Physics 2300
Spring 2018

Name _____

Lab partner _____

Project 3: Pendulum

In this project you will explore the behavior of a pendulum. There is no better example of a system that seems simple at first but turns out to hold intricate layers of complexity.

Figure 1 shows the basic setup: a fixed pivot, a massless rod of length L, and a point-mass m at the end, swinging in the plane of the page. It's easiest to analyze the motion in terms of torque and angular acceleration; recall that the angular version of Newton's second law is

$$\sum \tau = I\alpha. \tag{1}$$

On the left-hand side of this equation is the sum of all the torques acting on the object. On the right-hand side, I is the object's rotational inertia, simply mL^2 for our pendulum. The angular acceleration α is defined analogously to the ordinary acceleration:

$$\alpha = \frac{d\omega}{dt} = \frac{d^2\theta}{dt^2}, \tag{2}$$

where ω is the angular velocity and θ is assumed to be in radians.

From the diagram we see that the torque due to gravity is

$$\tau_g = -L|\vec{F}_g| \sin\theta = -Lmg \sin\theta, \tag{3}$$

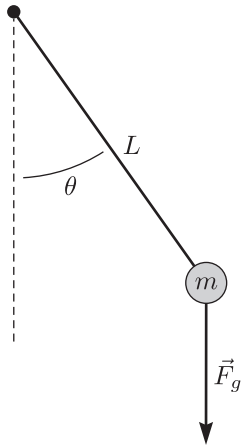

Figure 1: A simple pendulum of mass m and length L, acted upon by a gravitational force $\vec{F}_g$.

where the minus sign indicates that the torque is negative (clockwise) when θ is positive, and vice-versa. If there is no friction or other torque acting, then Newton's law (equation 1) says simply

$$-Lmg\sin\theta = mL^2\alpha, \qquad \text{or} \qquad \alpha = -\frac{g}{L}\sin\theta. \tag{4}$$

Although I drew the diagram for a small positive value of θ, this equation is valid for all angles—even angles greater than 90°, if the rod is rigid.

Because the mass m has canceled out of equation 4, there will be no need to specify a mass in your pendulum simulation. You might think you *do* need to specify a length L, and also a value of g (depending on which planet the pendulum lives on), but in fact you don't, because we still have the freedom to choose our *units* for measuring distance and time. (Yes, there are Unit Police who tell us we must always use SI units for all scientific work, but actual working scientists ignore the Unit Police and use whatever units are most appropriate for the work they're doing.) Our freedom to choose units means that whatever the actual values of L and g, we can simply specify that we're using units in which both of these constants are equal to 1. These units are called *natural units*, because they are natural to the physics that we're studying. One advantage of using natural units is that equation 4 becomes simply $\alpha = -\sin\theta$. But the real advantage is that your simulation will be applicable to *any* pendulum on *any* planet. The price is that in order to apply your results to a particular pendulum, you may need to convert them from natural units to more conventional units after running the simulation.

Exercise: Suppose (just for this exercise) that we want to study a pendulum on earth ($g = 9.8$ m/s^2) that is half a meter long. Then our unit of distance will be a half meter; let's call this unit the *ham*. Define a corresponding natural unit of time, called the *tic*, such that $g = 1.0$ ham/tic^2. How many seconds are in a tic?

Exercise: In order to draw a pendulum in the VPython graphics environment, you will need to express the position of the pendulum bob in rectangular coordinates. Taking the origin to be at the pivot and the x and y directions to be to the right and upward, respectively, what are the formulas for x and y in terms of θ?

Exercise: Write a VPython program called `Pendulum1` to simulate the motion of a pendulum swinging under the influence of gravity and no other torques. Use the Euler algorithm, with the variables `theta` (in radians), `omega`, and `alpha` playing the roles of y, v_y, and a_y in your original `Projectile1` simulation. Check carefully that the order of the lines in the algorithm is the same as in the example on page 2 of Project 2. In the 3D graphics space, represent the pendulum bob as a sphere, the pendulum rod as a cylinder, and the pivot at the other end of the rod as a short cylinder perpendicular to the plane of motion. You may optionally wish to draw a stand of some sort to "support" the pivot. Use natural units with $g = L = 1$ (so you don't need these variables at all!), and a time step of 0.01 in these units. Choose initial conditions such that the amplitude of the swing will be fairly small. Once everything seems to be working, let the simulation run for a while and describe what you observe.

