Modeling Free Fall

Now that you've done experimental measurements of an object in free fall, you will model the motion of an object in free fall using numerical methods and compare your results.

A typical computer program has four parts, and these instructions will guide you through generating each part, including all commands that will be needed, as follows.

- Setup statements
- Definitions of constants (if needed)
- Creation of objects and specification of initial conditions
- Calculations to predict motion or move objects (done repetitively in a loop)

1 Setup Statements

Open a VIDLE window and save it, adding "py" to the file name.

Enter the following two statements.

```
from __future__ import division
from visual import *
```

Every program we will run should start with these commands. Sometimes other setup statements will be required as well, but these will be explained as needed.

2 Constants

Next, you define any constants you'll need in the program. For free fall, you need only one constant.

`g = 9.81` (Note that the spaces around the equal sign are optional.)

Note that there are no units—it's up to you to use the constants and values in a way that correctly utilizes units. In other words, if you set g to be 9.81, you're working in meters and seconds, so any distances you later put in must be in meters, and so forth.

3 Objects and Initial Conditions

3.1 Initial Conditions

Set the following initial values for your objects.

```
m = 0.4
h = 2
vball = vector(0,0,0)
pball = m*vball
```

We've just defined values for the object's mass and the starting height (2 m, just like your first experimental trial), created a vector to represent the ball's velocity at the start of the trial—it's initially at rest—and defined the ball's momentum as mv. Note that while we've defined these now, the program will not know to actually do anything with them until we tell it how to use them.

3.2 Objects

Next, create the object to be dropped (a sphere) and the ground.

```
ball = sphere(pos=vector(0,h,0), radius=0.2, color=color.red)
ground = box(pos=vector(0,-0.005,0), size=(5,0.01,5), color=color.green)
```

This creates a green square with its top surface on the xy-plane representing the ground and a red sphere a vertical distance, h, above it. Writing the objects like this means you can change the starting height by only modifying the value of h rather than the definition of the sphere.

Note that if we had put these lines before the initial condition lines, we would get an error, as it would encounter these objects before we had defined h.

3.3 Time Step

To make the object move, we will use the position update formula or kinematic equations to repeatedly increment the ball's motion by small amounts of time Δt. At each step, the computer will calculate all the proper values for forces, velocities, and positions, and then will increment to the next time step and do it again.

Type the following lines after your object declarations.

```
deltat = 0.1
t = 0
```

4 Animating Objects

4.1 Beginning Loops

Watch **VPython Instructional Videos: 3. Beginning Loops**.

(http://www.youtube.com/VPythonVideos)

If you'd like to practice loops, you can do the following optional task, but remove it from your program when you move on to the next part.

- Create a loop that prints the variable t from 0 to 19 in increments of `deltat`.
- If you're stuck, watch the video again.
- Add a `print` command after the loop (not inside it) to print the text "`End of loop`".
- Run the program. Look at the shell window to make sure it worked.

4.2 Loops and Animation

Watch **VPython Instructional Videos: 4. Loops and Animations**,

(http://www.youtube.com/VPythonVideos),

to see how using loops can animate 3D objects. We will use loops and physical principles to build models of motion consistent with the natural world.

Your loop to animate the ball must do the following things.

- Calculate forces acting on the ball.
- Calculate how much the ball's momentum/velocity changes due to forces acting on it.
- Move the ball the appropriate distance for its momentum/velocity.
- Increase the time step by `deltat`.
- Recalculate all of the above for the ball's new position in the next time step, and repeat.

4.2.1 The `while` Command

We want the ball to stop when it hits the ground, so the `while` command is best used with the ball's height.

```
while ball.pos.y > 0.2:
```

`ball.pos.y` is the command for the y-component of the position vector of the ball. Since the ball's radius is 0.2, its lowest edge is on the ground when its center is at $y = 0.2$.

4.2.2 Controlling Loop Speed

The computer can do all the loops in a tiny amount of time. To make the simulation look more realistic, we want to slow it down. Put the following command inside the `while` loop.

```
rate(10)
```

This tells the computer to execute 10 loops per second. Since our `deltat` is set to 0.1, each loop represents 1/10th of a second, so the simulation will run in real time (one simulated second per actual second). If we later change our `deltat`, we should change this as well.

