Physics 2300
Spring 2018

Name _____

Lab partner _____

Project 2: Projectile Motion

You now know enough about VPython to write your first simulation program.

The idea of a *simulation* is to program the laws of physics into the computer, and then let the computer calculate what happens as a function of time, step by step into the future. In this course those laws will usually be Newton's laws of motion, and our goal will be to predict the motion of one or more objects subject to various forces. Simulations let us do this for *any* forces and *any* initial conditions, even when no explicit formula for the motion exists.

In this project you'll simulate the motion of a projectile, first in one dimension and then in two dimensions. When the motion is purely vertical, the state of the projectile is defined by its position, y, and its velocity, v_y. These quantities are related by

$$v_y = \frac{dy}{dt} \approx \frac{\Delta y}{\Delta t} = \frac{y_{\text{final}} - y_{\text{initial}}}{\Delta t}. \tag{1}$$

In a computer simulation, we already know the current value of y and want to predict the future value. So let's solve this equation for y_{final}:

$$y_{\text{final}} \approx y_{\text{initial}} + v_y \, \Delta t. \tag{2}$$

Similarly, we can predict the future value of v_y if we know the current value as well as the acceleration:

$$v_{y,\text{final}} \approx v_{y,\text{initial}} + a_y \, \Delta t. \tag{3}$$

These equations are valid for any moving object. For a projectile moving near earth's surface without air resistance, $a_y = -g$ (taking the $+y$ direction to be upward). In general, a_y is given by Newton's second law,

$$a_y = \frac{\sum F_y}{m}, \tag{4}$$

where m is the object's mass and the various forces can depend on y, v_y, or both.

In a computer simulation of one-dimensional motion, the idea is to start with the state of the particle at $t = 0$, then use equations 2 through 4 to calculate y and v_y at $t = \Delta t$, then repeat the calculation for the next time interval, and the next, and so on. Fortunately, computers don't mind doing repetitive calculations.

But there's one remaining issue to address before we try to program these equations into a computer. Equation 2 is ambiguous regarding *which* value of v_y appears on the right-hand side. Should we use the initial value, or the final value, or some intermediate value? In the limit $\Delta t \to 0$ it wouldn't matter, but for any nonzero

value of Δt, some choices give more accurate results than others. The easiest choice is to use the *initial* value of v_y, since we already know this value without any further computation. Similarly, the simplest choice in equation 3 is to use the initial value of a_y on the right-hand side.

With these choices, we can use the following Python code to simulate projectile motion in one dimension without air resistance:

```
while y > 0:
    ay = -g
    y += vy * dt    # use old vy to calculate new y
    vy += ay * dt   # use old ay to calculate new vy
    t += dt
```

This simple procedure is called the *Euler algorithm*, after the mathematician Leonard Euler (pronounced "oiler"). As we'll see, it is only one of many algorithms that give correct results in the limit $\Delta t \to 0$.

Exercise: Write a VPython program called `Projectile1` to simulate the motion of a dropped ball moving only in the vertical dimension, using the Euler algorithm as written in the code fragment above. Represent the ball in the 3D graphics scene as a sphere, and make a very shallow box at $y = 0$ to represent the ground. Use a light background color for eventual printing. The ball should leave a trail of dots as it moves. Be sure to put in the necessary code to initialize the variables (including `g`, putting all values in SI units), add a `rate` function inside the simulation loop, and update the ball's `pos` attribute during each loop iteration. Also be sure to format your code to make it easy to read, with appropriate comments. Use a time step (`dt`) of 0.1 second. Start the ball at time zero with a height of 10 meters and a velocity of zero. Notice that I've written the loop to terminate when the ball is no longer above $y = 0$. Test your program and make sure the animated motion looks reasonable.

Exercise: Most of the space on your graphics canvas is wasted. To fix this, set `scene.center` to half the ball's starting height (effectively pointing the "camera" at the middle of the trajectory), and set `scene.width` to 400 or less.

Exercise: Let's focus our attention on the *time* when the ball hits the ground, and on the *final velocity* upon impact. To see the numerical values of these quantities, add the following line to the end of your program:

```
print("Ball lands at t =", t, "seconds, with velocity", vy, "m/s")
```

Here we're passing five successive parameters to the `print` function: three quoted strings that are always the same (called *literal* strings), and the two variables whose values we want to see. If you look closely, you'll notice that the `print` function adds a space between successive items in the output.

Exercise: Are the printed values of the landing time and velocity what you would expect? Do a short calculation in the space below to show the expected values, as you would predict them in an introductory physics class.

