2. NON-CONSTANT FORCES AND THE SIMPLEST SIMPLE HARMONIC OSCILLATOR

I need to keep reminding myself that this isn't a physics textbook. I'm making some assumptions about what you already know. Maybe you are learning physics from this book - well, that's just peachy (which means great). However, if something goes too fast—just take a break and find some nice YouTube videos or a physics textbook to help you.

With that in mind let's use our new numerical calculation superpowers to do something super. Yes, it's time for a mass on a spring. I'm going to do this as an example of the power of python.

Let's start with a super simple case. Suppose I have a mass (m) connected to a spring with a spring constant (k) in some horizontal situation without any friction.

In this case, we can say that at the equilibrium position where the spring doesn't exert a force, the mass is at x = 0 meters. As you pull the mass in the positive x-direction, the spring will pull back in the negative direction. We can write the force due to the spring as:

$$F = -kx$$

Note: this is ONLY true in this situation that we set up. We will make a much better spring model later. If the mass is confined to the x-direction, then the net-force in the y-direction must be zero. We can just focus on the motion in the x-direction. Using Newton's second law, we get the following.

$$F_{\text{net}-\text{x}} = -kx = ma$$

Let's solve this expression for the acceleration (which is technically the acceleration in the x-direction).

$$a = -\frac{k}{m}x$$

Now we have a problem that isn't so trivial to solve (it's obviously not impossible). But here we have a non-constant acceleration. You can't use the kinematic equations to find the position of the mass since the acceleration keeps changing. Bummer.

Don't be surprised when I tell you that we can solve this problem numerically. I mean, read the room. Right? So, here's the plan. I'm going to break start with a mass at some initial position (x_0) with an initial velocity (v_0). Over some short time interval (Δt) the acceleration is constant. That's the case, I use the definition of acceleration as the change in velocity with respect to time to solve for the velocity at end of the time interval.

$$a = \frac{\Delta v}{\Delta t} = \frac{v_2 - v_1}{\Delta t}$$

$$v_2 = v_1 + a\Delta t$$

Now we are right back to the previous chapter. I can use this velocity to update the position.

$$x_2 = x_1 + v_2\Delta t$$

Let's do it. Of course for a numerical calculation, we need numbers. How about the following:

- Mass: $m = 0.1$kilograms.
- Starting position: $x_0 = 0.05$ meters
- Initial velocity: $v_0 = 0$ meters per second.
- Spring constant $k = 10$ Newtons per meter.
- Time interval: $\Delta t = 0.01$seconds.

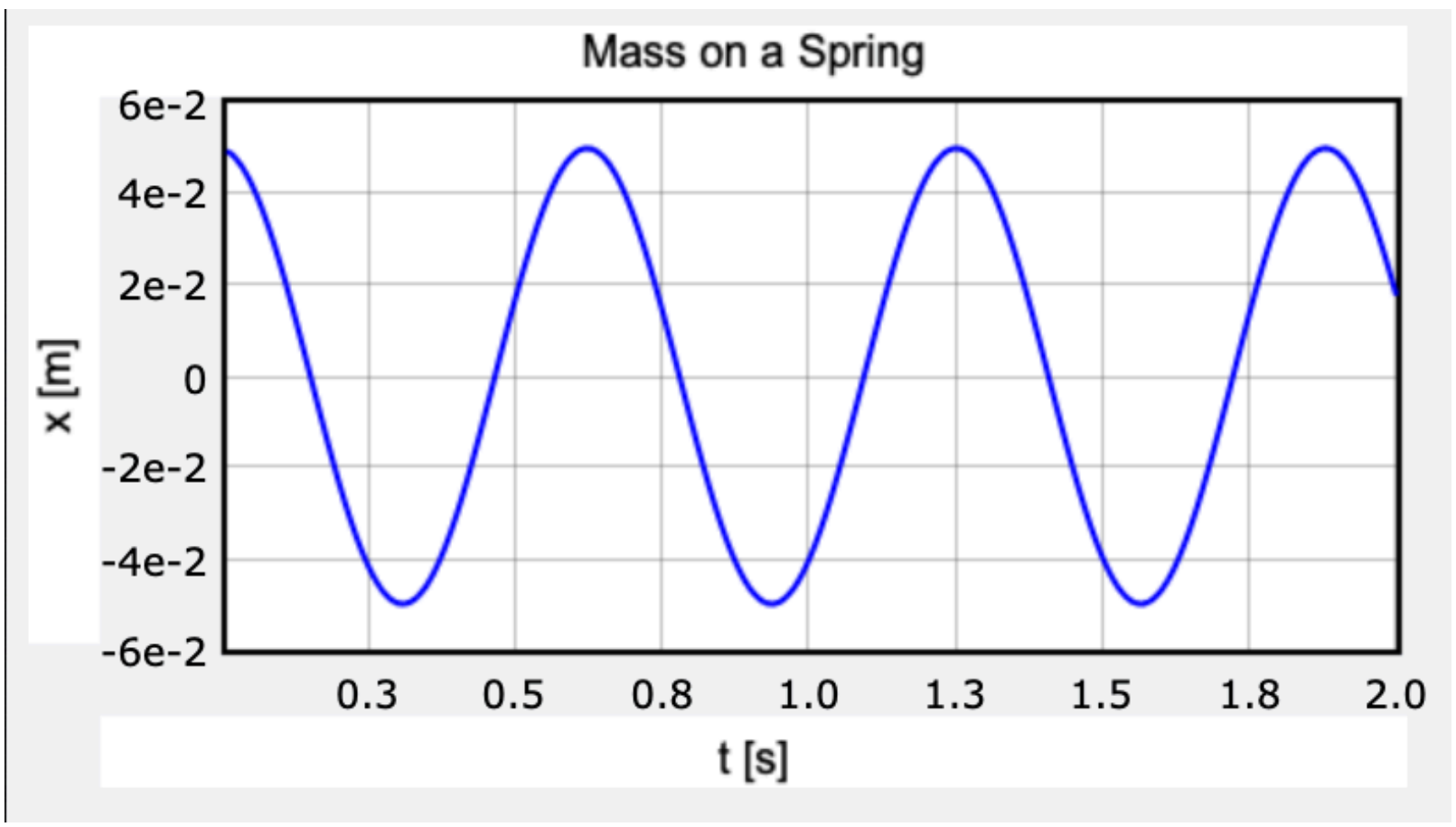
Here is the code for this calculation (don't forget about graphs and stuff from before). Comments to follow.

```
1  Web VPython 3.2
2  g1 = graph(title="Mass on a Spring", xtitle="t [s]", ytitle="x [m]",
3  width=500, height=250)
4  f1 = gcurve(color=color.blue)
5
6  m = 0.1
7  k = 10
8  x = 0.05
9  v = 0
10 t = 0
11 dt = 0.01
12
13 while t<2:
14     a = -k*x/m
15     v = v + a*dt
16     x = x + v*dt
17     t = t + dt
18     f1.plot(t,x)
19
```

