3. MOTION IN 3D AND ANIMATIONS

At this point you should ask the following: “hey, we just did motion in one dimension and now we are in three dimensions. Did you skip a chapter?”

Great question. No, I didn’t skip anything. In order to look at the motion of an object in two dimensions, we are going to need vectors. If you are going to use 2D vectors, you might as well use 3D vectors. That way, we can also make 3D animations. This is going to be great fun.

VECTORS

I know what you are going to say, “a vector has magnitude and direction”. I mean, that’s not wrong—but that’s not the version I like. Here’s how I describe vectors. My preferred definition is that a vector is a variable with more than one number. Three dimensional vectors have three numbers. In Cartesian coordinates, we can describe a position as the following vector (I will write it three different ways).

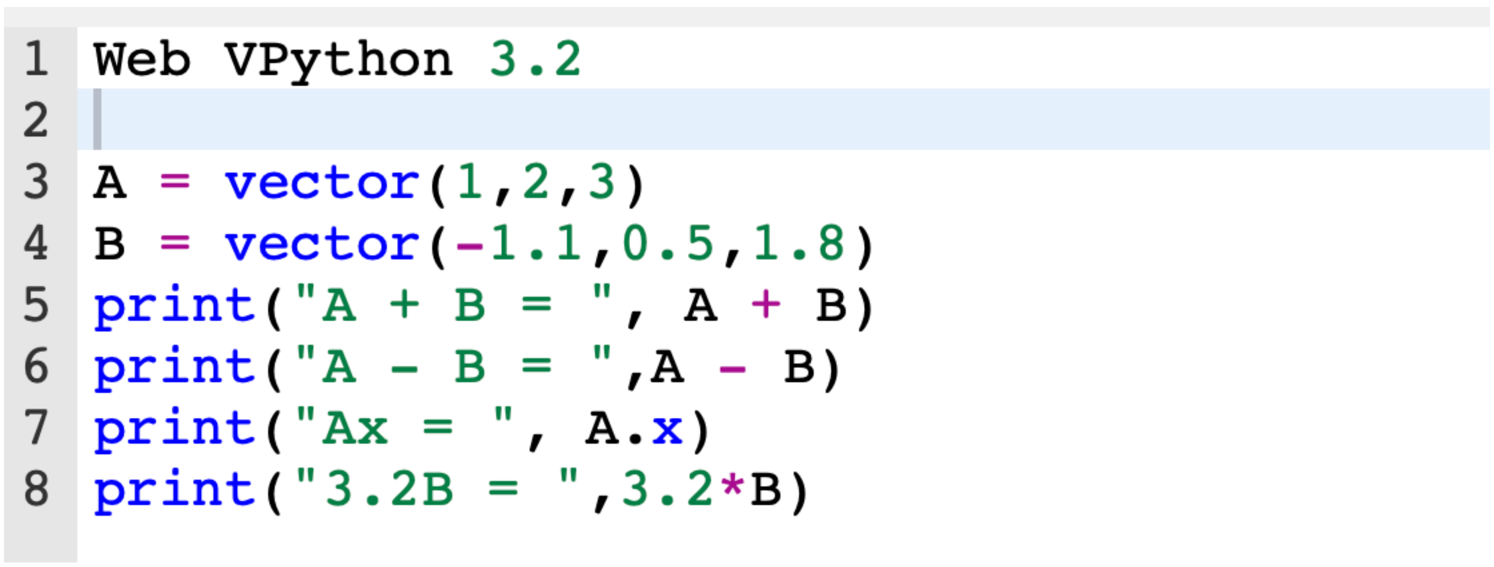
$$\vec{r} = x\hat{x} + y\hat{y} + z\hat{z}$$

$$\vec{r} = x\hat{i} + y\hat{j} + z\hat{k}$$

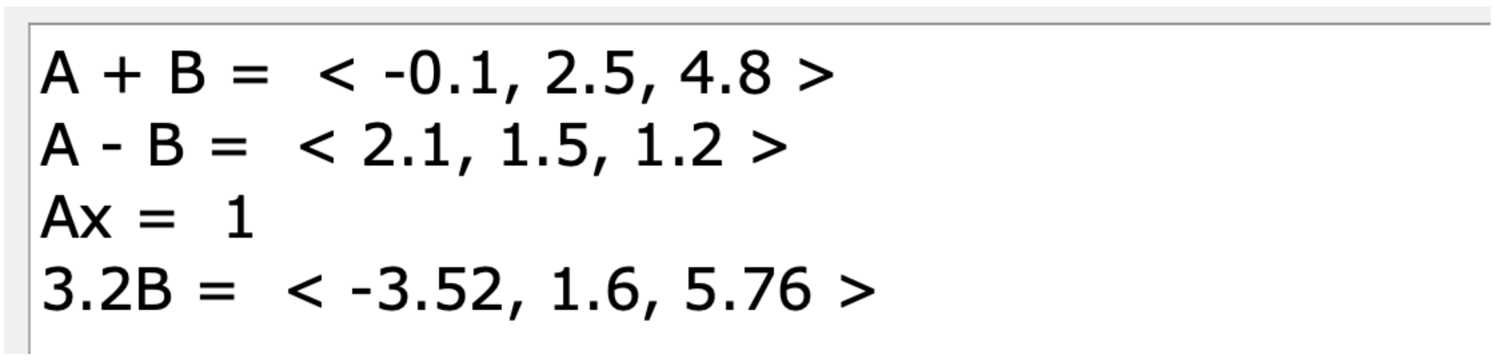
$$\vec{r} = < x, y, z >$$

These are all the same things, but I prefer the last version.

If we want to animate objects in 3D, we obviously need three dimensional vectors. Good news! Vectors are built into VPython. Here’s some quick vector stuff in VPython.