Exercise: The time and velocity printed by your program do not apply at the instant when the ball reaches $y = 0$, because that instant occurs somewhere in the middle of the final time interval. Add some code after the end of the `while` loop to estimate the time when the ball actually reaches $y = 0$. (Hint: Use the final values of `y` and `vy` to make your estimate. The improved time estimate still won't be exact, but it will be much more accurate than what your program has been printing so far.) Have the program print out the improved value of t instead, as well as an improved estimate of the velocity at this time. Write your new code and your new results in the space below. Please have your instructor *check* your answer to this exercise before you go on to the next.

Exercise: The inaccuracy caused by the nonzero size of `dt` is called *truncation error*. You can reduce the size of the truncation error by making `dt` smaller. Try it! (You'll want to adjust the parameter of the `rate` function to make the program run faster when `dt` is small. The slick way to do this is to make the parameter a *formula* that depends on `dt`. You'll also want to set the ball's `interval` attribute in a similar way, so the dot spacings are the same for any `dt`.) How small must `dt` be to give results that are accurate to four significant figures? Justify your answer.

Air resistance

There's not much point in writing a computer simulation when you can calculate the exact answer so easily. So let's make the problem more difficult by adding some air resistance. At normal speeds, the force of air resistance is approximately proportional to the *square* of the projectile's velocity. This is because a faster projectile not only collides with *more* air molecules per unit time, but also imparts more momentum to each molecule it hits. So we can write the magnitude of the air force as

$$|\vec{F}_{\text{air}}| = c|\vec{v}|^2, \tag{5}$$

for some constant c that will depend on the size and shape of the object and the density of the air. The *direction* of the air force is always directly opposite to the direction of $\vec{v}$ (at least for a symmetrical, nonspinning projectile).

Exercise: Assuming that the motion is purely in the y direction, write down a formula for the y component of the air force, in terms of v_y. Your formula should have the correct sign for *both* possible signs of v_y. (Hint: Use an absolute value function.) If you have any doubt about whether you've found the correct formula, have your instructor check it before you go on.

Exercise: Now modify your `Projectile1` program to include air resistance. Define a new variable called `drag`, equal to the coefficient c in equation 5 divided by the ball's mass. Then add a term for air resistance to the line that calculates `ay`. Python's absolute value function is called `abs()`. Run the program for the following values of `drag`: 0 (to check that you get the same results as before), 0.01, 0.1, and 1.0. Use the same initial conditions as before, with a time step small enough to give about four significant figures. Write down the results for the time of flight and final speed below.

Question: What are the SI units of the `drag` constant in your program?

Question: How can you tell that your program is accurate to about four significant figures, when you no longer have an "exact" result to compare to?

Exercise: Modify your `Projectile1` program to plot a graph showing the ball's velocity as a function of time. (By default the graph will appear above or below the graphics canvas, and you may leave it there if you like. If you would prefer to place the graph to the right of the canvas, you can do so by creating the graph first, setting its attribute `align="right"`, and immediately plotting at least one point on the graph to make it appear before you set up the canvas.) Use the `xtitle` and `ytitle` attributes to label both axes of the graph appropriately, including the units of the plotted quantities. Use the `interval` parameter of the `gdots` function to avoid plotting a dot for every loop iteration (which would be pretty slow). Run your program again with `drag` equal to 1.0, and print the whole window including your canvas and graph. Discuss the results briefly.

Exercise: When the projectile is no longer accelerating, the forces acting on it must be in balance. Use this fact to calculate your projectile's terminal speed by hand, and compare to the result of your computer simulation.

A better algorithm

Today's computers are fast enough that so far, you shouldn't have had to wait long for answers accurate to four signficant figures. Still, the Euler algorithm is sufficiently inaccurate that you've needed to use pretty small values of `dt`, making the calculation rather lengthy. Fortunately, it isn't hard to improve the Euler algorithm.

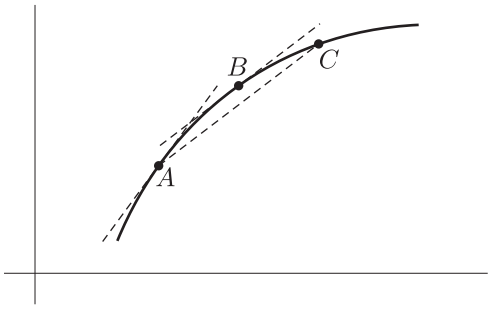

Figure 1: The derivative of a function at the middle of an interval (point B) is a much better approximation to the average slope (AC) than the derivative at the beginning of the interval (point A).

