UNIVERSITY OF TORONTO

Physics 151 Practical 3
Python Programming
Freefall Kinematics

Department of Physics
60 St. George St.

NOTE: Today's activities must be done in teams of **one** or **two**. There are now twice as many computers in the room, and twice as many lab notebooks. Please split up into pairs or find a computer and notebook to work alone; no teams of 3 or 4 today! As usual, don't forget to write your name or names on the first blank page of your notebook before you start. Also put the date and number your pages.

Python is a widely used and versatile programing language, often utilized by physicists and other scientists. VPython (i.e. Visual Python) is a module specifically used for 3D visualizations, making it a great tool for modelling physical phenomena.

In the following exercise you will use VPython to investigate the motion of a ball in free fall under the influence of gravity. If you are very new to programming, it might be useful to go through the VPython Intro document at http://compwiki.physics.utoronto.ca/Physics+with+VPython . However, this is not required, and the code for today's practical should be written "from scratch", according to the instructions below.

- There are questions asked of you throughout this Practical, labeled Question 1, Question 2, etc. Write down the answers in the lab notebook (to be handed in at the end of the practical).
- At the end of the Practical, hand in your lab notebook, along with a printed copy of your code for the Activity D.
- You must reach a consensus with your partner about an answer before moving on to the next part. Feel free to ask the TA for help if a consensus cannot be reached.
- We assume the motion is always close to Earth's surface, and therefore we will make the approximation that gravity is a constant $g = 9.8$ m/s^2.
- All quantities are in SI units; distance in meters (m), time in seconds (s), velocity in m/s, and acceleration in m/s^2.

Activity A 1-D Free-Fall with $v_i = 0$

In this activity you are going to drop a ball from a height of 20 with an initial velocity of 0. Open a new vpython window and instruct Python to use the Visual graphics module by typing:

```
from visual import *
```

This command tells python that you will be using functions from the visual module. The * symbol is a placeholder for any combination of characters. Putting * by itself in the line above indicates you are importing all the functions from the visual module.

You may as well go ahead and save the program now so you continue to save it to the same filename while working through this exercise. Call it "lab3exA.py"

It turns out you can control some properties of the visual window by setting 'scene' attributes. Let's set up some scene attributes to control the window properties. Enter the following commands:

```
scene.width=800
scene.height=800
scene.range=(16,16,16)
scene.autoscale = 0
```

```
scene.center=(0,12,0)
scene.background=color.white
```

(Don't worry too much about what these individual commands do, but if you are interested then you can alter them at home and see how they affect the python window.)

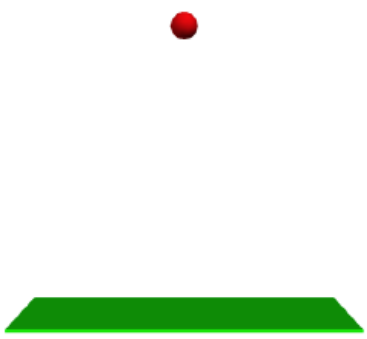
Next create a sphere called "ball" with an initial position vector (0,20,0), a radius of 1 and color red. You may want to refer to the tutorial introduction you did last week, or the python wiki page online to figure out the correct command for this.

Question 1: What command did you use to create the ball?

Now create a box called "ground" to represent the ground. You will want to make the top of the ground at a height of 0. To do this, set the ground position to (0,-0.1,0) and the ground size to (24,0.2,5). Make the color green.

Question 2: What command did you use to create the ground?

Save and run your program to make sure everything is ok so far. Your output should look like the following:



Question 3: To test your understanding of the code above, if we wanted the ground to be twice as thick, but still have its top at a height of 0, how would you change the values in the line of code?

Return the ground to the initial thickness. The ball will be under the influence of gravity, so the acceleration will be:

$$\vec{a} = -9.8\hat{j}$$

where $\hat{j}$ is a unit vector in the vertical direction. We will want to define a variable to represent this acceleration. To do so we will define a vector call 'accel'. Type the command:

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

Set the ball's initial velocity as (0,0,0) using the command:

```
ball.velocity=vector(0,0,0)
```

We want the ball to accelerate under the influence of gravity. This means the velocity and position of the ball will be changing with time. In class you learned the kinematic equations for position and velocity. These would be

called the "analytic" solution to the problem. However, instead of using those, let's use some ideas from numerical integration:

Numerical Integration:
In the VPython Intro we dealt with a ball bouncing between 2 walls, subject to a constant speed.
When speed is constant, you can write it in terms of displacement and time interval:

$$v = \frac{\Delta x}{\Delta t}$$

You can then rearrange this equation for the displacement:

$$\Delta x = v\Delta t$$

which can be used to write the position at time $t + \Delta t$ in terms of the position at time t :

$$x(t + \Delta t) = x(t) + v\Delta t$$

This was the equation used in the Intro to update the position of the ball in the "while" loop. In today's example, the acceleration is a constant, so we can update the velocity of the ball in the "while" loop using the equation:

$$v(t + \Delta t) = v(t) + a\Delta t$$

After we update the velocity, we would like to update the position at each time-step. Although technically, the velocity is not a constant in this case, if we take the time interval Δt small enough, the velocity can be approximated as a constant (this of course suggests that the shorter the time Interval Δt , the more accurate our approximation).

