Physics 2300
Spring 2018

Name _____

Lab partner _____

Project 4: Orbits

In this project you will simulate the motion of planets orbiting the sun.

The only important force in this situation is gravity. According to Newton, the gravitational force between two objects is always attractive, with magnitude

$$|\vec{F}_g| = \frac{Gm_1m_2}{r_{12}^2}, \tag{1}$$

where m_1 and m_2 are the masses of the two objects and r_{12} is the distance between them. The associated potential energy, for later reference, is

$$U_g = -\frac{Gm_1m_2}{r_{12}}. \tag{2}$$

This formula takes some getting used to, because it is always negative. The important thing, though, is that as two objects get farther apart their potential energy increases (becomes less negative).

We'll start with a single planet orbiting the sun. Because the sun is so much more massive than any planet, it's a reasonable approximation to neglect the sun's motion. Then we can put the sun at the origin of our coordinate system, and the formula for its force on the planet will simplify.

Exercise: Use equation 1 and an appropriate diagram to find the x and y components of the sun's gravitational force on a planet, in terms of the planet's coordinates x and y. *Be sure that you have the correct formulas before you try to write any code!*

Units

Rather than using SI units, it's usually a good idea to choose units that are natural for the problem being solved. In the case of planets orbiting the sun, a natural unit of distance is the *astronomical unit* (AU), defined as the average distance between the earth and the sun (about 150 million kilometers). A natural unit of time is the year (3×10^7 seconds), and a natural unit of mass is the sun's mass (2×10^{30} kg).

Exercise: What is earth's orbital speed in these natural units (assuming its orbit to be circular)? (In this exercise and the next, do not start with SI units and then convert; there's a much easier method.)

Exercise: Find the value of G in these natural units, by applying Newton's laws to earth's orbit.

Exercise: Suppose that a planet is in a circular orbit of radius r. Find a formula for its speed in terms of r, using natural units to simplify your answer.

A one-planet simulation

Exercise: Write a VPython simulation program called `Orbit1`, for a single planet orbiting the sun in the xy plane. Represent the planet as a sphere in the 3D simulation space, with a larger sphere at the origin to represent the sun. (Choose the radii of these spheres for easy visibility—not based on their sizes in the actual solar system!) Have the planet leave a trail to mark its orbital path. Also create two thin cylinders to represent the x and y axes, making each of them a few AU long. In this project you will always need to view the orbits directly from the perpendicular (z) direction, so set `scene.userspin = False` to disable rotating the view, and set `scene.fov = 0.01` to give a very small field of view (0.01 radians), as if you are viewing the scene from very far away through a telescope. Set the background color to white, since you'll be printing some of your orbital images. To simulate the planet's motion use the Euler algorithm for now, with a time step of 0.01 (in years). Start the planet at $x = 1.5$, $y = 0$, and an initial `vx` and `vy` that should give a circular orbit. Describe the result of your simulation for a running time of ten years. Try the simulation again with a time step of 0.001 (adjusting the `rate` parameter and the trail's `interval` accordingly), and discuss the implications of your results.

Exercise: Add a `totalE` function to your program, which computes and returns the planet's total energy per unit mass. Use equation 2 for the potential energy. Your function doesn't need to have any parameters, because x, y, v_x, and v_y are all global variables. Add code both before and after your simulation loop to call the `totalE` function and print out the result, labeled with appropriate text. Write down the results for a couple of different values of `dt`, and comment briefly.

Exercise: By now you should be anxious to replace the Euler algorithm in your simulation with something more accurate. Try the Euler-Richardson algorithm next (but comment-out your Euler algorithm code, rather than deleting it, so it will still be visible), and write down the results (initial and final energy) for a couple of values of `dt`. Comment briefly.

The Verlet algorithm

The Euler-Richardson algorithm is *so* much better than the Euler algorithm that you may be tempted to settle for it. However, in problems such as this where the force depends only on the positions of the particles (not on their velocities), there is another algorithm that usually does significantly better still, and is no harder to code.