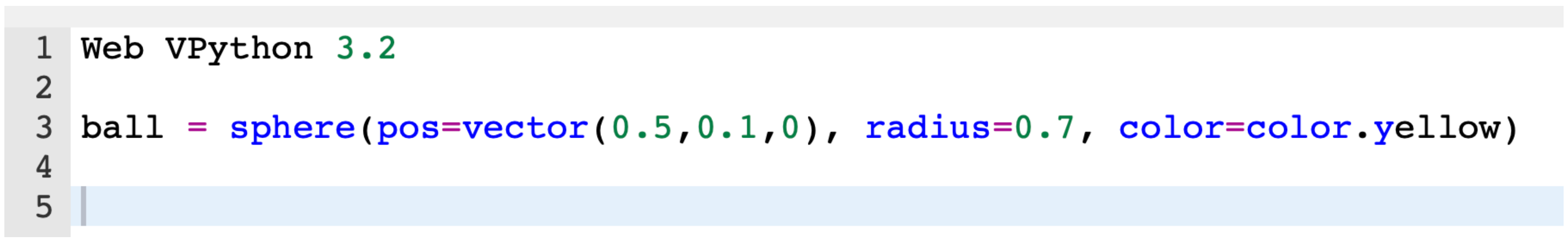
```
Web VPython 3.2

A = vector(1,2,3)
B = vector(-1.1,0.5,1.8)
print("A + B = ", A + B)
print("A - B = ",A - B)
print("Ax = ", A.x)
print("3.2B = ",3.2*B)
```


```
A + B =  < -0.1, 2.5, 4.8 >
A - B =  < 2.1, 1.5, 1.2 >
Ax =  1
3.2B =  < -3.52, 1.6, 5.76 >
```

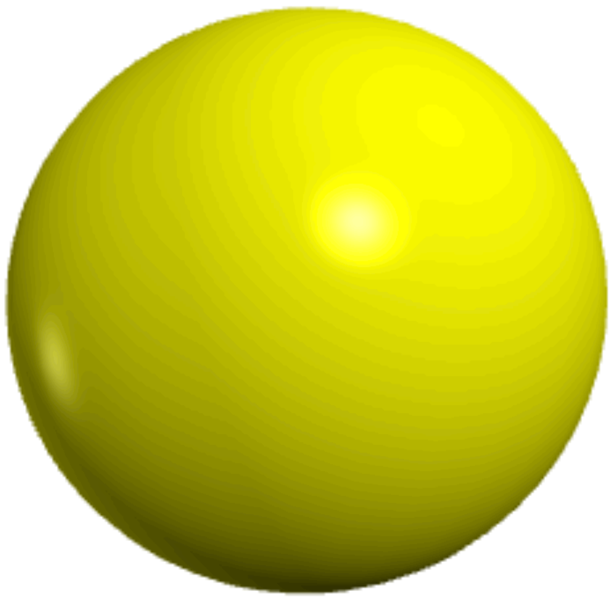
Of course there's much more stuff with vectors, but this is enough for now. Let's make our first 3D program.

SPHERE

This is a very simple program to make a 3D object. It looks like this.


```
Web VPython 3.2

ball = sphere(pos=vector(0.5,0.1,0), radius=0.7, color=color.yellow)

```

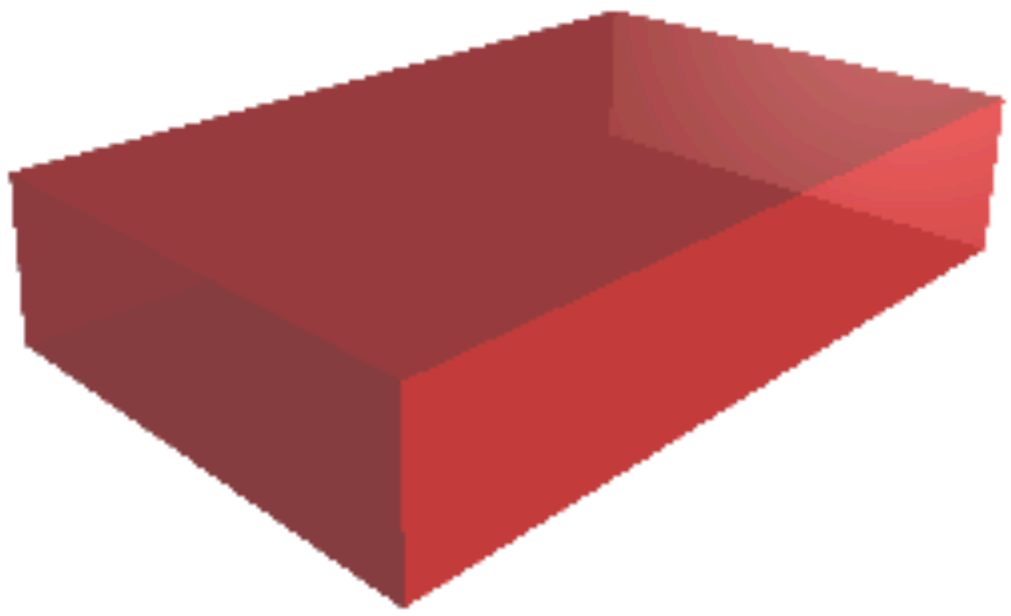
When you run it, this is what you get.



I lied. That's not what you get. The default background is black, but I changed it with the line "canvas(background=color.white). Just in case someone prints this stuff out, I didn't want a bunch of black ink going to waste. I lied twice actually. The other untruth is that you don't get a picture. You get an interactive 3D object. You can scroll your mouse to zoom in and out and rotate the view with a right-click drag. That's a 3D program in such simple code. Trust me, this is awesome.

BOX

I want to quickly show you one more 3D object and then we are going to make a program with some physics. Here is the box.



The code for the simple box program.

```
Web VPython 3.2
canvas(background=color.white)

thing = box(pos=vector(0.1,-0.1,0), size=vector(0.5,0.1,0.3),
color=color.red, opacity=0.5)

```

There you can see the code to make the background white (I don't want to hide any of my secret tricks). The box is a built in 3D object in VPython. The two important properties of the box are the position (pos) and the size. The position is the vector location of the center of the box. The size is a vector that gives the length (x), height (y) and width (z) for the box.

Bonus—here I made the box red with an opacity of 0.5. This makes it partly transparent so you can see the dimensions and stuff. Oh, quick note. If I have box with pos=vector(0,0,0) then the top of the box is NOT at y = 0. Just be sure that if you want a floor to be at your = 0, you need to move your position down a little bit.

There's a bunch of other objects and things you can do to them, but let's get back to physics.

TOSSED BALL

Suppose you have a basketball with a mass of 400 grams (I'm just guessing the mass—I'm not a basketballologist). You throw the ball straight up with a velocity of 5 meters per second. Can we model this? Yes, we can.

First, some quick physics (as a reminder). Once the ball leaves the hand, there's essentially only one force acting on it—the gravitational force. Yes, we could also include air resistance and WE WILL in a later chapter. We can calculate that gravitational force as:

$$\vec{F}_g = m\vec{g}$$

Where $\vec{g}$ is the gravitational field. On the surface of the Earth, this has a value:

$$\vec{g} = <0, -9.8, 0> \text{ N/kg}$$

Just a quick note—$\vec{g}$ is a vector. If we want to look at the magnitude of this vector, it would just be g and not negative g. Instead, the y-component of $\vec{g}$ is in the negative direction. OK, just wanted to clear that up.