That's just 18 lines of code—but really it's just 15 (two empty lines and the title line). The point is that it's not super complicated. Here are a few comments.

- Lines 2-4: Graphing stuff. Remember that line 2 wraps to 3 so it would look better. But I don't need to say anything else about graphing.
- Lines 6-11: Constants. I left off the units as comments because I like to live dangerously.
- The rest: the only thing new here is the calculation of the acceleration in the loop.

Here's the output (spoiler alert: it's amazing).



Why is this so awesome? It's because it's correct. Imagine that you started with Newton's second law and solved the equation of motion using your differential equation tricks. You would get the following function.

$$x(t) = x_0 \cos(\omega t) + \frac{v_0}{\omega} \sin(\omega t)$$

$$\omega = \sqrt{\frac{k}{m}}$$

Yes, you get a nice trig function. If the initial velocity is zero (like in our case), it's just a cosine function with an angular frequency that depends on the mass and spring constant. You could check if the two solutions agree, but if you would get a silly plot. It's silly because the two curves would be RIGHT on top of each other. The solutions agree very well.

Let's think about this for a second. We know the analytical solution depends on a trig function (and that means there is a π in there). But where does the π come into the python code? What line deals with circles and stuff? It's not in there. This is the amazing part, you get a cosine function without any circles or π or trig. That's just cool. Oh. In fact, if you measure the period of oscillation (T) from the numerical calculation, you could set that equal to the theoretical period.

$$T_{\text{numerical}} = 2\pi\sqrt{\frac{m}{k}}$$

You could then use the values of m and k to solve for π. What? Yup. Finding π without a circle.

NUMERICAL VS. ANALYTICAL

We have two solutions for a mass on a spring—the numerical calculation an the analytical solution. Does it matter which one we use? In this case, no—it doesn't matter. Let's think about the pros and cons of the two methods:

Analytical: The main thing is that you can get a solution without using numerical values. This is really nice since you can just plug in any numbers for any situation after you have a

solution. Also, you can see how the solution behaves—such as what happens if I double the mass.

However, some problems just don't have an analytical solution. Or maybe the solution is extremely complicated. Another thing to consider is that you might not actually have a real analytical solution. In the example above with a mass and a spring, we know that the position can be expressed using the cosine function. But what the heck is cosine? How do you find values for the the cosine of 0.3 or something? The answer: you use a numerical calculation. So, essentially you have solved the problem in terms of other functions and those functions need numerical solutions.

Numerical: Just to be clear, a numerical solution DOES NOT mean you have to use python. You don't even have to use an electronic computer. All that is required is that you break the problem into smaller pieces and use numerical values. The awesome part is that the process is basically the same for a falling ball as it is for a mass on a spring. Oh, what if you want to add a drag force to the mass? Note: we will actually do this later. In that case, you just need to update the expression for the acceleration and boom—you are ready to rock and roll.

Of course the bad part is that you don't really have a solution. You just have numbers. With the oscillating spring above, we don't get a cosine function. No, we get numbers that when plotted LOOK like a cosine function—but they are just numbers.

Homework

1. Take the mass and spring. Let $x(0) = 0$ m but $v(0) = 0.1$ m/s. Show that your numerical solution is indeed a sine function.

