Solving the N-Body Problem using Finite Difference Methods

Rohen Agarwal, Sam Ahnert, Eric Wan, Christopher Young

April 1, 2021

*This paper was created using LaTeX.

1 Introduction

The N-body problem is an initial value problem, whose solution is dictated by a governing differential equation and the initial conditions. The governing differential equation cannot be solved by simple algebra, so another method needs to be used to approximate the solution. In this project, we will explore the solution to an N-body problem using finite difference methods.

The governing differential equation for the N-body problem is given by

$$m_i \frac{d^2 \boldsymbol{r}_i}{dt^2} = \sum_{\substack{j=1 \\ j \neq i}}^{N} \frac{G m_i m_j (\boldsymbol{r}_j - \boldsymbol{r}_i)}{||\boldsymbol{r}_j - \boldsymbol{r}_i||^3} \qquad i, j = 1, 2, ..., N \tag{1}$$

where m_i is the i^{th} mass, r_j, r_i are the positions of the j^{th} and i^{th} bodies respectively such that $i \neq j$, and G is the gravitational constant.

Recognizing that velocity is the first derivative of position as acceleration is the second derivative of position, this 2nd order differential equation can linearized, yielding

$$\frac{d}{dt}\left[\boldsymbol{v}_i\right] = \begin{bmatrix} \dot{\boldsymbol{r}}_i \\ \ddot{\boldsymbol{r}}_i \end{bmatrix} = \begin{bmatrix} \boldsymbol{v}_i \\ \sum\limits_{j \neq i} \frac{G m_j (\boldsymbol{r}_j - \boldsymbol{r}_i)}{||\boldsymbol{r}_j - \boldsymbol{r}_i||^3} \end{bmatrix} \tag{2}$$

The right hand side of the linearized matrix expression of the governing differential equation resembles $\frac{du}{dt} = f(u, t)$, and thus can be used in implementing the differential solver in code, as will be described in

We will be simulating two different systems:

System 1 will be the major bodies in the solar system (the sun, the eight planets and Pluto) in stable orbits as determined by the real life dynamics of our solar system. This system will be our means of verifying that our implementation works for a general system, as we have real life data to compare to. After verification, we will explore the results using three different methods: the 2nd order Runge-Kutta method (Heun's Method/RK2), the 4th order Runge-Kutta method (RK4), and the 7th order Runge-Kutta method (RK7). Comparing convergence error, run time, stability, energy, and expected results from NASA HORIZONS data, we will choose a final method and respective Δt for an accurate yet efficient simulation. We can then extrapolate to system 2 with relative confidence in the accuracy of the results.

System 2 will again be the major bodies in the solar system (the eight planets and Pluto) but with a black hole in place of the sun. This will be a significantly more chaotic system, and we will investigate the dynamics of the system with a black hole of mass $5 \times mass_{sun}$.

2 Numerical method

2.1 Description and derivation of method

One of the methods we have chosen to investigate for our problem is the RK4 (Runge-Kutta 4^{th} method. It is a method we can use to solve ordinary differential equations that are either a Boundary Value Problem (BVP)

or an Initial Value Problem (IVP). To understand how RK4 works we must first understand the difference between and IVP and a BVP. While a BVP is a problem in which we know the *static equilibrium* conditions after a long time t, IVP defines the *dynamic time evolution* of the system since initialization (Goza, 2021). In our case, the governing equations along with the initial conditions of the "N-bodies" resemble an IVP problem. An example of an IVP is

$$\dot{\boldsymbol{u}} = \mathbf{f}(\mathbf{u}, t)$$
$$\mathbf{u}(t_0) = \mathbf{u}_0$$

In order to approximate the motion of our N-bodies we used 2 types of method, a single step, and a multi step (predictor-corrector) method. The idea is to use the single step method to predict to find each body's velocity and acceleration vectors at each time step which can be integrated to find the position of each body at each time step. Some possible options for numerical methods are Forward Euler as a first order method, Heun's as a second order method and RK4 as a fourth order method. The equation for the Forward Euler method is:

$$y_{n+1} = y_n + h * f(u_n, t_n) \tag{3}$$

where $f(t_n, y, n)$ represents the velocity (first derivative of position) and the acceleration (second derivation of position). However, the error associated which this is very large. The method for calculating error is finding the norm between our approximated function and the true function at any time t. One workaround is reducing the size of the time step. While this reduces the error, the truncation error however is magnified. To recall, truncation error is the error approximating any function at a certain point. To balance between accuracy and computational efficiency, one might truncate the approximate solution. This truncated value has an associated error which is the difference between the actual value and the truncated value. By virtue of numerical methods being ones that approximate the next value based on the previous value, the truncation error propagates as well and increases over time. Therefore, smaller time steps mean more number of total steps, which cause the total truncation error to be much higher.

The next option is to use Heun's method which is described as:

$$y_{n+1} = y_n + \frac{h}{2} * [f(u_n, t_n) + f(u_{n+1}, t_{n+1})] \tag{4}$$

we see that this is an implicit method because of the $f(u_{n+1}, t_{n+1})$ term which we can be solved for using a first order single step method such as Forward Euler. This method proves to have much better accuracy and lesser truncation error then Forward Euler, however, we can do better. The fourth-order Runge Kutta (RK4) method proves to be the most efficient numerical method. The truncation error of the Forward Euler method scales as $O(h)$, that of Heun's method scales as $O(h^2)$ while that of RK4 scales as $O(h^4)$ where h denotes a time step. Therefore, it is evident that the truncation error for RK4 is the least and so in a process where accuracy is critical, RK4 is an optimal choice of a numerical method. The actual implementation of the RK4 method uses 4 approximations:

$$k_1 = f(u_n, t_n) \tag{5}$$
$$k_2 = f(t_n + \frac{h}{2}, u_n + \frac{h}{2} * k_1) \tag{6}$$
$$k_3 = f(t_n + \frac{h}{2}, u_n + \frac{h}{2} * k_2) \tag{7}$$
$$k_4 = f(t_n + h, u_n + h * k_3) \tag{8}$$

A general form of the RK4 method is represented as:

$$u_{n+1} = u_n + (a_1 k_1 + a_2 k_2 + a_3 k_3 + a_4 k_4)h \tag{9}$$

where k_1, k_2, k_3, and k_4 are those found in equations 5, 6, 7, and 8 respectively, $h = t_{n+1} - t_n$, the time step, and u_{n+1} at t_{n+1} can be found using u_n at t_n. Using Taylor's series on equation 12 for the first five terms gives:

$$y_{n+1} = y_n + \frac{du}{dt}(x_{n+1} - x_n) + \frac{1}{2!}\frac{d^2u}{dt^2}(x_{n+1} - x_n)^2 + \frac{1}{3!}\frac{d^3u}{dt^3}(x_{n+1} - x_n)^3 + \frac{1}{4!}\frac{d^4u}{dt^4}(x_{n+1} - x_n)^4 \tag{10}$$

We know that $\frac{du}{dt} = \dot{u}$ where u is:

$$u = \begin{bmatrix} \boldsymbol{r}_i \\ \boldsymbol{v}_i \end{bmatrix} \tag{11}$$

and therefore, from equation 3 we get:

$$\dot{u} = \frac{d}{dt}\begin{bmatrix} \boldsymbol{r}_i \\ \boldsymbol{v}_i \end{bmatrix} = \begin{bmatrix} \dot{\boldsymbol{r}}_{\boldsymbol{i}} \\ \dot{\boldsymbol{v}}_{\boldsymbol{i}} \end{bmatrix} = \begin{bmatrix} \boldsymbol{v}_i \\ \boldsymbol{a}_i \end{bmatrix} = f(u,t) \tag{12}$$

$$\ddot{u} = \frac{d^2}{dt^2}\begin{bmatrix} \boldsymbol{r}_i \\ \boldsymbol{v}_i \end{bmatrix} = \begin{bmatrix} \ddot{\boldsymbol{r}}_{\boldsymbol{i}} \\ \ddot{\boldsymbol{v}}_{\boldsymbol{i}} \end{bmatrix} = \begin{bmatrix} \dot{\boldsymbol{v}}_{\boldsymbol{i}} \\ \dot{\boldsymbol{a}}_{\boldsymbol{i}} \end{bmatrix} = \dot{f}(u,t) \tag{13}$$

and so on ...

Additionally, we also ascertained that time step $h = t_{n+1} - t_n$. so equation 13 becomes:

$$u_{n+1} = u_n + f(u_n, t_n)h + \frac{1}{2!}f'(u_n, t_n)h^2 + \frac{1}{3!}f''(u_n, t_n)h^3 + \frac{1}{4!}f'''(u_n, t_n)h^4 \tag{14}$$

using equations 8-11, 13, and 17, we can derive the accurate form on the RK4 method as:

$$u_{n+1} = u_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)h \tag{15}$$

because $a_1 = \frac{1}{6}, a_2 = \frac{2}{6}, a_3 = \frac{2}{6}$, and $a_4 = \frac{1}{6}$

2.2 Justification of our Numerical Methods

If we are to claim that the data produced by our simulation is reliable for use in an application where an N-body model is required, it is vitally important that we have a valid way of verifying our results. Even more importantly, we must explore the extent to which our model provides valid data by investigating the truncation error and stability regions of the RK methods we chose to implement and figuring out under what conditions does our model fail. Beyond just the truncation error and stability of our numerical methods, the model will be justified further by testing whether energy is conserved throughout the simulation, and by comparing our data to data obtained from NASA/JPL horizons database on the real orbits of planets in our Solar System (these will be covered in later sections).

Truncation Error

Proof by induction for the truncation error of n-stage RK-methods

The model we implemented uses RK4 to iteratively solve for the progression of the state ($\boldsymbol{u} = f(\boldsymbol{r}, \boldsymbol{v})$)) of each body in the simulation. The type of Runge-Kutta methods that have been explored in class only use information from the current timestep to iterate to the next time step. These are, by definition, one-step methods. Although these methods are one-step methods, to get the state at time t_{i+1}, an n-stage RK method complies multiple stages to increase the accuracy of approximation on a scale proportional to the number of stages used. These stages are actually functions of the state, and they are conveniently named the stage-functions of Runge-Kutta. While we know the truncation error of an n-stage RK method will vary depending on how many stages are used, and that it generally increases as more stages are used, the exact relationship between the number of stages and order of accuracy of the global truncation error has not been established in this course. The proposed *hypothesis* is that the truncation error of any s-step RK method is on the magnitude of $O(\Delta t^s)$. This hypothesis will be evaluated via a proof by induction.

Setup

In general, Runge-Kutta methods will take multiple RK stage-functions as their inputs and will combine these stage functions to approximate the value of the next step of the IVP. It is important to begin by defining exactly what class of Runge-Kutta methods will be evaluated by this proof, as there are multiple other types of RK-methods outside of the scope of this class that do not conform to the principle laid out by the hypothesis. To be specific, the type of Runge-Kutta approximation that this proof will cover is an explicit, one-step Runge-Kutta method which takes the form:

$$u_{i+1} = u_i + \Delta t \sum_{i=1}^{s} b_i y_i \tag{16}$$

With y representing each stage function, s being the number of stage-functions desired and b being a constant coefficient for each y that will vary for different values of s.

Each RK stage-function y_s will have an input that includes the state and any previously defined y_i $\forall i \in 1, 0, \cdots, s-1$. Generally, each RK-function would also take an adjusted form of the current time t_i as an input. However, for the purposes of this project, f is not a function of time, and therefore, the equations for each RK stage-function simplify to:

$$\begin{aligned} y_1 &= f(u_i) \\ y_2 &= f(u_i + h(a_{21}(y_1))) \\ y_3 &= f(u_i + h(a_{31}(y_1) + a_{31}(y_2))) \\ &\vdots \\ y_s &= f(u_i + h(a_{s1}(y_1) + a_{s2}y_2, + \cdots + a_{s,s-1}y_{s-1})) \end{aligned} \tag{17}$$

The coefficients a_{ij} are known and vary based on the chosen number of stages s for any RK-method. They are generated by something called the *Runge-Kutta matrix* or the *Butcher Tableau* [1], which is a topic out of the scope of this class. However, we are not concerned with generating the correct formula for an arbitrary s-stage RK method, we only wish to determine the global truncation error of these methods. Consequently, we can use the fact that each coefficient is just a number whose value is ultimately left up to the discretion of the numericist to decide to prove the *hypothesis* without concerning ourselves the very elaborate, non-trivial algorithm that is used to generate the coefficients. In this sense, each a_{ij} will be treated almost like a free variable, or a "free coefficient".

Note that there are undoubtedly some slick individuals who have come up with clever alternative definitions for stage functions, but the stage-functions used to create the N-body simulation in this paper ascribe to the definition above. Thus, the proof in this paper will only explore Runge-Kutta methods whose stage functions can be defined exactly as above.

For the final step to set up this proof, the definition of an RK stage-function is used to show that any s-stage Runge-Kutta function will include n terms for $n \in 1, 2, ..., s$. Each term contains the nth derivative of the state $\mathbf{u}$ with respect to time (where the state raised to a number n will represent the nth derivative of the state, i.e. $\boldsymbol{u}^{(n)}$). Along with the nth derivative, each term will contain a constant coefficient and a factor of $(\Delta t)^{(n-1)}$ to accompany each derivative in the function. To make this pattern clear we begin with $s = 1$:

$$y_1 = f(u_i) = \dot{u}_i$$

Observe that, by definition, f serves as an operator to take the derivative of its input with respect to time. It is then clear that the above expansion of y_1 conforms to the specified pattern. y_1 is comprised of one term (corresponding to $n = 1$) containing the first derivative of the state with respect to time, coefficient of 1 and is multiplied by a factor of $(\Delta t)^{1-1} = (\Delta t)^0$. For $s = 2$:

$$y_2 = f(u_i + \Delta t(a_{21}(y_1))) = \dot{u_i} + \Delta t(a_{21}\ddot{u_i})$$

Again for $s = 2$, y_2 conforms to the specified pattern. y_2 is defined by two terms (corresponding to $n = 1, 2$) containing the first and second derivative of the state respectively. The first term (or n=1 term) has a coefficient of one and is multiplied by a factor of $(\Delta t)^{n-1} = (\Delta t)^0$. The second term (or n=2 term) has a newly defined coefficient, a_{21}, and it is multiplied by a factor of $(\Delta t)^{n-1} = (\Delta t)^1$. For $s = 3$:

$$\begin{aligned} y_3 &= f(u_i + \Delta t(a_{31}(y_1) + a_{32}(y_2))) \\ &= f(u_i + \Delta t(a_{31}\dot{u_i} + a_{32}(\dot{u_i} + \Delta t(a_{21}\ddot{u_i})))) \\ &= \dot{u_i} + \Delta t((a_{31} + a_{32})\ddot{u_i} + \Delta t(a_{32}a_{21}\dddot{u_i}))) \\ &= \dot{u_i} + \Delta t(a_{31} + a_{32})\ddot{u_i}) + \Delta t^2(a_{32}a_{21}\dddot{u_i}) \end{aligned}$$

Again for $s = 3$, y_3 conforms to the specified pattern. y_3 is defined by three terms containing the first, second and third derivative of the state respectively, each of the n terms multiplied by $(\Delta t)^{n-1}$. Importantly, two new coefficients are introduced, a_{31} on the $\ddot{u_i}$ term and both a_{31} anda_{32} on the $\dddot{u_i}$. This is significant because, despite the fact that a_{21} is previously defined for y_2, the two coefficients a_{31} and a_{32} are not previously defined and can be chosen such that each of the nth derivatives of $\mathbf{u}$ can be any number regardless of what a_{21} was. For convenience, the unique coefficients of the case $s = 3$ are called α_3 and β_3. Continuing this pattern, the coefficients of each of the nth derivatives of $\mathbf{u}$ can be proven to be unique to the stage-function they belong to by looking at the matrix a_{ij}. a_{ij} is a matrix of coefficients such that:

$$a_{ij} = \begin{bmatrix} a_{11} & & & \text{sym.} \\ a_{21} & a_{22} & & \\ \vdots & \vdots & \ddots & \\ a_{i1} & a_{i2} & \cdots & a_{ij} \end{bmatrix}$$

Each ith row of coefficients corresponds to the coefficients for each stage-function in an RK-method with $s = i$ stages. Due to this, every additional ith row will add exactly i new coefficients to the system of equations used to find y_s (since every row i only contains i unique coefficients, not j unique coefficients as a_{ij} is symmetric). This means that, each y_s has exactly s new terms coming from the matrix $[a_{ij}]$, giving a system of s new coefficients and s terms per individual y_s. This is great because such a system will have exactly one solution for the terms of the matrix $[a_{ij}]$ since there are the same number of equations as unknowns. The exact values of the entries of $[a_{ij}]$ depend on what the final, single-number coefficients of each term in y_s are, but remember, we don't care about assembling an actual working general method with the correct $[a_{ij}]$ coefficients. Instead, this knowledge can be used to claim that no matter what the final numeric values are of the coefficient corresponding to each term in the sequence $(\Delta t)^{(n-1)}\boldsymbol{u}^{(n)} \forall n \in 1, 2, ...s$ for each y_s, there exists a unique set of coefficients a_{ij} that will make that coefficient true. This means that for $s = 3$ we can write the coefficients as some arbitrary numbers (represented by Greek letters) instead of a combination of a_{ij}:

$$
\begin{aligned}
y_3 &= \dot{u_i} + \Delta t(a_{31} + a_{32})\ddot{u_i} + \Delta t^2((a_{32}a_{21}\dddot{u_i})) \\
&= \alpha_3\dot{u_i} + \Delta t(\beta_3\ddot{u_i}) + \Delta t^2(\lambda_3\dddot{u_i})
\end{aligned}
$$

The general case is then written:

$$
\begin{aligned}
y_s &= f(\alpha_s u_i + \Delta t(\beta_s y_1 + \lambda_s y_2, + \cdots + \gamma_s y_{s-1})) \\
&= f(\alpha_s u_i + (\Delta t)\beta_s\dot{u} + (\Delta t)^2\lambda_s\ddot{u}, + \cdots + (\Delta t)^{s-1}\gamma_s u^{(s-1)}) \\
&= \alpha_s\dot{u} + (\Delta t)\beta_s\ddot{u} + (\Delta t)^2\lambda_s\dddot{u}, + \cdots + (\Delta t)^{s-1}\gamma_s u^{(s)}
\end{aligned}
$$

Continuing this pattern to $s = s + 1$:

$$
\begin{aligned}
y_{s+1} &= f(\alpha_{s+1} u_i + \Delta t(\beta_{s+1} y_1 + \lambda_{s+1} y_2, + \cdots + \gamma_{s+1} y_s)) \\
&= f(\alpha_{s+1} u_i + (\Delta t)\beta_{s+1}\dot{u} + (\Delta t)^2\lambda_{s+1}\ddot{u}, + \cdots + (\Delta t)^s\gamma_{s+1} u^{(s)}) \\
&= \alpha_{s+1}\dot{u} + (\Delta t)\beta_{s+1}\ddot{u} + (\Delta t)^2\lambda_{s+1}\dddot{u}, + \cdots + (\Delta t)^s\gamma_{s+1} u^{(s+1)}
\end{aligned}
$$

The knowledge that this pattern holds for any sized RK-n method of our definition will become useful in the upcoming proof.

