A simulation of the Solar System

Computational Physics

Mathilde Simoni

May 11, 2023

Abstract

We simulated the motion of the Sun and the 8 planets of the solar system from January 1st, 1750, to January 1st, 2000. The gravitational interaction between these 9 bodies was modeled using Newton's second law of motion and the law of universal gravitation. The Bulirsch-Stoer and 4^{th}-order Runge-Kutta numerical methods were then used to deduce the bodies' motions which we visualized with Blender. We eventually compared the simulation results with NASA's Horizons database and found that despite circular orbits roughly located on the same plane, the accuracy of the predictions decreases over time, with oscillations but generally following a linear trend. We showed that reducing time-step increases the stability of the orbits of the 4 innermost planets in the solar system. However, further research needs to be done to reduce prediction errors for all bodies.

1 Introduction

This project is based on a classic problem in celestial mechanics called the ***n*-body problem**. It refers to understanding how n particles (or celestial bodies) interact with each other with the influence of gravity. Bodies have arbitrary masses, initial velocities, and positions and interact through Newton's law of gravitation.

Unfortunately, the equations of motion do not allow analytical solutions for systems with 3 bodies or more, and theory is limited to studying the qualitative behaviors of the systems. However, numerical solutions are used and have given approximations with reasonable accuracy. The n-body problem is very expensive computationally: computer time per step is proportional to n^2 since it involves determining all pairwise interactions between n bodies. The recent developments of supercomputers and high-performance computing have helped reduce the issues induced by this quadratic growth. More in-depth methods minimizing the costs of calculations have been developed [1] and have enabled to run simulations with more bodies and longer timespans.

An example of an n-body problem involves a classic 3-body problem with the Sun, the Earth, and Jupiter, to study the influence of Jupiter on Earth's trajectory. Other studies, and specifically this paper, focus on the solar system. It includes predicting the motion of $n = 9$ bodies (Sun included) over timescales approaching 4.6-billion-years, the age of our solar system (we limit our study to 250 years due to time constraints). These numerical simulations investigate the past of the solar system, but also questions of stability - whether current planets' configurations will be maintained in the future, or whether planets will eventually leave their orbits, the solar system, and alter other planets' trajectories [2].

2 Methods

2.1 Equations of Motion

The n-body problem considers n points masses $m_i, i = 1, 2, 3 \dots n$ in a 3-dimensional space $\mathbb{R}^3$ moving under the influence of mutual gravitational attraction. Each point mass has a position vector $\boldsymbol{r_i}$ and an acceleration vector $\boldsymbol{a_i}$. By Newton's second law of motion:

$$\boldsymbol{F_i} = m_i \boldsymbol{a_i} \tag{1}$$

Where F_i is the force applied on the point mass m_i.

Moreover, Newton's Law of universal gravitation states that the gravitational force $\boldsymbol{F_{ij}}$ felt on m_i by m_j is:

$$\boldsymbol{F_{ij}} = \frac{G m_i m_j}{\|\boldsymbol{r_j} - \boldsymbol{r_i}\|^2} \left(\frac{\boldsymbol{r_j} - \boldsymbol{r_i}}{\|\boldsymbol{r_j} - \boldsymbol{r_i}\|} \right) \tag{2}$$

Our model simulates a system with $n = 9$ bodies, represented by point masses $m_i, i = 1, 2, 3 \dots 9$. The total gravitational force felt by each m_i, defined as the sum of the gravitational forces exerted on m_i by $m_j, i = 1, 2, ..., 9, j \neq i$ can then be calculated as follows:

$$\boldsymbol{F_i} = \sum_{\substack{j=1 \\ j \neq i}}^{9} \frac{G m_i m_j}{\|\boldsymbol{r_j} - \boldsymbol{r_i}\|^2} \left(\frac{\boldsymbol{r_j} - \boldsymbol{r_i}}{\|\boldsymbol{r_j} - \boldsymbol{r_i}\|} \right) \tag{3}$$

Grouping (1) and (3), we get:

$$\frac{\partial^2 \boldsymbol{r_i}}{\partial t^2} = \boldsymbol{a_i} = \sum_{\substack{j=1 \\ j \neq i}}^{9} \frac{G m_j (\boldsymbol{r_j} - \boldsymbol{r_i})}{\|\boldsymbol{r_j} - \boldsymbol{r_i}\|^3} \tag{4}$$