So we can use the following equation to update the position as long as the time interval is very small:

$$x(t + \Delta t) = x(t) + v\Delta t$$

where *v* is the updated velocity at each time step. This is what we are going to implement.

Let's implement this numerical integration in the code. We are going to create a loop in time. Start by setting the time-step to a small number:

```
dt=0.005
```

Now for the first tricky part: We want the loop to run (i.e. for the velocity and position of the ball to be updated, and hence for the ball to keep moving) until it hits the ground. But the ball and the ground are shapes that have size and the ball will hit the ground when its lowest point hits the highest point on the ground. When you created the positions of the ball and the ground, these were the positions of the CENTER of these objects. The size of the ground is the total length in each dimension.

Question 4: What should your "while" statement look like, such that it stops when the ball hits the ground? Although we set the top of the ground is at height zero, make the statement as general as possible. You should

have the following variables in the statement: ball.pos.y, ball.radius, ground.pos.y, ground.size.y.

Enter the appropriate while statement into your code.

After the while statement, any commands you want to be part of the loop (i.e. repeated over and over again until the ball hits the ground) need to be indented. So make sure to indent the following statements.

We will set the rate of the loop using the command:

```
rate(1.0/dt)
```

You could also just use a number in the rate: e.g. rate(100).

Question 5: What commands will you use inside the "while" loop to update the ball's velocity and position?

After writing the commands into your notebook, type them into your program. Don't forget to indent the commands. Now run your program and see what happens.

Things to check:

- **5.1** Does the ball stop when it hits the ground?
- **5.2** Can you see the ball and ground throughout the simulation?
- **5.3** Was the motion as you expected?

This last question may be somewhat difficult to answer. What you should expect is that the ball, under constant acceleration, continues to speed up as it falls until it hits the ground. It might be difficult to tell just by looking whether the ball was speeding up correctly.

One way to check is to add the ball's velocity vector on the graph. To do so, you will need to define a new object (an arrow) for the ball's velocity. Initialize it before the while loop, but after you've defined the ball's initial velocity using the command:

```
bv=arrow(pos=ball.pos, axis=ball.velocity, color=color.yellow)
```

Now we need to update the position of the vector and its length and direction (i.e. axis) inside the while loop. In this way, every time the ball moves, the velocity vector will stay attached to the ball and change its length to the current velocity. Add the following commands at the end of your while loop (but still in the while loop, so make sure the lines are indented):

```
bv.pos=ball.pos
bv.axis=ball.velocity
```

Now run your program. You should see an arrow attached to the ball that increases in size as the ball falls.

Even though one can now see clearly that the velocity is increasing, it is hard to determine at what rate it is increasing just by looking at the vector size. So we probably need some quantitative output to answer the question "5.3 Was the motion as you expected?"

You can output values to your python shell window using the "print" command. If you want to print a character string like hello world, for example, you type:

```
print 'hello world'
```

If you want to print the output of a variable, say the variable "accel", you type:

```
print accel
```

If you want to know what you are printing, you can combine the character strings and the variables:

```
print "The acceleration is:", accel
```

Try adding this last line to your program right after you define the acceleration, but before the while loop. Now run your program and make sure you get a line in your python shell window that says:

The acceleration is: -9.8

The "print" command can help us get quantitative info from our simulations. For example, in the while loop you can add the line:

```
print ball.velocity.y, ball.pos.y
```

and it will tell you the ball's velocity and position every time the program runs through the loop. Add this line and run your program. Check your python shell window for all the output. You should have 2 long columns of output. The first column is the y component of the velocity, the second column is the y component of the position.

Question 6: Are your velocities becoming increasingly negative at a constant rate? Check by subtracting a few back-to-back pairs of velocities. What should the change in velocity be at each step given your time interval "dt" and acceleration "accel"?

Question 7: Are your displacements between each step becoming larger with time? Check by subtracting a few pairs of positions.

Activity B Plotting graphs of your output

In answering questions Questions 6 and 7 in Activity A, you might have realized that it could become tedious to always check through values to see if your simulation is doing what it is suppose to. Another option is to plot the velocity and position as a function of time and check if these make sense. In this activity we will learn how to make graphs of your output.