Base case

First, let's investigate the accuracy of the simplest Runge-Kutta method, the one-stage method (aka Forward-Euler), to serve as our base case:

$$
u(t_{i+1}) = u(t_i) + \Delta t f(u_i, t_i)
$$

There is no need to write this in terms of stage functions since the only one would be $y_1 = f(u_i, t_i)$, and this the quantity can easily be substituted in directly. The truncation error for this method can be found by assuming we have the true solution for $u(t_{i+1})$, and then solving for the remainder that is produced by iterating through one time step via the finite difference method used. First setup the equation for truncation error:

$$
\begin{aligned}
\tau &= \frac{u(t_{i+1}) - u(t_i)}{\Delta t} - f(u_i, t_i) \\
&= \frac{u(t_{i+1}) - u(t_i)}{\Delta t} - (\dot{u}(t_i))
\end{aligned}
$$

Next we find $\dot{u}(t_i)$ in terms of $u(t_{i+1})$ and $u(t_i)$ via a Taylor expansion about $u(t_{i+1})$

$$u(t_{i+1}) = u(t_i) + \Delta t \cdot \dot{u}(t_i) + (\Delta t^2)\frac{\ddot{u}(t_i)}{2} + (\Delta t^3)\frac{\dddot{u}(t_i)}{6} + \cdots$$
$$\implies \dot{u}(t_i) = \frac{u(t_{i+1}) - u(t_i)}{\Delta t} + c \cdot O(\Delta t) + O(\Delta t^2) + H.O.T.$$

Where c is just $\frac{\dot{u}(t_i)}{2}$ simplified as a constant. The constant isn't particularly relevant to the base case, but will prove vital in the inductive step. Plugging back into our equation for truncation error we find:

$$\begin{aligned}\tau &= \frac{u(t_{i+1}) - u(t_i)}{\Delta t} - \frac{u(t_{i+1}) - u(t_i)}{\Delta t} + c \cdot O(\Delta t) + O(\Delta t^2) + H.O.T. \\ &= c \cdot O(\Delta t) + O(\Delta t^2) + H.O.T.\end{aligned}$$

Thus, the truncation error of an $s = 1$-stage RK-method is of the form $c \cdot O(\Delta t) + O(\Delta t^2) + H.O.T.$ which is ultimately on the order of $O(\Delta t)$.

Inductive Step

Now suppose that the truncation error of an s-staged RK method is $O(t^s)$ which can take the form $c \cdot O(\Delta t^s) + O(\Delta t^{s+1})$ where c is just some constant multiplied by $O(\Delta t^s)$. We can prove that the $s+1$ RK method will have a truncation error on the order of $O(t^{s+1})$. Substituting into equation (18) an $s+1$ RK method yields:

$$\boldsymbol{u}_{i+1} = \boldsymbol{u}_i + \Delta t \sum_{i=1}^{s+1} b_i y_i$$

To find the truncation error we assume we have the real solution for $\mathbf{u}$ at t_{i+1}, just like we did for Forward Euler, and then find the difference between the true solution and the approximate solution from our $n = s + 1$-stage RK method:

$$\begin{aligned}\tau_{s+1} &= \frac{u(t_{i+1}) - u(t_i)}{\Delta t} - \sum_{i=1}^{s+1} b_i y_i \\ &= \frac{u(t_{i+1}) - u(t_i)}{\Delta t} - (\sum_{i=1}^{s} b_i y_i + b_{s+1} y_{s+1})\end{aligned}$$

We know that via the definition of truncation error that:

$$\tau_s = \frac{u(t_{i+1}) - u(t_i)}{\Delta t} - \sum_{i=1}^{s} b_i y_i$$

Which is equal to $c \cdot O(\Delta t^s) + O(\Delta t^{s+1})$ via our inductive hypothesis. Thus τ_{s+1} becomes:

$$\tau_{s+1} = \frac{u(t_{i+1}) - u(t_i)}{\Delta t} - (b_{s+1})(y_{s+1}) + c \cdot O(\Delta t^s) + O(\Delta t^{s+1})$$

Perfect, now solving for the expression $\frac{u(t_{i+1})-u(t_i)}{\Delta t} - (y_{s+1})$ using the definition of (y_{s+1}):

$$\frac{u(t_{i+1}) - u(t_i)}{\Delta t} - (y_{s+1}) = \frac{u(t_{i+1}) - u(t_i)}{\Delta t} - \left(\alpha_{s+1}\dot{u} + (\Delta t)\beta_{s+1}\ddot{u} + (\Delta t)^2\lambda_{s+1}\ddot{u}, + \cdots + (\Delta t)^s\gamma_{s+1}u^{(s+1)}\right)$$

In this case, because each term in y_{s+1} is multiplied by an yet to be defined coefficient, b_{s+1} can be distributed into the coefficient alongside each term of the stage function. Then, the Taylor expansion of $u(t_{i+1})$ is taken to rewrite the $\dot{u}$ term in the expression above:

$$\begin{aligned}
u(t_{i+1}) &= u(t_i) + \Delta t \cdot \dot{u}(t_i) + \frac{(\Delta t)^2\ddot{u}(t_i)}{2} + \frac{(\Delta t)^3\dddot{u}(t_i)}{3!} + \cdots + \frac{(\Delta t)^{s+1}u^{(s+1)}}{(s+1)!} + \frac{(\Delta t)^{s+2}u^{(s+2)}}{(s+2)!} + O(\Delta t^{s+3}) \\
\implies \dot{u}(t_i) &= \frac{u(t_{i+1}) - u(t_i)}{\Delta t} - \frac{\frac{(\Delta t)^2\ddot{u}(t_i)}{2} + \frac{(\Delta t)^3\dddot{u}(t_i)}{3!} + \cdots + \frac{(\Delta t)^{s+1}u^{(s+1)}}{(s+1)!} + \frac{(\Delta t)^{s+2}u^{(s+2)}}{(s+2)!} + O(\Delta t^{s+3})}{\Delta t} \\
&= \frac{u(t_{i+1}) - u(t_i)}{\Delta t} - \left[\frac{(\Delta t)\ddot{u}(t_i)}{2} + \frac{(\Delta t)^2\dddot{u}(t_i)}{3!} + \cdots + \frac{(\Delta t)^s u^{(s+1)}}{(s+1)!} + \frac{(\Delta t)^{s+1}u^{(s+2)}}{(s+2)!} + O(\Delta t^{s+2})\right]
\end{aligned}$$

Plugging back into the expression from solving for $\frac{u(t_{i+1})-u(t_i)}{\Delta t} - \alpha_{s+1}\dot{u}$ where α_{s+1} is given the value 1:

$$\begin{aligned}
\frac{u(t_{i+1}) - u(t_i)}{\Delta t} - \dot{u} &= \frac{u(t_{i+1}) - u(t_i)}{\Delta t} - \frac{u(t_{i+1}) - u(t_i)}{\Delta t} + \\
&\quad \left[\frac{(\Delta t)\ddot{u}(t_i)}{2} + \frac{(\Delta t)^2\dddot{u}(t_i)}{3!} + \cdots + \frac{(\Delta t)^s u^{(s+1)}}{(s+1)!} + \frac{(\Delta t)^{s+1}u^{(s+2)}}{(s+2)!} + O(\Delta t^{s+2})\right] \\
&= \left[\frac{(\Delta t)\ddot{u}(t_i)}{2} + \frac{(\Delta t)^2\dddot{u}(t_i)}{3!} + \cdots + \frac{(\Delta t)^s u^{(s+1)}}{(s+1)!} + \frac{(\Delta t)^{s+1}u^{(s+2)}}{(s+2)!} + O(\Delta t^{s+2})\right]
\end{aligned}$$

Now substituting this simplified expression back into the expression for τ_{s+1} we get:

$$\begin{aligned}
\tau_{s+1} = \frac{(\Delta t)\ddot{u}(t_i)}{2} + \frac{(\Delta t)^2\dddot{u}(t_i)}{3!} + \cdots + \frac{(\Delta t)^s u^{(s+1)}}{(s+1)!} + \frac{(\Delta t)^{s+1}u^{(s+2)}}{(s+2)!} + O(\Delta t^{s+2}) - \\
(\Delta t)\beta_{s+1}(\ddot{u}) + (\Delta t)^2\lambda_{s+1}\ddot{u}, + \cdots + (\Delta t)^s\gamma_{s+1}u^{(s+1)} + c \cdot O(\Delta t^s) + O(\Delta t^{s+1}) + H.O.T.
\end{aligned}$$

Recall that because n new free coefficients were brought in for every new term in each of our n stage functions, $\beta_{s+1}, \lambda_{s+1}, \cdots, \gamma s + 1$ can be any number we choose. We tactfully choose them such that as many of the terms from the Taylor series that resulted from expanding $u(t_{i+1})$ cancel out with the terms from each RK stage-function as possible *as well as* the $c \cdot O(\Delta t^s)$ term brought in from the truncation error of the previous RK-method with $n = s$ stages. After cancelling out as many terms as possible, which we just justified our ability to do, we are left with:

$$\tau_{s+1} = \frac{(\Delta t)^{s+1}u^{(s+2)}}{(s+2)!} + O(\Delta t^{s+1}) + H.O.T.$$

The truncation error of the $n = s+1$ RK-n method is then dominated by terms on the order of $O(\Delta t^{s+1})$ and takes the form $c \cdot O(\Delta t^{s+1}) + H.O.T.$ which is what we needed to show.

Because the forward Euler(RK1), Heun's(RK2) and RK4 methods all fit the category of explicit, one-step RK methods as defined by equation (18) and have step functions that follow the equation (19), we can apply the finding of this proof to these RK methods. Hence, we can say the forward Euler method will have a truncation error of $O(\Delta t)$, the Heun's method will have a truncation error of $O(\Delta t^2)$ and the RK4 method will have a truncation error of $O(\Delta t^4)$. This means that as we decrease our step size Δt, each timestep through our simulation for each of our RKn methods will produce a result that is erroneous by a factor of $O(\Delta t^n)$. Therefore, if each method is given the same value for Δt, the RK4 should provide the most accurate results of our methods, followed by Heun's and forward Euler. Note, that the RK-7 method investigated later in this paper *does not* follow the definitions and assumptions used in the proof, and therefore, it will be analyzed by the results it produces rather than on a theoretical level (the theory behind this method is a bit outside the scope of this course).

2.3 Theoretical Stability

The stability of our methods can first be probed by determining the region of stability for each method. Finding the exact stability matrix for our specific model is particularly tricky given that we have a large, non-linear system. However, we are limiting our choice of numerical methods to only include explicit Runge-Kutta methods, so we can try and visualize the region of the complex plane for which some RK methods are absolutely stable and compare them. The absolute stability regions for forward Euler(RK1), Heun's method(RK2) and RK4 are as follows:

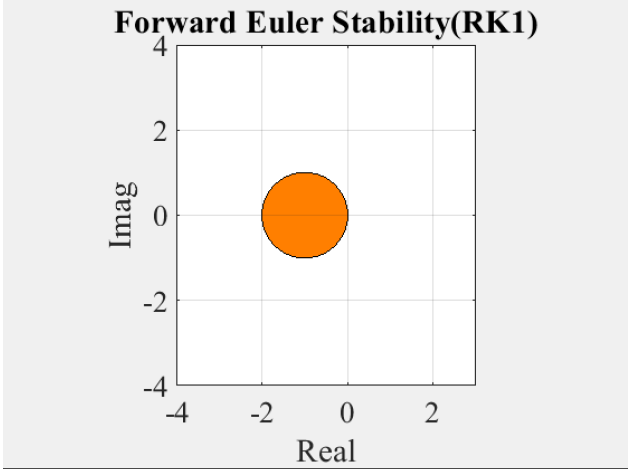

Figure 1: Absolute stability region for the forward Euler method.

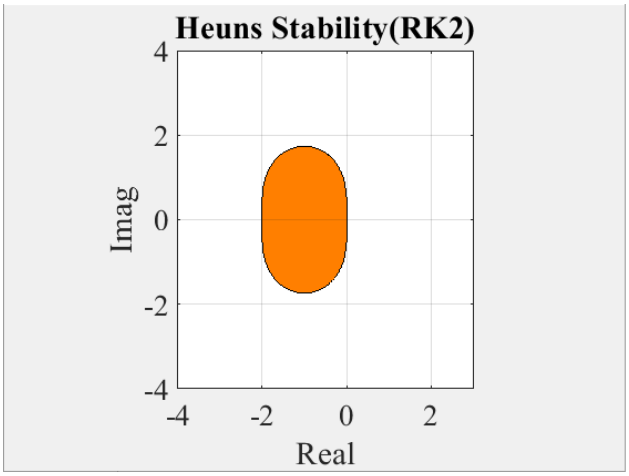

Figure 2: Absolute stability region for Heun's method.

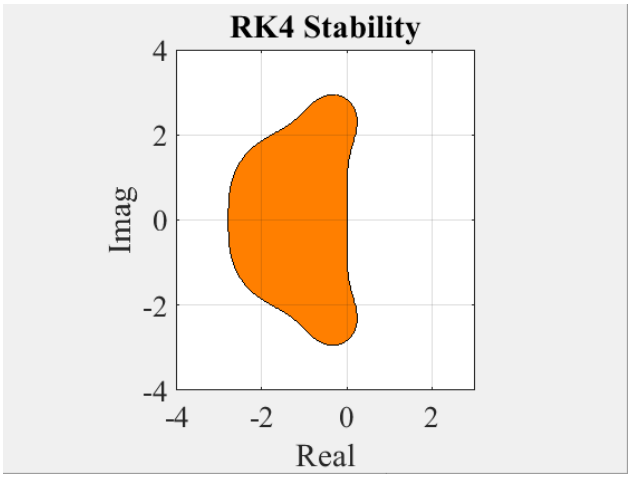

Figure 3: Absolute stability region for the RK4 method.

The graphs indicate that as the number of stages, n, increases in a given RK-n method, the area of absolute stability will also increase. This gave even more of a reason for avoiding Forward Euler and Heun's, and we decided that going with at least a four stage RK method would likely be ideal in order to achieve a larger region of absolute stability. However, this isn't really enough information to truly justify that our model is stable. Instead of continuing to test the general form of the methods on an more abstract level, we need to start testing the actual data produced by our model.

All relevant code to generate stability plots can be found in Code Appendix B.2.

3 Simulation Experimentation

3.1 Implementation

The code implementation is set up by taking advantage of Python's object oriented coding. A class object called Body is defined having the class variables name, mass, position, and velocity. For each body in the system, this allows us to keep track of the mass, position and velocity at any given time. Each Body is initiated with the initial position and velocity, and this is how we implement the initial conditions for N bodies.

```
class Body():
    def __init__(self, name, mass, pos0, vel0):
        self.name = name
        self.mass = mass
        self.pos = pos0
        self.vel = vel0
```

Figure 4: Body Class

A second class object called Simulation is then defined with the meat and potatoes of the code. It is defined having the class variables G (the gravitational constant), an array of bodies of class Body, a softening term, and total energy of the system. These are variables the simulation will need to calculate each time step, and total energy will be tracked over time for stability reasons, as will be explained later.

```
class Simulation():
    def __init__(self, G, softening, bodies):
        self.G = G
        self.bodies = bodies
        self.softening = softening
        self.energy = 0
        self.update_energy()
```

Figure 5: Simulation Class

After setting up a Body and Simulation class, we can use them together by creating instances of the Body class, and calling them into the simulation via a vector of body objects. This way, we can easily add an arbitrary amount of bodies, and the rest of the simulation methods have been generalized to work for any N bodies.

```
Body1 = Body('Body1', mass1, pos1, vel1)
Body2 = Body('Body2', mass2, pos2, vel2)

simulation = Simulation(G, softening, [Body1, Body2])
```

Figure 6: General Body Objects in Simulation Class

u_k represents the entire state of the N body system, so a general N body system will contribute N entries to the u_k matrix. To account for an arbitrary N bodies, a dynamic array u_k is initialized of size $(N \times 2 \times 3)$. This gives us an array of size N, with each entry consisting of two vectors of size 3: position (3×1) and

velocity (3×1). We can now iterate through the bodies, filling in the current position and velocity state for each body by accessing the body variables for position and velocity as defined in Figure 4. u_k can then be returned and manipulated as needed.

The function $f(u, t)$ can be expressed using equation 2. Note that equation 2 does not depend on time, so we can simplify our function to $f(u)$. This is implemented in the code relatively straight forwardly by taking a state u_k, decomposing it based on the structure described above, calculating $\ddot{\boldsymbol{r}}_i$ as shown in the right hand side of 2, and rearranging. This solution can then be outputted and used in our finite difference method of choice.

Once we have u_k and our function $f(u)$, we can implement a finite difference method to solve the IVP. We implemented RK2 and RK4 identically to implementation in previous homeworks, so this will not be covered. RK7 was implemented very similarly, except using 7 intermediate steps with various constants as found from [2].

Finally, we implement a run function for the Simulation class, which loops through a total time T with steps dt, calculating u_{k+1} and updating the Body variables for position and mass with each step. This loop also records the positional data of each body, and after the loop, the data is plotted to draw the dynamics of the system over time.

All relevant code is Code Appendix B.1.

3.2 Simulation Stability

A test of stability for the simulation can be done by creating a convergence plot for each method using various Δt. A relevant vector of Δt was chosen through trial and error to show a key transition in the behavior of the system, which will be explored further below. The Δt were chosen as $\Delta t = 1.5 \times 10^{-2}$, 1.25×10^{-2}, 1×10^{-2}, 7.5×10^{-3}, 5×10^{-3} and 2.5×10^{-3} years, with each respective error normalized by the solution at 1×10^{-3}. Also, in the interest of time, only the sun and the 4 inner most planets will be simulated. Because the orbital periods of the planets get smaller the closer the planet is, the effect of changing Δt becomes less pronounced for planets any further than Mars. This is because the stability is a relative relation between period of dynamics and Δt. In fact, at the chosen time steps, the only planet that noticeably fluctuates orbital path is Mercury, which makes sense because it has the fastest orbital period. The set of chosen time steps is also limited by run time, as anything below $\Delta t = 1 \times 10^{-3}$ starts to take unrealistic run times to complete.

We chose a total simulation time of 50 years, despite the orbital period of all bodies being under 2 years, because we want to be able to capture any small error that can creep up over many periods. Ultimately, we want a normalized error of at least below 0.1, or 10%, so we will also plot a horizontal line at this desired error. Running this convergence plot for a total time of 50 years, we get the following plot.