Now, let's look at Newton's second law.

$$\vec{F}_{\text{net}} = m\vec{a}$$

Since the only force is gravity, we get:

$$\vec{F}_{\text{net}} = m\vec{g} = m\vec{a}$$

$$\vec{a} = \vec{g}$$

The acceleration of a free falling object is the same as the gravitational field (which also has units of meters per second per second). But now we are essentially back to the same methods that we used in the previous chapter—constant acceleration.

But wait! There's one big difference. Now we have the acceleration as a vector. Velocity and position are also both vectors. It's fine. It's fine. Let's go back through our numerical model. Here is the vector definition of acceleration over some short time interval (Δt).

$$\vec{a} = \frac{\Delta \vec{v}}{\Delta t} = \frac{\vec{v}_2 - \vec{v}_1}{\Delta t}$$

From this we get the velocity update formula.

$$\vec{v}_2 = \vec{v}_1 + \vec{a}\Delta t$$

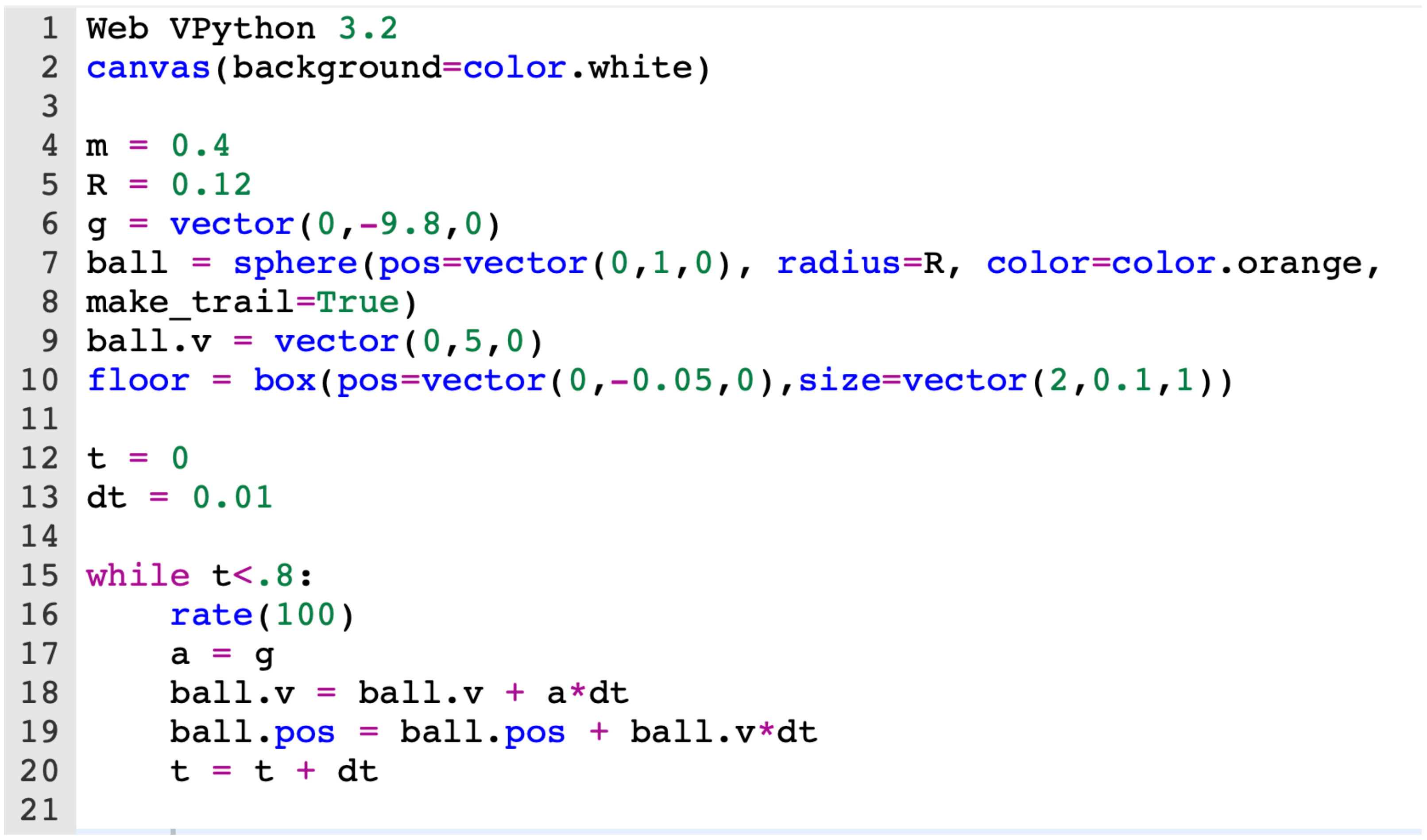
Next, our vector definition of velocity.

$$\vec{v}_2 = \frac{\Delta \vec{r}}{\Delta t} = \frac{\vec{r}_2 - \vec{r}_1}{\Delta t}$$

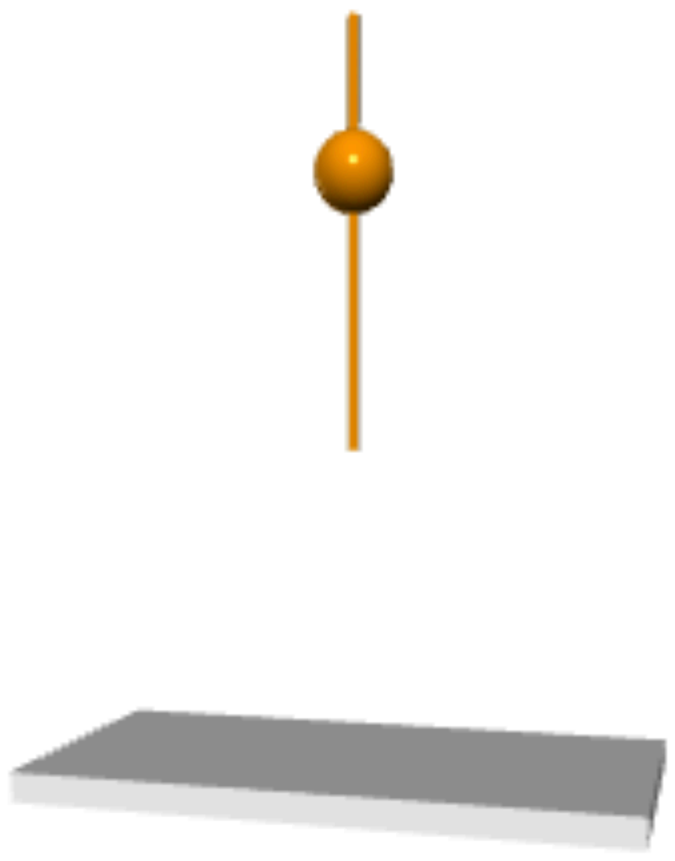
The position is the vector $\vec{r}$ so that the new position (and the position update formula) would be:

$$\vec{r}_2 = \vec{r}_1 + \vec{v}_2\Delta t$$

See, it doesn't matter that we have vector values for position and velocity. Stuff still works. OK, let's model this basketball. Here's the code and output followed by comments.



```
Web VPython 3.2
canvas(background=color.white)

m = 0.4
R = 0.12
g = vector(0,-9.8,0)
ball = sphere(pos=vector(0,1,0), radius=R, color=color.orange,
make_trail=True)
ball.v = vector(0,5,0)
floor = box(pos=vector(0,-0.05,0),size=vector(2,0.1,1))

t = 0
dt = 0.01

while t<.8:
    rate(100)
    a = g
    ball.v = ball.v + a*dt
    ball.pos = ball.pos + ball.v*dt
    t = t + dt

```



The output in this text doesn't look nearly as cool as an actual animation. But if you want to see that, you are going to have to actually to one of the implementation of web VPython and enter the code and run it (which you probably should do anyway).

Now for some very important comments about the code.

- Line 8: One of the best properties of the sphere() object is make_trail=True. When this option is used, the ball will leave a trail. I hope that's not very surprising. Sometimes code makes sense. Oh, notice that I wrapped the line so it's still part of the sphere function.

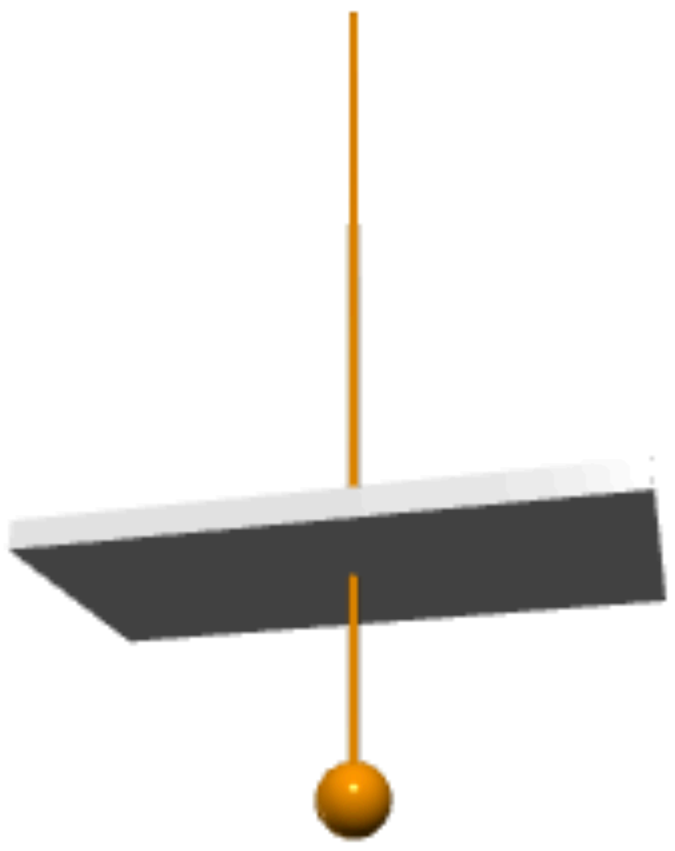
- Line 9: Instead of making a separate variable for the velocity of the ball, this line puts it as a property of the ball object. It doesn't really make a difference now, but using object properties will be a huge help if you have a bunch of balls moving around. Oh, but this is the initial velocity and it's a vector (obviously).

- Line 10: The floor is a box. I wanted the top of the floor to be at y = 0 so that I had to put the position (the center of the box) down lower.

- Line 16: Since we are making an animation (that you can't see in a book unless I made it one of those page-flip animations), we need to control the speed of the animation. You can't just say "oh, use the time scale"—because recall that the computer ONLY looks at the numbers. It has no idea what they actually mean (that's the human's job). So, we have this rate(100) statement. This tells VPython to only do 100 loops every second (if it can). If you want to run the animation faster, increase the number. If you want it to run slower, decrease the number. If you can't remember what the rate(100) does, just change it and see what happens.

- Line 17: This line looks dumb, but it has a purpose. Yes, the acceleration is equal to g and that doesn't change. However, I'm just setting this up for future calculations where the acceleration might not be constant.
- Line 18: Update the velocity. Remember, we have the velocity as a property of the ball.
- Line 19: Update the position. Here, we aren't using x like we did in 1D. Instead the vector position is ball.pos—yes, a property of the ball object. By changing the ball's position, it will move in the 3D space.

It's a whole new world when dealing with 3D stuff. Notice that I ran the code for 0.8 seconds. What happens if it's run for a little bit longer? Here's the ball after 1.3 seconds.



Notice that it went RIGHT THROUGH THE FLOOR! Do you know why? It's because there is no floor. It's just python code, it's not The Matrix (which might also be in python—

I don't know). If we don't model some interaction between the ball and floor, then there's no interaction. The ball just moves right along just like we told it it.

Homework

1. What if the ball starts at y = 2.5 m (above the floor) with the same initial velocity—how long would it take to reach the floor?

2. Physics textbooks state that when you toss a ball up with some initial velocity, it will be going just as fast (but in the opposite direction) when it gets back to the starting point. Can you check that by modifying the code?

3. How high would you have to jump to have a hang time of 2 seconds? You can modify the code and let the ball be you. You be the ball. The ball is you.

Answer key

1. It's easy to start the ball at a different position, but how long do you run the loop? Of course, the simple answer is to just keep changing the final time in the while loop until the ball ends at the correct place. This solution might seem barbaric, but it's fine. The code serves the human, not the other way around. All too often people are afraid to solve things because it's not the best or most efficient way. Don't worry about that.