Finally, we define $\boldsymbol{V_i} = \frac{\partial \boldsymbol{r_i}}{\partial t}$ (and $\boldsymbol{a_i} = \frac{\partial \boldsymbol{v_i}}{\partial t}$) and write the equations in terms of the 3 velocity and position components (Vx_i, Vy_i, Vz_i) and (x_i, y_i, z_i):

$$\begin{cases} \frac{\partial V x_i}{\partial t} = \sum\limits_{\substack{j=1 \\ j \neq i}}^{9} \frac{G m_j (x_j - x_i)}{\|\boldsymbol{r_j} - \boldsymbol{r_i}\|^3} \\ \frac{\partial x_i}{\partial t} = V x_i \end{cases} \tag{5}$$

$$\begin{cases} \frac{\partial V y_i}{\partial t} = \sum\limits_{\substack{j=1 \\ j \neq i}}^{9} \frac{G m_j (y_j - y_i)}{\|\boldsymbol{r_j} - \boldsymbol{r_i}\|^3} \\ \frac{\partial y_i}{\partial t} = V y_i \end{cases} \tag{6}$$

$$\begin{cases} \frac{\partial V z_i}{\partial t} = \sum\limits_{\substack{j=1 \\ j \neq i}}^{9} \frac{G m_j (z_j - z_i)}{\|\boldsymbol{r_j} - \boldsymbol{r_i}\|^3} \\ \frac{\partial z_i}{\partial t} = V z_i \end{cases} \tag{7}$$

2.2 Numerical Methods

We used the Bulirsch-Stoer and the 4^{th}-order Runge-Kutta integration methods to solve the ordinary differential equations (5), (6), (7) described above, and deduct the trajectory of the 9 bodies.

2.2.1 Fourth Order Runge-Kutta

The 4^{th}-order Runge Kutta Method is an integration method which has a fifth order accuracy ($\propto \Delta t^5$). It approximates $x(t+\Delta t)$ from an equation of the form $\frac{\partial x}{\partial t} = f(x,t)$ in the following way:

1. $x_1 = x(t)$ $\qquad$ $t_1 = t$
2. $x_2 = x(t) + f(x_1, t_1)\frac{\Delta t}{2}$ $\qquad$ $t_2 = t + \frac{\Delta t}{2}$
3. $x_3 = x(t) + f(x_2, t_2)\frac{\Delta t}{2}$ $\qquad$ $t_3 = t_2 = t + \frac{\Delta t}{2}$
4. $x_4 = x(t) + f(x_3, t_3)\Delta t$ $\qquad$ $t_4 = t + \Delta t$
5. $x(t+\Delta t) = x(t) + \frac{1}{6}\left[f(x_1,t_1) + 2f(x_2,t_2) + 2f(x_3,t_3) + f(x_4,t_4)\right]\Delta t$

This method uses 4 approximations to the slope at time t and subsequently calculates the weighted sum of these 4 slopes to get the final estimate at time $t + \Delta t$. The idea is that the midpoints slopes ($f(x_2,t_2)$ and $f(x_3,t_3)$) are expected to be better estimates of the actual slope than $f(x_1,t_1)$ and $f(x_4,t_4)$, hence their weighting is higher than those at the endpoints.

We used this technique to solve (5), (6), (7) by simultaneously performing each step for velocity and position components. A screenshot of the code written in a jupyter notebook environment is displayed in the Appendix.

2.2.2 Bulirsch-Stoer

Bulirsch-Stoer is an integration method that uses two powerful techniques, Richardson extrapolation and the modified midpoint method. It aims to obtain an approximation of $x(t+\Delta t)$ from an equation of the form $\frac{\partial x}{\partial t} = f(x,t)$ with high accuracy and little computational effort.

The Modified Midpoint method is a technique that splits the interval Δt into n smaller internals of size h, and then computes $x(t+\Delta t)$ in the following way:

- $x_0 = x(t)$
- $x_1 = x_0 + hf(x_0, t)$
- $x_2 = x_0 + 2hf(x_1, t+h)$
- $x_3 = x_1 + 2hf(x_2, t+2h)$
- $x_4 = x_2 + 2hf(x_3, t+3h)$
- ...
- $x_n = x_{n-2} + 2hf(x_{n-1}, t+(n-1)h)$
- $x(t+\Delta t) = \frac{1}{2}[x_n + x_{n-1} + hf(x_n, t+\Delta t)]$

Bulirsch-Stoer consists of using different values of h to approximate $x(t + \Delta t)$ with the modified midpoint method. Concretely, the interval Δt is split by intervals consisting of finer and finer steps h. Then, it uses Richardson extrapolation to get a value for $x(t + \Delta t)$ when $h \to 0$ (or $n \to \infty$) by using polynomial extrapolation.

Define $F(h)$ as the output of the modified midpoint method $x(t + \Delta t)$ with parameter h. Then suppose $F(h) = a_0 + a_1 h^p + O(h^r)$, $r > p$.

If:

$$F(h) = a_0 + a_1 h^p$$

$$F(\frac{h}{q}) = a_0 + a_1 \left(\frac{h}{g}\right)^p$$

Then, we obtain:

$$a_0 = F(h) + \frac{F(h) - F(\frac{h}{q})}{q^{-p} - 1} \tag{8}$$

With a_0 being the extrapolated value for $x(t + \Delta t)$ when $h \to 0$.