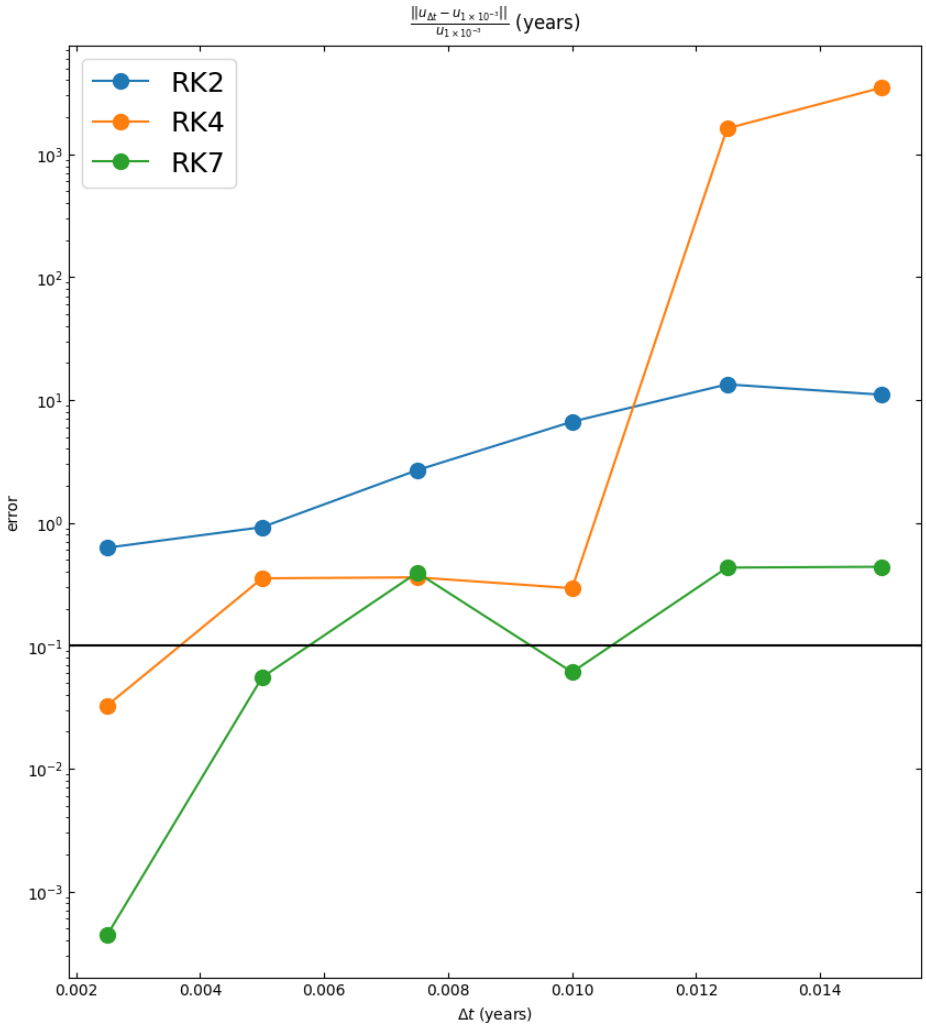

Figure 7: Error after 50 years for $\Delta t = 1.5 \times 10^{-2}, 1.25 \times 10^{-2}, 1 \times 10^{-2}, 7.5 \times 10^{-3}, 5 \times 10^{-3}, 2.5 \times 10^{-3}$ years

We will revisit this convergence plot, but for now we will also tabulate the run times it takes for each combination of method and Δt.

	Run Time (sec)		
Δt (year)	**RK2**	**RK4**	**RK7**
1.5×10^{-2}	3.06	4.05	6.77
1.25×10^{-2}	3.63	4.77	7.89
1×10^{-2}	4.59	5.67	9.92
7.5×10^{-3}	6.03	7.70	13.01
5×10^{-3}	8.91	11.74	19.45
2.5×10^{-3}	17.63	22.64	39.33

Table 1: Run times for $\Delta t = 1.5 \times 10^{-2}, 1.25 \times 10^{-2}, 1 \times 10^{-2}, 7.5 \times 10^{-3}, 5 \times 10^{-3}, 2.5 \times 10^{-3}$ years

Observing the convergence plot and run time table together, we see that there are a few attractive combinations of method a and time step to choose. However, these tests have been done with a total simulation time of 50 years and only including the four inner planets. The total time solution for the full

solar system would need to be at least 248 years, or the orbital period of Pluto, in order to capture the full dynamics of the system. As a result, we want the smallest time step that gives us an accurate solution.

Recognizing that this is normalized error, we want an error at least below 0.1, or 10%. This rules out all of the time steps for RK2, and a good chunk of time steps for RK4 as well.

3.2.1 RK2 Position and Energy

For curiosities sake, let's explore some of the "bad" combinations of method and time step, as we may be able to glean some reason's for why these don't work, why smaller time steps may work better, and hopefully arrive at a suitable combination of method and time step. We will be doing this by exploring the plot of the positions over time, as well as the plot of the total energy of the system over time. Calculation of total energy was implemented by calculating and adding the kinetic and potential energy of each body in the system at each time, recording this data and plotting.

The equation used for the calculation of energy is shown below [3]:

$$KE = \frac{1}{2}\sum_{i=1}^{N} m_i \boldsymbol{v}_i^2 \tag{18}$$

$$PE = -\sum_{1\leq i<j\leq N} \frac{G m_i m_j}{||\boldsymbol{r}_j - \boldsymbol{r}_i||} \tag{19}$$

$$TE = KE + PE \tag{20}$$

Using RK2 with $\Delta t = 1.5 \times 10^{-2}$, we get the following position and energy plots:

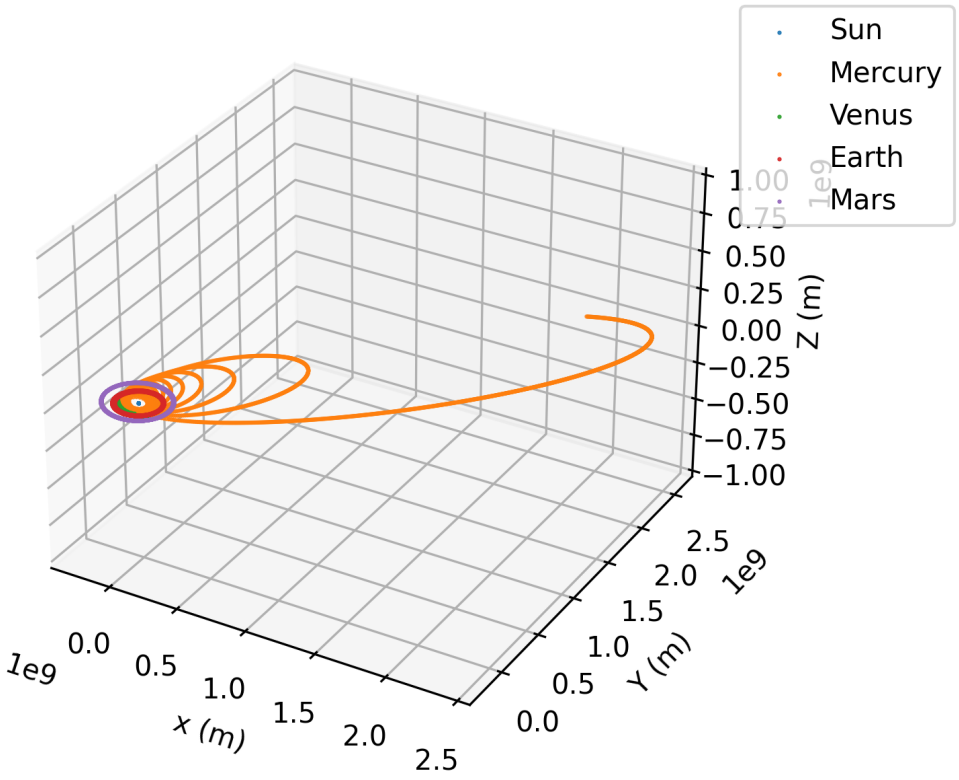

Figure 8: Simulation position for RK2 after 50 years for $\Delta t = 1.5 \times 10^{-2}$ years

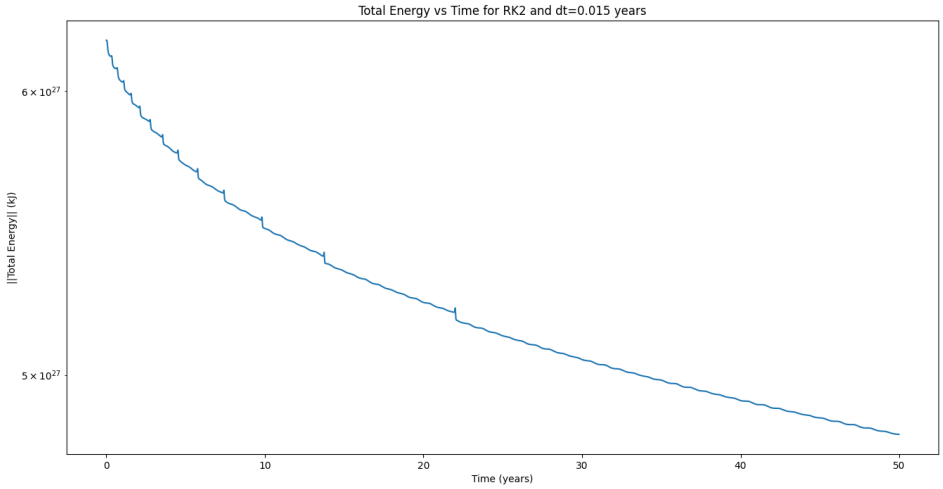

Figure 9: Simulation Energy for RK2 after 50 years for $\Delta t = 1.5 \times 10^{-2}$ years

Clearly, this is not a stable system as Mercury's orbit starts to diverge drastically over time, and this is verified by the plot of the total energy, which shows that total energy changes over time. We can calculate relevant energy changes, as tabulated below. This quantifies the error that Mercury going rogue introduces into the solution at this time step for RK2.

RK2 with $\Delta t = 1.5 \times 10^{-2}$ years			
Run time (sec)	**$\|\|$Initial Energy$\|\|$**	**$\|\|\Delta TE\|\|$**	**Percent change in Energy**
2.96	6.1975×10^{27}	1.3846×10^{27}	22.34

Table 2: Run time and Energy Change for RK2 at $\Delta t = 1.5 \times 10^{-2}$

Doing the same set of tests for RK2 with $\Delta t = 2.5 \times 10^{-3}$, we get the following plots and table.

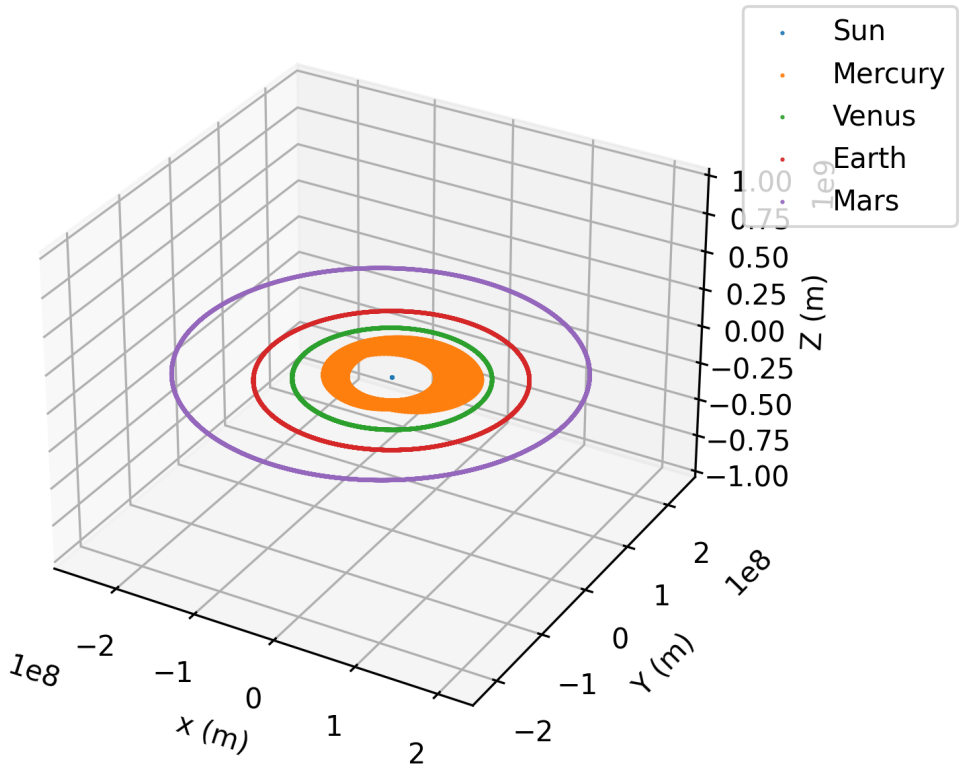

Figure 10: Simulation position for RK2 after 50 years for $\Delta t = 2.5 \times 10^{-3}$ years

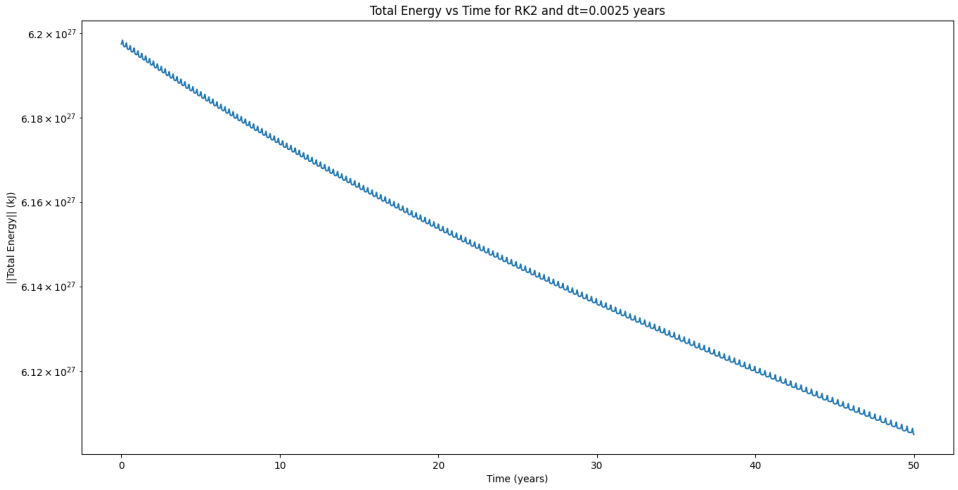

Figure 11: Simulation Energy for RK2 after 50 years for $\Delta t = 2.5 \times 10^{-3}$ years

RK2 with $\Delta t = 2.5 \times 10^{-3}$ years			
Run time (sec)	**||Initial Energy||**	**$\|\|\Delta TE\|\|$**	**Percent change in Energy**
16.95	6.1975×10^{27}	9.2371×10^{25}	1.54

Table 3: Run time and Energy Change for RK2 at $\Delta t = 2.5 \times 10^{-3}$

Looking at just the energy change, one could think that this system is a good approximation of the real N-body system. A first glance at the energy over time plot may imply that energy does change, but this change is relatively small as calculated in the table. Yet, when observing the positional plot, we see that Mercury's orbit is still not as expected, and this confirms that indeed this system is also not stable.

3.2.2 RK4 Position and Energy

Let's take a look at some Δt values for RK4. A relatively attractive time step to use based on the convergence plot is $\Delta t = 1 \times 10^{-2}$, as it is both faster to run and apparently just as accurate or more accurate than error for $\Delta t = 7.5 \times 10^{-3}, 5 \times 10^{-3}$. Though it does not reach our desired error of 10%, the logarithmic scale makes it look farther than it truly is. Plotting the positional data and total energy data with calculations, we get the following:

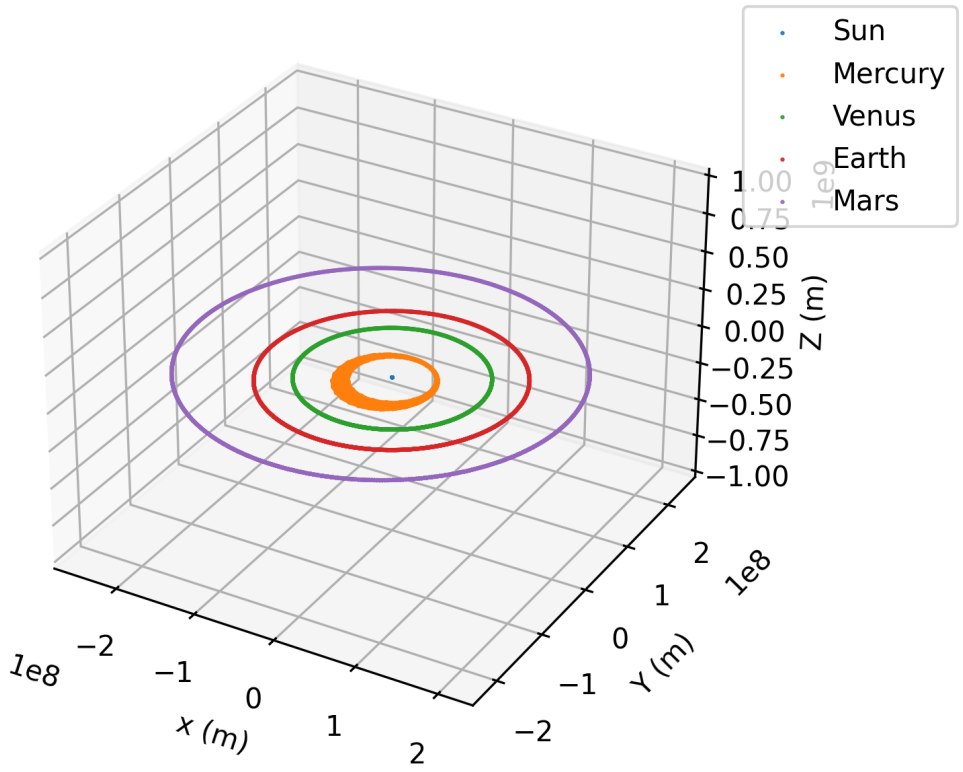

Figure 12: Simulation position for RK4 after 50 years for $\Delta t = 1 \times 10^{-2}$ years

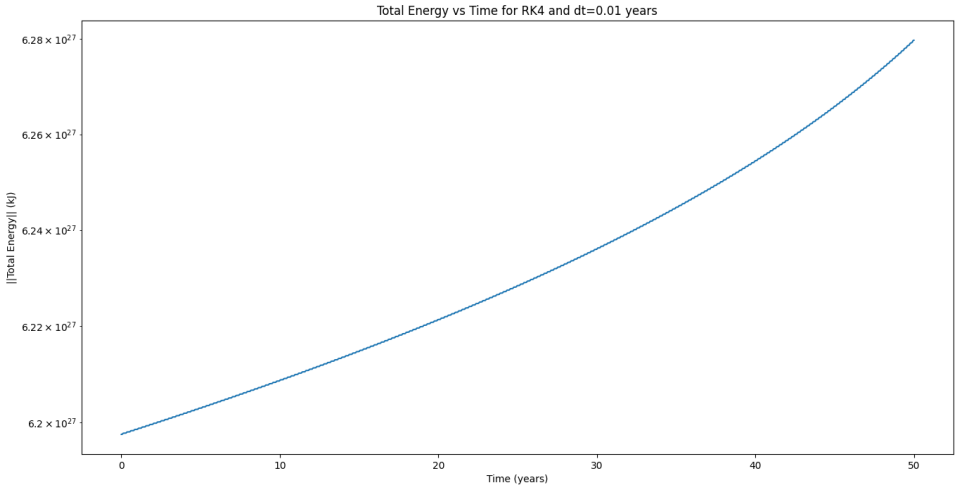

Figure 13: Simulation Energy For RK4 after 50 years for $\Delta t = 1 \times 10^{-2}$ years

RK4 with $\Delta t = 1 \times 10^{-2}$ years			
Run time (sec)	**$\|\|$Initial Energy$\|\|$**	**$\|\|\Delta TE\|\|$**	**Percent change in Energy**
5.45	6.1975×10^{27}	8.2213×10^{25}	1.33

Table 4: Run time and Energy Change for RK4 at $\Delta t = 1 \times 10^{-2}$

Similarly to RK2 at $\Delta t = 2.5 \times 10^{-3}$, looking at just the change in energy would indicate this is an accurate method to use. Even the convergence plot indicates that this combination of method and time step is below our threshold for 10% error, yet looking at the positional plot, we still see some undesired behavior in Mercury's orbit. As a result, this system is not stable either.