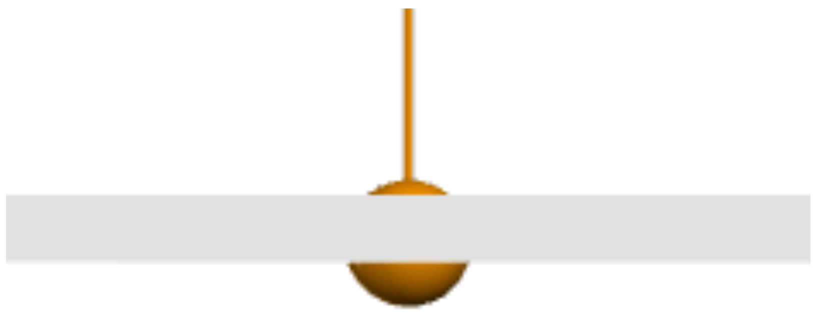
Of course I'm still going to show you a better way. Here's my code.

```
m = 0.4
R = 0.12
g = vector(0,-9.8,0)
ball = sphere(pos=vector(0,2.5,0), radius=R, color=color.orange,
make_trail=True)
ball.v = vector(0,5,0)
floor = box(pos=vector(0,-0.05,0),size=vector(2,0.1,1))

t = 0
dt = 0.01

while ball.pos.y>0:
    rate(100)
    a = g
    ball.v = ball.v + a*dt
    ball.pos = ball.pos + ball.v*dt
    t = t + dt
print("t = ",t," sec")

```

Really, the only big difference is in the while loop. Instead of the usual "while t<1.2:" or whatever—I'm using the ball's position as the condition of the loop. WARNING: if you aren't careful, it's very possible to make a loop that goes on forever.

In this case, the loop runs as long as the ball's y-position is greater than zero. Check that out. You can't compare a vector (the position) and the scalar value zero. So, instead I'm just looking at the y-component of the vector ball.pos. Yes, if you call "ball.pos.y" you just get the y-component of the vector. Very useful.

Oh, I print the final time. The answer is 1.39 seconds. Wait, check this out. If you look at the animation when it ends, you get this.



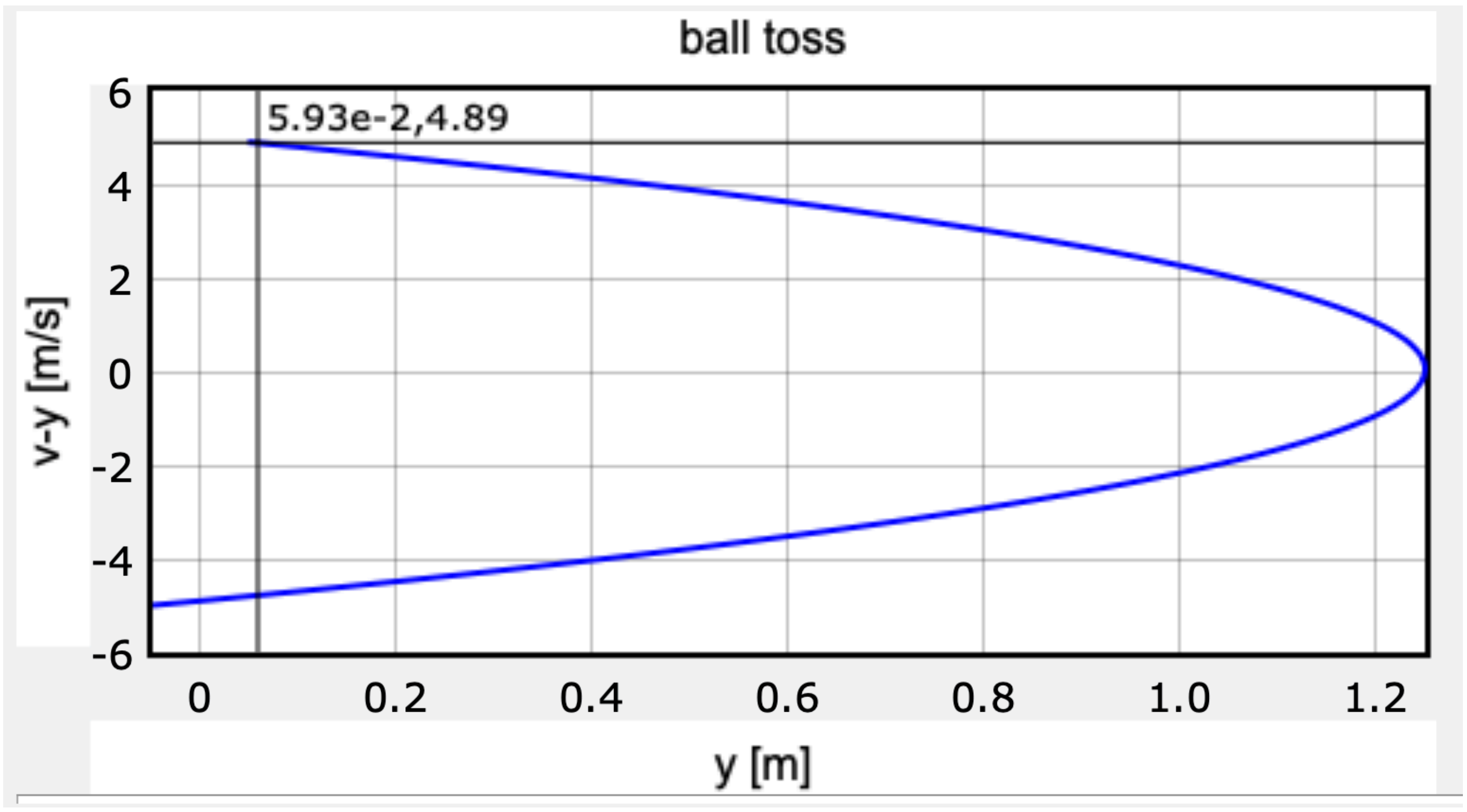
Notice that the ball "pokes" through the floor. That's because the position of the ball is less than zero and that's what stops the code. Remember, the position of the ball is the center of the sphere, not the bottom. It's possible to make this look better, but this is fine for now.

Once you understand the basics of narrative, you don't have to follow all of the rules. In fact, some of your favorite books that you enjoy as a reader might break away from typical narrative structure. As the writer, you have the freedom to structure your book in whatever way feels most powerful and effective, even if that means breaking some of the rules.