The choice of the modified midpoint instead of Runge-Kutta despite its lower accuracy (second-order accuracy instead of fifth-order accuracy for Runge-Kutta) is motivated by the fact that it only requires one derivative evaluation per step, instead of the two evaluations that Runge-Kutta necessitates. Hence, it is particularly relevant for Bulirsch-Stoer to reduce the algorithm's execution time.

3 Implementation

3.1 Non-Dimensionalization

The distances, masses, and timescales in the context of a solar system simulation are too big to be efficiently stored in computers, as well as to be used to perform mathematical operations. Define $AU = 1.496 \times 10^8\,\text{km}$ (distance between Earth and Sun), $year = 3.156 \times 10^7\,\text{s}$ (Earth year), and $M_* = 1.989 \times 10^{30}\,\text{kg}$ (mass of the Sun). Let:

- $r_i = AU.\hat{r_i}$
- $t = year.\hat{t}$
- $m_i = M_*.\hat{m_i}$

Replacing in (4), we get an equation with the non-dimensional variables $\hat{r_i}$, $\hat{t}$, and $\hat{m_i}$:

$$\frac{\partial^2 \hat{\boldsymbol{r_i}}}{\partial \hat{t}^2} = \sum_{\substack{j=1 \\ j \neq i}}^{9} \left(\frac{GM_* year^2}{AU^3}\right) \frac{\hat{m_j}(\hat{\boldsymbol{r_j}} - \hat{\boldsymbol{r_i}})}{\|\hat{\boldsymbol{r_j}} - \hat{\boldsymbol{r_i}}\|^3} \tag{9}$$

Therefore, in the code using non-dimensional variables, the constant $\hat{G} = \left(\frac{GM_* year^2}{AU^3}\right)$ replaces the gravitational constant G. The rest of the equations stays unchanged.

3.2 Initial Conditions

NASA's JPL (Jet Propulsion Laboratory) Horizons database provides online access to highly accurate positions and velocities for solar system objects. The web interface is publicly accessible at https://ssd.jpl.nasa.gov/horizons/app.html#/.

We used this database to obtain the initial position (x, y, and z) and velocity (Vx, Vy, and Vz) for each body in the solar system. We chose a Cartesian coordinate system centered at the solar system barycenter (center of mass of the solar system) and set the initial conditions to January 1st, 1750. There was no option to obtain data for an earlier date due to the late discovery of Uranus.

file_earth_init_2.txt

```
!$$SOF
MAKE_EPHEM=YES
COMMAND=399
EPHEM_TYPE=VECTORS
CENTER='500@0'
START_TIME='1750-01-01'
STOP_TIME='1750-01-02'
STEP_SIZE='2 DAYS'
VEC_TABLE='3'
REF_SYSTEM='ICRF'
REF_PLANE='ECLIPTIC'
VEC_CORR='NONE'
CAL_TYPE='M'
OUT_UNITS='KM-S'
VEC_LABELS='YES'
VEC_DELTA_T='NO'
CSV_FORMAT='NO'
OBJ_DATA='YES'
```

```python
bodies = ['sun', 'earth', 'jupiter', 'saturn', 'neptune', 'uranus', 'venus', 'mars', 'mercury']
data_bodies = []
```

```python
for body in bodies:
    input_file = "batch_files/file_%s_init_2.txt"%(body)
    f = open(input_file)
    url = 'https://ssd.jpl.nasa.gov/api/horizons_file.api'
    r = requests.post(url, data={'format':'json'}, files={'input': f})
    f.close()
    data = json.loads(r.text)
    data = data['result'].split('\n')
    for i in range(len(data)):
        if data[i] == '$$SOE':
            i_start = i+1
        if data[i] == '$$EOE':
            i_end = i
    data = data[i_start: i_end]
    # print(data)
    data_bodies.append(data)
```

Figure 1: Batch file for Earth and request code

The data was extracted using "batch mode" with a batch data file for each body. This enabled us to automate the process and easily get the data for any date. Figure 1 shows a batch file for Earth and a snippet of the code to make the requests to the Horizons database. The position (km), velocity(km/s), and mass(kg) of each body on January 1st, 1750, are displayed in Figure 2.

body	x	y	z	Vx	Vy	Vz	mass
sun	-1.355e+06	1.421e+04	3.144e+04	0.002	-0.016	8.962e-05	1.989e+30
earth	-2.682e+07	1.449e+08	2.413e+04	-29.815	-5.296	-5.494e-04	5.972e+24
jupiter	7.224e+08	1.564e+08	-1.681e+07	-2.913	13.384	9.666e-03	1.899e+27
saturn	1.217e+09	-8.235e+08	-3.415e+07	4.873	7.982	-3.327e-01	5.683e+26
neptune	4.451e+09	-4.402e+08	-9.351e+07	0.499	5.441	-1.236e-01	1.024e+26
uranus	1.999e+09	2.158e+09	-1.788e+07	-5.046	4.310	8.140e-02	8.681e+25
venus	8.259e+07	-6.916e+07	-5.762e+06	22.050	26.878	-9.029e-01	4.867e+24
mars	7.753e+06	2.338e+08	4.708e+06	-23.292	2.986	6.344e-01	6.416e+23
mercury	1.764e+07	4.206e+07	1.725e+06	-54.131	21.933	6.759e+00	3.285e+23