2. Get a real spring and some real masses. Hang the masses to stretch the spring and plot force as a function of position. Use the slope of this plot to calculate the spring constant. Now let the mass oscillate and record the period for one oscillation. Use these values in the code above to see if reality agrees with your calculation.

3. Oh, you have a spring and some masses? Find the period for different masses oscillating. Use your data to find the value of π.

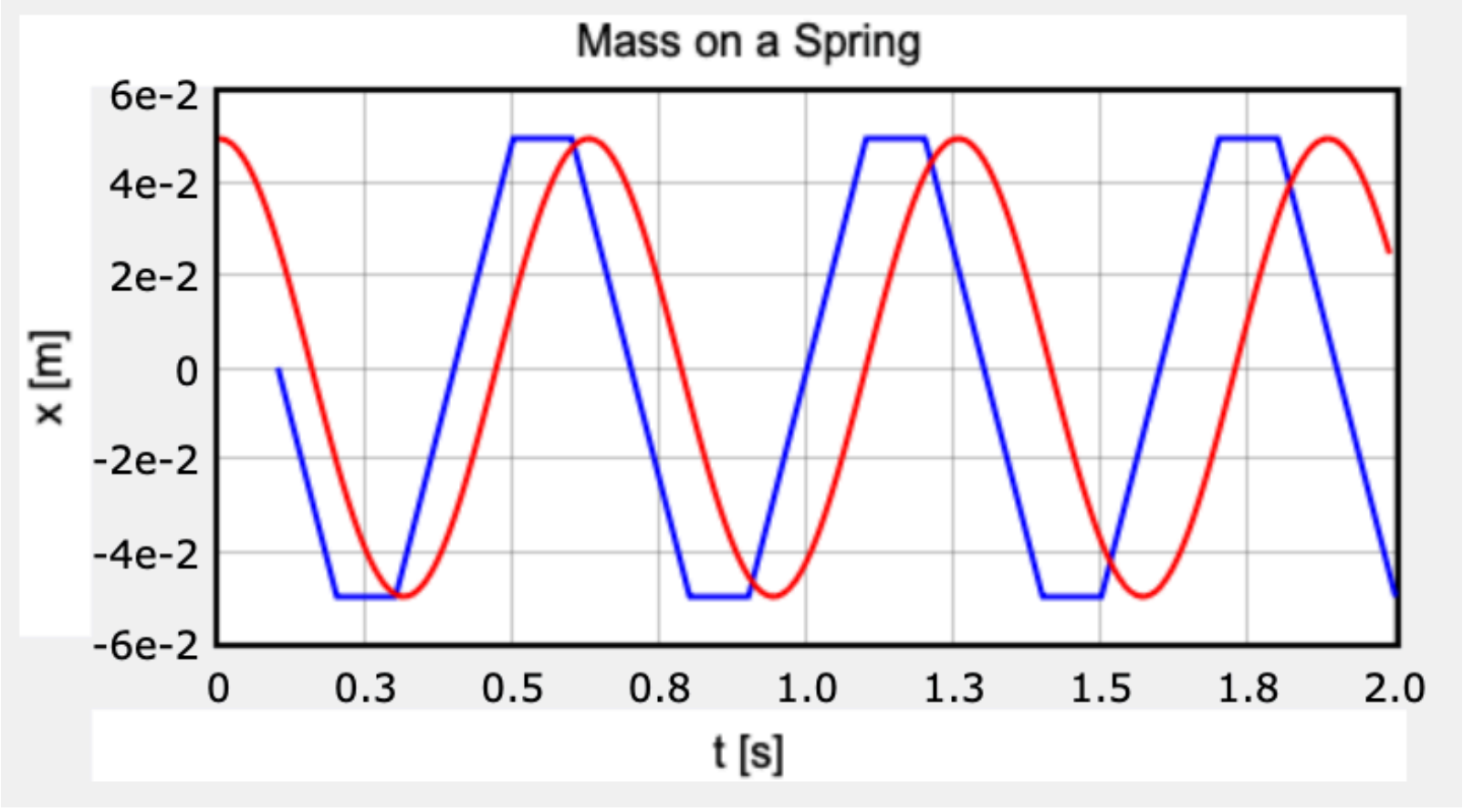
Answer Key

There is no answer key. You can do this.

TESTING THE NUMERICAL MODEL

Let's play around with our numerical recipe just a little bit. Nothing crazy.

