which you will use to visualize the momentum vector for the ball. The tail of the arrow is at the location given by `pos`, and we choose that to be the location of the ball. The tip of the arrow is at the location that is a vector displacement `axis` away from the tip (in this case, the location of the ball plus the momentum of the ball).

\[ \text{varr} = \text{arrow(pos=ball.pos, axis=ball.momentum, color=color.yellow)} \]

It is important to create the arrow before the `while` loop. If we put this statement in the indented code after the `while`, we would create a new arrow in every iteration. We would soon have a large number of arrows, all at the same location. This would make the program run very slowly.

> Run your program.

You should see a yellow arrow with its tail located at the ball’s initial position, pointing in the direction of the ball’s initial momentum. Let’s scale down the size of the arrow, by multiplying by a “scalar”, a single number. Multiplying a scalar times a vector changes the magnitude of a vector but not its direction, since all components are scaled equally. Change your arrow statement to use a scale factor to scale the axis, like this, then run the program:

\[ \text{vscale} = 0.1 \\
\text{varr} = \text{arrow(pos=ball.pos, axis=vscale*ball.momentum, color=color.yellow)} \]

Run the program. Now the arrow has a reasonable size, but it doesn’t change when the ball moves. We need to update the position and axis of the arrow every time we move the ball.

> Inside the `while` loop, update the position and axis attributes of the arrow named `varr`, so that the tail of the arrow is always on
the ball, and the axis of the arrow represents the current vector momentum. Remember to use the scale factor \( vscale \) to scale the axis of the arrow. Run the program.

The arrow representing the ball’s momentum should move with the ball, and should change direction every time the ball collides with a wall.

**C. Leaving a trail**

Often we are interested in the trajectory of a moving object, and would like to have it leave a trail. We can make a trail out of a `curve` object. A `curve` is an ordered list of points, which are connected by a line (actually a thin tube). We’ll create the `curve` object before the loop, and append a point to it every time we move the ball.

> After creating the ball, but before the `while` loop, insert the following statement to create a `curve` object

```python
ball.trail = curve(color=ball.color)
```

This creates a `curve` object, associated with the ball, whose color is the same as the color of the ball, but without any points in the curve as yet.

> Inside the while loop, after updating the ball’s position, add this statement:

```python
ball.trail.append(pos=ball.pos)
```

This statement appends a point to the trail. The position of the added point was chosen to be the same as the current position of the ball. Run your program. You should see a cyan trail behind the ball.

**D. Making the ball bounce around inside a box**

With the program you’ve written so far, the ball escapes and bounces off nothing. To make a more realistic model of the motion, do the following:

- Add top and bottom walls, and make the ball bounce off these walls.
- Make the walls touch, forming part of a large box.
- Add a back wall, and an “invisible” front wall, and make the ball bounce off these walls. Do not draw a front wall, but include an `if` statement to prevent the ball from coming through the front.
• Give your ball a component of momentum in the $z$ direction as well, so that it will bounce off the back and front walls. Make the initial momentum of the ball be this:

$$\text{ball.momentum} = \text{vector}(25,5,15)$$

The completed program should include these features:
• Complete box (except that the front is open).
• Correct initial momentum.
• Continuous display of momentum arrow that moves with the ball.
• Ball leaves a trail.

It is worth pointing out that this program represents a simple model of a gas in a container. The pressure on the walls of the container is due to the impacts of large numbers of gas molecules hitting the walls every second.