Figure 2: Initial conditions for the simulation (January 1st, 1750)

3.3 Simulations

We ran simulations on NYUAD's High-Performance Computing Jubail server, with the following parameters:

- Time-span: 250 Earth years (until January 1st, 2000)
- Integration methods: Runge-Kutta and Bulirsch-Stoer
- Time-step: 0.125, 0.25, 0.5, 1, and 2.5 Earth days

3.4 Visualizations

We used the software Blender for visualizations in three dimensions. This software has special packages for astrophysical visualizations, is free, and available on Mac, Linux, and Windows operating systems. We created a night sky effect using shaders.

4 Results

4.1 Matplotlib Plots

Figures 3 and 4 display the sun and planets' trajectories for a simulation of 165 Earth years which is the duration of Neptune's orbit, the outermost planet of the solar system. We observe that the trajectories are very close to being circular on the x-y plane. Moreover, Figure 4 highlights that all bodies are roughly on the same plane, as we expected.

4.2 Blender Visualizations

We also used Blender to produce 3D animations, comprising a night sky, a Sun emitting light, and planets following the trajectory computed with the simulation. Figure 5 shows some pictures of the animations.

5 Analysis

5.1 Time Performance

We evaluated the time performance of simulations performed with Runge-Kutta and Bulirsch-Stoer. Figure 6 displays the execution time of a solar system simulation for 10 Earth years, depending on time-step (x-axis) and integration method (color of the curve). We observe that, as expected, simulations with smaller time-steps take more time to run. In particular, we calculated that the execution time is inversely proportional to the time-step. For example, if the time-step is reduced by 2, the execution time is multiplied by 2. In addition, simulations with the Runge-Kutta method seem to be consistently faster than simulations using Bulirsch-Stoer. This makes sense as Bulirsch-Stoer has more intermediate calculations.