You may have already noticed, while coding the Euler-Richardson algorithm, that there's no actual need to know the velocity at the interval's midpoint when this velocity won't be needed to calculate the force. In this case, you could instead combine this velocity calculation with the calculation of the updated position. So, for the x components, the equations

$$v_{x,\text{mid}} = v_{x,\text{initial}} + \tfrac{1}{2}a_{x,\text{initial}}dt \qquad \text{and} \qquad x_{\text{final}} = x_{\text{initial}} + v_{x,\text{mid}}dt \tag{3}$$

can simply be combined into the single formula

$$x_{\text{final}} = x_{\text{initial}} + v_{x,\text{initial}}dt + \tfrac{1}{2}a_{x,\text{initial}}(dt)^2. \tag{4}$$

You should recognize this formula from introductory physics: It is the *exact* formula for the motion of an object whose acceleration is *constant*. We'll continue to use it (for small dt) even when the acceleration isn't constant, as we've already been doing, in effect, with the Euler-Richardson algorithm.

Our improvement on the Euler-Richardson algorithm will be in the calculation of the updated velocity. Once we've updated the position using equation 4, we can use this updated position to calculate the acceleration at the *end* of the time interval (so long as the force depends only on the position, not on the velocity). We can then estimate the *average* acceleration as the average of the initial and final accelerations, and use this average to update the velocity:

$$v_{x,\text{final}} = v_{x,\text{initial}} + a_{x,\text{average}}dt \approx v_{x,\text{initial}} + \Big(\frac{a_{x,\text{initial}} + a_{x,\text{final}}}{2}\Big)dt. \tag{5}$$

This estimate of the average acceleration is more symmetrical, and hence more accurate in most situations, than the Euler-Richardson method of calculating a_x from an estimated value of x_{mid}. The combination of equations 4 and 5 is known as the *Verlet algorithm*, or alternatively as the *second Taylor approximation* (STA). For two-dimensional motion, of course, each step would entail updating the y components as well as the x components.

The slick way to implement the Verlet algorithm is to break equation 5 into two parts, separated by the calculation of the new acceleration, as follows:

0. Before the loop begins, calculate the initial acceleration.
1. Update the position using equation 4.
2. Update the velocity half-way, adding $\frac{1}{2}a_x dt$.
3. Update the acceleration, calculating it from the new position.
4. Finish updating the velocity, again adding $\frac{1}{2}a_x dt$.
5. Repeat steps 1 through 4 in each loop iteration.

Notice that this procedure requires only *one* evaluation of the acceleration for each time interval. In the next project, where execution speed will be an issue, this will be another significant advantage over the Euler-Richardson algorithm.

Exercise: Comment-out the Euler-Richardson code in your `Orbit1` program, then implement the Verlet algorithm. Since the acceleration needs to be calculated once outside the loop and once inside it, move this code into a separate function and call it from both places. (The function needn't have any parameters, or return any value; just make `ax` and `ay` global variables.) Test your program for the same initial conditions, running time, and values of `dt` that you used in the previous exercise, and compare the accuracy to which the two algorithms conserve energy, writing your results below. Be sure to display enough decimal places in the energy values to allow you to see the change!

Kepler's laws

In the early 1600s, Johannes Kepler was the first to discover simple mathematical laws to accurately describe the observed motions of the planets. Nowadays we number Kepler's important discoveries 1, 2, and 3; I like to throw in a 0th law that may have gone without saying in Kepler's time, but isn't at all obvious from a modern perspective:

0. Planetary orbits are closed paths; each planet returns to the same point after one full orbit.

1. The shape of each orbit is an ellipse, with the sun at one focus of the ellipse.

2. The planets move faster when they are closer to the sun, in such a way that a line drawn from the sun to any planet sweeps out equal areas in equal times.

3. The outer planets move slower than the inner ones, in such a way that the cube of the length of the ellipse's semimajor axis is proportional to the square of the period of the orbit.

Later, in 1687, Isaac Newton published a book showing how all of Kepler's laws (and much more) can be *deduced* from his more fundamental laws of motion and gravity. Even today, however, a rigorous deduction of Kepler's laws from Newton's laws requires some sophisticated and lengthy calculations. On the other hand, you now have a computer program that simulates planetary motion according to Newton's laws. Let's check, then, whether the resulting motion obeys Kepler's laws.

Exercise: (Kepler's 0th Law.) If you haven't already, try out some other initial conditions for your simulated planet. A good way to start is to make the initial values of x, y, and v_x the same as for earth, and vary the value of v_y from slightly lower than earth's speed to slightly higher. Describe the results in the space below. Does your simulated planet obey Kepler's zeroth law? How small a value of `dt` must you use to obtain consistent results?

