Your Second Physics Simulation: A Mass on a Spring

I. INTRODUCTION

At this point I think everybody has a working "bouncing ball" program. In these programs the ball moves under the influence of a very simple force: the constant force of gravity (except for the time during impact). But what if the force isn't constant? What if it depends on the position, like the force exerted by a spring, or the force of gravity when you move very far away from the surface? What if it depends on velocity, like the force of air resistance? The goal of this exercise is to modify your working bouncing ball program so that it is is easy to use for just about any force.

The question is: What do we have to modify in our working programs? The "guts" of the calculation is carried out in the `while` loop. Inside the loop there are commands that tell the computer how to increment the velocity and the position according to the rules

$$\vec{r}_2 \simeq \vec{r}_1 + \vec{v}_1 \Delta t \tag{1}$$

$$\vec{v}_2 \simeq \vec{v}_1 + \vec{a}_1 \Delta t \tag{2}$$

In my program the lines that implement these *vector* rules look like

```
    ball.pos = ball.pos + ball.velocity*dt
    ball.velocity = ball.velocity + a*dt
```

where I have previously defined the acceleration as

```
    a = vector(0,-9.8,0)
```

If I change the force, are the updating rules expressed in Eqs. (1) & (2) still valid? Yes — they just come from the definitions of velocity and acceleration. This means that I don't have to change anything about the structure of my loop that updates the values of position and velocity. What does change when the force changes is the acceleration — it's no longer the constant vector `(0,-9.8,0)`. Newton's second law tells us that we can figure out the new acceleration if we know the net force: $\vec{a} = \vec{F}_{\rm net}/m$.

The plan is to define a force *function* near the top of your program. Then you can rewrite the line that updates velocity in terms of the force function and mass instead of in terms of a constant acceleration. Then all it will take to change forces is to change the function definition; no additional changes will be necessary in the `while` loop. The topic of the next section is how to define functions in Python.

II. PYTHON AND FUNCTIONS

As a first example of a Python function, let's do a very simple example of a function that calculates the square of a number. We will give this function the name `square` and it is a function of a single argument that we call `x` in the function definition. In an IDLE window enter the following:

```
from visual import *
def square(x):
     y = x*x
     return y
a = 4
b = 3
print square(2)
print square(a)
print square(a*b)
```

Executing this program should result in the output values 4, 16, and 144. Note that the `square` function could be defined in a shorter way without the intermediate value of `y`:

```
def square(x):
     return x*x
```

Now let's define the function $f(x) = x^2 - 3$. I'm going to do this in an overly long manner to make a point.

```
def f(x):
    y = square(x)
    c = 3
    y = y - c
    return y
```

If you define `f(x)` as above, and follow it with the statement

```
print f(6)
```

you will get the expected value of 33. (The definition of `square` must still be in your program for this to work as written.) But if you follow it with the statement

```
print f(c)
```

you will get an error message. You might think that you should get the value 6, because of the statement `c=3` in the function definition, and $f(3) = 3^2 - 3 = 6$. But the assignment of the value 3 to the variable `c` is *local* to the function; the rest of the program doesn't "know about" this assignment.

In the examples above the functions returned scalar values. It's also possible to define functions that return vectors. For example, a constant gravitational force in the negative y-direction can be defined as follows:

```
g = 9.8
def forceG(m):
    return vector(0,-m*g,0)
```

If you follow the function definition above with the statement

```
print forceG(2)
```

the output will be

```
<0, -19.6, 0>
```

Some of you defined the variable `g` as a vector instead of a scalar, and in this case the force definition would look something like

```
g = vector(0,-9.8,0)
def forceG(m):
    return m*g
```

which should give the same output as before.

In the case of a mass attached to a linear spring stretched in the y-direction the force on the ball is $F_{\text{spring}} = -ky$, where k is a constant. A definition of a vector force imparted by this spring looks like:

```
k = 2
def forceSpring(y):
    return vector(0,-k*y,0)
```

If you follow the function definition above with the statement

```
print forceSpring(3)
```

the output will be

```
<0, -6, 0>
```

The definition of `forceSpring` assumes that the input value `y` is a scalar, but if you want to input the position as a vector, the function definition could look like this:

```
    k = 2
    def forceSpring2(r):
        return vector(0,-k*r.y,0)
```

If you follow the function definition above with the statements

```
    r = vector(1,3,2)
    print forceSpring(r)
```

the output will be

```
    <0, -6, 0>
```

Finally, it's possible to define functions of more than one variable. The force on a particle might depend on the position of the particle, the velocity of the particle, the mass of the particle, or some combination of these, so in a bit of forward thinking it's useful to define a function that can handle all such cases. For example, we can re-write the spring force with "extra" variables for mass (`m`), and velocity (`v`) that aren't used in this case:

```
    def force(m,r,v):
        return vector(0,-k*r.y,0)
```

III. IMPROVING YOUR BOUNCING BALL PROGRAM

1. You should start today by opening a simple bouncing ball program that you have written previously and save it with a new name. We will modify this program, but it should still produce the same output as before.

2. Include a statement near the top of your program defining the mass of your ball. (A mass of 1 is fine to start with.)

3. Include near the top of your program a definition of a force function describing the constant downward force of gravity that is acting on the the ball. I encourage you to write this as a function of mass, position, and velocity, even though the gravitational force only depends on the mass.

4. As I said in Section I, my program has a line that updates the *vector* velocity that

looks like

```
ball.velocity = ball.velocity - a*dt
```

(where `a` has been defined as a vector). I want you to find the analogous line in your program, and I want you to change it so that the acceleration is calculated in terms of the force function you have defined and the mass.

5. If you made the changes correctly, your bouncing ball should bounce just like it did before you made the changes.

IV. PROGRAMMING A MASS ON A SPRING

1. Save the program you wrote in the previous section with a new name.
2. Modify the program to simulate the motion of a mass on a spring. (You can leave the `floor` where it was for the time being.) You may:
   - Assume that the equilibrium position of the mass is `(0,0,0)`.
   - Assume that the restoring force on the mass is given by $F_{\text{spring}} = -ky$.
   - Assume that $m = 0.5$ and $k = 1$.
   - Assume that the mass starts at rest.
   - Remove any statements that make the ball bounce; the ball will change direction under the influence of the spring.
3. Your program should show sinusoidal oscillation of the mass about the equilibrium position.

V. MAKING THINGS PRETTY

After you get the mass oscillating, let's make it look nice.

1. Move the `floor` down so that it is below the lowest point of the mass's oscillation.
2. Add a spring (a `helix` object) with one end attached to the `floor` and one end attached to the mass.