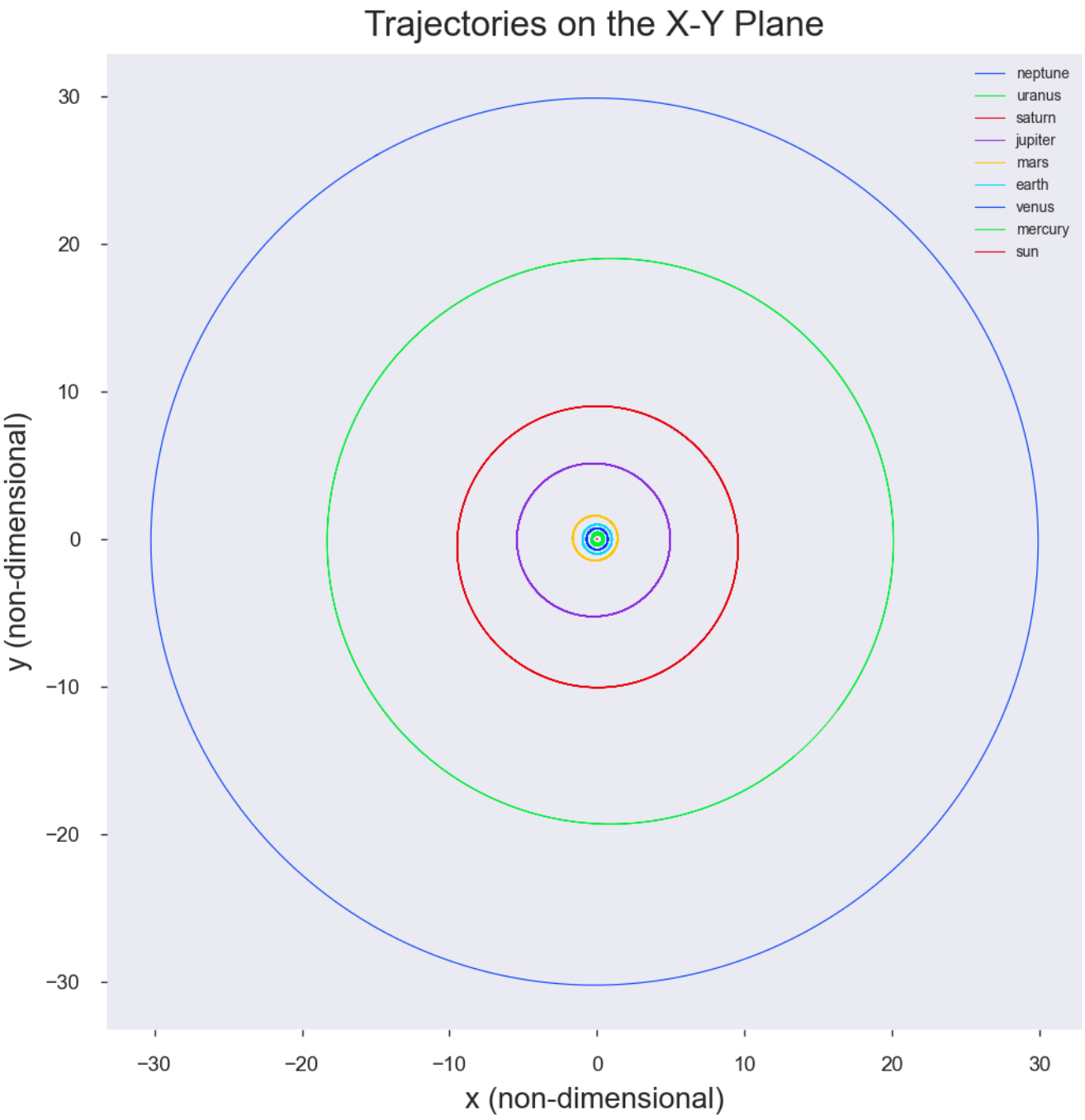

Figure 3: Trajectories of the 8 planets in the Solar System on the x-y plane

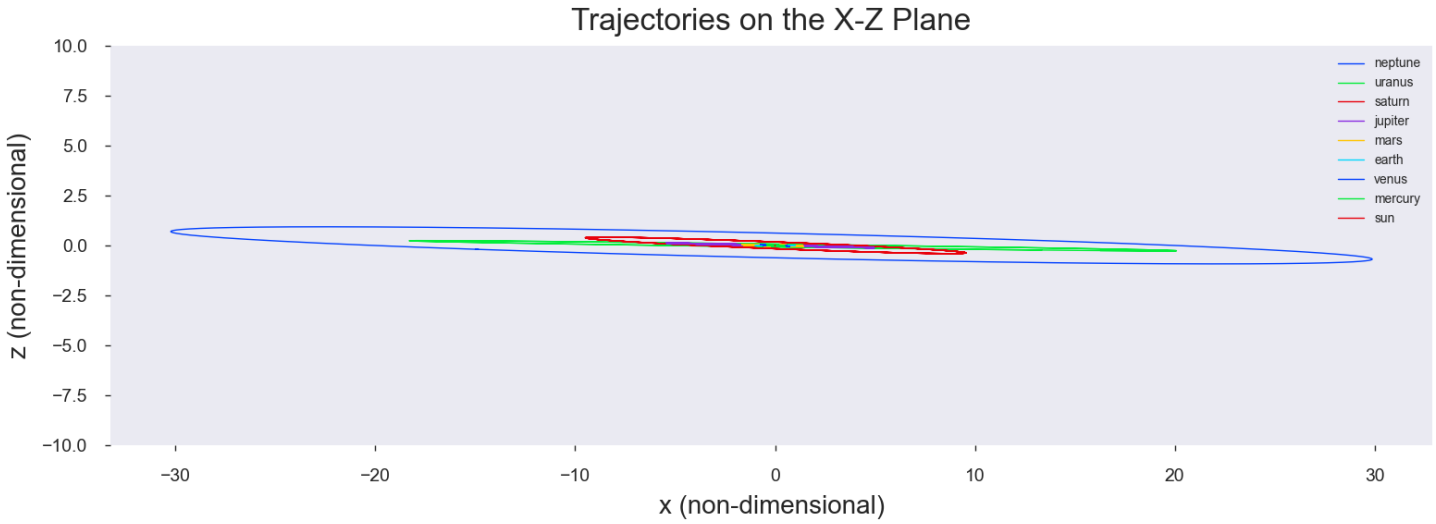