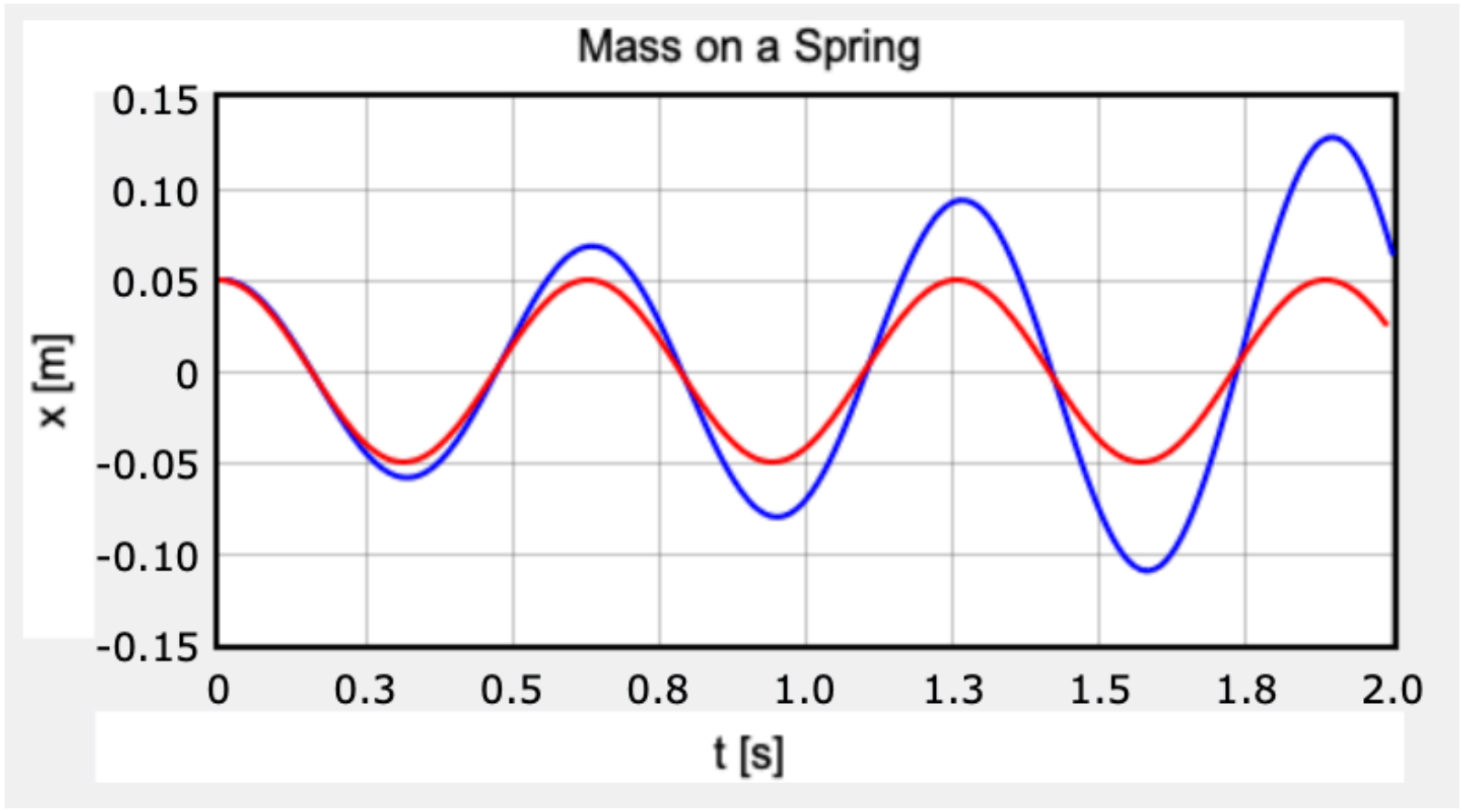
The first thing to change is the time step. How LARGE of a time step can we get and still have a nice solution. The cool thing is that we already know what we are supposed to get. So, I can plot my numerical solution with a large time step along with the analytical solution for comparison. Here is the numerical model with a time step of $\Delta t = 0.1$ seconds.



