Landon Buell

Mark Lyon

MATH 753.01

9 Dec 2020

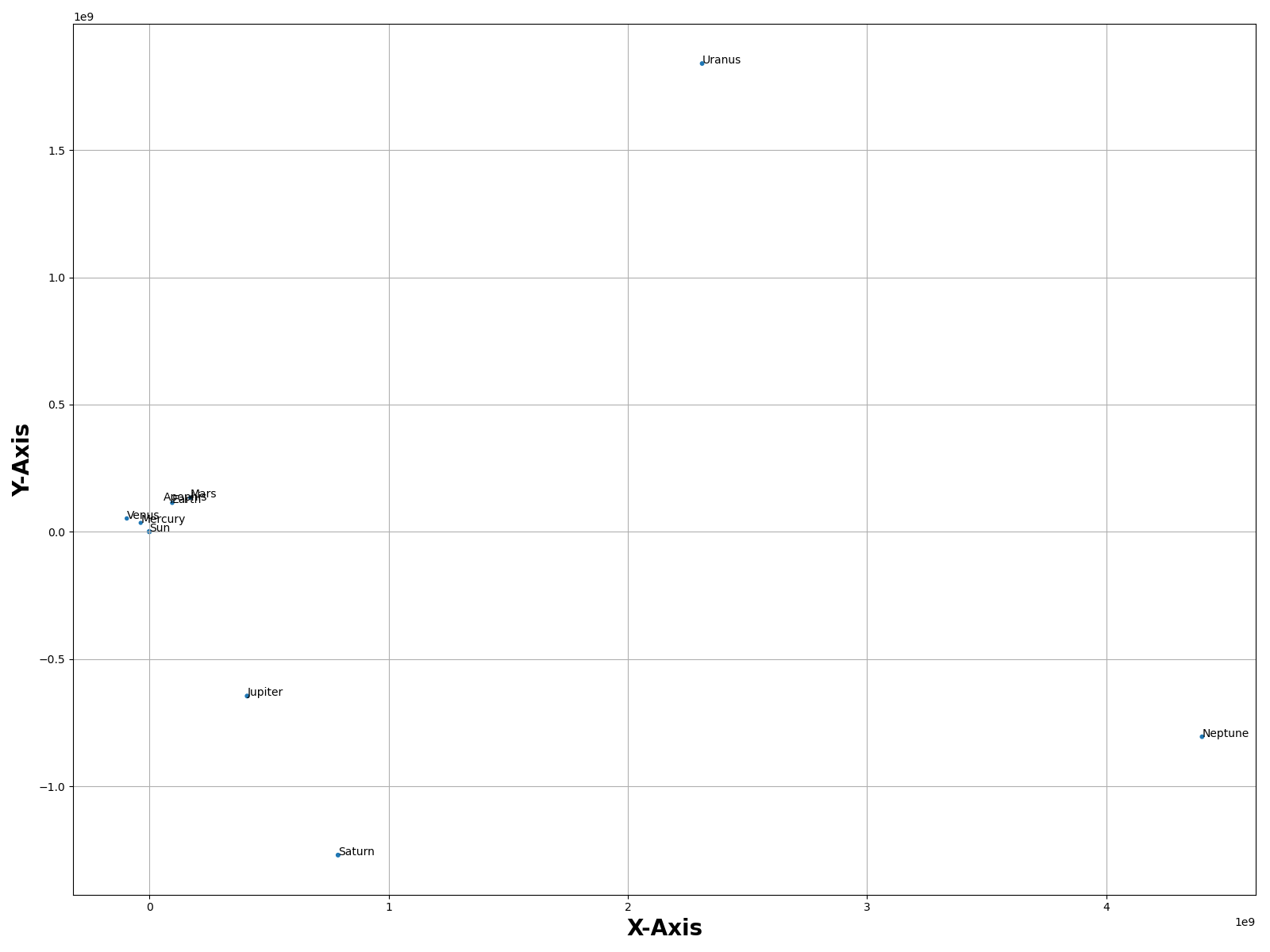
Problem Set #10 – Apophis Model

1. Initialize Sun, Planets & Apophis Asteroid

System was initialized using data from a CSV file. The program reads in the data from the CSV and constructs each the system – masses, positions and velocities, etc. for computation. See “ApophisModel\_2020.11.12” for details

Graphical user interface, application, table, Excel

Description automatically generated

 I load this table into python in “ApophisModelMain.py”, line 22. I use units of kg, km and km/s. Therefore the universal gravitational constant has a magnitude of G = 6.67e-20. A projection of the system onto the XY-plane allows us to see the initial conditions of the system in 2D. The Initial System looks like:

Chart, scatter chart

Description automatically generatedThe Inner terrestrial planets are hard to see, so we can zoom in on them.

1. Physics Function that incorporates gravity.

The “callable” function to solve is found in “OrbitalSystem.py”, line 80. This function takes a time *t* and a 1D state-vector *y* and computes the “left” side of the differential equation.

We use a for-lop (line 87) to loop through each body. This computes the velocity and acceleration for each body in the system (including Apophis & the Sun) using Newton’s Law of gravitation. The inner for-loop iterates through every other planet (excluding itself, we skip if I = j) and adds the acceleration contribution due to gravity. With each inner iteration, we add to the acceleration 3-vector.

The “self” parameter is because the method is attached to the “OrbitalSystem” class. It allows us to access a list of all the masses in the system, and access tat value of G, rather that constantly assigning a new variable every time. The function iterates through all bodies in the systems, planets, the sun and Apophis alike, so all objects are subject tot eh same rules of the gravity engine.



1. Implementation of Dormand Prince Algorithm

The Dormand Prince Algorithm has been (unsuccessfully) implemented in “Solver.py”

The Solver requires a callable function (an ODE) and a time-range to solve within.

We call the Solver in line 68 where we give a set of initial conditions, and a tolerance parameter. The solver then sets up the set size given the time range, and prepares to run the system. (We prepare some arrays to track the state-vector at every time step)

In the main while loop, we use the DP-45 algorithm as outline in Eq. 6.65 of the 3rd Edition book. We return the “Z” vector, the “S” vector and the error as computed from Eq. 6.66. These values are computed in a separate “NextStep” (line 40) function for readability.

As per the DP algorithm, we test the error in the step with Eq. 6.66. If it is acceptable we compute the value of the next state vector and time vector and store them in the arrays, and increment the step counter. If the error is unacceptable, the step size either grows or shrinks accordingly. (multiple or divide by 2) This is the adaptive step rate from the algorithm. I believe this is where my mistake lies?

I have not gotten this to work properly. When run in it’s current state, the stepsize of h either gets too small (like 10e-12) or gets too large (a few weeks). Thus it may “finish” after 12 steps or so, or take hundreds of thousands. In either case, plotting the outcome shows a very poor representation of the solar system (Sir Newton would not be happy with me)



I was unable to implement DP-45 fully, I think my error likes somewhere in lines 40 – 108 of “Solver.py”. Otherwise, I had fun setting the problem up with a sort of “object-oriented style”. I wanted to make the program work for a general input (any solar system) but I was unable to do so.