Topics in Physics: Problem Set #4

Topics: gravity, computational physics

Practice Problems (approx. 30 min)

You should try to do these problems individually. None of them should take very long to solve; if you get stuck, ask a TA for help!

1. Our solar system slowly circles the center of our galaxy, caused by a very small centripetal force. How does this force compare to earth's gravitational force?

2. Two baseballs with mass 0.2 kg float in deep space 100 meters apart. What should their velocity be so that they orbit each other in a circle? How long will one full orbit take? (Hint: the baseballs circle a point directly between them, and their gravitational attraction should cause their centripetal acceleration around that point.)

3. The first United States spacecraft to orbit the Moon arrived in 1966. Its orbit was circular with a radius of 310 km. How fast was it moving relative to the moon? How long did it take to orbit the moon?

4. The TRAPPIST-1 is a small, cool star 40 light years away. It is remarkable because we have observed 7 planets orbiting the star, all of which are similar to earth in size, and 3 of which orbit in the "habitable" zone where liquid water might exist. We were able to detect the presence of the planets because they happen to pass between TRAPPIST-1 and our vantage point, causing periodic dips in the star's brightness. As a result, we can measure the time each planet takes to complete an orbit:

Planet	Orbital period (days)
b	1.5
c	2.4
d	4
e	6
f	9.2
g	12.4
h	18.8

Table 1: Planetary orbital times for the TRAPPIST-1 system.

If we happen to know that planet e is 4.38 million km from TRAPPIST-1, what are the orbital distances of the other planets?

Experiment: simulating the solar system (approx. 120 min)

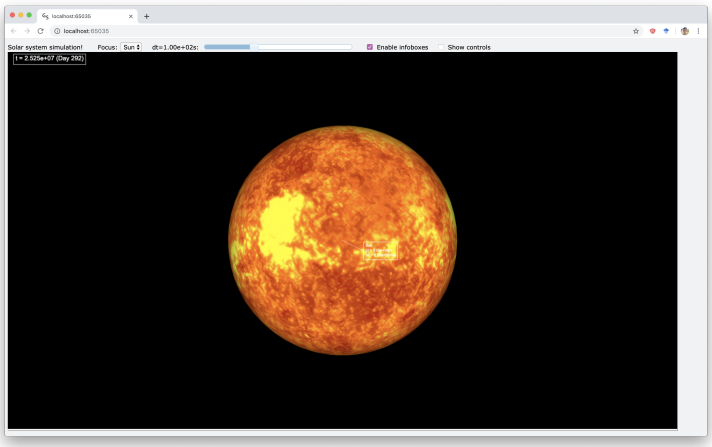
In this activity, you'll be programming the core of an N-body gravitational simulation and configuring it to simulate the solar system. Students in this course have a wide range of programming experience, so I have included questions of increasing difficulty at the end in case some people finish early. Don't worry if you don't get to everything!

Getting set up

First, install `vpython` by opening a terminal (command prompt if you are on Windows) and running `pip install vpython`. Make sure that it is successfully installed by opening a python interpreter (type `python` into the terminal) and running `import vpython`. Ask your TAs for help if it didn't install correctly.

Download the `solar_system_simulator.py` file from the Google classroom page and put it in an easily accessible directory. In your terminal navigate to the directory. You can change your directory with the `cd` command and list the contents of a directory with `ls` (Mac/Linux) or `dir` (Windows). Alternately, in most systems you can drag a folder onto the terminal window and it will automatically switch to that directory.

Once you are in the right directory, run `python solar_system_simulator.py` in the terminal. It should open your browser and you should see something like this:



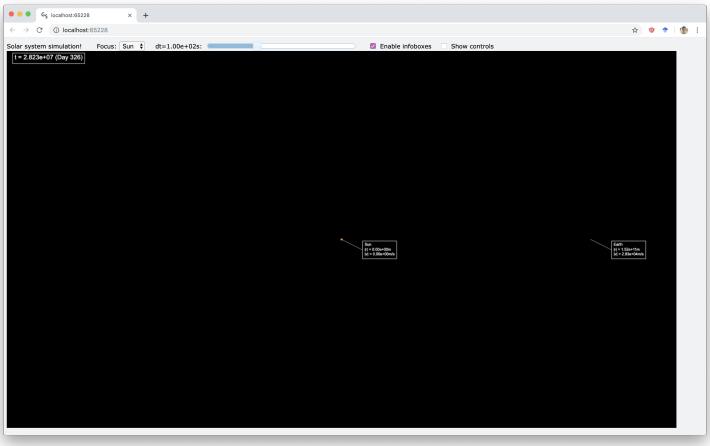
(Note: when I was coding this simulator, I only tested it running on Google Chrome, so if you are having problems, you may need to temporarily set Chrome to your default browser to get it to work correctly.)

Making your first planet

Open `solar_system_simulator.py` in your favorite text editor or IDE. (If you don't have an IDE or a code-friendly text editor, I would recommend Sublime Text: `https://www.sublimetext.com/`.)

Take a few minutes to look over the source code to see how things work. I've tried to comment it thoroughly so that you can see what is going on. Don't worry about understanding the code that renders the visuals (anything starting with `vis.*`), the classes that represent the physical objects and the code that adjusts their properties is what is important. You'll notice that some bits are missing (marked with `# TODO`); you'll need to fill these out as part of this assignment, as I'll discuss below.

Try adding another planet to the solar system. You can find a list of planetary parameters at `https://nssdc.gsfc.nasa.gov/planetary/factsheet/index.html`. Remember to also add your `Planet` instance to the `solar_system` object! When you add your planet, you should see something like this:



You can zoom in and out with scroll, pan around with right click and drag, and switch focus by changing the "Focus" menu.