Exercise: Add a graph of θ (vertically) vs. t (horizontally) to your program. Be sure to label the axes appropriately. Set the plotting `interval` to 10, in order to reduce the slow-down that will occur as more and more points are added to the graph. (Throughout the rest of this project, use your discretion to adjust the plotting interval as appropriate.) Use your graph to determine the approximate period of the pendulum in natural units, and write down the result. Does your result agree with what you learned in introductory physics? Explain carefully.

Accuracy over long time periods

Exercise: Set your `Pendulum1` program to run for about 50 units of time, then run it. Sketch the appearance of the graph of θ vs. t in the space below, and explain what's wrong with it.

Exercise: Work out a formula for the total energy (kinetic plus gravitational) of the pendulum, in terms of θ, ω, and constants. Take the gravitational energy to be zero at the lowest point of the pendulum's swing. (Use a picture and some trigonometry to relate θ to the height of the pendulum bob. Do not set $g = L = 1$ in this exercise.)

Exercise: Add some lines to your program to compute and print the total energy of the pendulum at both the beginning and the end of the simulation. For the purpose of this computation you should continue to take L, g, and m all to equal 1 (in natural units). Write the results below, and explain why something must be wrong.

Exercise: Run the simulation again with a smaller value of `dt`. (Be sure to adjust your `rate` function and graphing `interval` accordingly.) How do the results change? Is this what you would expect?

The preceding exercises illustrate a common problem with numerical calculations: small truncation errors can add up over time to produce large inaccuracies. When a quantity such as the total energy is supposed to be exactly conserved, you should always monitor this quantity for any significant drift that might come from compounded truncation errors. While using a smaller time step can provide a

brute-force solution to this problem, it's usually better to switch to a more accurate algorithm (if possible).

Exercise: Modify your `Pendulum1` program to use the Euler-Richardson algorithm instead of the Euler algorithm. For a time step of 0.01 and a total running time of 100 time units, how much does the total energy drift?

At this point, please make a copy of your `Pendulum1` program and call it `Pendulum2`. The remaining changes that you'll make to `Pendulum1` will not be needed for your subsequent work, so this will save you from having to undo those changes later.

Large-angle motion

Now that you've minimized your program's numerical inaccuracies, let's examine how the motion varies when the amplitude is changed.

Although you can read the approximate period of the pendulum from your graph, it's much more accurate to have the program calculate and print the period. Here's one way to do this: Introduce a variable called `lastTheta`, and set it equal to `theta` just before each time `theta` is updated. After updating `theta`, check whether `theta` is positive and `lastTheta` is negative. If so, the pendulum has just crossed $\theta = 0$ while moving from left to right. In this case, do an interpolation to compute the time when the crossing occurred (like when you computed the time the projectile landed in the previous project) and remember this time in another variable. The time between one one such crossing and the next equals the period of the pendulum.

Exercise: In your `Pendulum1` program, comment-out the code to calculate and display the pendulum's initial and final energy (by placing a `#` character in front of each line). Then implement the algorithm just described for determining the period of the pendulum. Check to make sure it works. Also modify your simulation loop so it stops right after calculating and printing the period.

Exercise: Modify your `Pendulum1` program to run the simulation successively for amplitudes 10°, 20°, and so on up to 170°, printing out a table with the amplitude (in degrees) in the first column and the period in the second column. To do this, you'll want to embed your entire simulation loop inside a larger `while` loop that loops over the amplitude values, reinitializing the pendulum each time through. Be sure to adjust the indentation appropriately, and be careful converting angles from degrees to radians and then back again. If you would like to reinitialize the graph

of θ vs. t for each new amplitude, you can do this by calling the `delete()` function of your `gdots` object. Another useful trick is to put a tab character, `"\t"`, between the amplitude and period in your `print` output. Check that everything works, so you get a table as intended in the print area.

Of course a graph of period vs. amplitude would be better than a mere table of numbers. You could easily produce such a graph right in the GlowScript window, by adding just a few lines to your code. Often, however, for graphing final results we use a different software environment than what we used for the simulation itself. In this case, it is convenient to use an ordinary spreadsheet program.