Remember that the Euler algorithm uses the values of v_y and a_y at the *beginning* of the time interval to estimate the changes in the position and velocity, respectively. A much better approximation would be to instead use the values of v_y and a_y at the *middle* of the time interval (see Figure 1). Unfortunately, these values are not yet known. But even a rough estimate of these values should be better than none at all. Here is an improved algorithm that uses such a rough estimate:

1. Use the values of v_y and a_y at the beginning of the interval to estimate the position and velocity at the middle of the time interval.

2. Use the estimated position and velocity at the middle of the interval to calculate an estimated acceleration at the middle of the interval.

3. Use the estimated v_y and a_y at the middle of the interval to calculate the changes in y and v_y over the whole interval.

This procedure is called the *Euler-Richardson algorithm*, also known as the *second-order Runge-Kutta algorithm*.

Here is an implementation of the Euler-Richardson algorithm in Python for a projectile moving in one dimension, without air resistance:

```
while y > 0:
    ay = -g                     # ay at beginning of interval
    ymid = y + vy*0.5*dt        # y at middle of interval
    vymid = vy + ay*0.5*dt      # vy at middle of interval
    aymid = -g                  # ay at middle of interval
    y += vymid * dt
    vy += aymid * dt
    t += dt
```

The acceleration calculations in this example aren't very interesting, because a_y doesn't depend on y or v_y. Still, the basic idea is to estimate y, v_y, and a_y in the middle of the interval and then use these values to update y and v_y. Although each step of the Euler-Richardson algorithm requires roughly twice as much calculation as the original Euler algorithm, it is usually many times more accurate and therefore allows us to use a much larger time interval.

Exercise: Write down the correct modifications to the lines that calculate `ay` and `aymid`, for a projectile falling *with* air resistance. (Be careful to use the correct velocity value when calculating `aymid`!)

Question: One of the lines in the Euler-Richardson implementation above is not needed, even when there's air resistance. Which line is it, and why do you think I included it if it isn't needed?

Exercise: Modify your `Projectile1` program to use the Euler-Richardson algorithm. For a `drag` constant of 0.1 and the same initial conditions as before ($y = 10$ m, $v_y = 0$), how small must you now make `dt` to get answers accurate to four significant figures? (You should find that `dt` can now be *significantly* larger than before. If this isn't what you find, there's probably an error in your implementation of the Euler-Richardson algorithm.)

Your `Projectile1` program is now finished. Please make sure that it contains plenty of comments and is well-enough formatted to be easily legible to human readers.

Two-dimensional projectile motion

Simulating projectile motion is only slightly more difficult in two dimensions than in one. To do so you'll need an x variable for every y variable, and about twice as many lines of code to initialize these variables and update them within the simulation loop.

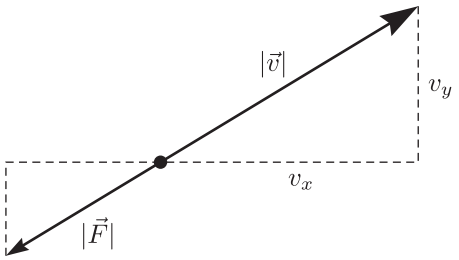

Figure 2: Assuming that the drag force is always opposite to the velocity vector, the similar triangles in this diagram can be used to express the force components in terms of the velocity components.

To calculate the x and y components of the drag force, it's helpful to draw a picture (see Figure 2).

Exercise: Finish labeling Figure 2, and use it to derive formulas for the x and y components of the drag force, written in terms of v_x and v_y. The magnitude of the drag force is again given by equation 5. (Hint: This is not an easy exercise if you've never done this sort of thing before. *Do not simply guess the answers!* You should find that the correct formula for F_x involves *both* v_x and v_y. Have your instructor check your answers before you go on.)

Exercise: In the space below, write the code to implement the Euler-Richardson algorithm for the motion of a projectile in two dimensions, with air resistance. The Python function for taking a square root is `sqrt()`. To square a quantity you can either just multiply it by itself, or use the Python exponentiation operator, `**`. As mentioned above, for every line that calculates a y component, you'll need a corresponding line for the x component. Think carefully about the correct *order* of these lines, remembering that to calculate the acceleration at the middle of the interval, you need to know both v_x and v_y at the middle.

Exercise: Create a new VPython program called `Projectile2` to simulate projectile motion in two dimensions, for the specific case of a ball launched from the origin at a given initial speed and angle that are set near the top of the program. Again use a sphere to represent the ball, leaving a trail as it moves, and use a very shallow box to represent the ground. Allow for the ball to travel as far as about 150 meters in the x direction before it lands, sizing the box and setting `scene.center` appropriately. Set the background to a light color for eventual printing. Use the Euler-Richardson algorithm, with a time step of 0.01 s. Run your program and check that everything seems to be working.