2. You forgot this was the answer key. It's cool, the previous answer was super long. If you start the ball at y = 0 and run it while y>0, it's not going to work because the ball starts at zero. You can fix this with "while ball.pos.y >= 0" (greater than OR equal to). Then just print the final velocity at the end of the loop.

Or, how about a graph? Here's some code to plot the velocity (y-component) as a function of y-position. If you already forgot the graph stuff, it's OK. Just look back at the previous stuff. You aren't going to get better at coding by just looking at stuff though, you need to get out the and make some coding errors to see some improvements.

Here's the output.



You can see that for any value of y, the ball has a positive and negative velocity with the same magnitude. Yes, the curve ends weird—that's the same thing as the ball moving through the floor just a little bit.

3. How do you get a hang time of 2 seconds? If you want to solve this with python, perhaps the most straightforward method is to use the code for question 2 and just change the initial velocity until you get a total time of 2 seconds. Then you can look at the graph to see the maximum height. If you have a launch speed of 10 m/s, the ball will travel a height of 5 meters. No one can jump that high. Right?

PROJECTILE MOTION

Vertical motion is boring. What about throwing a ball at an angle? That would be cool, but we've already done that. Yes, if you take the falling ball code above—you can make it a projectile motion model by just changing the initial conditions. OK, we should probably make some other changes—like the size of the "floor".

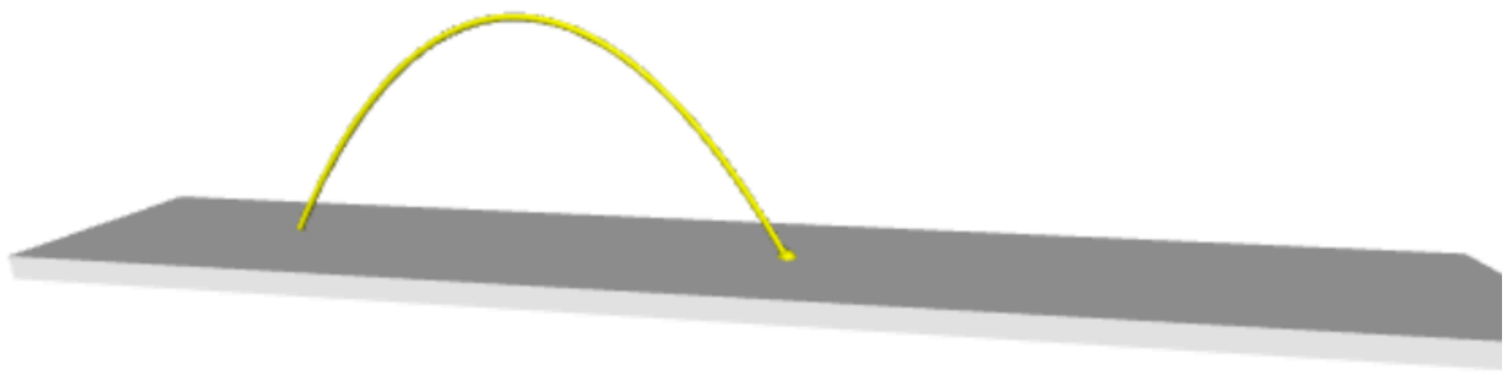
Let's throw a tennis ball with an initial velocity of 5 m/s launched at an angle of 60 degrees above the horizontal. Here's the code.

```
 4 g = vector(0,-9.8,0)
 5 ball = sphere(pos=vector(-2,0,0), radius=0.04, color=color.yellow,
 6 make_trail=True)
 7
 8 ground = box(pos=vector(0,-0.07,0), size=vector(6,0.1,1))
 9 v0=5
10 theta = 60*pi/180
11 ball.m = 0.06
12 ball.v = v0*vector(cos(theta),sin(theta),0)
13
14 t = 0
15 dt = 0.01
16
17 while ball.pos.y>=0:
18     rate(100)
19     ball.v = ball.v + g*dt
20     ball.pos = ball.pos + ball.v*dt
21     t = t + dt
```

What's new here? You know I like to give comments.

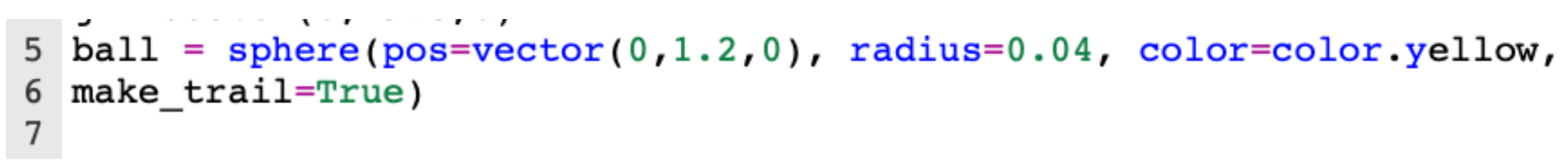
- Notice that the ball starts at x = -2. This just makes the motion take up more of the screen. You don't actually have to do that.
- For the initial velocity vector of the ball, I need a magnitude an angle. I'm using a launch angle of 60 degrees (line 10), but that needs to be converted to radians.
- The launch velocity vector will then have an x-component of $v_x = v_0 \cos\theta$ and a vertical component of $v_y = v_0 \sin\theta$.
- For the while loop, I'm running it as long as the y-position of the ball is greater than or equal to 0 (the floor).

Here's what it looks like.



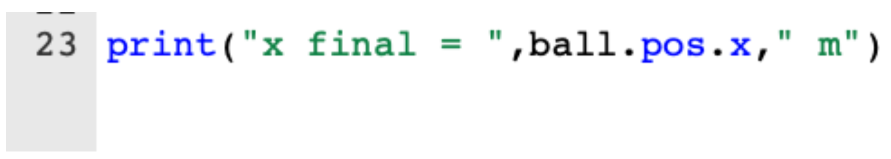
That's pretty. Oh, but what about a ball launched off the floor? This is one of those classic intro physics problems. Suppose the ball is launched at a 60 degree angle with a velocity of 5 m/s that starts 1.2 meters above the ground. How far horizontally will it travel?

You can solve this with two small modifications to the above code. Here's the first thing.



```
ball = sphere(pos=vector(0,1.2,0), radius=0.04, color=color.yellow,
make_trail=True)
```

I put the starting location of the ball at vector(0, 1.2, 0). Then I need this at the end of the code.



```
print("x final = ",ball.pos.x," m")
```

This prints out the final x-position (you get 2.75 meters). Isn't that simple? Just a little note—later, we will create code to find the angle that gives the ball the maximum range. It's going to be great.

