Simulating the N-Body Problem in Python

Christopher Heidelberg
Department of Physics and Astronomy, University of Southern California, Los Angeles, California 90089, USA
(Dated: December 4, 2021)

I developed a versatile tool to visualize and analyze any arbitrary N-Body system. The tool is developed such that a user inputs parameters choosing the number of bodies as well as each bodies initial conditions, then selects the time and resolution for the simulation. After the simulation is run, users have access to several post-processing options as well as the ability to save their simulation run eliminating the need to redo work when the program is closed. In this paper I show that the tool is not only capable of simulating any arbitrary N-Body system, but also accurate in simulating real physical systems as observed. Finally, I go over the weaknesses of my approach, showing that the runtime complexity of my methods are suboptimal compared to more sophisticated approaches used by larger-scale N-Body simulators.

Keywords: N-Body Problem, Simulation, Celestial Objects

I. INTRODUCTION

The N-Body problem is a way to predict the location and velocities of multiple bodies over time given an initial mass and velocity. Since each body will be attracted to the others due to Newton's the Third Law of Motion, over time the bodies tend to move in chaotic periodic dances around one another such as in figure 4. In some cases, the movement of these bodies can become stable and we get formations such as the solar system. Figure 2 shows an example of this. Historically speaking the N-body problem has been used to model the movement of celestial bodies. For instance, take Fujii et al.[1] where the research team used a novel implementation of the N-Body simulation to model the evolution of star clusters near the center of the Milky Way galaxy. However, simpler implementations of the N-Body problem could be used to model systems such as : the two body earth and moon, our 8 planet solar system, or even bodies as large as the milky-way galaxy and so on.

This paper aims to detail the steps used to create an N-Body simulator with an emphasis on ease of usability to general audiences as well as creating tools to better help the user model and analyze their results. To ensure optimal usability for both experienced users and novice users alike I break apart and focus on three main aspects of this project: 1). Easy to configure starting parameters and data visualization settings 2). Accurate and predictable simulation of the N-Body problem for any given (logical) inputs 3). Straightforward post processing tools to help showcase their results and draw conclusions.

In order to best insure accuracy I will show that this N-Body simulation program can be used to model the position of known celestial bodies by simulating the bodies in the solar system and comparing to their observed movements in reality.

II. UNDERSTANDING THE N BODY PROBLEM

Predicting the movement of celestial objects is important for understanding both the future and the past of our universe. To better understand the N-Body problem, lets imaging a universe with only two planets. This is the smallest non-trivial example for the N-Body problem. We can intuitively imagine that, because there are only two objects in this universe, the position of one body will have some type of equal and opposite reaction to the force acting on the other body at any given time t.

A. Deriving the N-Body formula

For simplicity we can think of these two planets as point masses with some (x, y, z) location p_0 in space and some starting (x, y, z) velocity vector v_0. Thanks to Newton's laws we know the force acting on each star can be expressed as:

$$F = \frac{Gm_1m_2}{r^2}\hat{r} \tag{1}$$

Where m_1 is the mass of our first object, m_2 is the mass of our second object, G is the gravitational constant and r^2 is the squared distance between the two objects.

We can use $F = ma$ where $\ddot{r} = a$ and $\hat{r} = \frac{1}{|r|}$ to solve for a differential equation representing the motion of the two bodies.

$$m_1\frac{d^2r_1}{dt^2} = \frac{Gm_1m_2}{r^3} \tag{2}$$

In this example it can be proven that there is a closed-

form solution for this 2-Body problem[1]. However, the problem arises that if even one more body is added into the system, there is no general closed-form solution to the problem[2]. This means that no simple formula exists where we can plug any arbitrary time value t into a function with the initial body positions $f(t, ...)$ and get our solution to the problem. In order to calculate the solution for an arbitrary N-Body problem, we have to manually solve for every state of the system from $t = 0$ to $t = t_f$ jumping only by some small time step Δt. Thankfully, with the power of modern day computers, this type of calculation doable by almost anyone.