Figure 4: Trajectories of the 8 planets in the Solar System on the x-z plane

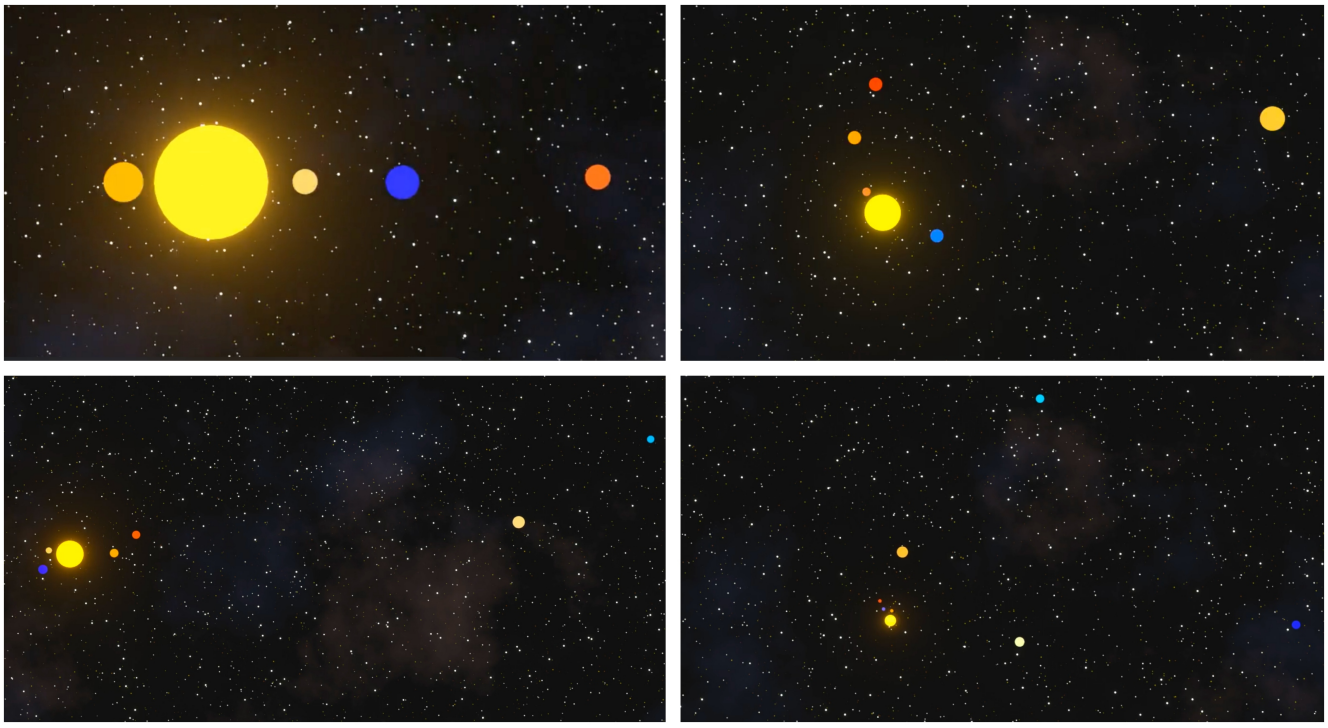

Figure 5: Blender visualizations

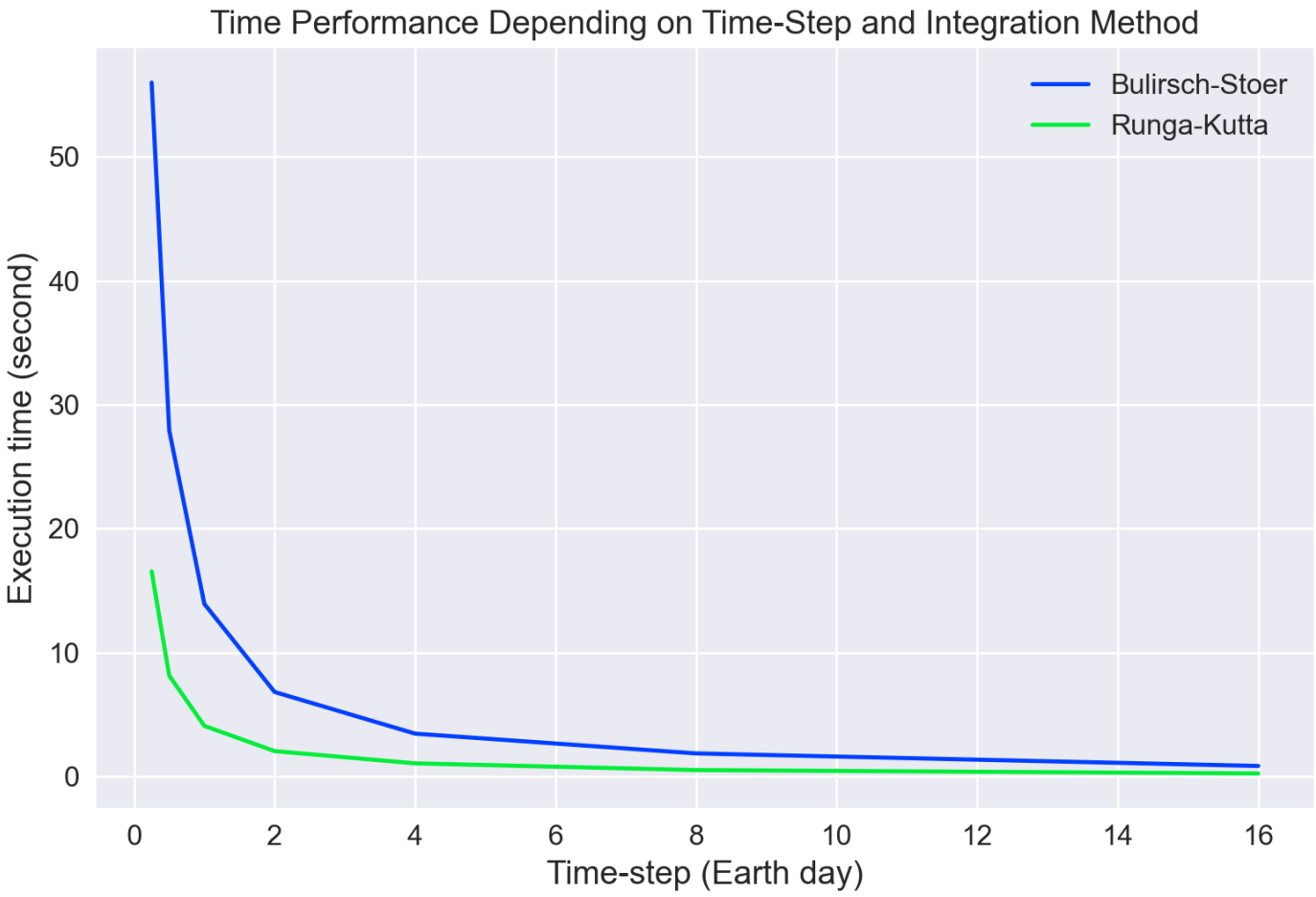