BALL WITH AIR RESISTANCE

What if we replace the tennis ball with one that has the same radius, but a much lower mass? Suppose that it's only 5 grams. In that case, it seems unreasonable to ignore the air resistance force on the object. OK, well then let's add it into our calculations.

I'm going to use the following model for the air drag.

$$\vec{F}_{\text{drag}} = -\frac{1}{2}\rho A C|\vec{v}|^2\hat{v}$$

In this model (it's just a model but it will work fine here), we have the following:

- The density of air: ρ
- The cross sectional area of the object: A
- A drag coefficient that depends on the shape of the object: C

- The square of the magnitude of the velocity: $|\vec{v}|^2$
- A unit vector in the direction of the velocity: $\hat{v}$

Let's talk about magnitudes and unit vectors. Suppose we have a vector:

$$\vec{A} = < A_x, A_y, A_z >$$

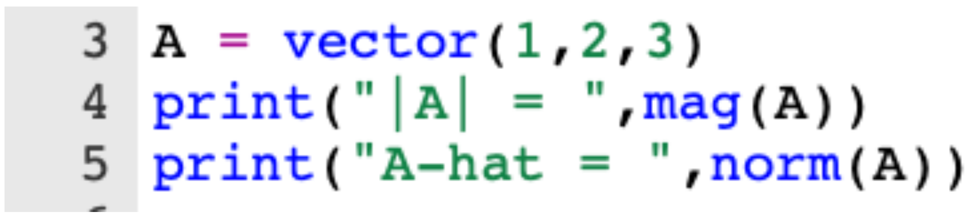
We can find the magnitude of this vector as:

$$|\vec{A}| = A = \sqrt{A_x^2 + A_y^2 + A_z^2}$$

The unit vector is defined as:

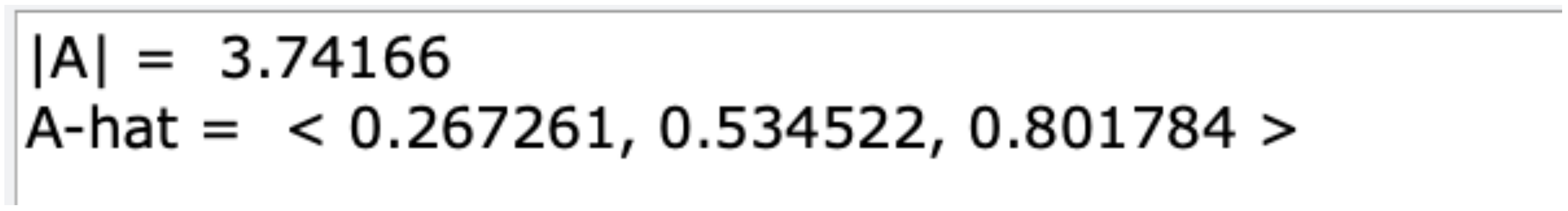
$$\hat{A} = \frac{\vec{A}}{|\vec{A}|}$$

In VPython we can easily calculate both the magnitude and unit vector with built in functions.



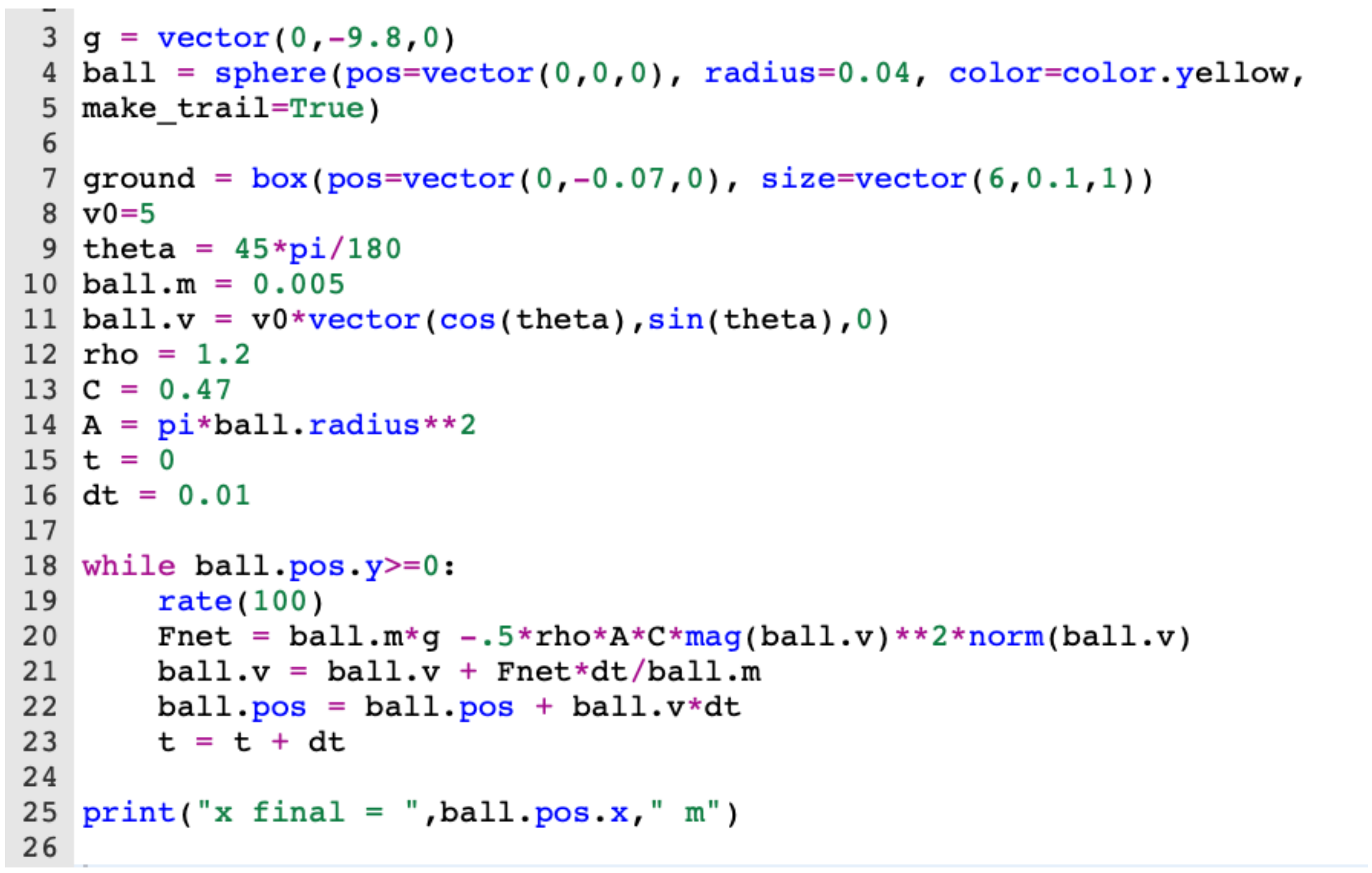
```
A = vector(1,2,3)
print("|A| = ",mag(A))
print("A-hat = ",norm(A))
```