To account for arbitrary N-Bodies we can generalize our differential equation above 2 to:

$$m_i \frac{d^2 r_i}{dt^2} = G \sum_{i=1}^{N} \sum_{j=1;j\neq i}^{N} \frac{m_i m_j}{|r_i - r_j|^3} \tag{3}$$

Note that I have split up the r^3 to explicitly be the magnitude between any two arbitrary points now that we have more than two total bodies. Using this equation it is possible to develop a program to solve the N-Body problem through brute force solving. For the simulation, I follow the steps lined out in the book[3]

B. Calculating Center of Mass

Calculating the center of mass for the system has some benefits when it comes to displaying the results. For example, take the case of figure 1, a system where we simulate three points and one diverges from the other two.

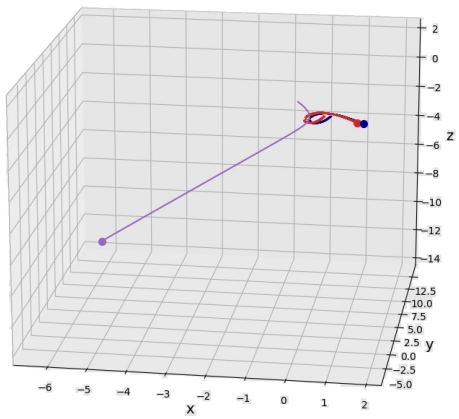

FIG. 1. In this system the center of mass would move along 3D space with a linear motion as the third (purple) point got further further away from the first two.

[1] I wont go into the details of deriving it here but you will find it is rather simple if you sit down and try it yourself

The center of mass of a system with two bodies will have no movement, the center of mass of a system with three bodies will have a center of mass that is either zero, or linear. And the center of mass of a system with N-Bodies can have a center of mass that fluctuates in the x,y or z directions.

$$r_c = \frac{1}{M} \sum_{1=i}^{N} m_i r_i \tag{4}$$

We can also calculate the velocity of our center of mass to determine how much it is moving:

$$v_c = \frac{1}{M} \sum_{1=i}^{N} v_i r_i \tag{5}$$

III. IMPLEMENTING THE SIMULATION

A. Configuration

As stated in the introducing, one of my main focuses points for this project was user accessibility. I wanted to make a program that could be used by not just individuals experienced with the problem, but also users who have never heard of the N-Body problem. One of the easiest ways to increase usability is by not having hard-coded values. To avoid hard-coded values the simulation is designed to read in two configuration files. The first one, bodies.cfg, is designed to take an any arbitrary number of bodies the user wants to simulate as well as their initial starting conditions. Below is an example of the first few lines of this file from one run of the simulation. Bodies.cfg allows the user of the simulation to fine-tune their initial conditions without needing to go into the code and manually create an object in the code for each body then append it to a list of bodies. Each body is specified with a name (placed at the top between the square braces), and 4 associated metadata fields below it. The user is free to add as many bodies as they choose.

```
[body1]
locations = [10,0,0]
velocities = [-3,10,0]
mass = 0.5
color=red

[second-body]
locations = [-4,2,0]
velocities = [0,-1,2.5]
...
```

In the bodies.cfg example above you can see two bodies defined: body1 and secondbody. Both of these bodies

have 4 associated metadata fields. The first three determine starting conditions for the simulation: location an (x, y, z) array, velocities an (x, y, z) array, and mass a float representing kg mass. The final parameter determines what color the body should be drawn with when plotted to the figure.