Exercise: Add code to your program to calculate and display the landing time, the value of x at this time (that is, the *range* of the projectile), and the projectile's maximum height. Use interpolation for the first two quantities, as you did in `Projectile1`. For the maximum height, you'll need to test during each loop iteration whether the current height is more than the previous maximum. To do this you can use an `if` statement, whose syntax is similar to that of a `while` loop:

```
if y > ymax:
    ymax = y
```

Use `print` functions to display all three of your calculated results, along with the initial speed and angle, and the `drag` constant.

Exercise: Check that your `Projectile2` program gives the expected results when there is no air resistance, and briefly summarize this check.

Exercise: For the rest of this project there is no need to display the numerical results to so many decimal places. To round these quantities appropriately, you can use the following (admittedly arcane) syntax:

```
"Maximum height = {:.2f}".format(ymax)
```

Here we're creating a string object (in quotes) and then calling its associated `format` function with the parameter `ymax`. This function replaces the curly braces, and what's between them, with the value of `ymax`, rounded to two decimal places. (To change this to three decimal places, you would just change `.2f` to `.3f`; the `f` stands for *floating-point* format.) Make the needed changes to display all three of the calculated results to just two decimal places.

A graphical user interface

Are you tired of having to edit your code and rerun your programs every time you want to change one of the constants or initial conditions? A more convenient approach—at least if you'll be running a simulation more than a handful of times—is to create a *graphical user interface*, or *GUI*, that lets you adjust these numbers and repeat the simulation while the program is running.

Creating a graphical user interface for a computer program can be a lot of work. You need to plan out exactly how broad a set of options to offer the user who runs your program, then design a set of graphical *controls* (buttons, sliders, etc.) to control those options, and finally write the code to display the controls and accept input from them. Fortunately, VPython makes these tasks about as easy as possible. The documentation refers to its GUI controls as *widgets*, and in this project we'll use three of them: the button, the slider, and the dynamic text widget.

Here's some minimal code to create a button that merely displays a message:

```
def launch(b):
    print("Launching the projectile!")
button(text="Launch!", bind=launch)
```

The first two lines define a *new function* called `launch`, and the last line creates a button that is *bound* to this function, so the function is called whenever the button is pressed. (Don't worry about the parameter `b` of this function; although a parameter name is required, there is no need to make use of it in this project.)

Exercise: Put this code into your `Projectile2` program and try it.

You might wonder how to control *where* the button appears. VPython doesn't give you nearly as much control over the placement as would an environment for developing commercial software. By default, new widgets are placed immediately below the `scene` canvas, in what's called its *caption*. There are a few other placement options that you can read about on the Widgets documentation page if you like.

More importantly, your Launch! button doesn't yet do what we want, namely launch the projectile! In a local installation of VPython you could make it do so by moving all your initialization and simulation code into the definition of the new `launch` function. But the GlowScript environment doesn't allow a `rate` function to appear inside a bound function, so we have to do it a different way. Although it's more difficult in the present context, the following program structure will also be more useful in future projects.

The basic idea is to turn your `while` loop into an *infinite* loop that runs forever, but to execute most of the code inside the loop only if the simulation is supposed to be "running". The code on the following page provides a basic outline.

```
while True:
    rate(100)
    if running:
        # carry out an Euler-Richardson step
        if y < 0:
            running = False
            # print out results
```

The big new idea here is the introduction of a *boolean* variable (named after logician George Boole) that I've chosen to call `running`, whose value is always one of the two boolean constants `True` or `False` (note that these are capitalized in Python). When `running` is `False`, we merely call the `rate` function to delay a bit before the next loop iteration. When `running` is `True`, we carry out a time-integration step (using the code you've already written, omitted here for brevity), then test whether the ball has dropped below ground level, in which case we set `running = False` and print out the results.

Exercise: Insert this new code into your `Projectile2` program, replacing the two comments with the code you've already written to carry out the time step and print the results. Also add a line near the top of the program to initially set `running = True`. Test the program to verify that it works exactly as before.

Exercise: You're now ready to activate your Launch! button. To do so, insert the following two lines into the indented body of the `launch` function definition:

```
global running
running = True
```

Also change your initialization line near the top of the program to set `running = False` instead of `True`. Test your program again and verify that the projectile is not launched until you press the button.

Exercise: To allow multiple launches, simply move all of the relevant code to initialize the variables into your `launch` function. You'll also need to add their names, separated by commas, to the `global` statement. Leave the initializations of `dt` and `g` outside the function definition, since those values never change. Check that you can now launch the projectile repeatedly.