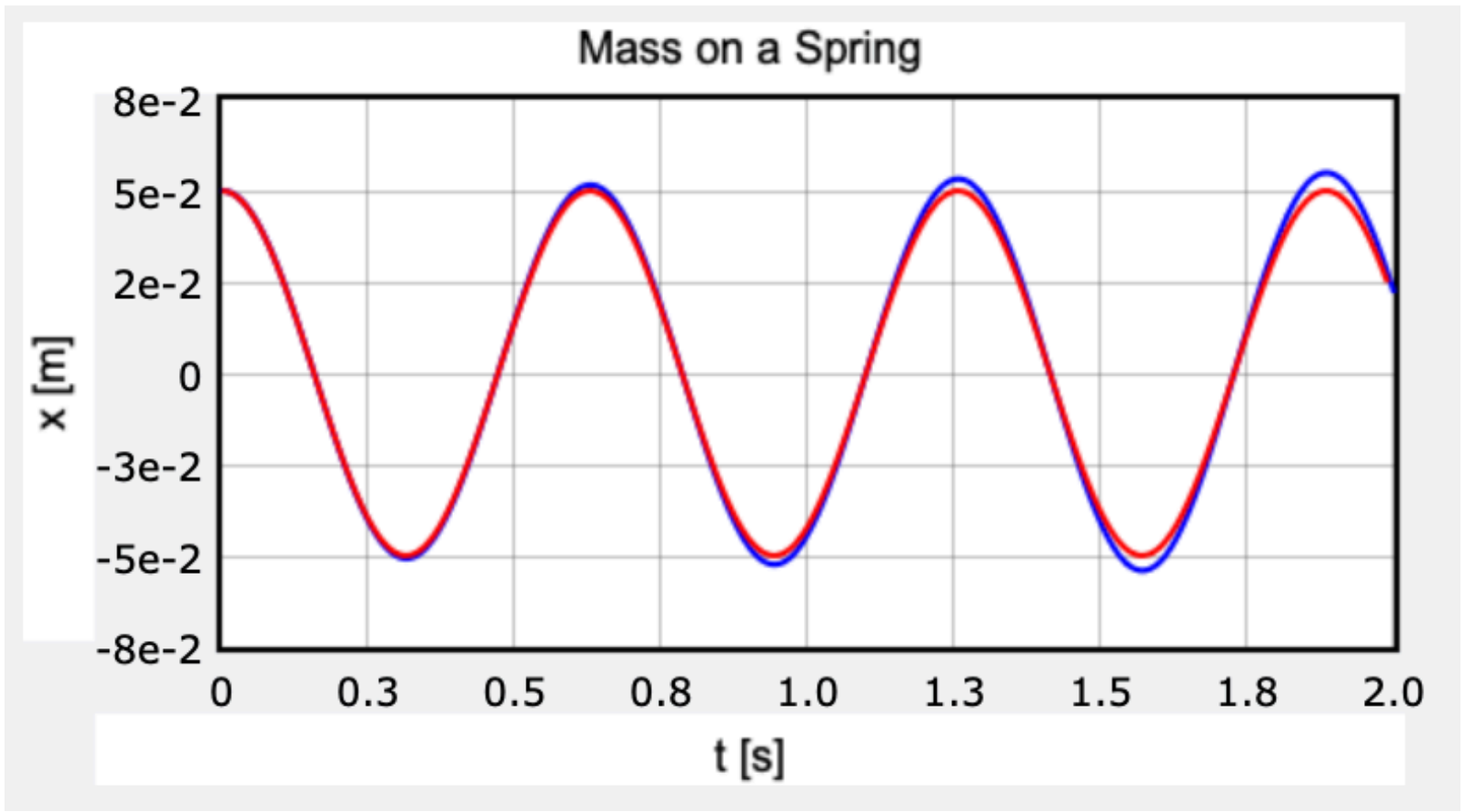
I mean, it's not the same—but it's really not that bad. Right? Just think—we are just updating the position and velocity 10 times per second. It's amazing that the whole thing didn't just go crazy. You can play around with the time interval and see what happens.

What if we don't know what the solution should look like? How do we know how small of a time interval we need? One rule of thumb that we can use is to decrease the time interval by a factor of 2 until the new solution isn't significantly different from the old one. That's what we can go with for now.

What about the order of operations? In the calculation above, we calculated the acceleration and then updated the velocity and then updated the position. What if we change this order and do it backwards? Here's the output (along with the analytical solution again). Oh, this is with a time interval of 0.01 seconds.



Yeah, that doesn't look right. The numerical model shows the amplitude of oscillation increasing as time increases. So, energy is not conserved. Can we fix this by using a smaller time interval? Sort of. Here's a time step of 0.001 seconds.



It's better—but still not as good as the original order with a larger time interval. Here's the rule. In general, do the calculations for the highest order derivatives first. Since the

acceleration is the second derivative of position with respect to time, we do that one first. Then velocity (first derivative) and finally the position. We can look at other numerical recipes later.

FALLING COFFEE FILTER

Here's another great example for numerical calculations. Imagine you take a tennis ball and drop it from a height of 2 meters above the floor. You can easily build a numerical model for this and you can also calculate the time it takes to hit the ground using the kinematic equations.

But what if I take a coffee filter and let it go from 2 meters? That's a different problem. With the coffee filter, there's both a downward pulling gravitational force (mg) AND an upward pushing air resistance force. Yes, there is also an air resistance force on the tennis ball, but it's tiny compared to the gravitational force.

Let's assume that we can model the air resistance force on the coffee filter as the following equation:

$$F_{\text{air}} = cv^2$$

Don't worry, we will look at more sophisticated air resistance models later. But for now, this should work. It says that the magnitude of the drag force is proportional to the square of the velocity. Here I'm using c is a coefficient for our particular case.

So, just think about what's going on here.

- When the coffee filter is released, it has a speed of 0 m/s so that there is no drag force and the filter increases in speed in the negative y-direction (because of the gravitational force).
- After a short time, the coffee filter is moving so that now there is a combination of a downward gravitational force and an upward pushing drag force. This gives a net force and acceleration LESS than g so that it still speeds up but at a lower rate.
- Eventually (almost eventually) the magnitude of the drag force will equal the downward pulling gravitational force and the net force will be zero. The filter will move at a constant speed. We call this terminal velocity.