The other configuration file, config.cfg, allows the user to fine-tune the parameters used to reenter the simulation's animation output without needing to look through or modify the code. Below is an example of the config.cfg. In this configuration file, the user should not change any of the profile names or variable name (the sections defined by the square braces). Instead the user can alter the values of each variable to modify the simulation. See the example config.cfg below

```
[figure]
window_size = 5
background_color = (0.9,0.9,0.9)
draw_axis = on
draw_legend = off
title = ""

[simulation]
resolution = 5600
time = 25
center_camera_on_body = none
```

From the example above notice there are two categories for config.cfg: figure, file, and simulation. 1) The figure profile affects metadata defining how the figure is drawn. The window size defines how large (in inches) do draw the figure. Background color affects the background color of the figure window. Draw axis (on/off) determines whether or not the figure axis is drawn. Draw legend affects whether or not a legend is added to the top left of the figure. Finally, title determines the title placed at the top of the figure. 2) The simulation profile affects variables modifying how the simulation is run. Resolution affects how many steps the simulation will run between zero and the total time (δt). The time variable affects how long the simulation should run for. The center camera on body variable lets the user put in the name of the body the camera should be centered on. This shifts the reference frame so that body is always at the origin and all other bodies are shown relative to that reference frame. Leaving it as none makes the reference frame not center on any bodies.

With these two config files the user is easily able to enter initial conditions for their systems. Additionally

B. User Menu

To help new users access the program, by default when the simulation is called from the main method, a command line interface (CLI) prompt will appear for the user asking if they would like to use an interactive menu. This menu walks the user through the process of setting up their own N-Body simulation. Additionally, several presets are available for the user to select from so they can get a feel for using the program without needing to define their own N-Body configuration files. Once the user has input the path to their configuration files (or selected one of the presets) the program will start the simulation. After every simulation is run the user is able to choose from a selection of options: 1. The user can open an interactable figure to view the line plot of their run 2. The user can save the simulation results to a file for faster loading next time 3. The user can select from a few post-processing options. I will discuss the details of post processing in a future section.

More information on the menu can be found in Appendix A

C. Argument Passing

Despite my focus on user friendliness, I want the simulator to convenient to use for more experienced users too. Because of this, the program also supports command line arguments. This means that the program can be run from the shell. All options and settings can be passed in from command line flags. This enables the N-Body simulation to be called by other programs allowing advanced users to utilize this tool in their own project, without needing to use a clunky cli menu. This includes the ability to automatically call the N-Body simulator with other programs.

Further details about the command line arguments this program accepts can be found in the appendix B.

D. Visualizing Results

Data visualization is a system that has been written, deleted and rewritten several times while creating this project. Initially, I planned on creating and saving a gif animation visualizing the movement of each body over time. However, after several large code changes and direction switches I ultimately abandoned the idea of creating an animated output. A lot of the foundation code still exists and I hope to add animated outputs once the simulation's code is a little more stable.

Currently, the user has one choice when it comes to rendering results. When the simulation is done iterating the user can open the results as an interactable matplotlib figure and view the plotted trails of each body in 3D space. All of the figures in this paper were generated by taking a screenshot of the matplotlib. Because it is an interactable window, the user is able to rotate, pan and zoom the camera where they please.

Since data visualization is a critical step in explaining results I included several properties the user can set in the config.cfg file enabling them to edit how the figure is drawn. A list of configurable figure settings can

be found in the config example monotext of the configuration section. Additionally a description of how the user can modify their figure with the menu is located in Appendix A.

E. Post processing

Due to the long wait times between rendering. Enabling the post processing on the data, without needing to rerun the simulation was a must. Since the time complexity of post processing the data is $O(r * b)$ where r is the resolution and b is the number of bodies, it is a level of magnitude faster to edit the data after the simulation has been run. There are three main post processing tools available to the user: 1. Change the reference frame 2. Trim the run by setting the start and end times to be drawn 3. Modify the figure settings (these settings are already discussed in Section III A.

While the post processing tools were very helpful for generating each of the figures in this paper, I wont go into detail about them here. More information on how post processing fits in with the program can be found in the Appendix A.