Figure 6: Time Performance

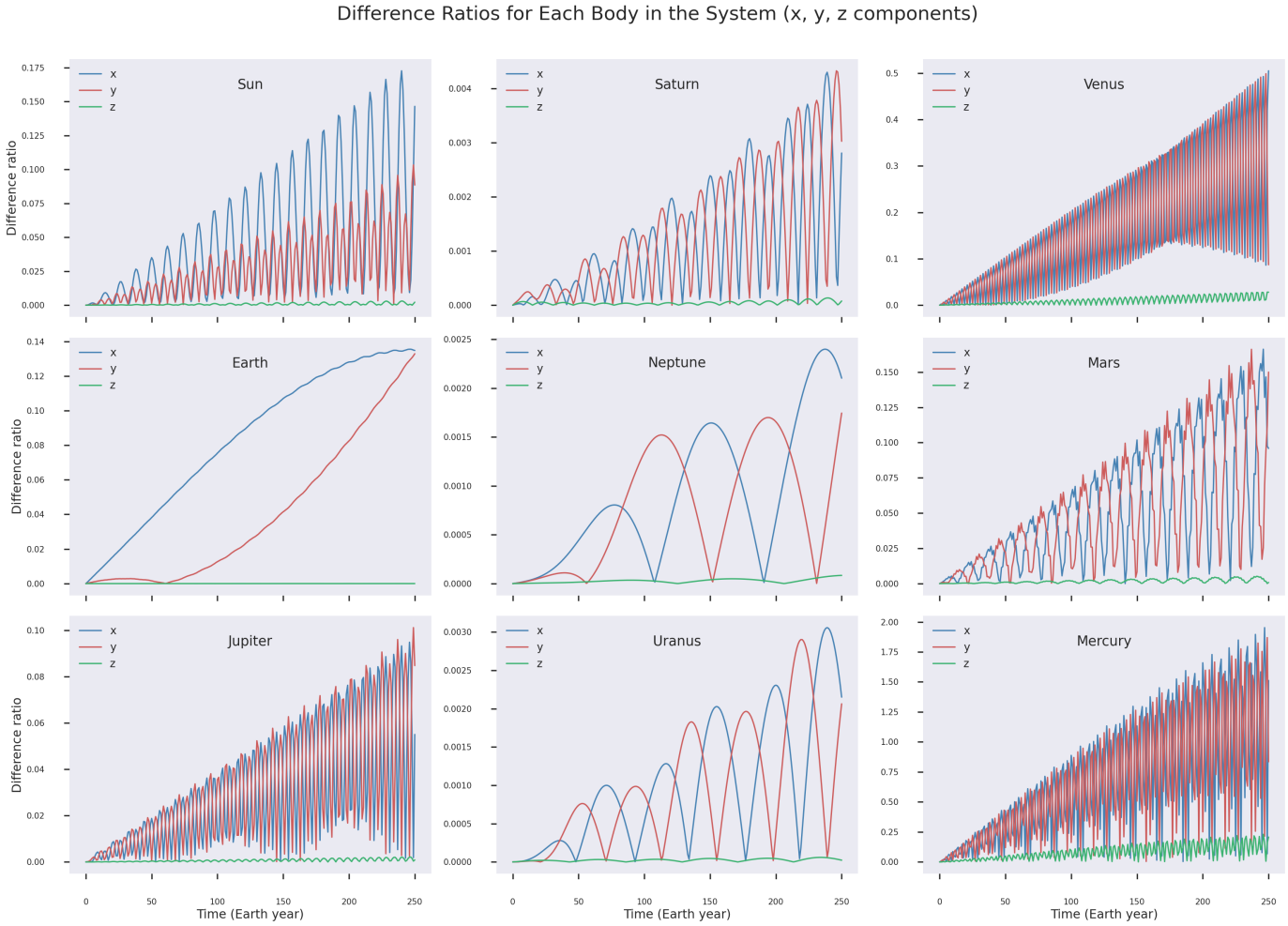

Figure 7: Accuracy Plots

5.2 Accuracy

We compared the simulations results with NASA's JPL Horizons database.