Observing the convergence plot, the next most intriguing time step to test for RK4 is at $\Delta t = 2.5 \times 10^{-3}$. Doing the tests for this setup, we get the following plots and table:

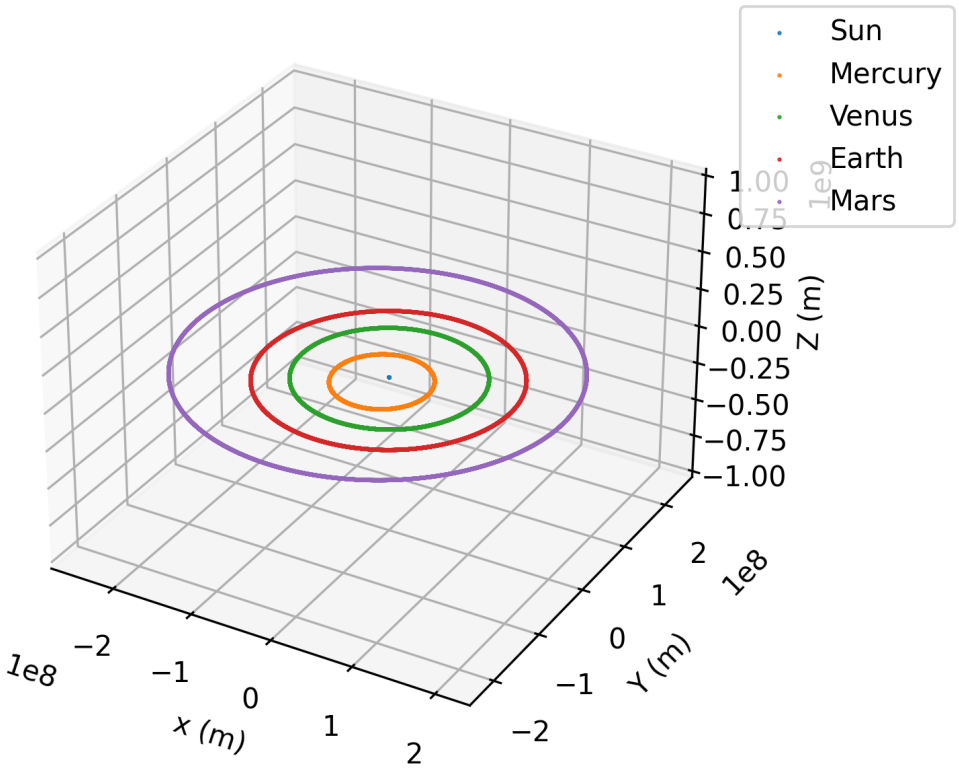

Figure 14: Simulation position for RK4 after 50 years for $\Delta t = 2.5 \times 10^{-3}$ years

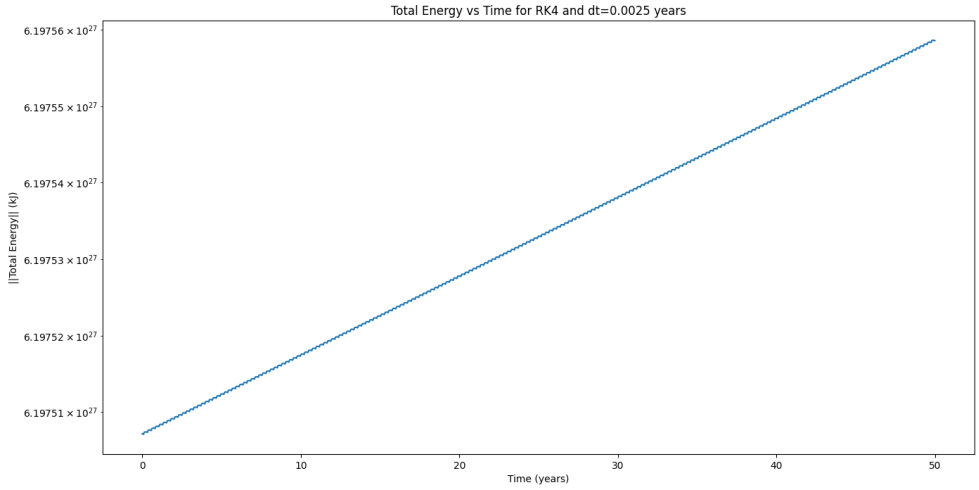

Figure 15: Simulation Energy For RK4 after 50 years for $\Delta t = 2.5 \times 10^{-3}$ years

RK4 with $\Delta t = 2.5 \times 10^{-3}$ years			
Run time (sec)	**||Initial Energy||**	**$\|\|\Delta TE\|\|$**	**Percent change in Energy**
21.35	6.1975×10^{27}	5.1400×10^{22}	8.29×10^{-4}

Table 5: Run time and Energy Change for RK2 at $\Delta t = 2.5 \times 10^{-3}$

The choice of RK4 with $\Delta t = 2.5 \times 10^{-3}$ appears to be a very attractive choice, as the convergence plot, positional plot, and energy plot all agree that there is relatively low error. Keeping a bookmark on this choice of method and time step, we will now investigate RK7.

3.2.3 RK7 Position and Energy

A very attractive time step to use for RK7 is $\Delta t = 1 \times 10^{-2}$, which looks as accurate or more accurate than error for $\Delta t = 7.5 \times 10^{-3}, 5 \times 10^{-3}$. Plotting the position, energy and tabulating the energy change yields the following:

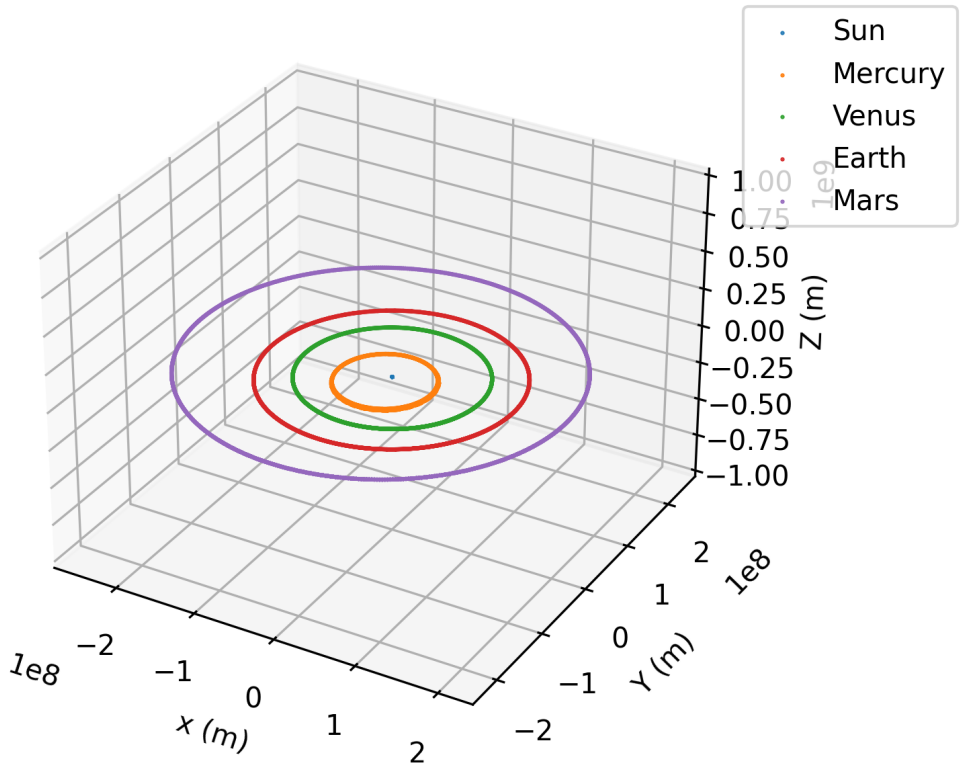

Figure 16: Simulation position for RK7 after 50 years for $\Delta t = 1 \times 10^{-2}$ years

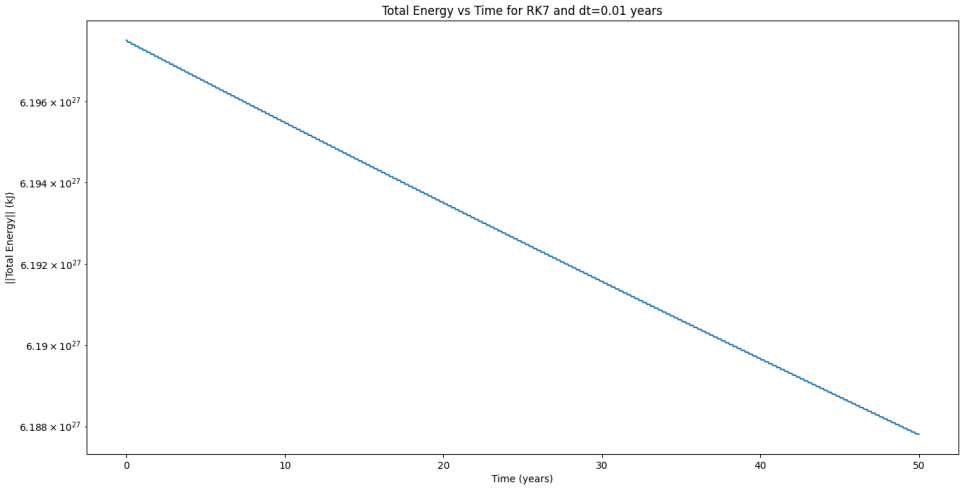

Figure 17: Simulation Energy For RK7 after 50 years for $\Delta t = 1 \times 10^{-2}$ years

RK7 with $\Delta t = 1 \times 10^{-2}$ years			
Run time (sec)	**$\|\|$Initial Energy$\|\|$**	**$\|\|\Delta TE\|\|$**	**Percent change in Energy**
9.51	6.1975×10^{27}	9.7031×10^{24}	0.157

Table 6: Run time and Energy Change for RK7 at $\Delta t = 1 \times 10^{-2}$

While energy over time and the convergence plot indicate that this is a stable system, there is a very subtle discrepancy in the positional plot. While it is hard to see in the pdf document, zooming in reveals the orbit path being slightly thicker on the side of the orbit closest to the x axis, and this suggests a slight deviation of Mercury's orbit. While this may be minor, over a total simulation time of 248 years, this discrepancy can add up, and cause a more significant error. As a result, while this system is very close, it is not quite what we are looking for.

Investigating the next attractive time step, $\Delta t = 5 \times 10^{-3}$, we get the following position plot, energy plot, and table:

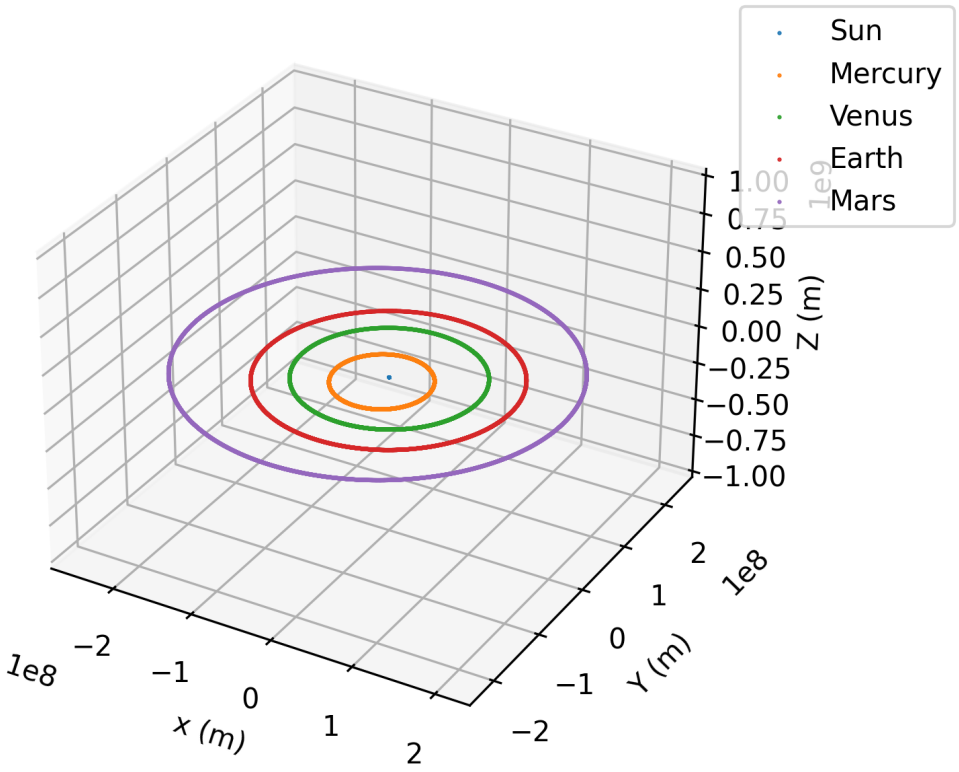

Figure 18: Simulation position for RK7 after 50 years for $\Delta t = 5 \times 10^{-3}$ years

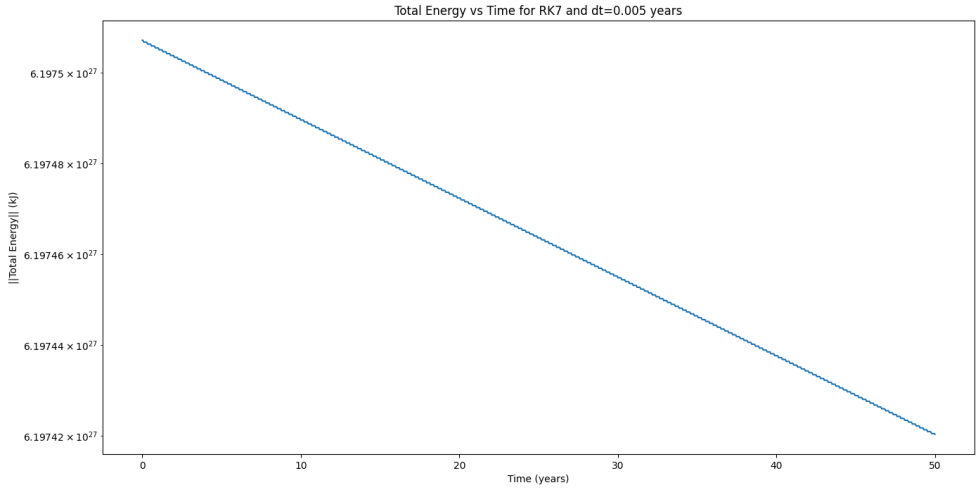

Figure 19: Simulation Energy For RK7 after 50 years for $\Delta t = 5 \times 10^{-3}$ years

RK7 with $\Delta t = 1 \times 10^{-2}$ years			
Run time (sec)	**||Initial Energy||**	**$\|\|\Delta TE\|\|$**	**Percent change in Energy**
18.92	6.1975×10^{27}	8.6799×10^{22}	1.40×10^{-3}

Table 7: Run time and Energy Change for RK7 at $\Delta t = 5 \times 10^{-3}$

The position plot for RK7 with $\Delta t = 5 \times 10^{-3}$ looks ideal, and the convergence error and energy change agree. Comparing this combination of RK7 and time step with RK4 with $\Delta t = 2.5 \times 10^{-3}$, RK7 is slightly less accurate than RK4, but it is also slightly faster to run. We will choose to stick with RK7, both for run time concerns and the relatively small difference in error. Also, we are curious how an untaught method RK7 will fare in the full simulation of the entire solar system, as well as with a black hole in place of the sun.

We will not consider RK7 with $\Delta t = 2.5 \times 10^{-3}$, as although it cuts down error significantly, it takes twice as long to run, and in the interest of significantly longer simulation times and including the rest of the planets, we will stick with RK7 with $\Delta t = 5 \times 10^{-3}$.

Thus, we have shown experimentally using our simulation that both RK4 with $\Delta t = 2.5 \times 10^{-3}$ and RK7 with $\Delta t = 5 \times 10^{-3}$ are indeed stable in this solar system model.

4 Results

We came up with two scenarios to run through our n-body simulator. These include one which models our solar system, and a theoretical scenario where the sun is replaced with a black hole of mass $= 5 \times mass_{sun}$. We obtained ephemeris data for the planets from JPL HORIZONS Web-Interface. The initial conditions used for our solar system simulations were based on data from 2021-Mar-26 at 00:00:00.0000 (Appendix A). The Sun was positioned as a stationary non-moving body and all target body positions and velocities, $(\mathbf{r}, \mathbf{v})$, were measured relative to it.

4.1 Solar System Sim - Inner Planets

We ran the simulation for the inner planets for a period of two years and the results are shown in Fig. 20. Note that while the data for only the inner planets are shown, the simulation did include all 10 bodies, but plotting the outer planets orbits at the same time would make the inner planet orbits practically invisible. A two year period was chosen because that is approximately how long Mars takes to complete an orbit around the Sun and the other planets would have completed more than one orbit as well. All the planets appear to have elliptical orbits and lie, more or less, in the same plane (Mercury not withstanding).

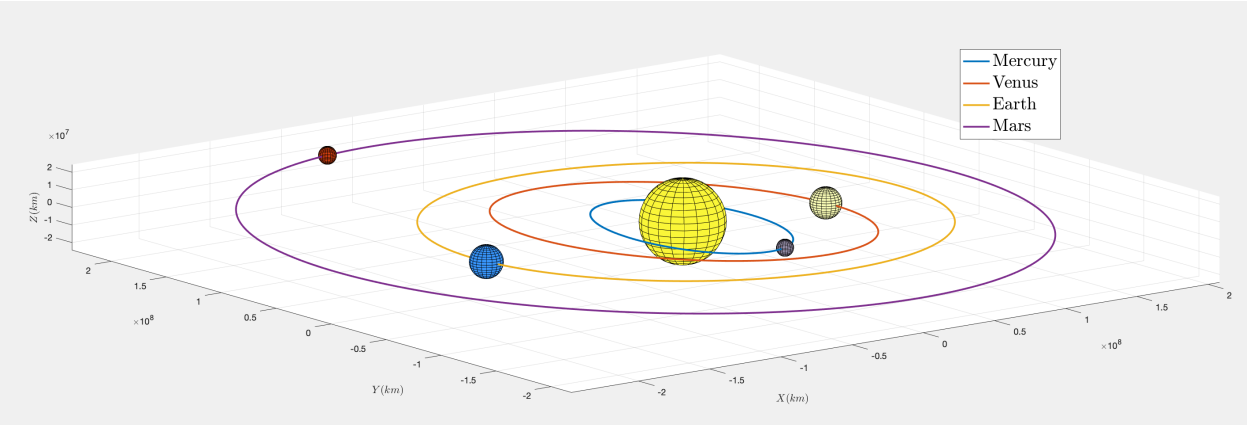

Figure 20: Orbit Propagation of Inner Planets

4.2 Solar System Sim - Outer Planets

The simulation for the outer planets covered a time span of 275 years due to the length of time to capture Pluto's orbit. Again, while the data for only the outer planets are shown, the simulation did include all 10 bodies. Figure 21 plots the results and we again see the outer planets with elliptical orbits and in the same plane (except for that rogue Pluto...).

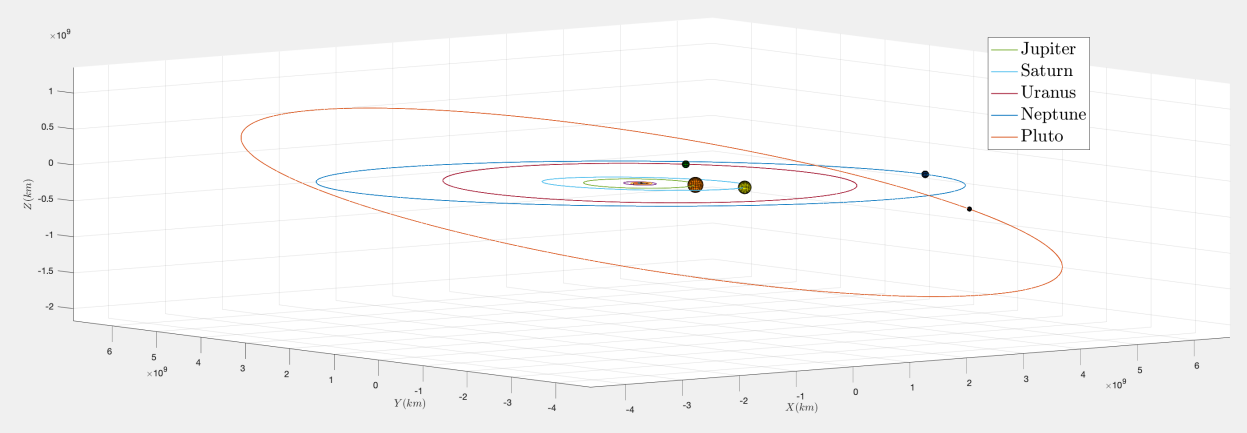

Figure 21: Orbit Propagation of Outer Planets

4.3 Sim Verification with JPL HORIZONS

In order to verify how accurately our simulation was running, we plotted our data versus JPL HORIZONS ephemeris data. We obtained two years worth of ephemeris data (2021 - 2023) for the inner planets and 85 years worth of ephemeris data (2021 - 2106) for the outer planets from JPL HORIZONS Web-Interface. NOTE: HORIZONS was unable to provide full orbital data for Neptune and Pluto this far into the future. This data was plotted in overlays with our RK7 simulator as shown in Figs. 22, 23, 24 and 25. Our simulator appears to match the JPL HORIZONS ephemeris data closely.