F. N-Body Simulation

After the major components were put together, it was time to test the simulation. The first test I wanted to try was simple. Add 9 bodies separated by relatively small distances with random initial velocities. This first test proved not only was the simulator working, but that it would also handle a relatively large number of bodies (compared to the 2 and 3-body problem). A plot of this simple test can be found in the Appendix under Figure 4.

Simulating the movement of 9 bodies with random initial starting conditions is a good start. However, the simulation is not helpful unless it can be used to describe physical systems that exist in our universe. To test the authenticity of my simulator, I chose to try to model the solar system. Because the simulation works best with smaller numbers (I will go into detail about the limitations in a later section) I decided to normalize the distance unit to be relatively small. The unit I decided to use for distance was the Astronomical unit. This put Earth's starting position at 1.0 AU away from the sun. Each of the other planets were placed at their respective average orbital distance from the Sun. For mass I chose to use kg, however, I opted to normalize the mass by the Earth's weight. This means that the mass of every other planet was divided by the mass of the Earth and is about 10^{24} times smaller than it would be un-normalized. Using lower mass values speeds up the computation's time because larger numbers take more ALU cycles to compute (this is especially apparent for multiplication and division). Additionally, due to the way floating point numbers are represented in binary, smaller numbers will yield a much more precise result. This means less error as the simulation runs for longer periods of time. Finally, I chose to use kilometers per second of the planet's initial velocity. For the sake of this proof-of-concept I opted to start all the planets on the x-axis and set initial velocity in the $\hat{y}$ direction. Although it might seem like placing all the planets on the x-axis could have a larger impact on the result of the simulation, I calculated the effect is rather minimal. The change to the system's center of mass by having every planet lined up on the x-axis would only shift the center of mass by roughly 5.2×10^{-6} AU (or roughly 1,000 Kilometers). Because the shift the the center of mass is so small, the initial setup of all bodies on the x-axis will not have very large consequences on the simulation. To add another layer of complexity, I also wanted to simulate the moon orbiting around the earth. I set its initial starting position as the Earth's distance from the Sun plus the distance from Earth to Moon. For its initial velocity it got the orbital velocity of the earth plus the orbital velocity of the Moon's orbit around Earth. Once the initial conditions were setup I ran the simulation.

I chose to simulate 365 days (one orbital period of Earth). The results can be seen in 2. Additionally, a zoomed in picture showing the Earth-Moon orbit around the sun can be found in the Appendix (Figure 5). One way to show this is correct is by analysing how far the other planets made it on their orbits in the time it took Earth to complete one revolution. In the simulation, Mars had completed 55% of its orbit when Earth completed one revolution. In reality Mars has an Orbital period of 687 days. Dividing Earth's 365 day orbital period by the orbital period of Mars we get 0.53. In other words, Mars should be 53% done with its orbit by the time Earth finishes one revolution. Overall I am pretty happy with this result. We are off by a factor of 2%, however, this error can likely be attributed to rounding issues or the starting positions not reflecting their actual location for our Solar System down as I started all the bodies on the x-axis.

G. Runtime Analysis

A major factor of simulators is their computational runtime. There are many different ways runtime can be analyzed but one of the most common algorithm classification strategies is Big-O notation[2]. This is a technique used to describe the worst case behaviour of an algorithm as you put n inputs into the program, how fast does the time complexity of that algorithm grow.

[2] Since the program will never shortcut like you might have in a search or sort algorithm, the runtime here will be Big-O $\approx$ Big-Ω $\approx$ Big-Θ