This won't be too difficult to model in python, so let's do it. We need some starting values, how about the following:

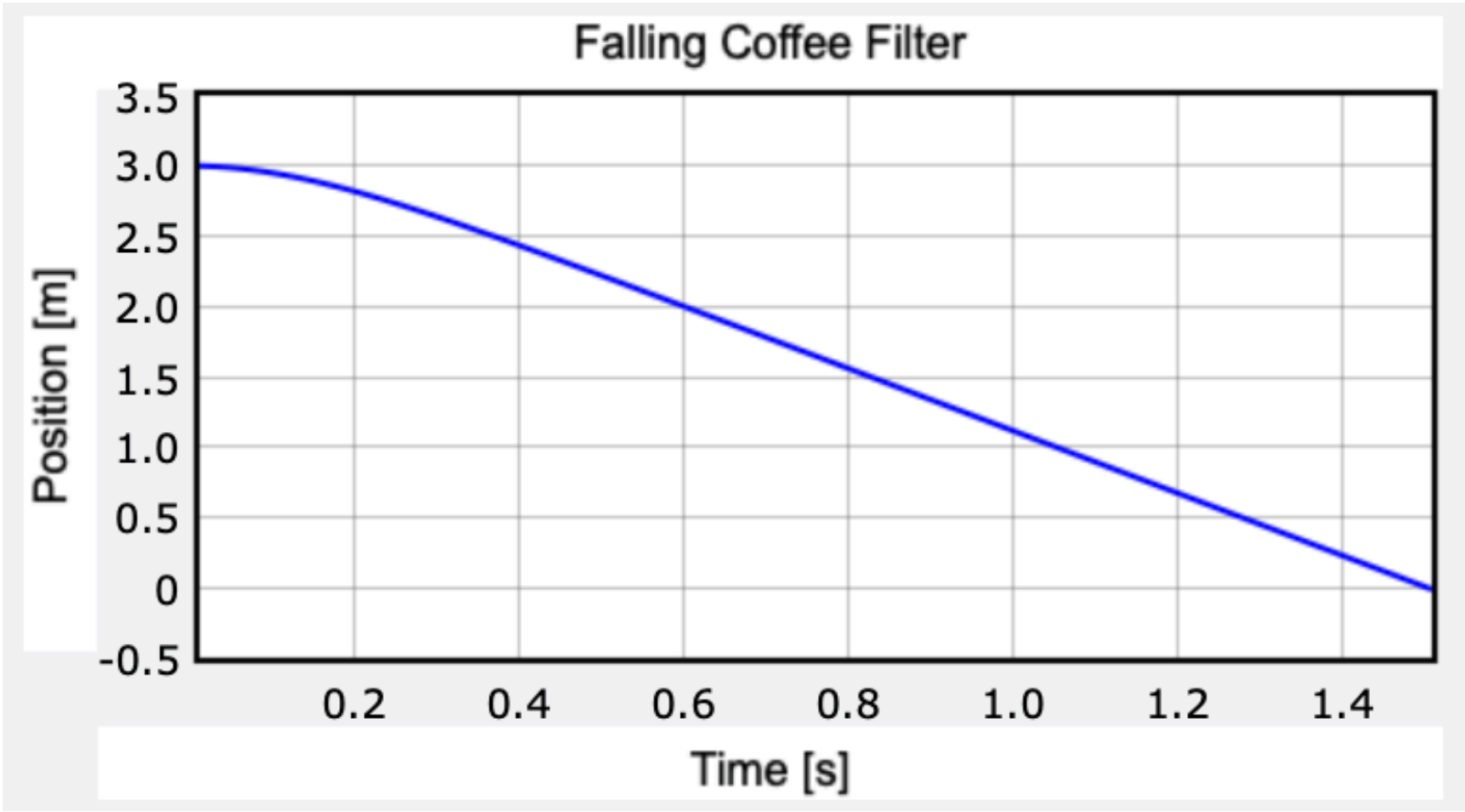
- Mass = 0.005 kilograms.
- $c = 0.01$ Ns^2/m^2
- $y = 3$ meters
- $v = 0$ m/s.

Here's a starter program for you to model this motion. Don't worry, there are only three lines you need to add. You can do it.