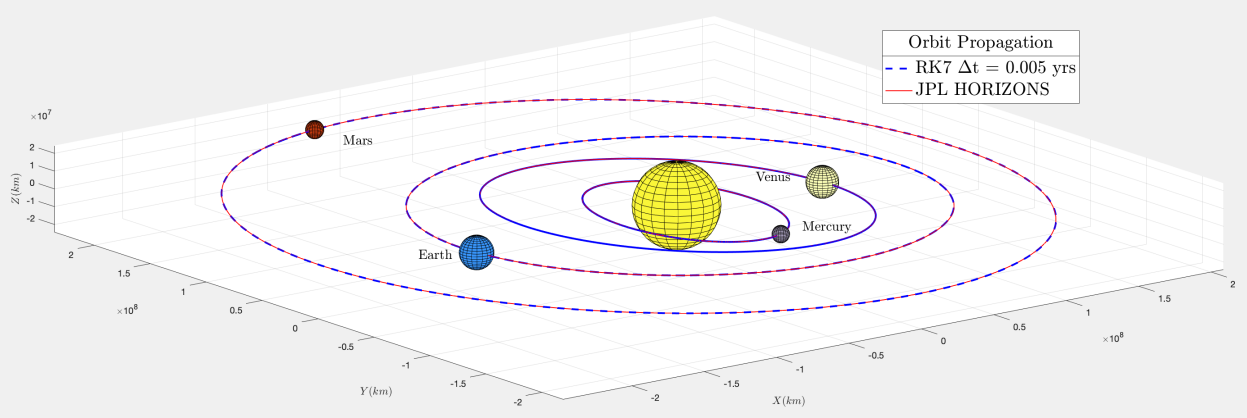

Figure 22: Sim Verification of Inner Planets

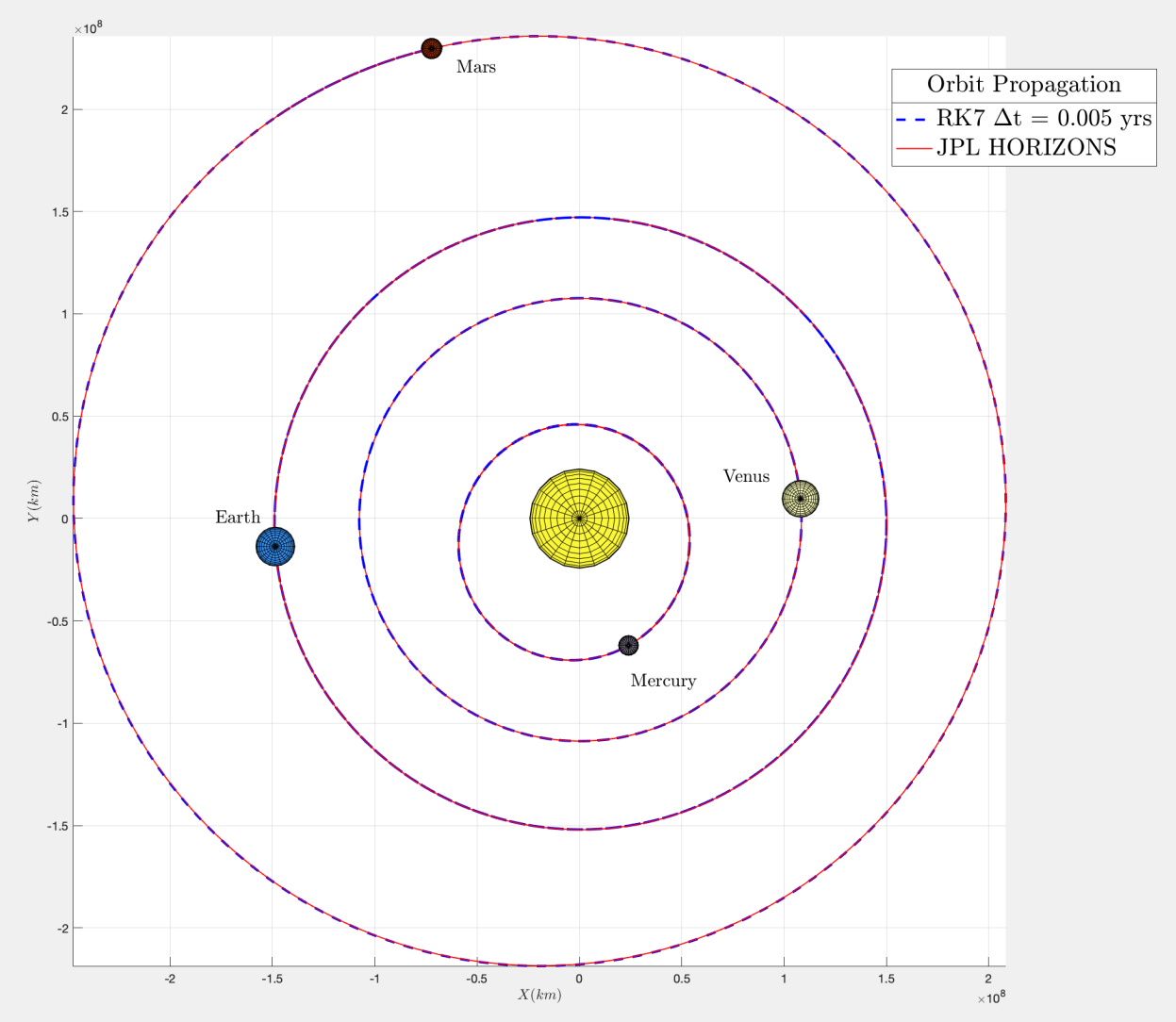

Figure 23: Sim Verification of Inner Planets along XY Plane

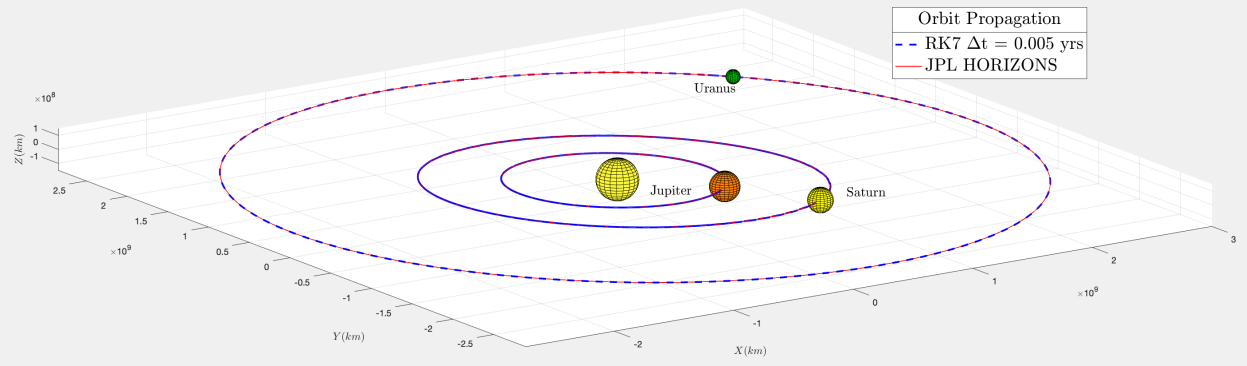

Figure 24: Sim Verification of Outer Planets

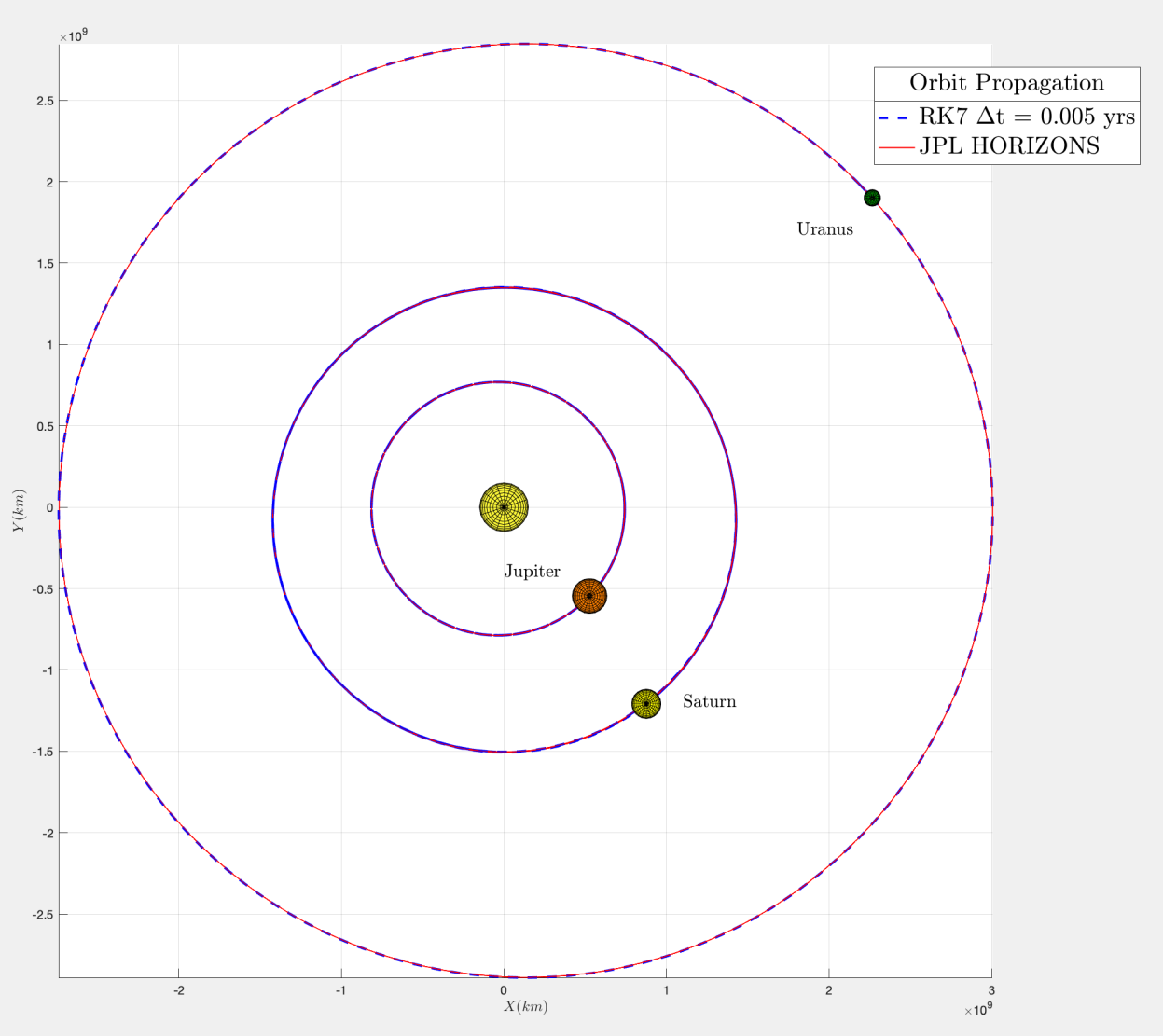

Figure 25: Sim Verification of Outer Planets along XY Plane

As another method of sim verification, we plotted the norm of the distance from the Sun to the planets as they propagated through their orbits:

$$\mathbf{r} = \sqrt{\mathbf{r_x}^2 + \mathbf{r_y}^2 + \mathbf{r_z}^2}$$

Again, this was compared to JPL HORIZONS ephemeris data: two years for the inner planets, Fig. 26, and 85 years for the outer planets, Fig. 27.

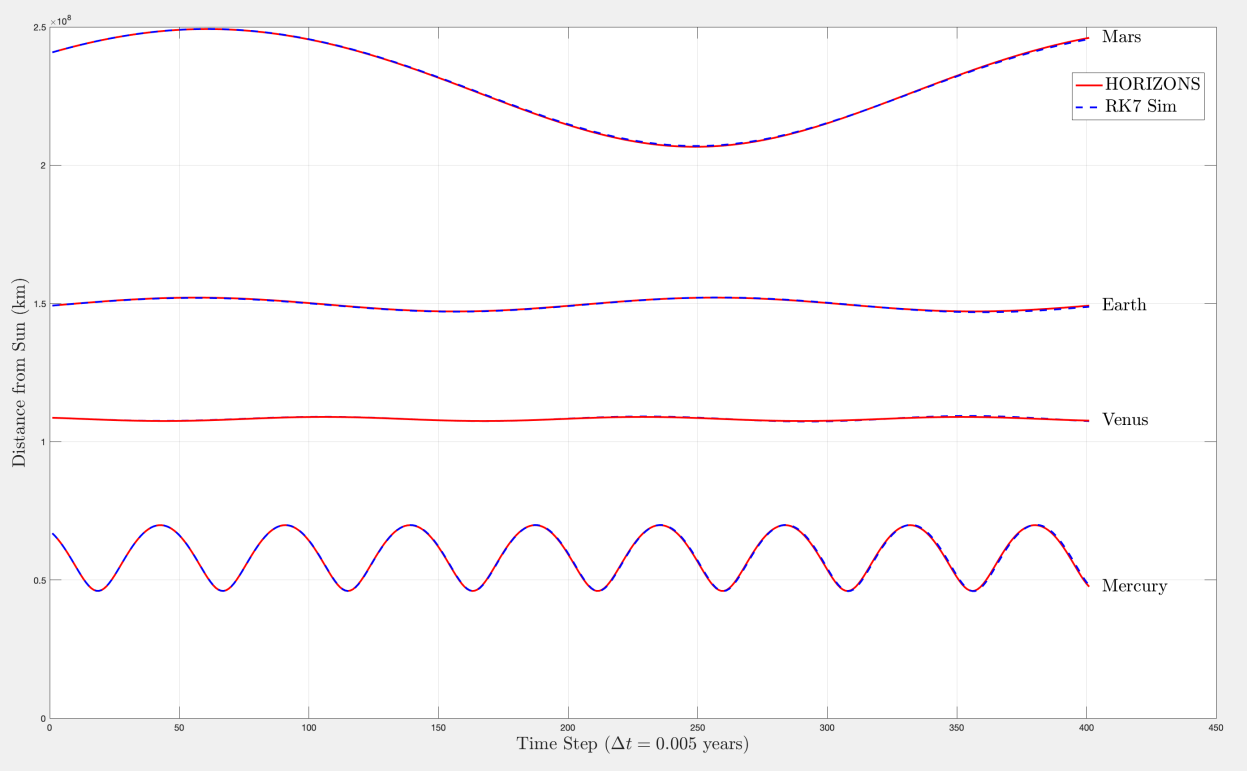

Figure 26: Sim Verification - Orbital Distance Norm, Inner Planets

As can be seen in Figure 26, the simulation data for the inner planets perfectly matches the HORIZONS data for 400 time steps, or 2 years. While this is not a significant amount of time, especially for Mars, we can see that Mercury stays on par with HORIZONS data for 8 periods, which is very accurate. There does not appear to be any phase misalignment or magnitude discrepancy over this time period.

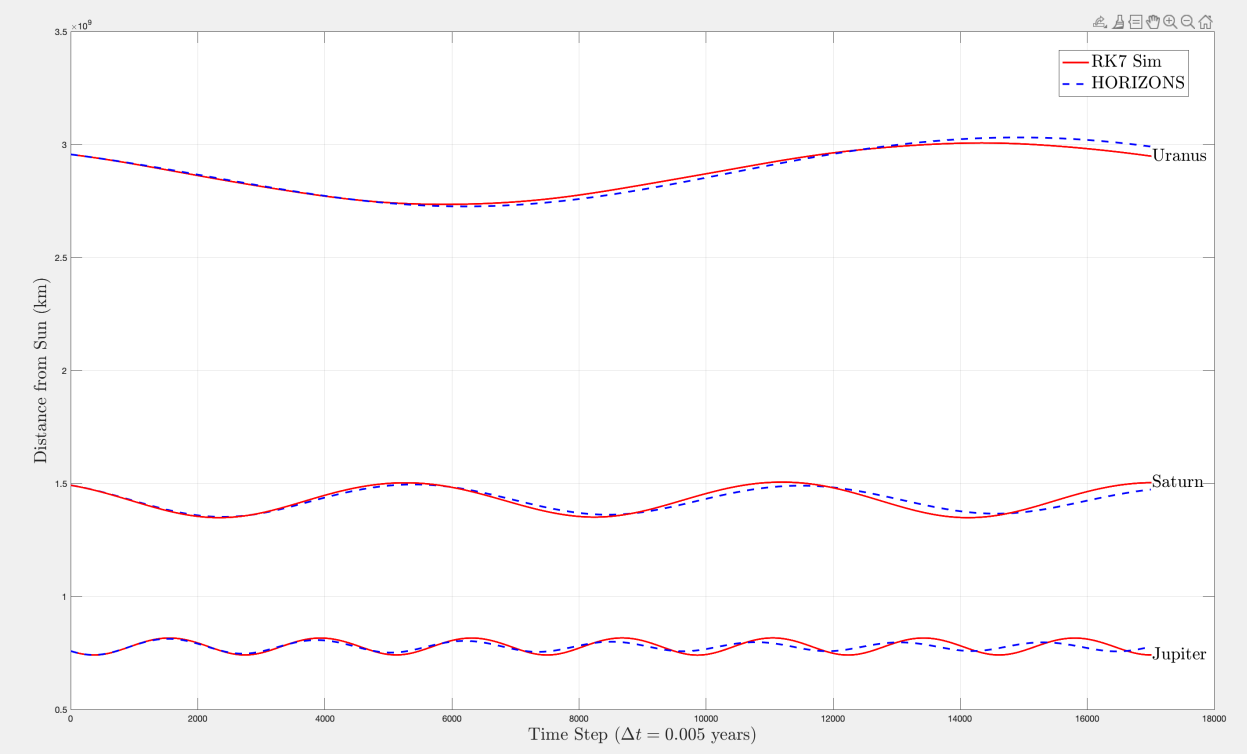

Figure 27: Sim Verification - Orbital Distance Norm, Outer Planets

As can be seen in Figure 27, the simulation data for the outer planets does not fare quite as well when compared to HORIZONS data. The simulation data seems to be slightly shortening the orbital period of Uranus and Saturn, and the distance between these bodies and the Sun appears to decrease slightly over time. On the other hand, the simulation data seems to be elongating the orbital period of Jupiter, while also increasing the distance between Jupiter and the sun over time. By the end of the simulation (275 years), Jupiter is almost 180° phase shifted compared to the HORIZONS data, which would place it on the opposite side of its normal orbit.

While this is not a great sign for the accuracy of our simulation, it is expected that our simulation will not perfectly model the dynamics of all planets. One possible theory for why the inner planets fared a lot better than the outer planets is because of simulation run time. The inner planets only being run for 2 years as opposed to 275 years may not have allowed enough time for errors to accumulate. On the other hand, the number of orbits for the inner bodies vs the outer bodies is similar, and it would also be expected that error creeps in on the scale of periods not years.