Exercise: (Kepler's 1st Law.) An ellipse is defined as the set of all points for which the sum of the distances to the two foci is a constant. According to Kepler's first law, one focus of the ellipse should be at the sun, which is at the origin of your coordinate system. For the initial conditions suggested in the previous exercise, the other focus should be symmetrically located exactly 1 AU from the left end of the ellipse. (The second focus may lie either left or right of the first.) Adjust your initial conditions to obtain a reasonably elongated orbit, and zoom in or out until the orbit fills most of the scene. Print this image, making sure it fills most of a page, and label the two foci on the printout. Then pick at least three dissimilar points along the orbit and for each, use a ruler to measure the sum of the distances to the two foci. Show your measurements and calculations on the printout, and discuss (on the printout) whether Kepler's first law seems to hold for your orbit.

Exercise: (Kepler's 2nd Law.) Modify the `interval` setting of your moving planet so that only two or three dozen trail points appear around the orbit. Use the same `dt` and initial conditions as in the previous exercise. Run the simulation for just a single orbit and again print the resulting image, making sure it fills most of a page. Then pick three dissimilar intervals along the orbit, identical in duration, and for each, use a ruler to determine the approximate area swept out by an imaginary line from the sun to the planet. Show your measurements and calculations on the printout, and discuss (on the printout) whether Kepler's second law seems to hold for your orbit.

Exercise: (Kepler's 3rd Law.) The semimajor axis of an ellipse is defined as half of its widest width. As long as you use an initial v_x of zero, this width should be along the x axis of your image. You can then test Kepler's third law as follows. Add code to your program to determine when the planet crosses the x axis, and for each crossing, to print out (to the screen) the value of x and the time. From these data, you can calculate (by hand is fine) both the semimajor axis and the period of the orbit. Run the simulation for three dissimilar orbits, recording the data below. Then for each orbit, calculate the cube of the semimajor axis and the square of the period. Discuss whether your results are consistent with Kepler's third law.

Elongated orbits and variable time steps

By now you may have noticed that the closer your simulated planet gets to the sun, the smaller you need to make `dt` to reduce truncation errors to an acceptable level. For highly elongated orbits this is awkward, because the small value of `dt` is needed only near one end of the orbit, while a much larger value of `dt` would suffice elsewhere.

Fortunately, there's no law that says we have to use the same value of `dt` throughout the simulation. Instead, we can use what is called *adaptive step size control* to continually adjust `dt` as appropriate. Nearly all of the "professional quality" algorithms for numerically solving differential equations employ some sort of adaptive step size control. Most of these algorithms are quite complex and sophisticated—not worth our time in a first course on computer simulations. In your `Orbit1` program, however, there is a very simple way to add adaptive step size control.

Think about it: We want `dt` to be small when the planet is close to the sun (where its acceleration is large and rapidly changing) but large when the planet is farther away (where its acceleration is small and slowly changing). A natural way to accomplish this would be to make `dt` proportional to r, or to some positive power of r. Equivalently, we could make `dt` proportional to some negative power of the acceleration, $|\vec{a}|$, which is proportional to $1/r^2$:

$$\texttt{dt} \propto |\vec{a}|^{-n} \propto r^{2n}. \tag{6}$$

Let's work with $|\vec{a}|$, since this will make it easier to generalize our approach to multi-planet simulations in the next section. The optimum power of $|\vec{a}|$ is not easy to guess, but I've done a bit of pencil-and-paper analysis that seems to indicate that $n = 1$ is a good choice. In practice, this choice seems to work just fine.

Exercise: Modify your `Orbit1` program to set `dt` equal to a constant (call it `tolerance`) times $|\vec{a}|^{-1}$. This should be done at the beginning of each loop iteration. To select a good value of `tolerance`, either do a quick calculation or just try some values and see what works. Explain how you chose your value of `tolerance` in the space below.

Exercise: Use your `Orbit1` program to model the orbit of Halley's comet, which is 0.58 AU from the sun at its closest approach and has a period of 76 years. What is its maximum orbital speed in AU/year? How far from the sun is the orbit's most distant point? What did you have to do to determine these quantities?

Your `Orbit1` program is now finished, so please make sure the code is well organized, easy to read, and adequately commented.