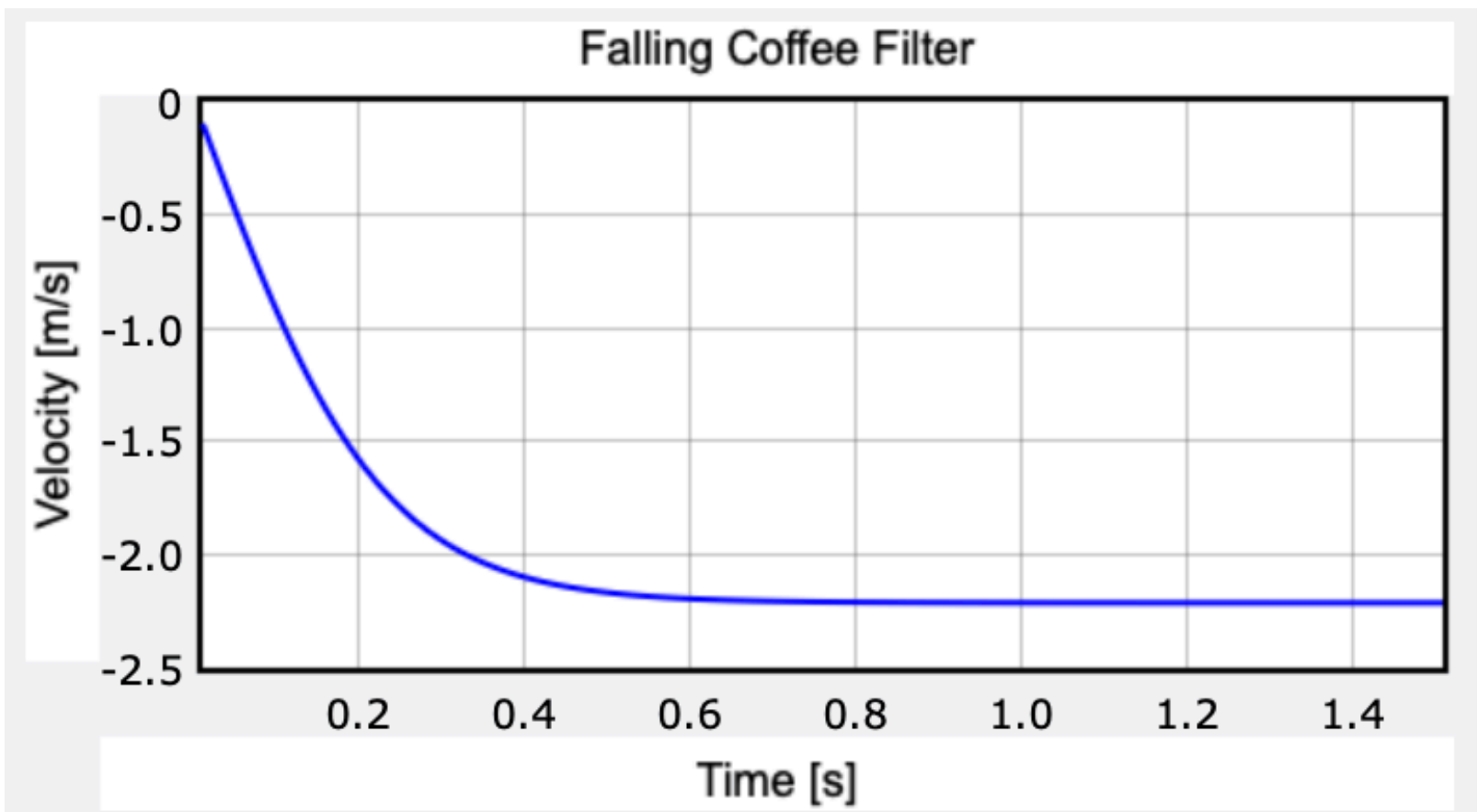
```
Web VPython 3.2
tgraph=graph(title="Falling Coffee Filter",
xtitle="Time [s]", ytitle="Position [m]")
f1=gcurve(color=color.blue)
g=9.8
m=0.005
c=0.01
y=3
v=0
t=0
dt=0.01

while y>=0:
  F= #stuff
  a=F/m
  v= #stuff
  y= #stuff
  t=t+dt
  f1.plot(t,y)

```

Remember that the gravitational force is negative and the air resistance is positive in this case. I know this is dangerous code because my loop is based on position. If you mess up, it's gonna run forever. No pressure though.

When you get it to work, it will look like this.



Notice that the coffee filter quickly reaches a terminal velocity. Just for fun, let's plot the velocity as a function time.



That's exactly what you would expect. You can see that horizontal line representing the terminal velocity of the filter.

Homework

1. There's just one question (but this is a great one). Suppose you are working for a TV show (something that rhymes with Just Musters) and you are going to drop a dummy human so that it hits the ground at terminal velocity. How high do you need to drop the dummy so that it reaches 95 percent of terminal velocity? Let's say that the terminal velocity of a skydiver is 120 miles per hour (54 m/s) for a 75 kilogram person.

Answer Key

Did you really try to solve the problem? You are only cheating yourself.

Fine, here's the answer. The first thing we need to do is to find the "drag" coefficient (not the real coefficient that we will use later). At terminal velocity (v_T), we have the following:

$$F_{\text{net}-\text{y}} = 0 = cv_T^2 - mg$$

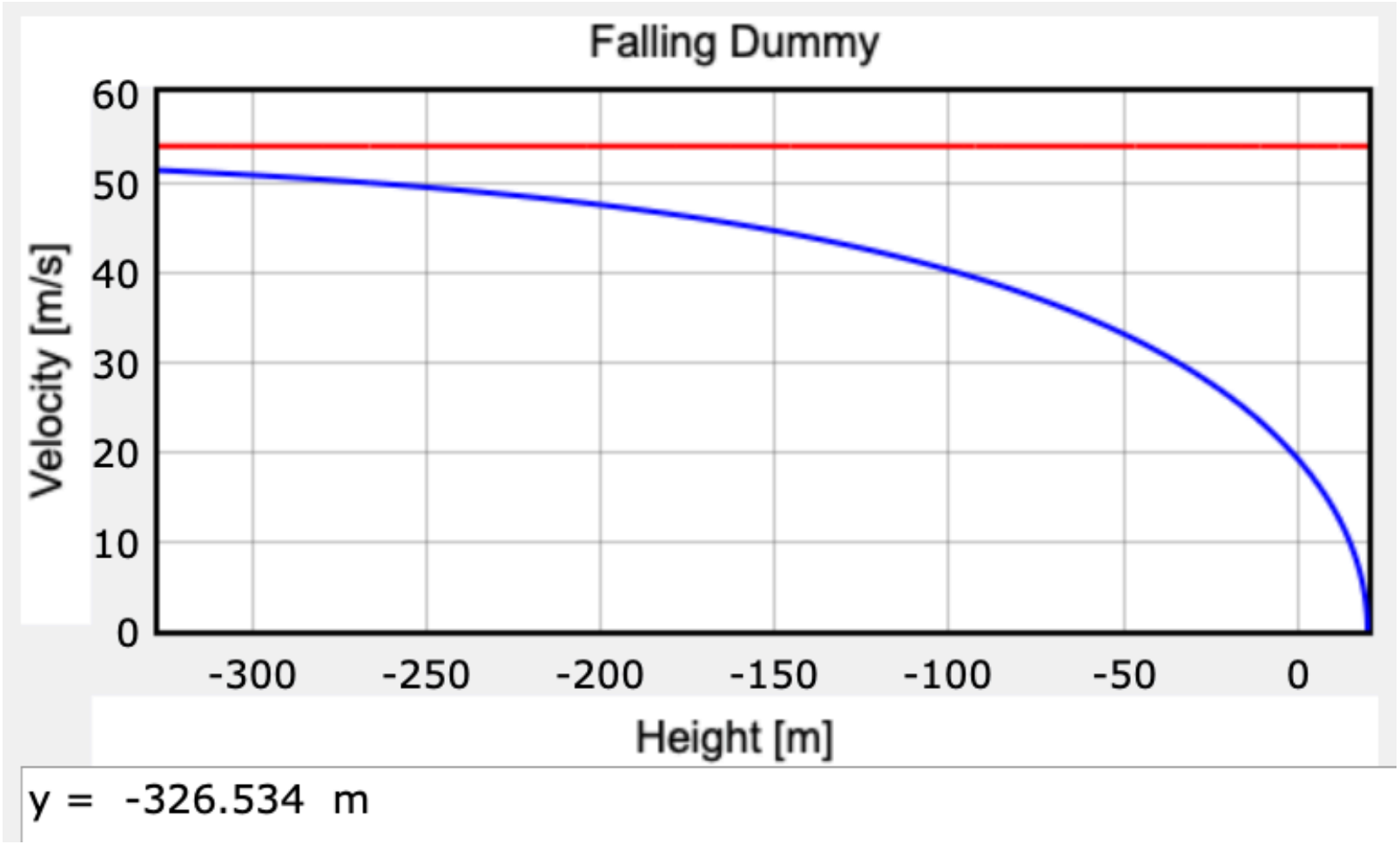
$$c = \frac{mg}{v_T^2}$$

I'm not even going to punch the numbers to get a value for c. I can do that in python. With this value and the mass of the human I could just reuse the coffee code above. Pick a starting value for y and print the final velocity as well as the terminal velocity. If it's within 95 percent, you win. If the final velocity is too low, change the starting height and rerun it. Oh, I know—that seems barbaric to just keep changing the code. There should be a more civilized way. Yes, but don't be afraid to do something that works even if it's not the most elegant. I mean, Han Solo used a blaster instead of the lightsaber because it worked for him —no matter what Ben Kenobi said.

Here is my lightsaber code. Note that it's very dangerous (just like a real lightsaber in the hands of a young moisture farmer).

```
Web VPython 3.2
tgraph=graph(title="Falling Dummy",
xtitle="Height [m]", ytitle="Velocity [m/s]",
width=500, height=250)
f1=gcurve(color=color.blue)
f2 = gcurve(color=color.red)
g=9.8
m=75
vT = 54
c=m*g/vT**2
y=20
v=0
t=0
dt=0.01

while v>-0.95*vT:
  F= c*v**2-m*g
  a=F/m
  v= v+a*dt
  y= y + v*dt
  t=t+dt
  f1.plot(y,-v)
  f2.plot(y,vT)

print("y = ",y," m")

```

Do you see what I did? First, I started my falling body at y = 0 meters. This means that as it falls it will have negative positions (I don't want it to hit the ground). Next, my while loop tells the code to run as long as the velocity (which will be negative) is greater than negative 0.95 times the terminal velocity. If you put this as "greater than negative terminal velocity", it will run forever. Forever is a long long time. Also, I plotted velocity vs position. Here's what I get.



Oh, maybe your TV show doesn't have money to drop a dummy from that height? Well, you could always change it to 80 percent of terminal velocity.

Could you solve this problem without using python? Yes, this is indeed solvable—but not quite trivial. However, you can see that the numerical solution is fairly straightforward. Also, we will look at a version of this problem that's practically impossible to solve analytically. Just wait.

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.

```
1 Web VPython 3.2
2
3 A = vector(1,2,3)
4 B = vector(-1.1,0.5,1.8)
5 print("A + B = ", A + B)
6 print("A - B = ",A - B)
7 print("Ax = ", A.x)
8 print("3.2B = ",3.2*B)
```