4.2.3 The Physics

Now we must tell the program what to do with the objects. To model real-world physics, we must enter the equations of real-world physics. Namely, the Momentum Principle, equations of kinematics, Newton's Second Law, or similar dynamical equations.

The following table lays out a series of commands that use the methods of M and H sections of PY205 on the left, and commands that use the methods of N sections on the right. They both accomplish the same thing, so use either one, but don't mix and match. Make sure everyone in your group understands at least one method and how it's translating physics into code.

<table>
<tr><th>Using M and H methods</th><th>Using N methods</th></tr>
<tr><td>

- Force

Define the forces acting on the ball (in this case, Earth's gravity).

```
Fgrav = vector(0,-m*g,0)
Fnet = Fgrav
```

(The second line isn't important with only one force, but is good practice for future programs, which will combine multiple forces.)

</td><td>

- Acceleration

Define the ball's acceleration (in this case, acceleration due to Earth's gravity).

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

</td></tr>
<tr><td colspan="2">

It's important to work with vectors for animating motion. It's not vital in this program to use 3-D vectors since the motion will be one-dimensional, but it's good practice for future problems.

Also, note that it was necessary that we define m and g before using them like this. Since these vectors are constant, they could have been defined with the other constants instead of inside the loop, but often we'll encounter nonconstant forces or accelerations that must be recalculated each iteration, so again, it's good practice.

</td></tr>
<tr><td>

- Momentum update

Net forces cause changes in momentum. Enter the Momentum Principle (in update form).

```
pball = pball + Fnet*deltat
```

To translate $\vec{p}_f = \vec{p}_i + \vec{F}_{net}\Delta t$ into computer code, we get rid of the f and i subscripts. This line tells the computer to take the value it did have for the ball's momentum (`pball`, defined above), add $\vec{F}_{net}\Delta t$, then redefine `pball` with this newly updated value. Thus, every time this command is run, it will change the momentum by the appropriate $\Delta\vec{p}$.

</td><td>

- Velocity update

Acceleration cause changes in velocity. Enter the following kinematic equation.

```
vball = vball + a*deltat
```

To translate $\vec{v}_f = \vec{v}_i + \vec{a}\Delta t$ into computer code, we get rid of the f and i subscripts. This line tells the computer to take the value it did have for the ball's velocity (`vball`, defined above), add $\vec{a}\Delta t$, then redefine `vball` with this newly updated value. Thus, every time this command is run, it will change the velocity by the appropriate $\Delta\vec{v}$.

</td></tr>
<tr><td colspan="2">

Note that we didn't have to tell the computer that these are vectors since we originally defined all of the values as vectors.

</td></tr>
</table>

• Position update Now that we have updated the ball's momentum, find its new position. Enter the position update formula. `ball.pos = ball.pos + (pball/m)*deltat` Note that `pball/m` is the ball's velocity, and with small time steps, we'll let that approximate the average velocity for the time step. This line actually redraws the ball onscreen for each loop to reflect its newly updated position.	• Position update Now that we have updated the ball's velocity, find its new position. Enter the following kinematic equation. `ball.pos = ball.pos + vball*deltat` Note that this is a rearrangement of the definition of $\vec{v}_{ave}$, and with small time steps we'll let the velocity at the end of the previous time step approximate the average velocity for the next time step. This line actually redraws the ball onscreen for each loop to reflect its newly updated position.

The remaining commands are common to either M and N, or H formulations.

4.2.4 Increment the time

The last thing that should happen in a loop is to increment whatever your incremental value is for the next iteration. For most of our programs, this means updating the time by the following incremental time value.

```
t = t + deltat
```

Like the equations above, this is the computer formulation of $t_f = t_i + \Delta t$, rewriting the variable t each time with the increased quantity.

4.3 After the Loop

To put commands outside of the loop, simply unindent and go back to writing commands flush with the left side. None of these commands will be executed until the loop goes through the indented commands until the `while` condition is satisfied.

Have the program print the fall time:

```
print "The fall time is", t
```

This will print the current value of t, which, since it won't be executed until the loop is done, is the time at which the ball hits the ground.

5 Evaluation and Modifications

How does the fall time for 2 m compare to your measured fall time? What might account for the difference?

Change the definition of h in your program to the other starting heights you measured in your experiment. How do the fall times compare?

You can make the simulation more accurate by having small time steps. Change `deltat` to 0.01 and change `rate` to 100. Run the simulation again with your various drop heights. How do the values compare now?