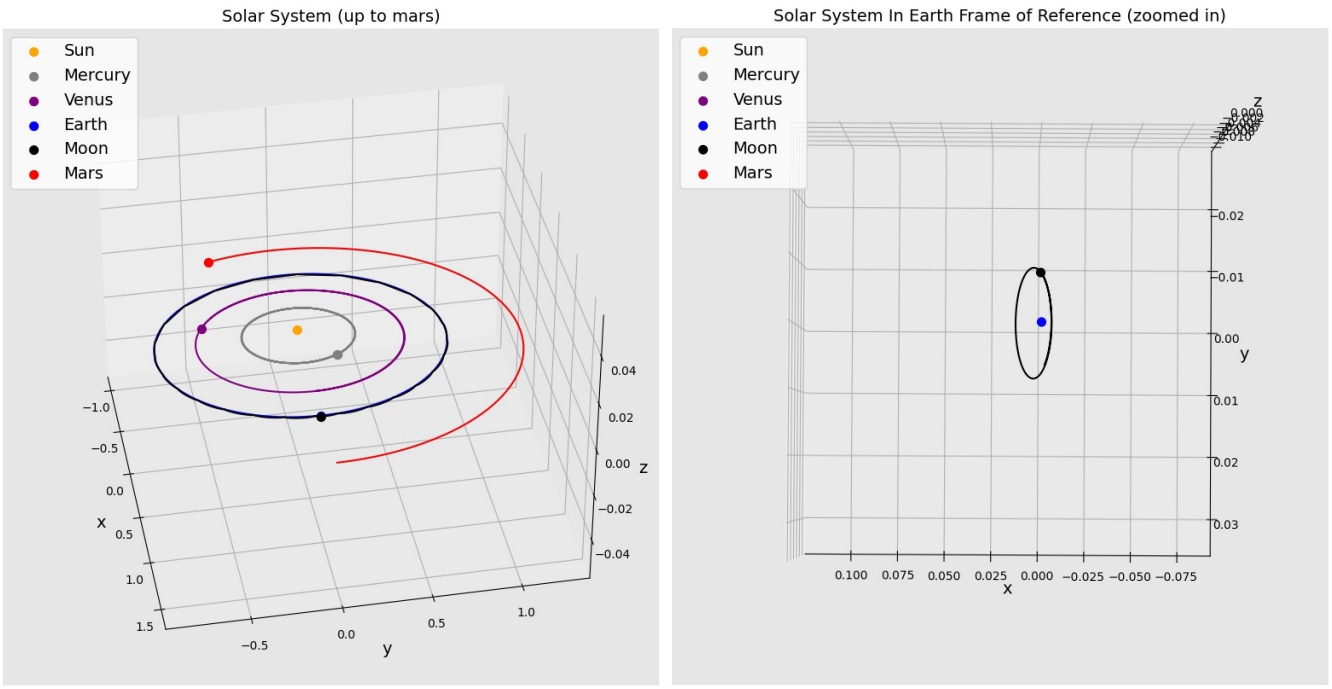

FIG. 2. Solar System with planets up until mars. **(left)** Shows the 4 plotted planets orbiting the sun for 365 days. The moon is also in orbit around the earth. **(Right)** Changing the reference frame so that Earth is in the center and zooming in we can see the moon make one orbit around the Earth. Plotted over 27 days. *Note:* The axis values are skewed here which at first glance might seem like the moon has a far more eccentric orbit than it should.

First lets define some variables, let $t = total\_time$, $r = resolution$, $b = number\_of\_bodies$

My simulator has a relatively 'bad' runtime complexity. for each run of the simulation the computer will solve for the position of each body between 0 and t for r number of times. That is, the simulation creates r evenly spaced timesteps between 0 and t. Next, for each of these timesteps, the computer must calculate the force acting on each body from every other body (as can be seen in equation 3). This leaves us with a Big-O complexity of $O(r * b(b-1))$ or roughly $O(r * b^2)$. However, there is also a large amount of constant and linear operations that are run with each step of the simulation too. Because of this, the simulation starts to take longer to complete for higher resolutions and becoems much more complex as the number of bodies grows. Due to the runtime complexity, it would be unpractical to use my simulator to simulate large-scale systems with thousands of bodies (such as the formation of a galaxy). In order to do something like this, a more efficient algorithm would need to be implemented such as the Barnes-Hut model[4] which solves a highly accurate approximation of the N-Body problem in $O(NlogN)$ time. Much faster than my $\approx O(N^2)$ solution.