4.4 Black Hole Sim

What if the Sun turned into a black hole with mass 5× what it is now? What would happen to the planets and their orbits? We used the sim to propagate the inner planets two years and the outer planets 275 years. All planets were used in both runs of the simulation, but for plot clarity, one group of planets was plotted without the other. All the planets had the same initial conditions $(\mathbf{r}, \mathbf{v})$ used in the normal simulations.

The inner planets, as shown in Fig. 28, faced some difficulty. Mercury, Venus, and Earth were pulled into the Black Hole directly and appeared to be pushed away, possibly on non-orbiting trajectories. This is likely due to the immense force generated as the distance between these bodies and the black hole approach zero.

The magnitude of the distance being in the denominator term, it causes a spike in acceleration at one time step, and it is likely that by the next time step the position of the body has been calculated to be far on the other side of the black hole, traveling too fast to be recaptured by the gravity of the black hole. This could be made more realistic by decreasing Δt, as it would smooth out the jaggedness of these ejected planets, and perhaps at a small enough Δt, the body would not teleport vast distances through the black hole and not be ejected. Another potential way to combat this curious dynamic of our simulation as distances between objects get very small is the introduction of a softening term. This softening term would be summed into the denominator magnitude of distance between objects, to "soften" the great forces observed at these distances. Introduction of a softening term would change the magnitude of forces between all bodies, so it would add some inaccuracies for other bodies behaving relatively well, and so this softening term needs to be small enough as to not cause large inaccuracies, yet large enough to make the close body dynamics more realistic. While softening has been implemented in the code, we were not able to explore the subject, as we had more essential aspects of the paper to clean up and finish. Mars entered a decreasing spiral orbital path and made approximately 11 orbits in the same time span it usually takes to make one.

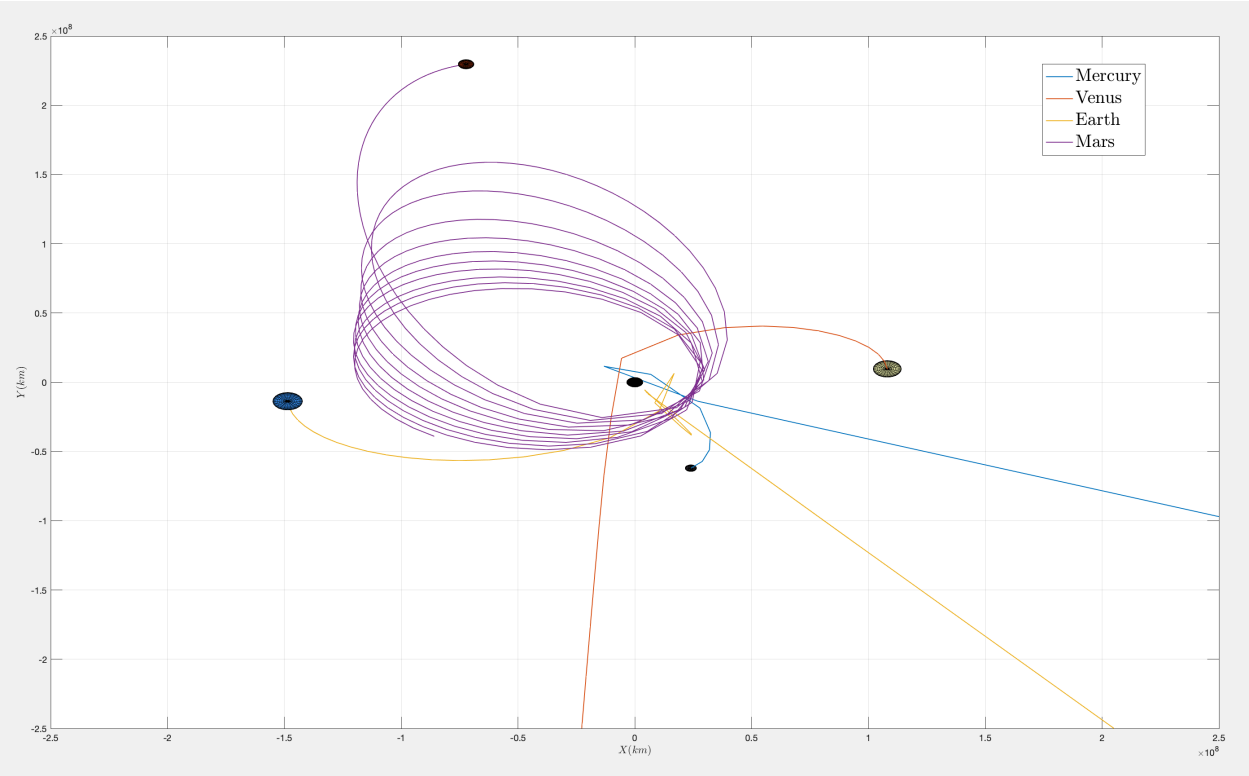

Figure 28: Black Hole Sim with Inner Planets

The outer planets, Fig. 29, fared better than the inner planets. All outer planets maintained orbits around the black hole but the frequency and speed of the orbits increased. Curiously, or maybe not, the black hole definitely aligned all the remaining orbiting planets into one plane...even Pluto. While we have no real world data or HORIZONS data to compare our results to, based on our testing of our simulation with the real world solar system, we can be confident that these results for the well behaving bodies (all bodies but Mercury, Venus and Earth) at least capture the general dynamics of if the sun suddenly increased mass 5 times. Relevant code for all plots in Results can be found in Code Appendix B.3.

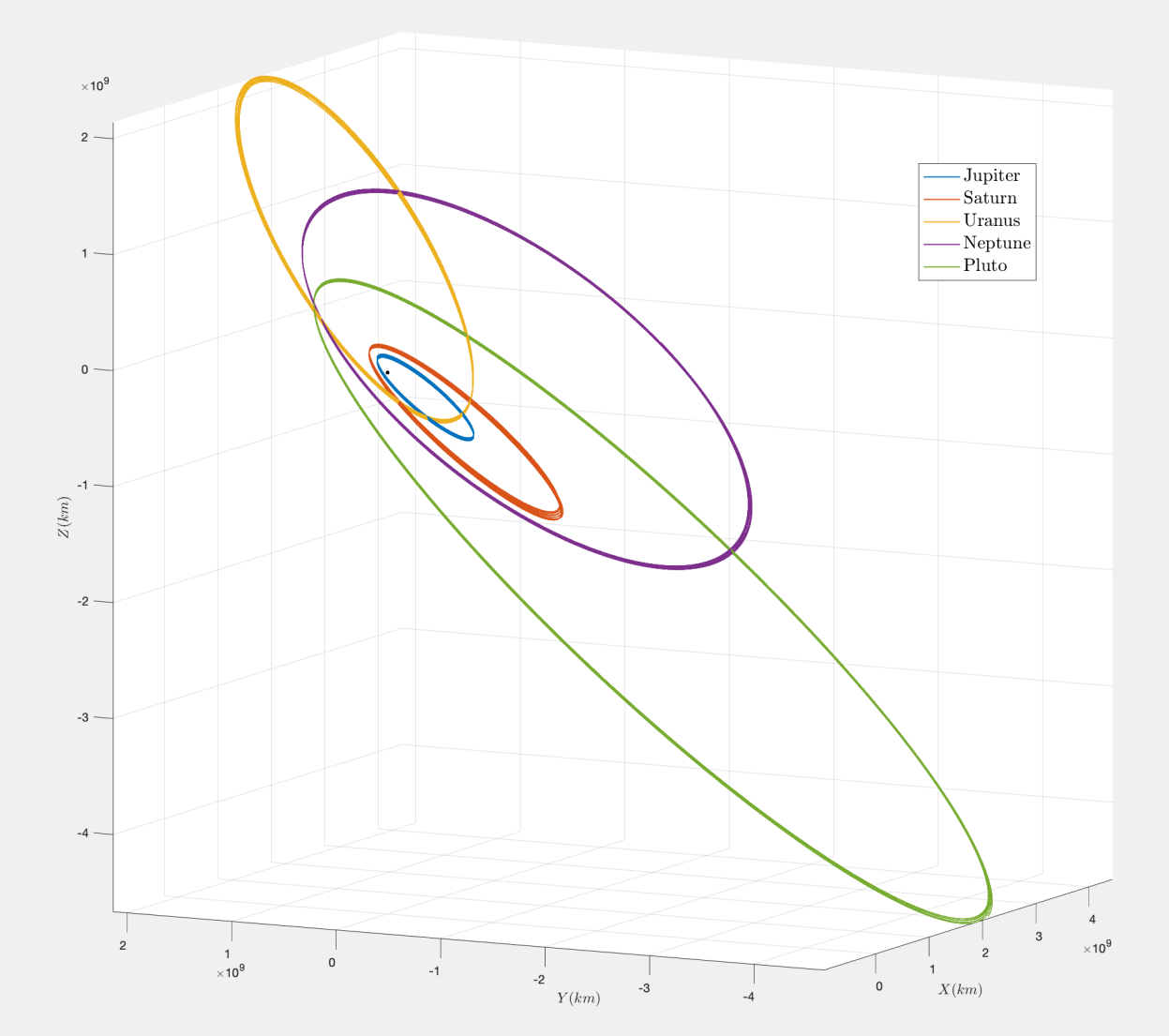

Figure 29: Black Hole Sim with Outer Planets

5 Conclusion

Using our understanding of the solar system, we were able to effectively and accurately model the n-body problem specific to the solar system. We justified the method we used, both theoretically by investigating truncation error and stability, and experimentally by investigating simulation results compared to expected results. Expected results came in the form of comparing convergence error, visually observing the stability of the simulation positional data, comparing initial and final total energy, and comparing simulation positional data to HORIZONS predictions. We finally settled on RK7 with $\Delta t = 5 \times 10^{-3}$, which seemed to give a compromise of accuracy and run time. When doing trial and error, simulation time really does add up, and in retrospect, it would have been unrealistic to choose $\Delta t = 2.5 \times 10^{-3}$, as all simulations would have taken twice as long. That being said, choosing a smaller $\Delta t = 2.5 \times 10^{-3}$ may have helped solve some of the issues we encountered with our simulation over time. For the regular solar system, a smaller time step may have allowed our simulation to stay on course with HORIZONS data longer before diverging. For the black hole simulation, a smaller time step may have given more realistic motions for Mercury, Venus and Earth, which

were violently flung out of orbit. Overall however, we are very happy with our project as a rudimentary model of our solar system.

A Planetary Initial Conditions for Sim

Retrieved from NASA/JPL HORIZONS Web-Interface
Data is for 2021-Mar-26 at 00:00:00.0000.

$$Bodies = \begin{bmatrix} Sun \\ Mercury \\ Venus \\ Earth \\ Mars \\ Jupiter \\ Saturn \\ Uranus \\ Neptune \\ Pluto \end{bmatrix} \qquad mass_{bodies} = \begin{bmatrix} 1.989 \times 10^{30} \\ 3.302 \times 10^{23} \\ 4.8685 \times 10^{24} \\ 5.972\,19 \times 10^{24} \\ 6.4171 \times 10^{23} \\ 1.898\,187 \times 10^{27} \\ 5.6834 \times 10^{26} \\ 8.6813 \times 10^{25} \\ 1.024\,126 \times 10^{26} \\ 1.307 \times 10^{22} \end{bmatrix}$$

mass units kg
$\vec{r}$ units km
$\vec{v}$ units km/s

$$\vec{r}_x = \begin{bmatrix} 0 \\ 2.397\,512\,607\,514\,394 \times 10^{7} \\ 1.080\,429\,709\,155\,349 \times 10^{8} \\ -1.485\,874\,787\,293\,754 \times 10^{8} \\ -7.222\,548\,929\,681\,751 \times 10^{7} \\ 5.262\,100\,914\,609\,202 \times 10^{8} \\ 8.752\,039\,476\,003\,931 \times 10^{8} \\ 2.264\,574\,284\,515\,496 \times 10^{9} \\ 4.413\,771\,818\,953\,279 \times 10^{9} \\ 2.140\,348\,545\,900\,892 \times 10^{9} \end{bmatrix} \quad \vec{r}_y = \begin{bmatrix} 0 \\ -6.201\,865\,125\,794\,719 \times 10^{7} \\ 9.656\,721\,872\,978\,827 \times 10^{6} \\ -1.369\,700\,631\,085\,473 \times 10^{7} \\ 2.296\,141\,187\,698\,652 \times 10^{8} \\ -5.457\,557\,836\,218\,837 \times 10^{8} \\ -1.208\,229\,829\,565\,184 \times 10^{9} \\ 1.900\,107\,166\,695\,706 \times 10^{9} \\ -7.434\,876\,157\,131\,478 \times 10^{8} \\ -4.653\,259\,655\,247\,819 \times 10^{9} \end{bmatrix} \quad \vec{r}_z = \begin{bmatrix} 0 \\ -7.267\,199\,604\,103\,021 \times 10^{6} \\ -6.102\,135\,646\,085\,277 \times 10^{6} \\ 6.972\,378\,362\,230\,957 \times 10^{2} \\ 6.583\,525\,826\,588\,318 \times 10^{6} \\ -9.506\,356\,235\,333\,622 \times 10^{6} \\ -1.382\,578\,451\,909\,006 \times 10^{7} \\ -2.228\,222\,756\,247\,282 \times 10^{7} \\ -8.641\,964\,456\,127\,855 \times 10^{7} \\ -1.209\,828\,896\,350\,319 \times 10^{8} \end{bmatrix}$$

$$\vec{v}_x = \begin{bmatrix} 0 \\ 3.567\,792\,640\,575\,497 \times 10^{1} \\ -3.257\,141\,829\,462\,026 \\ 2.254\,339\,636\,701\,395 \\ -2.219\,702\,361\,854\,693 \times 10^{1} \\ 9.259\,420\,222\,709\,933 \\ 7.294\,581\,138\,660\,418 \\ -4.418\,847\,631\,137\,454 \\ 8.774\,194\,505\,964\,793 \times 10^{-1} \\ 5.088\,364\,631\,334\,085 \end{bmatrix} \quad \vec{v}_y = \begin{bmatrix} 0 \\ 2.002\,238\,928\,246\,063 \times 10^{1} \\ 3.472\,097\,248\,658\,434 \times 10^{1} \\ -2.976\,371\,417\,125\,200 \times 10^{1} \\ -5.212\,936\,215\,527\,516 \\ 9.695\,513\,443\,591\,937 \\ 5.653\,516\,068\,318\,001 \\ 4.908\,981\,327\,356\,346 \\ 5.404\,189\,815\,072\,675 \\ 1.112\,702\,387\,534\,124 \end{bmatrix} \quad \vec{v}_z = \begin{bmatrix} 0 \\ -1.636\,595\,698\,116\,885 \\ 6.644\,540\,807\,741\,954 \times 10^{-1} \\ 1.173\,751\,460\,282\,091 \times 10^{-3} \\ 4.352\,643\,730\,629\,382 \times 10^{-1} \\ -2.473\,639\,086\,097\,603 \times 10^{-1} \\ -3.886\,511\,261\,802\,323 \times 10^{-1} \\ 7.522\,607\,324\,237\,862 \times 10^{-2} \\ -1.312\,085\,852\,380\,689 \times 10^{-1} \\ -1.605\,457\,519\,092\,396 \end{bmatrix}$$