Coding the laws of physics

You might notice that nothing is happening, which makes for a pretty boring universe. The first thing you'll need to do for this exercise is to implement the `compute_acceleration()` function, which takes two `Body()` instances as arguments and returns a tuple of the $\hat{x}, \hat{y}, \hat{z}$ components of the gravitational acceleration of the second body exerted upon the first.

Once you have implemented this function, you still need to plug it in. Write the code that updates the velocities of each body based on the acceleration to each other body in that timestep. Then write the code that updates the positions of the bodies based on their velocities.

Now run the simulator. If you did everything right, you should see your planet happily orbiting the sun! If something's not working, try checking your `compute_acceleration()` function with some manually computed examples.

Making it realistic

Now go ahead and fill out the rest of your solar system. You will need to specify the initial position, velocity, and mass of each planet or moon. To get the right orbits, you should click on the fact sheet for each planet and use either the aphelion and minimum orbital velocity or the perihelion and maximum orbital velocity. (The main page only lists the average orbital velocity.) Don't worry about getting the relative initial angles between the planets or the $\hat{z}$ displacement from the orbital plane correct, although feel free to do this if you would like after you've finished the main assignment! The radius and color don't matter for getting the physics right, but they do help give you a sense of scale and differentiate planets from each other. For your convenience, I've provided a set of `COLOR_*` constants.

When you're done, run your solar system! Remember you can change focus to zoom into certain planets.

Playing with integration timesteps

In class we learned about Euler's method of numerical integration and how the error is affected by the timestep you choose. In the controls at the top of the screen, you have a slider which can be used to adjust the integration timestep dt from 10^0s to 10^7s on a logarithmic scale. (You can change the bounds on line 177 if you're curious, but you won't find terribly interesting behavior outside 0 to 7.)

Go ahead and play with the slider for a few minutes. Observe what happens when you push dt to increasingly high values.[1] Why do you think this is?

If you push the slider all the way to the max, you'll probably see several planets being ejected from the solar system. How do the velocities of the planets change over time once this has happened? Why do you think this is? (Although the ejection of the planets was obviously due to accumulated numerical errors, there is some understanding to be gained by looking at the velocities of ejected bodies.)

Launching a space telescope

The James Webb Space Telescope (JWST) is a planned successor to the Hubble space telescope scheduled to launch in 2021.[2] The telescope must be kept very cold in order for the infrared sensors to function, so it will be deployed in space at the L_2 Lagrange point of the Earth–Sun system, using the Earth as a gigantic sun shield! (The telescope also features a large shield that will block any remaining light and will keep its instruments around 50K.)

For this part of the exercise, comment out your definitions of all of the planets and comment out where you have included them in the `solar_system` object. (In most editors, you can select multiple lines and use Cmd+/ or Ctrl+/ to comment out blocks of text.) Make a copy of your definition for Earth, and give it a position and velocity that are the average orbital radius/velocity for Earth, which will give it a circular orbit.

The Earth-Sun L_2 Lagrange point (assuming a circular orbit) is situated at an orbital radius of 1.511×10^{11}m. Make a `Ship()` instance for the JWST and place it at the L_2 Lagrange point. What must its initial velocity be for it to stay in the Lagrange point? (Hint: the velocity must give a period which is the same as the period of the Earth's orbit, but must circle a larger radius in the same time.)

Give your JWST the correct initial velocity, then run your simulation. Quickly turn your timestep to a low value and observe the orbits. Your program should show you how the Earth can act as a gigantic sunshield!

The L_2 Lagrange point is an example of an *unstable equilibrium*: small perturbations away from the exact equilibrium point will result in the system evolving away from equilibrium. (Like a rock balancing on the top of a hill; a *stable equilibrium* returns to balance after small perturbations, like a rock at the bottom of a valley.) Turn your value of dt up a little bit. What happens to the orbit?

To solve this instability problem, the JWST will include thrusters that will allow it to make small adjustments to its orbit to allow it to stay at the L_2 Lagrange point.

Extra activities

If you finish early and want a bit of a challenge, try any of these activities (listed approximately by difficulty):

- Add some extra orbital bodies to your solar system! You could explore the 1:2:4 resonance between the Galilean moons of Jupiter, or see the orbit of Halley's comet that we looked at in the previous problem set.
- Make your solar system even more realistic by including the current vertical offsets from the orbital plane and the current relative angles of the planets!
- Create a `Ship()` instance which can apply thrust to accelerate over time! If you want to escape the solar system, have it accelerate in its prograde direction (along the direction of its orbit) over a long time. If you want it to crash into the sun, accelerate along its retrograde direction.
- Improve your integration accuracy! Look up the Runge-Kutta method on Wikipedia. The idea is pretty similar to Euler integration, except that instead of $y_{n+1} = y_n + \Delta t \cdot \frac{dy}{dt}$, you use an average of

[1] If you use values of $dt \approx 10^6$, you'll notice the inner planets following some crazy orbits without being flung out into space. This is actually mainly due to the refresh rate of the visualizer package I used not rendering every timestep and connecting rendered timesteps with straight lines.

[2] Although with two previous delays and a budget that has ballooned from $0.5 billion to $9.6 billion, I wouldn't count on it.

$\frac{dy}{dt}$ evaluated at four different points. This should allow you to crank your timestep up to much higher values without losing accuracy.

- Perform a flyby of Jupiter! This is how the Voyager probes escaped the solar system moving with such fantastic speeds. Create a `Ship()` instance and position it in an elliptical orbit which will intersect Jupiter's orbit and approach Jupiter coming from its retrograde direction.