Outline

* Statement of problem
  + Fnet = things
* Why I used Runge Kutta method
  + Complicated potential, easy to implement in RK4
* Adaptive time step
  + Have to pick the worst time step
  + Adjust to error
  + Have to do x and y error
* Design considerations
  + JSON files for easily adjusting the bodies
  + Sun is stuck in the center
    - A specific object called Sun is not moved
  + Writes directly to a csv file
    - Doesn’t hold the data so it doesn’t slow down

**The Problem**

The interactions of the solar system can be written as

where the second line is easier to use for a numerical simulation. refers to the specific body, and refers to the other bodies in the system, hence it runs from . This last part, , means that is not interacting with itself, and will therefore be implied from now on and not included in any equation. Also the subscript will be implied on all equations.

Analytically this is not possible to solve, hence rewriting the forces to be convenient for a numerical solution. Locking the system to a 2D plane, this is a rather trivial thing to implement in a Runga-Kutta 4th order algorithm. It requires rewriting the second order ODE into two first order ODEs

The iterative steps of a RK4 numerical solution are therefore

And

Where h is the time step.

Therefore the next step would be

However this is a bit of a waste: each step is taken at the same interval, and each body is accumulating an error of . Using an adaptive method can reduce the error (but not eliminate it): but the caveat is that the step size must be found for the *worst* possible condition. The time step for each body must be found, the worst time step picked, and the whole system has to be incremented by that amount. In addition, because the error must be found as . The velocity plays no role in this as it’s just an intermediate step created to facilitate the Runge-Kutta Method needing a first order ODE.

The adaptive step size for a 1 Dimensional system is

Because of , becomes

But it is , each body having it’s own error. Therefore, the worst condition must be found and h must be found from that. Two steps are computed and a step with twice h for all bodies, the smallest is determined, and the next step is computed with the new .

is a chosen value for an acceptable amount of error. In this case it was chosen to be and accumulated error of or . The second value is the one used each step, hence the conversion from to .

Each body is taken to be a point. While each planet will orbit a rather stable path, smaller bodies such as asteroids and comets will cross these orbits. Pluto and Neptune will eventually cross paths if the simulation is run long enough (or the right initial conditions are picked) because the simulation is in 2 dimensions. For this reason, a soft distance of 1000 meter is added to each : this prevents a possible divide by 0 error. The bodies are massive enough where an extra 1000 meters being factored in between them is irrelevant.

**Design choices**

Not totally relevant to the physics of the problem are some design choices of the script.

To start with, the Sun is taken at a locked position. Instead of trying to find how the Sun wobbles around the common center of mass of the planets, it is assumed to be at that center of mass. This isn’t that bad of an assumption: the Sun has 99.5% of the mass of the solar system, and Jupiter contains nearly all the remaining 0.5%. An object named “Sun” is therefore taken to be at coordinates (0,0), and it doesn’t move. However this can be circumvented by renaming it to something like “Sol.” The script hard codes in a search for “Sun” and ignores any change in position or velocity, but it wont do this for an object named something else.

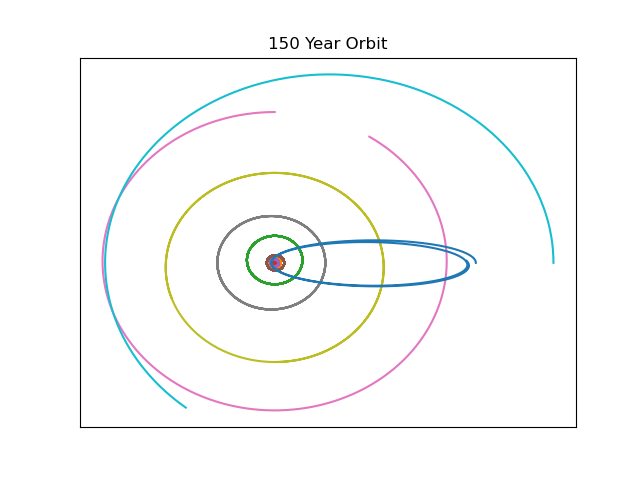
Each body is input into script via a JSON file. This way, instead of hard coding a body and having to recompile the script after adjustments to the orbital bodies, to adjust simply modify the JSON file. Each body has the following attributes: JSON identifier such as "Object012"; A unique name to label the item and used when output to the csv file; a mass in kg; an initial x and y position in meters; an initial x and y velocity in m/s. To make things easier, each planet is placed at aphelion (farthest from the sun and at their slowest velocity) and placed entirely on the x or y axis with everything orbiting counterclockwise. Of course, this means that to compile the script it requires compiling with an additional jsonjson.cpp and json folder, but additional compiling vs time saved for minor adjustments is worth it. Only the json file titled “AstronomicalObjects.json” is used. A second json file is included for copy and paste purposes.

A good source of orbital data is [Planetary Fact Sheet (nasa.gov)](https://nssdc.gsfc.nasa.gov/planetary/factsheet/).

Data is written directly to a csv file instead of being held in memory and being written at a later time. This is because during testing doing more iterations slowed the script down. Instead of holding the data and writing when finished, the data is written into a csv file right after. Writing to a csv has the advantage of dealing with animations and graphs as an afterthought. While it means two scripts are required to make a visual representation of the solar system, using Python after generating a csv file is much easier to make such representations than C++.

Statics plots are made in the Jupyter Notebook via matplotlib. Animations can also be made via Plotly. Plotly has the advantage over Matplotlib in that it doesn’t need an animation function and the animations can be saved to an html. The downside is that the files can be quit large and time consuming to open. A 100 year orbit graph took about 30 minutes to create and was about 150 mb. Opening was nearly impossible: I recommend using Edge for opening large Plotly animations as no one uses Edge for anything, so it shouldn’t slow down the browser.

**Analysis & Notes**



The elongated orbit is Haley’s Comet. Some orbit data I guessed at. Looking at Haley’s comet alone, for 2000 years, gives this plot.

Shape

Description automatically generated

Which is remarkably boring. Adding in Saturn for scale gives this plot

Diagram, venn diagram

Description automatically generated

This plot shows both the accumulated error for 2000 years but also the effects of Saturn on Haley’s Comet over time. Adding in Jupiter to this system (to give some stability to Haley’s Comet) yields

Diagram

Description automatically generated

Remember has been set to so after just 100 years this data will start to have divergence from the actual positions, let alone 2000 years. For planets they may hold out their positions for a bit longer, but for Haley’s Comet, a small object, will noticeably diverge. What’s of interest is the effect Jupiter has on the small body.