B Code Appendix

B.1 Code for Simulation (Python)

```python
import numpy as np
import matplotlib.pyplot as plt
import time
import csv

class Body():
    def __init__(self, name, mass, pos0, vel0):
        self.name = name
        self.mass = mass

        #Keep track of initial conditions for multiple runs
        self.pos0 = pos0
        self.vel0 = vel0

        #Updateable position and velocity
        self.pos = pos0
        self.vel = vel0


class Simulation():
    def __init__(self, G, softening, bodies):
        self.bodies = bodies
        self.G = G
        self.softening = softening
        self.energy = 0
        self.update_energy()


    def update_energy(self):
        #Sum up kinetic energy
        ke = sum([0.5*body.mass*(np.linalg.norm(body.vel)**2) for body in self.bodies])

        #Initialize potential energy
        pe = 0
        for i in range(len(self.bodies)):
            m_i = self.bodies[i].mass
            r_i = self.bodies[i].pos
            #range(i+1, len(self.bodies) makes sure not to double count
            for j in range(i+1, len(self.bodies)):
                m_j = self.bodies[j].mass
                r_j = self.bodies[j].pos

                #Sum potential energy from each other body other than body_i
                pe = pe + (-self.G * m_i * m_j) / (np.linalg.norm(r_j - r_i))
        #Updates class variable
        self.energy = ke + pe


    def get_u_k(self):
        #Initialize uy_k of size (N,2,3)
        u_k = np.ndarray((len(self.bodies), 2, 3))
        for i,body in enumerate(self.bodies):
            r = body.pos
            v = body.vel
```

```
55
56              #Fill u_k as uk[body_i] = [body_i_pos, body_i_vel]
57              u_k[i] = np.array([r, v])
58
59          return u_k
60
61      #du/dt = f(u)
62      def func(self, u_k):
63          #Initialize solution array of size (N,2,3)
64          sol = np.ndarray((len(self.bodies), 2, 3))
65          for i in range(len(self.bodies)):
66              r_i = u_k[i][0]
67              v = u_k[i][1]
68              a = np.zeros(3)
69              for j,body in enumerate(self.bodies):
70                  if i != j:
71                      m_j = body.mass
72                      r_j = u_k[j][0]
73
74                      #Sum acceleration from each other body
75                      a = a + ((self.G * m_j * (r_j - r_i)) / ((np.linalg.norm(r_j - r_i)**2 + self.
    softening**2)**(3/2)))
76
77              #Fill sol as sol[body_i] = [body_i_vel, body_i_acc]
78              sol[i] = np.array([v, a])
79
80          return sol
81
82
83      def rk2(self, dt, u_k):
84          u_kp1 = u_k + 0.5*dt*(self.func(u_k) + self.func(u_k + dt*self.func(u_k)))
85          return u_kp1
86
87
88      def rk4(self, dt, u_k):
89          y1 = self.func(u_k)
90          y2 = self.func(u_k + 0.5*dt*y1)
91          y3 = self.func(u_k + 0.5*dt*y2)
92          y4 = self.func(u_k + dt*y3)
93
94          u_kp1 = u_k + (1/6)*dt*(y1+ 2*y2 + 2*y3 + y4)
95          return u_kp1
96
97
98      def rk7(self, dt, u_k):
99          k1 = dt*self.func(u_k)
100         k2 = dt*self.func(u_k + k1)
101         k3 = dt*self.func(u_k + (3*k1 + k2)/8)
102         k4 = dt*self.func(u_k + (8*k1 + 2*k2 + 8*k3)/27)
103         k5 = dt*self.func(u_k + (3*(3*np.sqrt(21) - 7)*k1 - 8*(7 - np.sqrt(21))*k2
104                                + 48*(7 - np.sqrt(21))*k3 - 3*(21 - np.sqrt(21))*k4)/392)
105         k6 = dt*self.func(u_k + (-5*(231 + 51*np.sqrt(21))*k1 - 40*(7 + np.sqrt(21))*k2
106                                - 320*(np.sqrt(21))*k3 + 3*(21+121*np.sqrt(21))*k4
107                                + 392*(6+np.sqrt(21))*k5)/1960)
108         k7 = dt*self.func(u_k + (15*(22 + 7*np.sqrt(21))*k1 + 120*k2
109                                + 40*(7*np.sqrt(21) - 5)*k3 - 63*(3*np.sqrt(21) - 2)*k4
110                                - 14*(49 + 9*np.sqrt(21))*k5 + 70*(7 - np.sqrt(21))*k6)/180)
111
112         u_kp1 = u_k + (9*k1 + 64*k3 + 49*k5 + 49*k6 + 9*k7)/180
```

```
        return u_kp1


    def plot_once(self, dt, history, time_array, energy_history, method):
        fig1 = plt.figure()
        ax1 = fig1.add_subplot(111, projection='3d')

        # Plot the values
        for body in self.bodies:
            ax1.scatter(history[body.name][0], history[body.name][1], history[body.name][2],marker='.'
    ,s=1,label=body.name)
        plt.legend(bbox_to_anchor=(1.0,1),loc='upper left',borderaxespad=0.)
        ax1.set_zlim(-1e9, 1e9)
        ax1.set_xlabel('x (m)')
        ax1.set_ylabel('Y (m)')
        ax1.set_zlabel('Z (m)')

        #Convert back to years
        dt = dt/(365*24*60*60)
        time_array = time_array/(365*24*60*60)

        #Save sim position plot
        fig1.savefig('sim_' + method.lower() + '_' + str(format(dt, '1.1e')) + '.png',bbox_inches='
    tight', dpi=300)

        plt.figure(figsize=(16,8))
        plt.plot(time_array, np.abs(energy_history))
        plt.title('Total Energy vs Time for ' + method + ' and dt=' + str(format(dt, '1.1e')) + '
    years')
        plt.xlabel('Time (years)')
        plt.ylabel('||Total Energy|| (kJ)')
        plt.yscale('log')

        #Save energy plot
        plt.savefig('energy_' + method.lower() + '_' + str(format(dt, '1.1e')) + '.png')

        #Calculate and print relevant energy data
        print("Initial Energy: ", np.abs(energy_history[0]))
        print("Change in Energy: ", np.abs(energy_history[-1] - energy_history[0]))
        print("Percent change in Energy: ", np.abs((energy_history[-1] - energy_history[0])/
    energy_history[0]))


    def run(self, dt, T, method):
        t1 = time.time()

        #Initialize dictionary with key=body name and positional history as value
        history = {}
        for body in self.bodies:
            #[ [xpos], [ypos], [zpos] ]
            history[body.name] = [[body.pos[0]],[body.pos[1]],[body.pos[2]]]

        #Initialize total energy history to later plot
        energy_history = [self.energy]

        nsteps = int(T/dt)

        #Initialize time_array for plotting energy
        time_array = np.zeros(nsteps+1)
```

```
168
169         for step in range(nsteps):
170             #Fill time_array for plotting energy
171             time_array[step+1] = step*dt
172
173             u_k = self.get_u_k()
174
175             #Choose method based on user input in run()
176             if method == 'RK2':
177                 u_kp1 = self.rk2(dt, u_k)
178
179             elif method == 'RK4':
180                 u_kp1 = self.rk4(dt, u_k)
181
182             else:
183                 u_kp1 = self.rk7(dt, u_k)
184
185             #Update states
186             for i,body in enumerate(self.bodies):
187                 pos = u_kp1[i][0]
188                 vel = u_kp1[i][1]
189
190                 body.pos = pos
191                 body.vel = vel
192
193                 #Append x, y, z pos to history
194                 history[body.name][0].append(pos[0])
195                 history[body.name][1].append(pos[1])
196                 history[body.name][2].append(pos[2])
197
198             #Update total energy
199             self.update_energy()
200             energy_history.append(self.energy)
201
202         #Print runtime
203         t2 = time.time()
204         print('Time taken: ', t2-t1)
205
206         #Plot one dt
207         self.plot_once(dt, history, time_array, energy_history, method)
208
209         #Export data to MATLAB
210         '''
211         positRow = len(history['Earth'][0])
212         #print(positRow)
213
214         positionData = np.zeros((positRow,30))
215
216         positionData[:,0] = history['Sun'][0]
217         positionData[:,1] = history['Sun'][1]
218         positionData[:,2] = history['Sun'][2]
219
220         positionData[:,3] = history['Mercury'][0]
221         positionData[:,4] = history['Mercury'][1]
222         positionData[:,5] = history['Mercury'][2]
223
224         positionData[:,6] = history['Venus'][0]
225         positionData[:,7] = history['Venus'][1]
226         positionData[:,8] = history['Venus'][2]
```

```
227
228         positionData[:,9] = history['Earth'][0]
229         positionData[:,10] = history['Earth'][1]
230         positionData[:,11] = history['Earth'][2]
231
232         positionData[:,12] = history['Mars'][0]
233         positionData[:,13] = history['Mars'][1]
234         positionData[:,14] = history['Mars'][2]
235
236         positionData[:,15] = history['Jupiter'][0]
237         positionData[:,16] = history['Jupiter'][1]
238         positionData[:,17] = history['Jupiter'][2]
239
240         positionData[:,18] = history['Saturn'][0]
241         positionData[:,19] = history['Saturn'][1]
242         positionData[:,20] = history['Saturn'][2]
243
244         positionData[:,21] = history['Uranus'][0]
245         positionData[:,22] = history['Uranus'][1]
246         positionData[:,23] = history['Uranus'][2]
247
248         positionData[:,24] = history['Neptune'][0]
249         positionData[:,25] = history['Neptune'][1]
250         positionData[:,26] = history['Neptune'][2]
251
252         positionData[:,27] = history['Pluto'][0]
253         positionData[:,28] = history['Pluto'][1]
254         positionData[:,29] = history['Pluto'][2]
255
256         #print(positionData)
257
258         # name of csv file
259         filename = "sim_rk2_1.5e-2.csv"
260
261         # writing to csv file
262         with open(filename, 'w') as csvfile:
263             # creating a csv writer object
264             csvwriter = csv.writer(csvfile)
265
266             # writing the data rows
267             csvwriter.writerows(positionData)
268
269         '''
270
271
272     def run_convergence(self, dtvect, T):
273         t1 = time.time()
274         ndt = len(dtvect)
275
276         #Initialize array to store final solutions for each method/dt
277         u_rk2_keep = np.ndarray((ndt, len(self.bodies), 2, 3))
278         u_rk4_keep = np.ndarray((ndt, len(self.bodies), 2, 3))
279         u_rk7_keep = np.ndarray((ndt, len(self.bodies), 2, 3))
280
281         for j,dt in enumerate(dtvect):
282             ta = time.time()
283             nsteps = int(T/dt)
284
285             #RK2
```

```
            for step in range(nsteps):
                u_k = self.get_u_k()
                u_kp1 = self.rk2(dt, u_k)

                #Update states
                for i,body in enumerate(self.bodies):
                    pos = u_kp1[i][0]
                    vel = u_kp1[i][1]

                    body.pos = pos
                    body.vel = vel

            #Store final u_k for error comparison
            u_rk2_keep[j,:] = self.get_u_k()

            #Reset initial conditions for next method
            for body in self.bodies:
                body.pos = body.pos0
                body.vel = body.vel0

            #RK4
            for step in range(nsteps):
                u_k = self.get_u_k()
                u_kp1 = self.rk4(dt, u_k)

                #Update states
                for i,body in enumerate(self.bodies):
                    pos = u_kp1[i][0]
                    vel = u_kp1[i][1]

                    body.pos = pos
                    body.vel = vel

            #Store final u_k for error comparison
            u_rk4_keep[j,:] = self.get_u_k()

            #Reset initial conditions for next method
            for body in self.bodies:
                body.pos = body.pos0
                body.vel = body.vel0

            #RK7
            for step in range(nsteps):
                u_k = self.get_u_k()
                u_kp1 = self.rk7(dt, u_k)

                #Update states
                for i,body in enumerate(self.bodies):
                    pos = u_kp1[i][0]
                    vel = u_kp1[i][1]

                    body.pos = pos
                    body.vel = vel

            #Store final u_k for error comparison
            u_rk7_keep[j,:] = self.get_u_k()

            #Reset initial conditions for next dt
            for body in self.bodies:
```

```
                body.pos = body.pos0
                body.vel = body.vel0

            #Time for each dt
            tb = time.time()
            print('Time taken: ', tb-ta)

        #Total time for convergence plot
        t2 = time.time()
        print('Total time: ', t2-t1)

        #Initialize error vectors
        u_rk2_diff = np.zeros(ndt - 1)
        u_rk4_diff = np.zeros(ndt - 1)
        u_rk7_diff = np.zeros(ndt - 1)

        for j in range(ndt - 1):
            u_rk2_diff[j]= np.linalg.norm(u_rk2_keep[j,:] - u_rk2_keep[ndt - 1,:])/np.linalg.norm(
    u_rk2_keep[ndt - 1,:])
            u_rk4_diff[j]= np.linalg.norm(u_rk4_keep[j,:] - u_rk4_keep[ndt - 1,:])/np.linalg.norm(
    u_rk4_keep[ndt - 1,:])
            u_rk7_diff[j]= np.linalg.norm(u_rk7_keep[j,:] - u_rk7_keep[ndt - 1,:])/np.linalg.norm(
    u_rk7_keep[ndt - 1,:])

        #Convert dtvect into years
        dtvect = dtvect/(365*24*60*60)

        width = 10
        height = 1.1*width
        plt.figure(figsize=(width,height))
        plt.loglog(dtvect[:-1], u_rk2_diff, linestyle='-', marker='.', markersize=20, label='RK2')
        plt.loglog(dtvect[:-1], u_rk4_diff, linestyle='-', marker='.', markersize=20, label='RK4')
        plt.loglog(dtvect[:-1], u_rk7_diff, linestyle='-', marker='.', markersize=20, label='RK7')
        plt.axhline(y=1e-1, color='black')
        plt.legend(fontsize='18')
        plt.title(r'$\frac{||u_{\Delta t}-u_{1\times10^{-3}}||}{u_{1\times10^{-3}}}$ (years)', pad=10)
        plt.xlabel('$\Delta t$ (years)')
        plt.ylabel('error')
        plt.xscale('linear')
        plt.tick_params(axis="both",direction="in")
        plt.tick_params(right=True, top=True)
        plt.savefig('error.png', bbox_inches='tight', dpi=100)
        plt.clf()


def main():
    # G using km
    G = 6.67e-20
    softening = 0


    # Ephemeris of the planets (x,y,z) (vx,vy,vz)
    # (r,v) in (km,km/s)
    # NASA/JPL Horizons 2021-MAR-26 00:00.00
    x = np.array([0, 2.397512607514394E+07,1.080429709155349E+08, -1.485874787293754E+08,
            -7.222548929681751E+07, 5.262100914609202E+08, 8.752039476003931E+08,
            2.264574284515496E+09, 4.413771818953279E+09, 2.140348545900892E+09])

    y = np.array([0, -6.201865125794719E+07, 9.656721872978827E+06,  -1.369700631085473E+07,
```

```
            2.296141187698652E+08, -5.457557836218837E+08, -1.208229829565184E+09,
            1.900107166695706E+09, -7.434876157131478E+08, -4.653259655247819E+09])

    z = np.array([0, -7.267199604103021E+06, -6.102135646085277E+06, 6.972378362230957E+02,
            6.583525826588318E+06, -9.506356235333622E+06, -1.382578451909006E+07,
            -2.228222756247282E+07, -8.641964456127855E+07, -1.209828896350319E+08])

    vx = np.array([0, 3.567792640575497E+01, -3.257141829462026E+00, 2.254339636701395E+00,
            -2.219702361854693E+01, 9.259420222709933E+00, 7.294581138660418E+00,
            -4.418847631137454E+00, 8.774194505964793E-01, 5.088364631334085E+00])

    vy = np.array([0, 2.002238928246063E+01, 3.472097248658434E+01, -2.976371417125200E+01,
            -5.212936215527516E+00, 9.695513443591937E+00, 5.653516068318001E+00,
            4.908981327356346E+00, 5.404189815072675E+00, 1.112702387534124E+00])

    vz = np.array([0, -1.636595698116885E+00, 6.644540807741954E-01, 1.173751460282091E-03,
            4.352643730629382E-01, -2.473639086097603E-01, -3.886511261802323E-01,
            7.522607324237862E-02, -1.312085852380689E-01, -1.605457519092396E+00])

    #Sun
    Msun = 1.989e30
    PosSun = np.array([x[0],y[0],z[0]])
    VelSun = np.array([vx[0],vy[0],vz[0]])

    #Mercury
    Mmercury = .3302e24
    PosMercury = np.array([x[1],y[1],z[1]])
    VelMercury = np.array([vx[1],vy[1],vz[1]])

    #Venus
    Mvenus = 4.8685e24
    PosVenus = np.array([x[2],y[2],z[2]])
    VelVenus = np.array([vx[2],vy[2],vz[2]])

    #Earth
    Mearth = 5.97219e24
    PosEarth = np.array([x[3],y[3],z[3]])
    VelEarth = np.array([vx[3],vy[3],vz[3]])

    #Mars
    Mmars = .64171e24
    PosMars = np.array([x[4],y[4],z[4]])
    VelMars = np.array([vx[4],vy[4],vz[4]])

    #Jupiter
    Mjupiter = 1898.187e24
    PosJupiter = np.array([x[5],y[5],z[5]])
    VelJupiter = np.array([vx[5],vy[5],vz[5]])

    #Saturn
    Msaturn = 568.34e24
    PosSaturn = np.array([x[6],y[6],z[6]])
    VelSaturn = np.array([vx[6],vy[6],vz[6]])

    #Uranus
    Muranus = 86.813e24
    PosUranus = np.array([x[7],y[7],z[7]])
    VelUranus = np.array([vx[7],vy[7],vz[7]])

```

```
    #Neptune
    Mneptune = 102.4126e24
    PosNeptune = np.array([x[8],y[8],z[8]])
    VelNeptune = np.array([vx[8],vy[8],vz[8]])

    #Pluto
    Mpluto = 0.01307e24
    PosPluto = np.array([x[9],y[9],z[9]])
    VelPluto = np.array([vx[9],vy[9],vz[9]])

    #Initialize bodies
    Sun     = Body('Sun', Msun, PosSun, VelSun)
    Mercury = Body('Mercury', Mmercury, PosMercury, VelMercury)
    Venus   = Body('Venus', Mvenus, PosVenus, VelVenus)
    Earth   = Body('Earth', Mearth, PosEarth, VelEarth)
    Mars    = Body('Mars', Mmars, PosMars, VelMars)
    Jupiter = Body('Jupiter', Mjupiter, PosJupiter, VelJupiter)
    Saturn  = Body('Saturn', Msaturn, PosSaturn, VelSaturn)
    Uranus  = Body('Uranus', Muranus, PosUranus, VelUranus)
    Neptune = Body('Neptune', Mneptune, PosNeptune, VelNeptune)
    Pluto   = Body('Pluto', Mpluto, PosPluto, VelPluto)

    #Input bodies into sim
    simulation = Simulation(G, softening, [Sun, Mercury, Venus, Earth, Mars])
    # , Jupiter, Saturn, Uranus, Neptune, Pluto
    # pluto takes approx 248 years for an orbit
    year = 365*24*60*60

    #Run once
    method = 'RK7'
    time = 50*year
    interval = 5e-3*year
    simulation.run(interval, time, method)

    '''
    #Run multiple times for various dt for each method for convergence
    dtvect = np.array([1.5e-2*year, 1.25e-2*year, 1e-2*year, 7.5e-3*year, 5e-3*year, 2.5e-3*year, 1e
    -3*year])
    time_convergence = 50*year
    simulation.run_convergence(dtvect, time_convergence)
    '''

if __name__ == "__main__":
    main()
```

B.2 Code for Stability Plots (Python)

```
    %MatLab code to generate stability plots
    %We Acknowledge that the structure for the code generating these plots
    %comes from an in lecture example-- Week7 Wednesday
    clear all
    %define real and imaginary parts of w
    wr = -4 : 0.01 : 3;
    wi = -4 : 0.01 : 4 ;
    %create meshgrid
    [Wr, Wi] = meshgrid( wr, wi );

```

```
    %BE for forward EUler
    %BE_sc = abs((1 + (Wr + 1i*Wi)) );

    dt = Wr + 1i*Wi;
    %define |R(w)| for BE

    % 2nd order Runge-Kutta growth factor
    BE_sc = abs(1 + dt + 0.5*dt.^2);

    %RK4 region
    % BE_sc = abs( ( 1 + dt + .5*(dt.^2) + (dt.^3)/6 + (dt.^4)/24) );



    %saturate values for easier coloring
    BE_sc( BE_sc < 1 ) = 1;
    BE_sc( BE_sc > 1 ) = 2;
    %UIUC-themed colormap
    cmap = [1 1/2 0; 1 1 1];
    %Create a contour plot of the stability region
    contourf( Wr, Wi, BE_sc, [1 2])
    grid on
    colormap( cmap ), axis equal
```