Exercise: Copy your table of amplitudes and periods into a spreadsheet (Excel, Google Sheets, etc.), then make a scatter plot of period (vertically) vs. amplitude (horizontally), and label it appropriately. Arrange the graph so it fits on a single printed page with your data table, then print this page and attach it to your lab report.

Your `Pendulum1` program is now finished, so please make sure the code is well organized and adequately commented.

Friction and driving forces

Now let's add some further complications: Friction to slow the pendulum down, and a periodic external torque that continually adds energy to the system.

A simple way to add friction is to subtract a term proportional to ω from the right-hand side of equation 4 for the angular acceleration:

$$\alpha = -\frac{g}{L}\sin\theta - c\,\omega. \tag{5}$$

A linear resistive force (or torque) of this form is called *damping*, and the coefficient c is called the damping constant.

Exercise: In your `Pendulum2` program, remove the code that calculates and displays the initial and final energies (since we no longer expect energy to be conserved). Then add a damping term to the lines that compute `alpha` and `alphaMid`. Run the program for various values of the damping constant, ranging up to about 2.0. Describe the results briefly.

Damping removes energy from the pendulum, so the motion dies out. But we can keep the motion going by continually applying an external torque to the

pendulum. A simple yet interesting way to do this is to add a term to equation 5 that is sinusoidal in time:

$$\alpha = -\frac{g}{L}\sin\theta - c\,\omega + A\sin(ft). \tag{6}$$

This additional term is called a *driving* term. Physically, it would represent a smooth, back-and-forth twisting force, presumably applied at the pivot, that is unaffected by the pendulum's position and speed. The constants A and f represent the amplitude and angular frequency of the driving torque.

Because the Euler-Richardson algorithm requires two calculations of α (one at the beginning of the time interval and one at the middle), and because the formula for α is getting rather complex, it's a good idea to move this calculation into a separate function. Let's call the function `torque`, which is the same as angular acceleration in our system of units. This function should accept three parameters—θ, ω, and t—and return the value of the torque. Assuming that the constants `damp`, `driveAmp`, and `driveFreq` have already been defined, the `torque` function can be defined as follows:

```
def torque(theta, omega, t):
    return -sin(theta) - damp*omega + driveAmp*sin(driveFreq*t)
```

I should emphasize that the variables `theta`, `omega`, and `t` in this function definition (called *formal parameters*) are logically independent of the variables of the same names in the rest of your program; here we could just as well call them Larry, Moe, and Curly (at least as far as the computer is concerned). When you call the `torque` function from elsewhere in your program you can pass it any values you like for these three variables, and it will return the appropriate torque.

Exercise: Add this function definition to your program, and use it appropriately each time your program computes the angular acceleration. (Be sure to pass the correct values to your `torque` function for the mid-point calculation!) Test your program with the damping constant equal to 0.5, the drive amplitude equal to 0.5, and the drive frequency equal to 2/3. You should find that after an initial "transient" behavior, the motion becomes periodic. What is the approximate period of the motion, and what is the significance of this value?

Chaos!

Exercise: Now increase the drive amplitude to 1.2, and run the program again (for about 100 time units). Describe the behavior briefly.

When the drive amplitude is sufficiently large, the pendulum can swing over the top of the pivot and the motion becomes much more complex. In some cases, such as the one you just saw, the motion never settles down to become periodic. Because the motion seems so unpredictable, the word "chaos" often comes to mind.

Chaos is actually a technical term in dynamics, used to describe behavior that is unpredictable in practice because even a tiny change in the initial condition results in an exponentially increasing change in the motion. A good way to test for chaotic behavior is to run a simulation twice, with two slightly different initial conditions, and monitor the difference in the motion as a function of time.