IV. DISCUSSION

A. Strengths

Ultimately I am happy with how this program turned out. My goal when I set out was to create a user-friendly easily accessible tool to simulate the N-Body problem. I am confident in saying that with the tool I have created my program could be used both by a user looking to explore the N-Body problem by downloading and running the computer on his/her machine or something as advanced as the backend of a website which provides a user friendly GUI accessibly to anyone through the internet. As proven in the in Section III the simulator is accurate to real world observations. The configuration file and CLI support enable quick and easy editing of parameters without needing to dive into the code of the simulator. Finally, the user menu offers a robust and easy tool for users to access the simulator in a user friendly manor.

B. Weaknesses

Currently the simulator suffers from some instability when dealing with large numbers. This issue seems to be caused by my ODE solver library, however I have not been able to isolate exactly what is causing this issue to occur. Because of this issue, larger time values (or some-

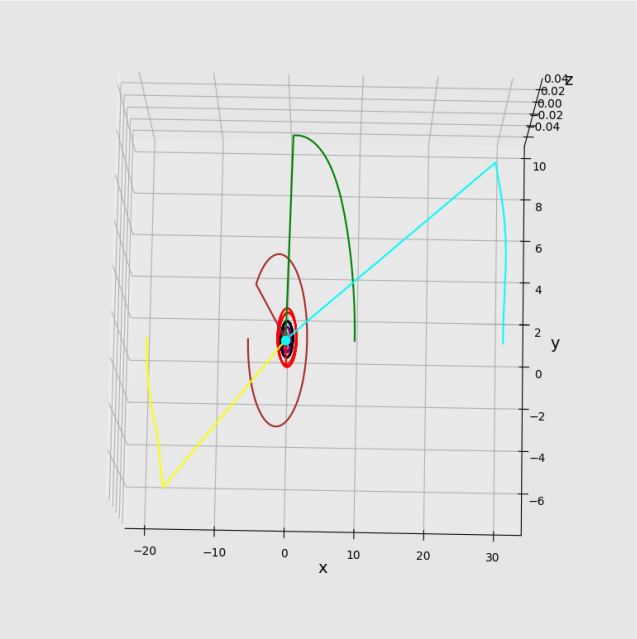

FIG. 3. An early attempt to plot all 8 planets in the solar system lead to many crashes. Each of the planets plotted moving along their orbits until the ODE failed and each planet's position jumped to zero.

times systems with several large massed body values that getting close to one another), cause all the bodies to suddenly jump to zero (See Figure 3. The main downside of this is that it can cause simulation runs that take a long time to perform, sometimes upwards of an hour, to suddenly crash, losing the user a lot of time and causing difficulty when it comes to configuring the time and resolution that will allow those systems to be rendered. The issue is difficult to track down since it tends to affect runs that take more than just a few seconds to run, so each test to track down the cause of it takes up several minutes.

Another major issue with the current system is due to an oversight in development, the save/load feature uses the default (root) configuration files to load all the metadata. That means that if the saved/loaded run is not run from the configs in the root of the project, such as one of the preset configs, the program will likely crash when the previously saved run is loaded in.

C. Future Work

As mentioned in the weaknesses section, this program is far from perfect. On top of the major bug fixes, there are future improvements I would like to make to the program too. Primarily, I want to add better ways to visualize the results. First, the user should be able to disable the automatic axis values when the figure is drawn. users will be able to enter an x and y value range for the axis. This would prevent confusing plots like the one on the right side of figure 2 from being drawn where the x and y axis are not symmetrical. Second, as mentioned in the visualization section, I would like to add gif support so the movement of each body can be shown over time. The users could chose to save their gif after it renders.