B.3 Code for Simulation Runs (Matlab)

```
clc
clear all
close all

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%   AE370 Project 1
%   This script imports the sim data and plots it over a
%   275 year period (2021 - 2296) for all the planets.
%   The sim used RK7 at delta t = 0.005 yrs
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% load SIM data


% Plot the paths of all the planets
figure(1)
for j=1:9

    h(j)=plot3(y(:,1+j*3)-y(:,1),y(:,2+j*3)-y(:,2),y(:,3+j*3)-y(:,3),...
        'linewidth',1);hold on;
end

[xs,ys,zs] = sphere;
sun = surf(xs*2*695700e1,ys*2*695700e1,zs*2*695700e1);
set(sun,'FaceColor',[1 1 0])

[xm,ym,zm] = sphere;
mercury = surf(xm*2440e3+y(1,4),ym*2440e3+y(1,5),zm*2440e3+y(1,6));
set(mercury,'FaceColor',[1 1 0])

[xv,yv,zv] = sphere;
venus = surf(xv*6051e3+y(1,7),yv*6051e3+y(1,8),zv*6051e3+y(1,9));
set(venus,'FaceColor',[0 1 0])

[xe,ye,ze] = sphere;
earth = surf(xe*6378e3+y(1,10),ye*6378e3+y(1,11),ze*6378e3+y(1,12));
set(earth,'FaceColor',[0.1 0.5 1])

[xma,yma,zma] = sphere;
mars = surf(xma*3396e3+y(1,13),yma*3396e3+y(1,14),zma*3396e3+y(1,15));
set(mars,'FaceColor',[1 0 0])

[xj,yj,zj] = sphere;
jupiter = surf(xj*1.5*71492e3+y(1,16),yj*1.5*71492e3+y(1,17),zj*1.5*71492
    e3+y(1,18));
set(jupiter,'FaceColor',[0.9961 0.5 0])

[xsat,ysat,zsat] = sphere;
saturn = surf(xsat*1.5*60268e3+y(1,19),ysat*1.5*60268e3+y(1,20),zsat
    *1.5*60268e3+y(1,21));
```

```
set(saturn,'FaceColor',[1 1 0])

[xu,yu,zu] = sphere;
uranus = surf(xu*2*25559e3+y(1,22),yu*2*25559e3+y(1,23),zu*2*25559e3+y
    (1,24));
set(uranus,'FaceColor',[0 1 0])

[xn,yn,zn] = sphere;
neptune = surf(xn*2*24764e3+y(1,25),yn*2*24764e3+y(1,26),zn*2*24764e3+y
    (1,27));
set(neptune,'FaceColor',[0.1 0.5 1])

[xp,yp,zp] = sphere;
pluto = surf(xp*5*6378e3+y(1,28),yp*5*6378e3+y(1,29),zp*5*6378e3+y(1,30));
set(pluto,'FaceColor',[0.1 0.5 1])

lgd = legend('','','','','Jupiter','Saturn','Uranus','Neptune','Pluto',...
            'Location','northwestoutside','interpreter', 'latex','fontsize
    ', 20);
%lgd.Title.String = 'Outer Planets';
axis equal;
grid on;
xlabel('$X (km)$','interpreter', 'latex','fontsize', 12)
ylabel('$Y (km)$','interpreter', 'latex','fontsize', 12)
zlabel('$Z (km)$','interpreter', 'latex','fontsize', 12)
```

```
clc
clear all
close all

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%   AE370 Project 1
%   This script imports the sim data and plots it over a
%   2 year period (2021 - 2023) for the inner planets.
%   The sim used RK7 at delta t = 0.005 yrs
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% load SIM data
% y = readmatrix('sim_rk7_2.csv');

% Plot the paths of the inner planets
figure(1)
for j=1:4

    % SIM
    h(j)=plot3(y(:,1+j*3)-y(:,1),y(:,2+j*3)-y(:,2),y(:,3+j*3)-y(:,3),...
        'linewidth',2);hold on;
end

% make bodies
[xs,ys,zs] = sphere;
sun = surf(xs*3.5*695700e1,ys*3.5*695700e1,zs*3.5*695700e1);
```

```
set(sun,'FaceColor',[.98 .964 .219])

[xm,ym,zm] = sphere;
mercury = surf(xm*2*2440e3+y(1,4),ym*2*2440e3+y(1,5),...
zm*2*2440e3+y(1,6));
set(mercury,'FaceColor',[.686 .65 .768])

[xv,yv,zv] = sphere;
venus = surf(xv*1.5*6051e3+y(1,7),yv*1.5*6051e3+y(1,8),...
zv*1.5*6051e3+y(1,9));
set(venus,'FaceColor',[0.976 0.988 0.713])

[xe,ye,ze] = sphere;
earth = surf(xe*1.5*6378e3+y(1,10),ye*1.5*6378e3+y(1,11),...
ze*1.5*6378e3+y(1,12));
set(earth,'FaceColor',[.219 .584 .98])

[xma,yma,zma] = sphere;
mars = surf(xma*1.5*3396e3+y(1,13),yma*1.5*3396e3+y(1,14),...
zma*1.5*3396e3+y(1,15));
set(mars,'FaceColor',[.87 .262 .07])

% make legend
lgd = legend('Mercury','Venus','Earth','Mars','Location','best',...
    'interpreter', 'latex','fontsize', 20);

axis equal;
grid on;
%view(0,90)

% label bodies and axes
% text(.25e8,-.8e8,'Mercury','interpreter', 'latex','fontsize', 15);
% text(0.7e8,.2e8,'Venus','interpreter', 'latex','fontsize', 15);
% text(-1.78e8,0,'Earth','interpreter', 'latex','fontsize', 15);
% text(-.6e8,2.2e8,'Mars','interpreter', 'latex','fontsize', 15);
xlabel('$X (km)$','interpreter', 'latex','fontsize', 12)
ylabel('$Y (km)$','interpreter', 'latex','fontsize', 12)
zlabel('$Z (km)$','interpreter', 'latex','fontsize', 12)
```

```
clc
clear all
close all

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%   AE370 Project 1
%   This script imports the sim data and HORIZONS data
%   and plots them over a 2 year period (2023) for inner
%   planet orbit propagation verification.
%   The sim used RK7 at delta t = 0.005 yrs
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% load SIM data
```

```
y = readmatrix('sim_rk7_2.csv');

% load HORIZONS data for inner planets
r_mercury = readmatrix('h_2023_mercury.xlsx');
r_venus = readmatrix('h_2023_venus.xlsx');
r_earth = readmatrix('h_2023_earth.xlsx');
r_mars = readmatrix('h_2023_mars.xlsx');

% combine HORIZONS data
r_2023 = [r_mercury(:,3:5) r_venus(:,3:5) r_earth(:,3:5) r_mars(:,3:5)];

% Plot the paths of the planets
figure(1)
for j=1:4

    % SIM
    h(j)=plot3(y(:,1+j*3)-y(:,1),y(:,2+j*3)-y(:,2),y(:,3+j*3)-y(:,3),'b--',...
        'linewidth',2);hold on;

    % HORIZONS
    hh(j)=plot3(r_2023(:,j*3-2),r_2023(:,j*3-1),r_2023(:,j*3),'r-','linewidth',1)
end

% make bodies
[xs,ys,zs] = sphere;
sun = surf(xs*3.5*695700e1,ys*3.5*695700e1,zs*3.5*695700e1);
set(sun,'FaceColor',[.98 .964 .219])

[xm,ym,zm] = sphere;
mercury = surf(xm*2*2440e3+y(1,4),ym*2*2440e3+y(1,5),zm*2*2440e3+y(1,6));
set(mercury,'FaceColor',[.686 .65 .768])

[xv,yv,zv] = sphere;
venus = surf(xv*1.5*6051e3+y(1,7),yv*1.5*6051e3+y(1,8),zv*1.5*6051e3+y(1,9));
set(venus,'FaceColor',[0.976 0.988 0.713])

[xe,ye,ze] = sphere;
earth = surf(xe*1.5*6378e3+y(1,10),ye*1.5*6378e3+y(1,11),ze*1.5*6378e3+y(1,12));
set(earth,'FaceColor',[.219 .584 .98])

[xma,yma,zma] = sphere;
mars = surf(xma*1.5*3396e3+y(1,13),yma*1.5*3396e3+y(1,14),zma*1.5*3396e3+y(1,15));
set(mars,'FaceColor',[.87 .262 .07])

% make legend
lgd = legend('RK7 $\Delta$t = 0.005 yrs','JPL HORIZONS',...
    'Location','best','interpreter', 'latex','fontsize', 20);
lgd.Title.String = 'Orbit Propagation';
axis equal;
```

```
grid on;
view(0,90)

% label bodies and axes
text(.25e8,-.8e8,'Mercury','interpreter', 'latex','fontsize', 15);
text(0.7e8,.2e8,'Venus','interpreter', 'latex','fontsize', 15);
text(-1.78e8,0,'Earth','interpreter', 'latex','fontsize', 15);
text(-.6e8,2.2e8,'Mars','interpreter', 'latex','fontsize', 15);
xlabel('$X (km)$','interpreter', 'latex','fontsize', 12)
ylabel('$Y (km)$','interpreter', 'latex','fontsize', 12)
zlabel('$Z (km)$','interpreter', 'latex','fontsize', 12)
```

```
clc
clear all
close all

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%   AE370 Project 1
%   This script imports the sim data and HORIZONS data
%   and plots them over a 85 year period (2106) for Jupiter,
%   Saturn, and Uranus orbit propagation verification.
%   The sim used RK7 at delta t = 0.005 yrs.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% load SIM data
y = readmatrix('sim_rk7_85');

% load HORIZONS data for outer planets
r_jupiter = readmatrix('h_2106_jupiter.xlsx');
r_saturn = readmatrix('h_2106_saturn.xlsx');
r_uranus = readmatrix('h_2106_uranus.xlsx');

% combine HORIZONS data
r_2106 = [r_jupiter(:,3:5) r_saturn(:,3:5) r_uranus(:,3:5)];

% Plot the paths of the planets
figure(1)
for j=5:7

    % SIM
    h(j)=plot3(y(:,1+j*3)-y(:,1),y(:,2+j*3)-y(:,2),y(:,3+j*3)-y(:,3),'b--'
   ,...
        'linewidth',2);hold on;

    % HORIZONS
    hh(j)=plot3(r_2106(:,j*3-14),r_2106(:,j*3-13),r_2106(:,j*3-12),'r-',...
        'linewidth',1);
end

[xs,ys,zs] = sphere;
sun = surf(xs*1.5e8,ys*1.5e8,zs*1.5e8);
set(sun,'FaceColor',[.98 .964 .219])
```

```
%
[xj,yj,zj] = sphere;
jupiter = surf(xj*1.5*71492e3+y(1,16),yj*1.5*71492e3+y(1,17),zj*1.5*71492
    e3+y(1,18));
set(jupiter,'FaceColor',[0.9961 0.5 0])

[xsat,ysat,zsat] = sphere;
saturn = surf(xsat*1.5*60268e3+y(1,19),ysat*1.5*60268e3+y(1,20),zsat
    *1.5*60268e3+y(1,21));
set(saturn,'FaceColor',[1 1 0])

[xu,yu,zu] = sphere;
uranus = surf(xu*2*25559e3+y(1,22),yu*2*25559e3+y(1,23),zu*2*25559e3+y
    (1,24));
set(uranus,'FaceColor',[0 1 0])

% make legend
lgd = legend('RK7 $\Delta$t = 0.005 yrs','JPL HORIZONS',...
    'Location','best','interpreter', 'latex','fontsize', 20);
lgd.Title.String = 'Orbit Propagation';
axis equal;
grid on;
%view(0,90)

text(0,-.4e9,'Jupiter','interpreter', 'latex','fontsize', 15);
text(1.1e9,-1.20e9,'Saturn','interpreter', 'latex','fontsize', 15);
text(1.8e9,1.7e9,'Uranus','interpreter', 'latex','fontsize', 15);
%text(-.1e9,.25e9,'Sun','interpreter', 'latex','fontsize', 15);
xlabel('$X (km)$','interpreter', 'latex','fontsize', 12)
ylabel('$Y (km)$','interpreter', 'latex','fontsize', 12)
zlabel('$Z (km)$','interpreter', 'latex','fontsize', 12)
```

```
clc
clear all
close all

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%   AE370 Project 1
%   This script imports the sim data and HORIZONS data
%   for a 2 year period (2021 -2023) for the inner planets.
%   The norm of the distance from the sun is plotted:
%   sim vs HORIZONS.
%   The sim used RK7 at delta t = 0.005 yrs (2628 minutes).
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% load SIM data
y = readmatrix('sim_rk7_2.csv');

% load HORIZONS data for inner planets
r_mercury = readmatrix('h_2023_mercury.xlsx');
r_venus = readmatrix('h_2023_venus.xlsx');
r_earth = readmatrix('h_2023_earth.xlsx');
```

```
r_mars = readmatrix('h_2023_mars.xlsx');


h_norm = zeros(size(r_earth(:,1),4));

% calculate distance norms for HORIZONS
for i = 1:size(r_earth(:,1))

    h_norm(i,1) = norm(r_mercury(i,3:5));
    h_norm(i,2) = norm(r_venus(i,3:5));
    h_norm(i,3) = norm(r_earth(i,3:5));
    h_norm(i,4) = norm(r_mars(i,3:5));

end

s_norm = zeros(size(y(:,1),4));

% calculate distance norms for sim
for j = 1:size(y(:,1))

    s_norm(j,1) = norm(y(j,4:6));
    s_norm(j,2) = norm(y(j,7:9));
    s_norm(j,3) = norm(y(j,10:12));
    s_norm(j,4) = norm(y(j,13:15));

end

% plot away
figure(1)
plot(h_norm(:,1),'r-','linewidth',2); hold on
plot(s_norm(:,1),'b--','linewidth',2);
text(length(h_norm)+5,h_norm(end,1),'Mercury','interpreter',...
    'latex','fontsize', 20)

plot(s_norm(:,2),'b--','linewidth',2);
plot(h_norm(:,2),'r-','linewidth',2);
text(length(h_norm)+5,h_norm(end,2),'Venus','interpreter',...
    'latex','fontsize', 20)

plot(h_norm(:,3),'r-','linewidth',2);
plot(s_norm(:,3),'b--','linewidth',2);
text(length(h_norm)+5,h_norm(end,3),'Earth','interpreter',...
    'latex','fontsize', 20)

plot(h_norm(:,4),'r-','linewidth',2);
plot(s_norm(:,4),'b--','linewidth',2);
text(length(h_norm)+5,h_norm(end,4),'Mars','interpreter',...
    'latex','fontsize', 20)

lgd = legend('RK7 Sim','HORIZONS','','','','','','',...
    'Location','northwest','interpreter', 'latex','fontsize', 20)
grid on
xlabel('Time Step ($\Delta t=0.005$ years)','interpreter',...
    'latex','fontsize', 20)
```

```
ylabel('Distance from Sun (km)','interpreter', 'latex','fontsize', 20)
%title('Inner Planets Distance from Sun Normalized','interpreter',...
%'latex','fontsize', 20)
```

```
clc
clear all
close all

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%   AE370 Project 1
%   This script imports the sim data and HORIZONS data
%   for an 85 year period (2021 - 2106) for Jupiter, Saturn,
%   and Neptune. The norm of the distance from the sun is
%   plotted: sim vs HORIZONS.
%   The sim used RK7 at delta t = 0.005 yrs (2628 minutes).
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% load SIM data
y = readmatrix('sim_rk7_85.csv');

% load HORIZONS data for outer planets
r_jupiter = readmatrix('h_2106_jupiter.xlsx');
r_saturn = readmatrix('h_2106_saturn.xlsx');
r_uranus = readmatrix('h_2106_uranus.xlsx');

h_norm = zeros(size(r_jupiter(:,1),3));

% norm the distances from HORIZONS
for i = 1:size(r_jupiter(:,1))

    h_norm(i,1) = norm(r_jupiter(i,3:5));
    h_norm(i,2) = norm(r_saturn(i,3:5));
    h_norm(i,3) = norm(r_uranus(i,3:5));

end

s_norm = zeros(size(y(:,1),3));

% norm the distances from sim
for j = 1:size(y(:,1))

    s_norm(j,1) = norm(y(j,16:18));
    s_norm(j,2) = norm(y(j,19:21));
    s_norm(j,3) = norm(y(j,22:24));


end

% plot that bad boy
figure(1)
plot(h_norm(:,1),'r-','linewidth',2); hold on
plot(s_norm(:,1),'b--','linewidth',2);
```

```
text(length(h_norm)+5,h_norm(end,1),'Jupiter','interpreter','latex',...
    'fontsize', 20)

plot(s_norm(:,2),'b--','linewidth',2);
plot(h_norm(:,2),'r-','linewidth',2);
text(length(h_norm)+5,h_norm(end,2),'Saturn','interpreter','latex',...
    'fontsize', 20)

plot(h_norm(:,3),'r-','linewidth',2);
plot(s_norm(:,3),'b--','linewidth',2);
text(length(h_norm)+5,h_norm(end,3),'Uranus','interpreter','latex',...
    'fontsize', 20)

lgd = legend('RK7 Sim','HORIZONS','','','','',...
    'Location','northwest','interpreter', 'latex','fontsize', 20);
grid on
xlabel('Time Step ($\Delta t=0.005$ years)','interpreter', 'latex',...
    'fontsize', 20)
ylabel('Distance from Sun (km)','interpreter', 'latex','fontsize', 20)
%title('Outer Planets Distance from Sun Normalized','interpreter',...
%'latex','fontsize', 20)
```

```
clc
clear all
close all

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%   AE370 Project 1
%   This script imports the data for a black hole sim with
%   the inner planets for two years.
%
%   The sim used RK7 at delta t = 0.005 yrs.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


y = readmatrix('sim_rk7_2bh.csv');

% Plot the paths of the planets
figure(1)

for j=1:4

    h(j)=plot(y(:,1+j*3)-y(:,1),y(:,2+j*3)-y(:,2),'linewidth',1);hold on;
end


[xs,ys,zs] = sphere;
sun = surf(xs*.5*695700e1,ys*.5*695700e1,zs*.5*695700e1);
set(sun,'FaceColor',[0 0 0])

[xm,ym,zm] = sphere;
mercury = surf(xm*2440e3+y(1,4),ym*2440e3+y(1,5),zm*2440e3+y(1,6));
```

```
set(mercury,'FaceColor',[.686 .65 .768])

[xv,yv,zv] = sphere;
venus = surf(xv*6051e3+y(1,7),yv*6051e3+y(1,8),zv*6051e3+y(1,9));
set(venus,'FaceColor',[0.976 0.988 0.713])

[xe,ye,ze] = sphere;
earth = surf(xe*6378e3+y(1,10),ye*6378e3+y(1,11),ze*6378e3+y(1,12));
set(earth,'FaceColor',[.219 .584 .98])

[xma,yma,zma] = sphere;
mars = surf(xma*3396e3+y(1,13),yma*3396e3+y(1,14),zma*3396e3+y(1,15));
set(mars,'FaceColor',[.87 .262 .07])

xlim([-2.5e8 2.5e8])
ylim([-2.5e8 2.5e8])
% zlim([-6 0])
lgd = legend('Mercury','Venus','Earth','Mars','Location','northwest',...
       'interpreter', 'latex','fontsize', 20);
% lgd.Title.String = 'Orbits';
% axis equal;
grid on;
xlabel('$X (km)$','interpreter', 'latex','fontsize', 12)
ylabel('$Y (km)$','interpreter', 'latex','fontsize', 12)
zlabel('$Z (km)$','interpreter', 'latex','fontsize', 12)
```

```
clc
clear all
close all

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%   AE370 Project 1
%   This script imports the data for a black hole sim with
%   the outer planets for 275 years.
%
%   The sim used RK7 at delta t = 0.005 yrs.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

y = readmatrix('sim_rk7_275bh.csv');

% Plot the paths of the planets
figure(1)
for j=5:9

    h(j)=plot3(y(:,1+j*3)-y(:,1),y(:,2+j*3)-y(:,2),y(:,2+j*3)-y(:,3),'
   linewidth',1);hold on;
end

[xs,ys,zs] = sphere;
sun = surf(xs*2*695700e1,ys*2*695700e1,zs*2*695700e1);
set(sun,'FaceColor',[1 1 0])

```

```
% [xj,yj,zj] = sphere;
% jupiter = surf(xj*1.5*71492e3+y(1,16),yj*1.5*71492e3+y(1,17),zj
    *1.5*71492e3+y(1,18));
% set(jupiter,'FaceColor',[0.9961 0.5 0])
%
% [xsat,ysat,zsat] = sphere;
% saturn = surf(xsat*1.5*60268e3+y(1,19),ysat*1.5*60268e3+y(1,20),zsat
    *1.5*60268e3+y(1,21));
% set(saturn,'FaceColor',[1 1 0])
%
% [xu,yu,zu] = sphere;
% uranus = surf(xu*2*25559e3+y(1,22),yu*2*25559e3+y(1,23),zu*2*25559e3+y
    (1,24));
% set(uranus,'FaceColor',[0 1 0])
%
% [xn,yn,zn] = sphere;
% neptune = surf(xn*2*24764e3+y(1,25),yn*2*24764e3+y(1,26),zn*2*24764e3+y
    (1,27));
% set(neptune,'FaceColor',[0.1 0.5 1])
%
% [xp,yp,zp] = sphere;
% pluto = surf(xp*5*6378e3+y(1,28),yp*5*6378e3+y(1,29),zp*5*6378e3+y(1,30)
    );
% set(pluto,'FaceColor',[0.1 0.5 1])

lgd = legend('Jupiter','Saturn','Uranus','Neptune','Pluto',...
             'Location','northwestoutside','interpreter', 'latex','fontsize
    ', 15);
%lgd.Title.String = 'Outer Planets';
axis equal;
grid on;
xlabel('$X (km)$','interpreter', 'latex','fontsize', 12)
ylabel('$Y (km)$','interpreter', 'latex','fontsize', 12)
zlabel('$Z (km)$','interpreter', 'latex','fontsize', 12)

% set(gcf, 'InvertHardCopy','off')
% set(gcf,'papertype','a4')
% print(figure(1), '-dpdf', sprintf('orbits_outer'),'-bestfit','-r300')
```

References

[1] "Runge-Kutta methods," *https://en.wikipedia.org/wiki/Runge-Kutta_methods*, 2021.

[2] Luther, H. A., "An Explicit Sixth-Order Runge-Kutta Formula," *www.jstor.org/stable/2004675*, 1967.

[3] "n-body problem," *https://en.wikipedia.org/wiki/N-body_problem*, 2021.