Exercise: Make a copy of your `Pendulum2` program and call it `Pendulum3`, for later use. Then modify `Pendulum2` to simulate the motion of *two* damped, driven pendulums simultaneously. Use two sets of variables for the two separate pendulums, and simulate both motions using a single `while` loop in your code. Call the new variables `theta2`, `omega2`, and so on. I'll refer to the angle of the original pendulum as θ_1, but you may leave it as simply `theta` in your code. In the 3D graphics space, put the second pendulum in front of the first, suspended from the same pivot. On your graph, plot both θ_1 and θ_2 vs. time, using two different colors (it's a nice touch to use the same colors for the pendulums in the 3D graphics space that you use on the graph). Start one pendulum at $\theta = 0$ and the other at $\theta = 0.001$ (radians), with $\omega = 0$ initially for both, and run for about 200 units of time. Use the same damping and driving constants as in the two previous exercises, and describe the results for both values of the driving amplitude. Print the more interesting of the two graphs, and attach it to your lab report. (To print a graph, first make a screen capture of just the graph, then open it in a suitable graphics program and print it from there. Consult with your instructor if you are unsure of how to do this.)

Exercise: To better see how the motions of the two pendulums relate to each other, modify your `Pendulum2` program to produce a second graph, plotting $\ln|\theta_2 - \theta_1|$ vs. t. The VPython function for the natural logarithm is `log()`. Run your program using the same constants as before, including both values (0.5 and 1.2) of the drive amplitude. This time it should suffice to run for about 80 units of time. Print the log-difference plot for both cases, and discuss the results in some detail. Why do both plots have several downward-pointing spikes?

The roughly exponential behavior of $\theta_2 - \theta_1$ as a function of time is a feature of many dynamical systems. We can write this behavior as

$$\theta_2 - \theta_1 \approx (\text{constant}) \times e^{\lambda t}, \tag{7}$$

where λ is called a *Lyapunov exponent*. When λ is positive the system is chaotic: even the smallest difference in initial conditions will grow over time until the two systems have radically different behavior. Since we can never know the initial conditions of a real physical system with infinite accuracy, it is effectively impossible to predict the behavior of a chaotic system over long time periods. Although our damped and driven pendulum is a somewhat contrived system, chaotic behavior is also quite common in the real world.

Exercise: Estimate the Lyapunov exponent of the pendulum system for each of the values of the constants used in the previous exercise. (Hint: Use a ruler to draw a reasonable straight line to fit your logarithmic graphs of $\theta_2 - \theta_1$, ignoring the downward-pointing spikes.)

Your `Pendulum2` program is now finished, so check everything over and be sure that the code is well organized and adequately commented.

When is the motion chaotic?

So far you have explored the motion of the damped and driven pendulum for only two sets of damping and driving parameters—one that produces chaotic behavior and one that doesn't. Your final task will be to map out in more detail which conditions produce chaos.

In principle you could vary the damping constant, the driving frequency, *and* the driving amplitude—but thoroughly exploring that three-parameter space would take a long time. Instead, you will keep the damping constant and the driving frequency fixed at the same values as before, and vary only the driving amplitude.

Exercise: In your `Pendulum3` program (which simulates only a single damped and driven pendulum), change the simulation loop to run indefinitely, but add a GUI button to pause and resume the simulation. Then add a slider to adjust the driving amplitude to any value between 0 and 2.0, in increments of 0.01. Be sure to create a numerical readout for the slider. Test these controls to make sure they work.

Exercise: Change the graph in `Pendulum3` to plot ω vertically vs. θ horizontally, instead of θ vs. t. This new graph is called a *phase space plot*. Use the `xmin` and `xmax` parameters of the `graph` function to set the horizontal scale on the graph to run from $-\pi$ to $+\pi$, and then, in your simulation loop, insert some `if` statements to shift θ by 2π whenever necessary to keep it within the range of the graph. (This doesn't affect the physics, because changing an angle by 2π doesn't really change it.) Set the `size` parameter of the `gdots` object to 1, and adjust the plotting interval to compromise between a more complete plot and a reasonable execution speed. Finally, add another GUI button to the program that clears all dots from the plot, by calling the `delete` function of your `gdots` object. Again, test everything to make sure it works.

Exercise: Run your simulation and systematically explore what happens as the drive amplitude increases from 0 to 2. For some settings the motion will settle into a repeating pattern (once any "transient" behavior has died out), while for others it will be chaotic, never exactly repeating itself. Print a few of the more interesting phase space plots (combining them onto a single page if you know a way to do that), being sure to label them with the drive amplitude settings. Describe the motion in words for each of these cases. Also, in the space below, make a list of the drive amplitude values for which the motion is chaotic.

This exercise completes Project 3. Be sure to attach all the printouts to this lab report, and to submit your code by email (or by emailing a link).