The second set of features I want to add are more data-analysis tools. The program already saves the forces acting on each body for every time-step. A helpful tool for analysis would be one that shows the forces acting on each body by drawing a force vector on the body as it moves through space. Additionally, centering the reference on the center of mass can be helpful in 2 and 3 body simulations to keep all the objects in view, however, in larger body simulations the center of mass can start moving chaotically creating a confusing perspective to try to interpret. It would be more useful to also have the option to plot the center of mass over time, similar to the way each body is currently plotted. The center of mass would be given a dotted line to distinguish it from the rest of the bodies.

Finally, the last feature I would like to add before declaring a version 1.0 release is a config editor. Despite the config files being much more user friendly than having users hard-code values into the program code. Creating a CLI menu that walks users through the process of creating their own config file (or modifying an existing one) would be much easier and less error prone for both new and experienced users.

V. CONCLUSIONS

In this paper I sought out to create an accurate and user friendly simulator for the N-Body problem. Overall I think the project succeeded in every major goal I set out from the beginning. The simulator is both easy to use and relatively quick to setup. The simulation is accurate to real-world observations. The data visualization is flexible and allows users a range of options when deciding how they would like to present their data. Finally, the simulator is capable of taking command line arguments allowing it to be used as a tool for developers creating larger automated scripts.

CODE AVAILABILITY

Code is available at https://github.com/Cmheidelberg/N-Body-Problem.

Disclaimer: The code is still in a fairly unstable state. All of the core functionality works but it is untested on other systems and some features are not as streamlined as I would like them to be. If you are interested in cloning and running the code yourself feel free to reach out to me (*cheidelb@usc.edu*) with any questions or if you run into difficulties getting it working.

Appendix A: User Menu

The menu is a user friendly and helpful tool to walk the user through the steps needed to generate a figure. There are three main steps to the menu: 1). The user selects a config.cfg and bodies.cfg file to run the simulation on 2). The simulation is run on the selected configs 3). The user selects from some post processing options to enhance their figure. Every figure shown in this paper has been created entirely using the user menu. Below I will discuss what options the user is given when using the menu. It should also be noted that everything done in the menu can also be done by passing pre-made config files to the program using the command line arguments. Thus, while this menu is a helpful tool, users are not required to use it.

1). Select config The user has 4 options to select which config they would like to run the simulation on. They can choose to: select from a set of preset configs (intended as examples), enter a path to their config.cfg/bodies.cfg file, use the default config (same as entering a path but selects from a cfg located in the root of the project which is included when cloning from GitHub) or the user can load a past run which skips the need to rerun the simulation from scratch. Overall I wanted the system to be as easy to use and convenient to new and experienced users alike.

2). Run Simulation If the user did not choose to load a past run then after loading the configuration files the N-Body simulator will now run the simulation. This can take anywhere from several seconds to hours depending on the number of bodies in the bodies.cfg file and the time to run.

3). Post processing After the simulation is run the user gets to select from several options. These include: Plotting the figure, changing the center of reference, trimming the start/end time, saving the run (for loading next time), changing the figure settings (axis on/off, legend, figure title, background color, etc), and finally the user can choose to rerun the simulation with a new time length or resolution.

I put a lot of detail into the user menu which I cant show in this paper. Overall I am proud with how it turned out. In the future I hope to add more sections to the interactive menu allowing the user to create and edit the bodies.cfg and config.cfg files. This would further streamline the process of setting up starting conditions for user made runs.

Appendix B: Command Line Arguments

When I added command line arguments I wanted to enable my simulator to be fully automated and used by other programs. This could be for something liek Python's os or Java's ProcessBuilder libraries or even just a shell script written by the user. Because of this, all the major core functionality from the menu system can be set with arguments. There are 8 main arguments the program accepts:

-h/–help: Print the help menu giving details about each flag

-c/–center_on: Input the name of the body for which the reference frame should be centered on. This will keep the input body at (0,0) and offset the positions of other bodies accordingly. The user can also enter 'c.o.m' to offset all other bodies by the center of mass of the system.