We want to start with the program you just finished in Activity A, but we are going to alter it, so save another copy under a different name (lab3exB.py). Now we can alter this version of the program.

We are going to create a graph that plots the velocity of the ball and the position of the ball as a function of time. In order to do this, we are going to need a variable keeping track of time. We will set the initial time to be 0 by entering the following command somewhere before the while loop:

```
time = 0.0
```

Now each time we go through the commands in the loop we want to update the time. It will increase by an amount dt, so enter the following command in the while loop right after your "rate" statement:

```
time = time + dt
```

Now each time this line is encountered in the loop, the time will become its previous value plus "dt".

In order to create our graphs of position and velocity vs time, we will tell vpython that we want to create curves to graph. The first thing we need to do is tell vpython to use a special module it has designed to create graphs. In order to load this module we need to type the following line after our first line of the program (which was: from visual import *):

```
from visual.graph import *
```

So you should now have 2 statements at the beginning of your program that start with "from".

Next, we are going to open a new display window to plot the curves in. You can control a lot of the properties of the new window, but all we will insist is that the background color be in white and the foreground color (for the axes) are black. Enter the following command just before the while loop:

```
display2=gdisplay(background=color.white,foreground=color.black)
```

The next thing we need to do is tell vpython that we want to create 2 curves on our graph (one for position, one for velocity). Type in the following commands right after the display2 command:

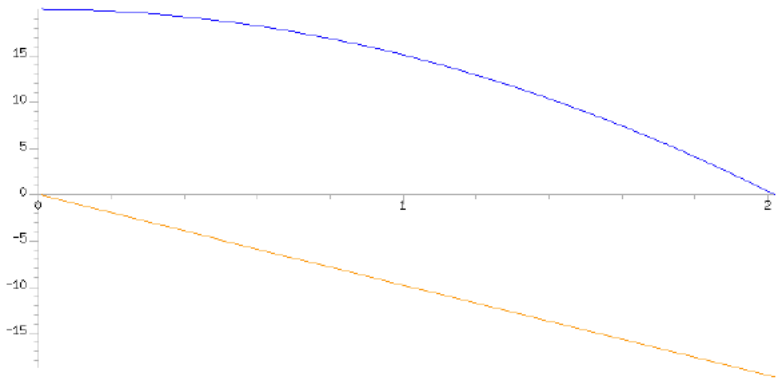
```
poscurve=gcurve(color=color.blue)
velcurve=gcurve(color=color.orange)
```

This tells the program we are going to create 2 curves: a blue curve for position and an orange curve for velocity. Note: vpython plots objects in the display it finds immediately before the object is defined. So in this example, the ball, ground and arrow all appear in the default display, but poscurve and velcurve appear in display2.

So we've told the program we want to create 2 curves, but we haven't told it what the points on this curve will be! What we want is for each iteration of the while loop, the set of points: (time, ball.pos.y) and (time, ball.velocity.y) to be graphed. We do this by appending ".plot" to our curve names and telling the program what the new point is we are plotting. In the while loop, after the ball.velocity.y and ball.pos.y are updated, add the lines:

```
poscurve.plot(pos=(time,ball.pos.y))
velcurve.plot(pos=(time,ball.velocity.y))
```

Now run your program and see what happens. You should see a second window that plots the position and velocity curves in real time. By the end of the run, it should look like this:



Question 1: Answer the question "Was the motion as you expected?" in quantitative detail. For example: is your velocity a linear function? Is your position a quadratic function? Do the signs of your position and velocity make sense?

Activity C Analytic vs Numeric Answers

You will be altering the version of your code from Activity B so let's save it under a new name: "lab3exC.py". Now we can alter this version.

You will notice that in Activities A and B, we solved the equations for position by numerically updating the position given the new velocity at each time-step. We didn't use the kinematic equation for position (which is called an "analytic equation"), namely:

$$y(t) = y(t_i) + v(t_i)[t - t_i] + 0.5a[t - t_i]^2$$

which in this example, plugging in the initial velocity, the constant acceleration and initial time becomes:

$$y(t) = 20 + 0.5 * accel * t^2$$

You might think: "We could have used the analytic equation". We could program it into the code just as easily as we did the numeric integration example. You would be right! We could do it in this case because we know the analytic solution for free fall. It turns out that many problems in physics (indeed most real-world problems) don't have an analytic solution and therefore you can't generally do this. This is why numerical integration is so important, it is harder, but (almost) always works.

To test whether numerical integration is any good, lets compare our numerical integration answer for free fall to the exact analytic answer. We won't change anything in the simulation window, but instead, we will plot another curve in our graphing window for the analytic solution.