Here's the output.



```
|A| =  3.74166
A-hat =  < 0.267261, 0.534522, 0.801784 >
```

Now we can go back to our projectile motion code and make some small modifications. Check it out.



```
 3 g = vector(0,-9.8,0)
 4 ball = sphere(pos=vector(0,0,0), radius=0.04, color=color.yellow,
 5 make_trail=True)
 6
 7 ground = box(pos=vector(0,-0.07,0), size=vector(6,0.1,1))
 8 v0=5
 9 theta = 45*pi/180
10 ball.m = 0.005
11 ball.v = v0*vector(cos(theta),sin(theta),0)
12 rho = 1.2
13 C = 0.47
14 A = pi*ball.radius**2
15 t = 0
16 dt = 0.01
17
18 while ball.pos.y>=0:
19     rate(100)
20     Fnet = ball.m*g -.5*rho*A*C*mag(ball.v)**2*norm(ball.v)
21     ball.v = ball.v + Fnet*dt/ball.m
22     ball.pos = ball.pos + ball.v*dt
23     t = t + dt
24
25 print("x final = ",ball.pos.x," m")
26
```

What's new here?

- Line 12-14: Of course I had to add values for the parameters in the air drag model. The density of air is about 1.2 kg/m^3 and I can calculate the area as the circular size of the sphere (not the surface area of a sphere). Finally, a sphere has a drag coefficient of about 0.47.

- Line 20: In the loop, I need to calculate the force during each time interval. Notice that I have the gravitational force (that should be fine). For the drag force, I have the same calculation as the drag model using mag(ball.v) and norm(ball.v).

Now it should run.

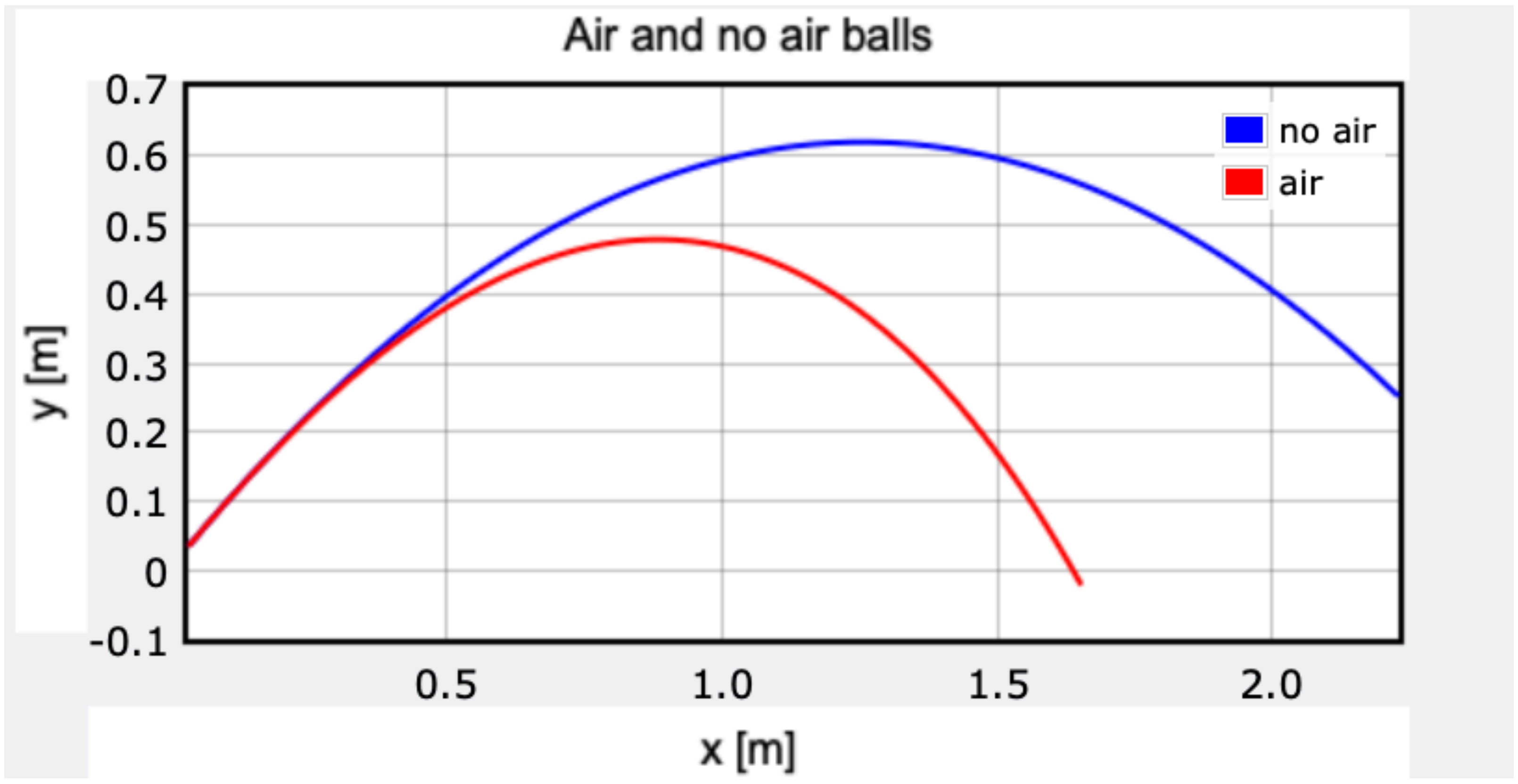
Homework

1. Modify the code so that there are two balls with the same initial conditions—but one of the balls has no air drag. Plot the trajectory for both balls.

2. For a ball with air resistance, try the following launch angle 43 degrees and 45 degrees. Which angle produces a greater range? Hint: it's not 45.

Key

1. Here's the output. I think you can figure out the code for this one.



2. We will make some cool code to solve this problem later.