-ts/–display_timesteps: Specify a percentage range of the figure to display. For example, to graph only the first half of the simulation input [0,50].

-p/–config_path: Specify a path to a config file to use for the run. If this is not specified the simulator will look for a config.cfg file in the project root.

-b/–body_config_path: Specify a path to a body configuration file to use for the run. If this is not specified the simulator will look for a body-config.cfg file in the project root.

-np/–no_prompt: Do not ask the user if they would like the interactive menu. This is to prevent the program from hanging (and waiting on input) when called from another script.

-s/–save_output: Save the simulation output to a file with the input name. (works if absolute path is given too) This file is not human readable and can only be used when loaded in from the load flag. This argument cannot be active with the load flag given too.

-l/–load_equation: Load a previously saved run. By default these files have the extension .nbp.

Appendix C: Additional Figures

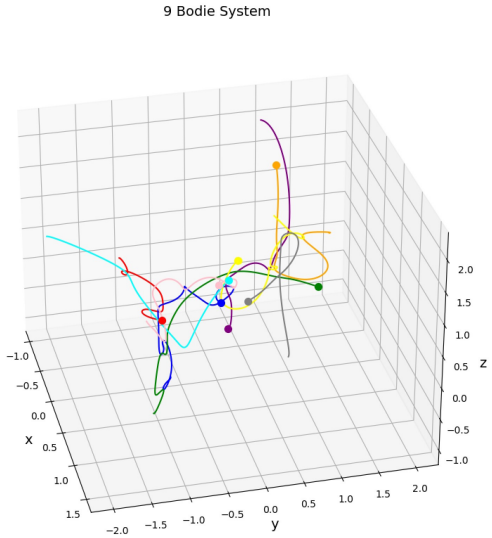

FIG. 4. 9 Body test run. each body has a similar mass and random starting velocities that were all within an order of magnitude of eachother

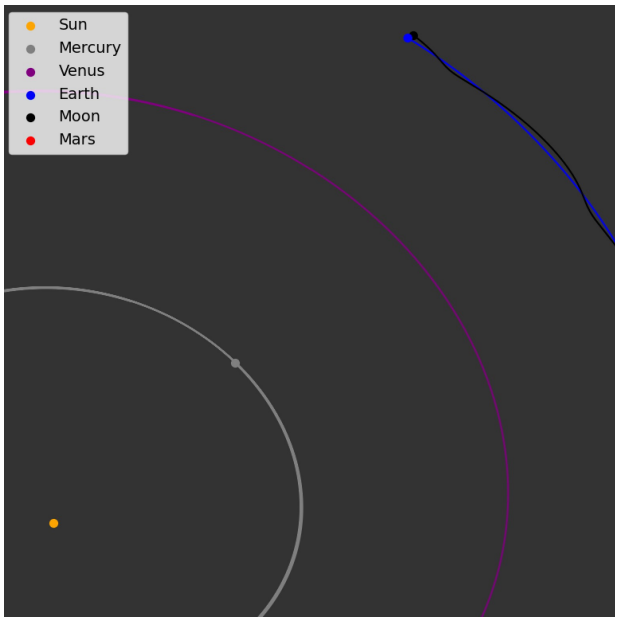

FIG. 5. Zoomed in picture of the simulator's Earth-Moon orbit around the Sun.

[1] M. Fujii, M. Iwasawa, Y. Funato, and J. Makino, Evolution of star clusters near the galactic center: Fully self-ConsistentN-body simulations, The Astrophysical Journal **686**, 1082 (2008).
[2] Z. E. Musielak and B. Quarles, The three-body problem, Reports on Progress in Physics **77**, 065901 (2014).
[3] S. J. Aarseth, *Gravitational N-Body Simulations* (Cambridge University Press, 2003).
[4] J. Barnes and P. Hut, A hierarchical o(nlogn) force-calculation algorithm, Nature **324**, 446 (1986).