Before going on, let me pause to explain that `global` statement. In Python, any variable that you change inside a function definition is *local* by default, meaning that it exists only within the function definition and not outside it. In longer programs this behavior is a very good thing, because it frees you, when you're writing a function definition, from having to worry about which variable names have already been used elsewhere in the program. But this means that when you *do* want to change a *global* variable (that is, one that belongs to the larger program), you need to "declare" it as `global` inside your function. (JavaScript, by contrast, has the

opposite behavior, making all variables global by default and forcing you to declare them with a `var` statement if you want them to be local to a particular function.) Notice that you need to use `global` only if your function is *changing* the variable; you don't need it just to "read" the value of a variable. And for technical reasons, this means that you don't need to declare graphics objects as global variables just to change their attributes.

Now let's add some controls to let you vary the launch conditions. The best way to adjust a numerical parameter is usually with a slider control. Here is how you can add one to adjust the launch angle:

```
def adjustAngle(s):
    pass    # function does nothing for now
scene.append_to_caption("\n\n")
angleSlider = slider(left=10, min=0, max=90, step=1, value=45,
                     bind=adjustAngle)
```

The `slider` function creates the slider, and most of its parameters indicate which numerical values to associate with the allowed slider positions. The `value` parameter is set to an initial value (here intended to be 45 degrees), but will change when you actually adjust the slider. The `left` parameter and the `append_to_caption` function on the previous line merely insert some space around the slider in the window layout ("`\n`" is the code for "new line"). The `bind` parameter, as before, binds a function to this control; that function (`adjustAngle`) will be called whenever you adjust the slider. (Again, don't worry about the unused parameter that I've called `s`.) For now I've used Python's `pass` statement to make the function do nothing.

Exercise: Put this code into your program, and check that the slider appears underneath the Launch button. Then modify the code that sets the ball's initial launch velocity so it uses `angleSlider.value` instead of whatever angle you were giving it before. You should now be able to launch multiple projectiles at varying angles. Try it!

Exercise: The only problem now is that you can't tell exactly what angle the slider is set to—at least not until you actually click Launch! and see the output of your `print` function. To give the slider a numerical display, add the following code right after the line that creates the slider:

```
scene.append_to_caption("    Angle = ")
angleSliderReadout = wtext(text="45 degrees")
```

Then replace the `pass` statement in the `adjustAngle` function with:

```
angleSliderReadout.text = angleSlider.value + " degrees"
```

Verify that the slider's numerical readout now works as it should.

Exercise: Add two more sliders, with numerical readouts, to control the ball's initial speed and the `drag` constant. Let the launch speed range from 0 to 50 m/s in increments of 1 m/s, and let the `drag` constant range from 0 to 1.0 in increments of 0.005 (in SI units). Test these sliders to make sure they work as they should.

Exercise: Add a second button (on the same caption line as the Launch! button) to clear away the trails from previous launches and start over. To do this, the bound function should set the ball's position back to the origin and then call the ball's `clear_trail` function (with no parameters). This button finishes your `Projectile2` program, so look over everything and make any final tweaks to the graphics, the GUI elements, the code, and the comments before turning it in.

Exercise: Run your `Projectile2` program, experimenting with various values of the drag coefficient, launch speed, and launch angle. For a drag coefficient of 0.1 and a launch speed of 25 m/s, what launch angle gives the maximum range? Record your data below and explain why you would expect the optimum angle to be less when there is air resistance. Also make a printout of your program window, showing the trails and the numerical results from three (or more) launches with significantly different settings.

Exercise: The maximum speed of a batted baseball is about 110 mph, or about 50 m/s. At this speed, the ball's drag coefficient (as defined in your program) is approximately 0.005 m^{-1}. Using your program with these inputs, estimate the maximum range of a batted baseball, and the angle at which the maximum range is attained. Write down and justify your results below. Is your answer reasonable, compared to the typical distance of an outfield fence from home plate (about 350–400 feet)? For the same initial speed and angle, how far would the baseball go without air resistance?

Question: Out of all the coding tasks and exercises you did in this project, which was the most difficult and why?

Question: Briefly discuss how you and your lab partner worked together on this project. Did one or the other of you find it easier to do the coding, or to understand the physics? Did you find enough time to do all your work together, or did you do some work separately? Please include an estimate of your own percentage contribution to this project (ideally 50%, but unlikely to be exactly 50%; please be as honest about this as you can).

Congratulations—you're now finished with this project! Please turn in these instruction pages, with answers written in the spaces provided, as your lab report. Be sure to attach your two printouts of the `Projectile1` and `Projectile2` program results. Turn in your code as before, either sending it by email or putting it into a public GlowScript folder and writing the folder URL below.