We won't need to plot the analytic solution in the "while" loop, because we will be plotting an

analytic function and we can do this more easily with another kind of loop, called a "for" loop. We will do this after the while loop, so enter a new line after all your commands in the while loop, but make sure it is NOT indented. That way it won't be part of the loop. Start by defining a new "gcurve". Since we are going to plot the analytic solution, call it analyticposcurve:

```
analyticposcurve=gcurve(color=color.green)
```

We are going to pick a range of time values in which to plot the analytic solution.

Question 1: Run your program as it is right now and determine approximately how long it takes for the ball to fall by looking at the position vs time graph.

You should find its close to 2. So we will want our time range to be from 0 to 2. Let's choose points separated by 0.005 in this interval and plot the analytic solution at each point. Note there is no numerical integration here, we are using 0.005 as the time separation between the points in our curve.

Then we create a new loop, called a "for" loop and tell it what range of values we want to use. After the last line in your program, type the command:

```
for t in arange(0, 2, 0.005):
```

This tells the program you want to do something (the commands you will write after this) for every t in the range 0 to 2, separated by .005. We want to plot the analytic solution for each t. To do this, enter the following command after the for loop (make sure its indented so its part of the loop):

```
analyticposcurve.plot(pos=(t,20+0.5*accel.y*t**2))
```

Run your program. Notice that the analytic solution isn't plotted until the very end (i.e. the green curve doesn't show up until the end). This is because the program doesn't plot the green curve until the "for" loop at the very end of the program.

Question 2: How well did the numeric integration solution match the analytic solution?

Question 3: Look back to the "Numerical Integration" section in Activity A. Did we say that numerical integration would always work? What condition was required in order for our numeric equation for position to be valid?

Your answer to question 3 should be that the time step had to be really small so that the velocity was approximately constant. Let's see what happens when the time step gets bigger. In your program, go to the line where you defined "dt" and make the value bigger (try 0.05, 0.1 and 1.0). Run your program. Do you notice any significant difference between your analytic and numerical solutions?

Question 4: Make dt= 0.1. Where in time is the difference between blue and green lines the largest? Why do you think this is?

Question 5: Answering this question will really determine if you understand the difference between the analytic and numerical solutions: In the "for" loop, change the time step that was set to "0.005" to the new "dt" value (i.e. 0.1). Will your analytic solution now match your new numeric solution or will it give you a similar analytic solution as before? What is the effect of a shorter time interval on the analytic solution plotted in the for loop? Run your simulation to see.

Activity D 1-D Free-Fall with non-zero v_i

We are going to start Activity D with your final program from Activity B (NOT Activity C), but you will want to save it under a different name so we can alter it. Open "lab3exB.py" and save it as "lab3exD.py". Now we can alter this version of the program.

This new program will be a slight alteration to the previous program. Rather than let the ball fall from a height of 20 with an initial velocity of 0, we are going to toss the ball in the air. We will start the ball at a height of 2 and give it an initial velocity of 15. Make these two changes to your program and run it.

Question 1: Was the motion as you expected? i.e. are the shapes of the velocity and position curves correct? Were your initial velocity and position correct?

Question 2: What is the analytic solution for y(t) in this example?

Plot the analytic solution you wrote as your answer to Question 2 using a "for" loop exactly like you did in Activity C (except with the new equation). You will want to change the range of your for loop so it goes from 0 to 3.1. You can keep the interval as 0.005.

Question 3: Does your numeric solution do a good job at approximating the analytic solution? Do a "screenshot" of your graph and hand it in with your lab notebook. Make sure to label it accordingly (i.e. QD3fig). (If you don't know how to take a screenshot, ask your TA).

Now increase your dt value until you get a significant difference between the analytic and numeric solutions.

Question 4: What dt value did you find gave you a significant difference? Take a screenshot of your graph and hand that in too (label it QD4fig).

If You Have Time Activity

In programming, it is generally frowned upon to put the same numerical values in your code multiple times, rather one names a parameter, assigns it a value once, then uses it as many times as needed (kind of how we did with dt for example).

However, when plotting the analytic graphs, we put in some numbers 'by hand' again, namely the initial height and velocity of the ball. We also put the upper limit of range of the for loop by hand (2 in Activity B, or 3.1 in this Activity D).

Supposing we had created variables for every parameter (e.g. a variable called y0 for initial height of the ball and v0 for initial velocity, and other variables if needed), in terms of the relevant variables, *write a python expression for the upper limit of range of the for loop*. *What is the physical meaning of this upper limit? What is its value for the values of y0 and v0 given in this exercise?*

Make sure to explain each variable you define. You may modify your python code with these additions to make sure the result of the expression is as expected.

[Note: This Module was written by Omar Gamel in September 2014, based on activities written by Sabine Stanley 2010-2014. Last modification by Jason Harlow Sep. 26, 2013.]