A two-planet simulation

Don't let this get you down, but all the results of your `Orbit1` program were already known to Newton more than 300 years ago. The real advantage of a computer simulation over pencil-and-paper methods comes at the next stage, when we want to model a system that is more complex than a single particle subject to an inverse-square force. For instance, you could easily explore what happens when the force law is modified, either in some hypothetical universe or in the real solar system due to the sun's nonspherical shape or the effects of general relativity. An even more difficult problem, though, is to predict the motion of three or more mutually gravitating objects. With the exception of a few unrealistically symmetrical configurations, there are no analytic formulas for the motions of such systems.

Your next task will be to write a new computer program (`Orbit2`) to model the behavior of *two* planets orbiting the sun, including the effects of their mutual gravitational attraction. Because these effects are often small, you'll need to run the simulation for a relatively long time—long enough to observe dozens or hundreds of orbits.

Exercise: Make a copy of your `Orbit1` simulation and call it `Orbit2`. In `Orbit2`, remove the commented-out code for the Euler and Euler-Richardson algorithms, as well as the code that prints the initial and final energies (but keep the function that calculates the energy). You may also remove the cylinders that represent the x and y axes. In anticipation of adding a second planet, change the planet's variable names to `x1`, `y1`, `vx1`, and so on. Set the simulation loop to run indefinitely, but introduce a boolean `running` variable and a "Start/stop" button that you can use to pause and resume the simulation. Change the planet's initial position and velocity to simulate earth's orbit (for now). Spend some time adjusting the graphics settings, including the size of the planet and its trail (which you could leave as "points" or change to "curve"), and the trail interval and animation rate. You want to be able to simulate dozens of orbits quickly, and to discern small changes from one orbit to the next.

Exercise: Add `wtext` objects to `Orbit2` to display the elapsed time and the system's total energy in the space below the graphics scene. Update these readouts during each simulation loop iteration, and use the string `format` function to round each of them to an appropriate number of decimal places.

Exercise: Now add a second planet to your simulation, orbiting like the first under the influence of the sun's gravitational pull. Don't worry yet about including the gravitational force *between* the two planets. For simplicity, please assume that the second planet also orbits in the xy plane, and start it out in a circular orbit of some

reasonable size. In the line that calculates `dt`, use the *larger* of the two planets' accelerations. (Use the `max` function.) Draw the two planets and their trails in different colors. Be sure to update your energy function to include the second planet's energy.

Exercise: Finally, modify your simulation code to include the gravitational force between the two planets. Up until now the masses of the planets have been irrelevant, but at this point you'll have to introduce variables to represent the masses (in units of the sun's mass). To calculate the force, it's helpful to first calculate the x and y separations between the two planets, storing these in temporary variables. Be very careful to check that you're using the correct formula for the force. Also be sure to include the inter-planet potential energy in your total energy calculation. To test your program, start the planets in orbits that would be circular if they did not interact, with radii of 1 and 1.5 (about right for earth and Mars). First set both masses equal to 10^{-6}, and verify that the orbits remain essentially circular. Then set both masses equal to 0.02 (about 20 times the mass of Jupiter), and describe what happens. Is the total energy reasonably constant?

Your `Orbit2` program is now finished! It can be used to explore a great variety of scenarios; feel free to experiment. The following exercise should give you a good start.

Exercise: Set the masses of your two planets to 0.001 and 10^{-8} (`1e-8` in Python syntax), to simulate Jupiter and an asteroid. Start both in orbits that would be circular if they did not interact with each other, with Jupiter at 5.2 AU from the sun and the asteroid at 3.0 AU. Run the program for a few hundred simulated years and describe the resulting orbits. Then repeat the simulation with the asteroid at 3.1 AU, 3.2 AU, and so on up to 4.0 AU (adjusting the asteroid's speed to keep its orbit circular if Jupiter weren't there), and describe the results. For what sizes of the asteroid's orbit does the influence of Jupiter appear to be the greatest? Can you explain why? Print out two representative images, labeling each with the asteroid's initial orbit size. Observations of the actual asteroid belt show that there are gaps in it, where very few asteroids are found. At what distances from the sun would you expect to find gaps? (Continue your answer on the following page.)

Question: Which part of this project did you find most difficult? Any other comments on how it went?

Question: How well did you and your lab partner work together on this project? What is your estimate of your own percentage contribution, and your lab partner's?

Congratulations! You are finished with this project. Please turn in your code by email or by emailing a public link, as usual.