A First Look at Simulating Motion

In the last activity, we managed to move objects on the screen by repeatedly evaluating the position function, x(t), for successively larger time values. This iterative evaluation of the function was done in the WHILE loop.

We were also able to graph the values in a position-time graph.

This approach was quite powerful already, but it also limits what we can do. For example, it would already be quite difficult to make an object move forward, stop for a while and then move backward using this approach, as you probably discovered in the last chapter. Furthermore, this approach also imposes solutions rather than numerically finding them, thus sort of bypassing the underlying physical laws, as we will see later.

For these reasons, we would like to move from a mere evaluation of a function to an actual simulation of a physical process in real time. Today, we will take our first step in the direction of a true numerical simulation of physics.

4.1 The Physics

Let's start with a piece of physics that you already know – the formula for average velocity in one dimension – and then translate it to a statement a computer can understand. So, average velocity is defined as,

$$v_{avg} = \frac{\Delta x}{\Delta t} = \frac{x_2 - x_1}{\Delta t} \tag{4.1}$$

In words, the average velocity over a time interval of motion is defined as the displacement divided by the elapsed time.

As a general rule, computers cannot handle continuous motion. Therefore, in order to make any simulation of real physical motion look *realistic*, we will have to "chop up" the true continuous motion into many, many frames separated by very small time intervals. The situation is not unlike watching a movie where we perceive continuous motion when in reality we are being exposed to a rapid sequence of still images.

The upshot is that we will want to make our time interval Δt really small in Equation (4.1). Over the course of this small time interval we would not expect the velocity to change very much, so a pretty good assumption is that it is simply constant over this interval. This also means that the average velocity is just the same as this constant velocity and we can drop the subscript "avg". If we then solve Equation (4.1) for x_2, we get:

$$x_2 = x_1 + v\Delta t \tag{4.2}$$

Here we assumed that Δt was sufficiently small, which allowed us to omit the subscript *avg*.

Let's describe Equation (4.2) in words: Between two frames separated in time by Δt, we can compute the position of the object in the new frame, x_2, based on the position in the current frame, x_1, and the velocity of the object, v. We simply have to add $v\Delta t$ to x_1.

Now all we have to do is repeat the step in Equation (4.2) over and over again. Every time we advance forward in time by the small amount Δt. And every time, we replace the starting position x_1 by the ending position x_2 of the previous step. In other words, we want to use Equation (4.1) recursively in order to evolve the position of our object forward in time, step by step.

So far, we have assumed that v in Equation (4.2) would remain constant in each successive step, but this does not have to be the case. In general, we can allow the velocity v to get updated as well with each new iteration. We will see some straightforward examples in the section 4.3. The procedure we have described here is also known in the numerical analysis community as the *Euler Method.*[1]

4.2 The Basic Code

Now that we a taste of the physics involved, how do we translate that idea into code a computer can interpret? Let's take a look at the following few lines of VPython code - see Figure 4.2.

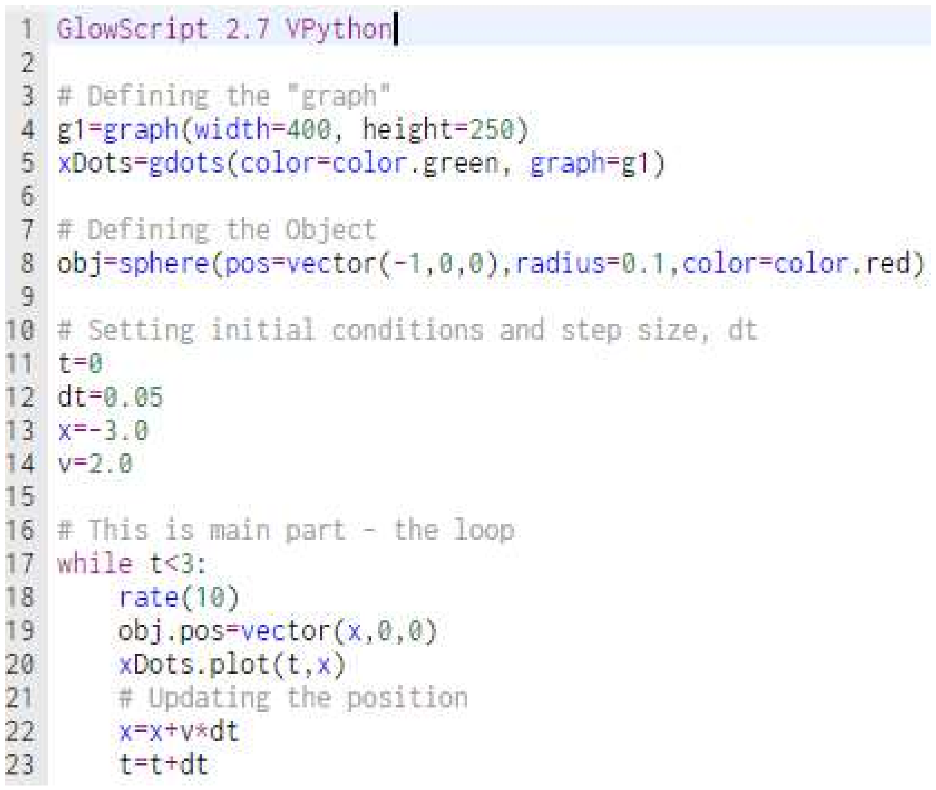
We see that the core of the program is contained in the WHILE loop, as before. Let's start our discussion with the central line, namely LINE 22:

$$x = x + v * dt \tag{4.3}$$

As a mathematical statement this is, of course, nonsense. But we should not interpret the equal symbol in the mathematical sense. It is not an equality. It is an ASSIGNMENT. "x" is a variable that holds a certain number, and whenever you see "x = something" this signals that the value stored in the variable "x" is about to be updated.

So Equation (4.3) should be read in two parts. The expression to

[1]Named after the mathematician and physicist Leonhard Euler.



```
1  GlowScript 2.7 VPython
2
3  # Defining the "graph"
4  g1=graph(width=400, height=250)
5  xDots=gdots(color=color.green, graph=g1)
6
7  # Defining the Object
8  obj=sphere(pos=vector(-1,0,0),radius=0.1,color=color.red)
9
10 # Setting initial conditions and step size, dt
11 t=0
12 dt=0.05
13 x=-3.0
14 v=2.0
15
16 # This is main part - the loop
17 while t<3:
18     rate(10)
19     obj.pos=vector(x,0,0)
20     xDots.plot(t,x)
21     # Updating the position
22     x=x+v*dt
23     t=t+dt
```

Figure 4.1: The basic code

the right of the equal sign is computed first based on the current value of x. Secondly, the "x = " part then assigns the result of that computation to the variable x. The upshot is that x is updated and now holds the new value.

Thus, if we had to translate the statement in Equation (4.3) into a mathematical equation, we would say:

$$x_2 = x_1 + v\Delta t,$$

where x_2 is the new value and x_1 is the old value of position. Note also that "dt" in Eq. 4.3 should not be interpreted as a differential in the Calculus sense but represents a small time interval, defined in LINE 12 as 0.05.

We are now in a position to discuss the entire code. Line 4 and 5 set up a position graph (time on the horizontal axis, position on the vertical axis). LINE 8 defines the object we want to move around in space – a red sphere of radius 0.1. Lines 11-13 set the initial conditions as well as the size of the small time interval, dt. Within the WHILE look, we keep updating the variable x and then, using this, to update the position of the object using the command in LINE 19: **obj.pos=vector(x,0,0)**. Here **obj** was defined as the sphere, and **pos** is an attribute of the sphere, namely the position (i.e. location) of the sphere's center. This position is a three-dimensional vector which we create using the **vector** command. Finally, the **xDots.plot(t,x)** command in LINE 20 appends the newest data point to our position graph.

The last line, Line 23, updates the time variable. This is necessary in order to test the conditional of the WHILE loop **(t<3)**, as well as for the purpose of plotting the position (and velocity) graph. It does not, however, come into play in calculating the position of the object – an important difference from the previous approach of simply plotting the function **x(t)**.

4.3 Exercises

1.
   - Run this program to make sure it works. Describe what you see.
   - Modify the code so that during the first second, the ball is moving to the right as before, but for the second second it stays still, and during the third second, it returns to the original starting point. Also have VPython generate the corresponding position-time graph.

     Hint: Think about how you could use **if-then** statements to accomplish this task.

2.
- Discuss with your group how you would have to modify the code printed above to make the object accelerate to the right at constant acceleration. *Hint*: Think about the role of "v"!
- Once you verify that the code really does produce an accelerating object on the screen, let's have VPython generate the position-time graph, as well as the velocity-time graph. Ideally, the data would be displayed in two separate graphs. You may have to duplicate and slightly modify some of the lines already appearing in the sample code.

  Take a screenshot of the code, as well as the two graphs to include in your lab notebook.
- Now that you have the position-time graph, verify that it agrees with the first kinematics equation: $x(t) = x_0 + v_0t + \frac{1}{2}at^2$. You can do this, for instance, by evaluating the formula at a particular time (for the initial velocity and the acceleration that appears in your code). Does this point lie on the the graph?