5.2.1 Difference Ratios

In order to evaluate the accuracy of the simulations, we first calculated difference ratios, defined as:

$$\text{difference ratio} = \frac{|\text{NASA's expected value} - \text{simulation value}|}{\text{planet's semi-major axis}}$$

Difference ratios were calculated for the x, y, and z position components of each body, every Earth year, and for 250 years. The resulting plots for a simulation with a time-step of 1 Earth day and the Bulirsch-Stoer Integration method are displayed in Figure 7. All plots except Earth's graph are characterized by oscillations and a trend of linear increase throughout time. Earth's plot does not show oscillations but can still be approximated linearly.

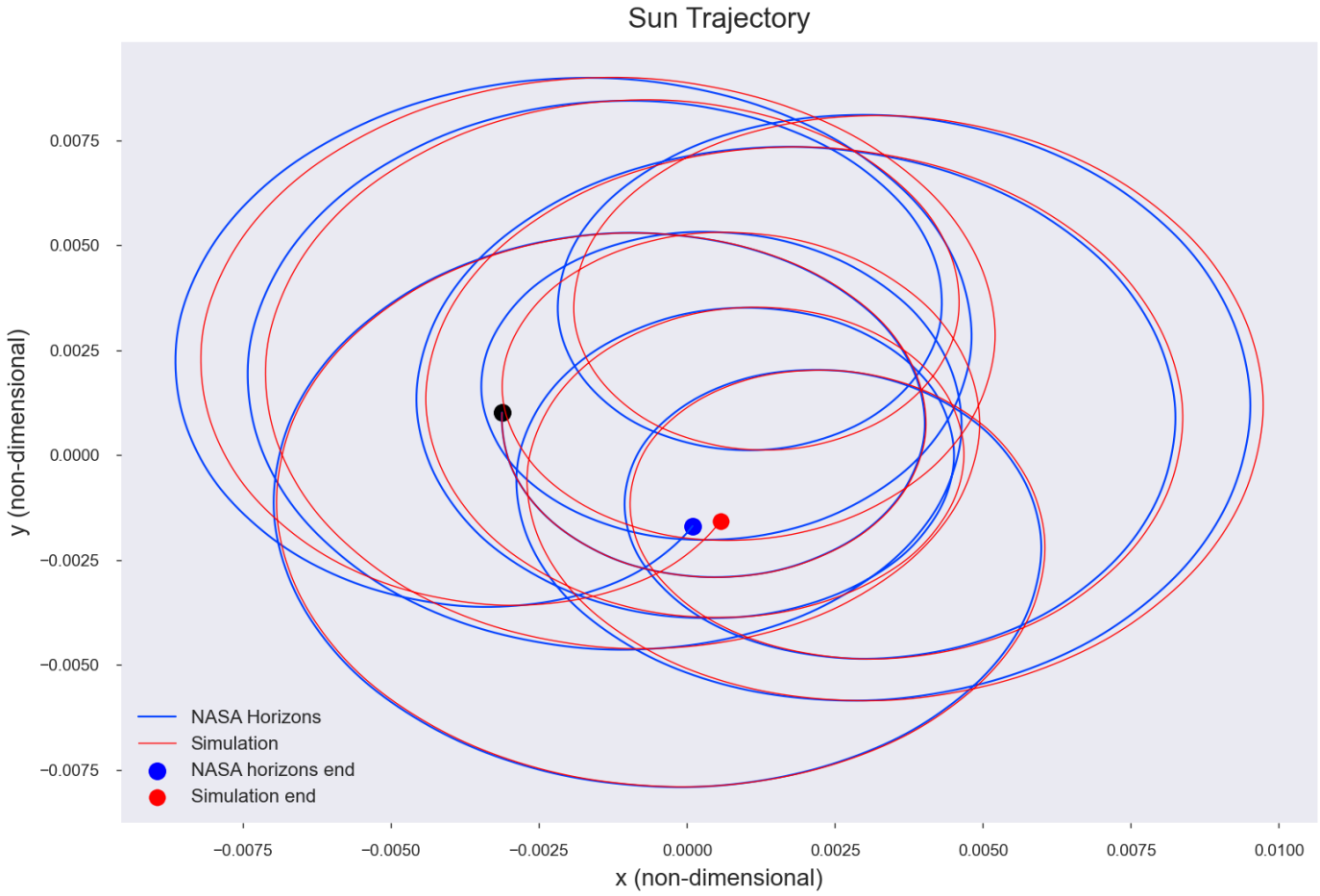

Figure 8: Trajectory of the Sun (our simulation and NASA Horizons database)

5.2.2 Trajectory of the Sun

In order to understand the oscillations, we plotted the Sun's trajectory over 10 years according to our simulation and the Horizons Database (Figure 8). First, we notice that the Sun does not follow a specific orbit around the solar system barycenter. This is due to the fact that the barycenter, the center of mass of all 9 bodies orbiting each other in the solar system, evolves depending on where the planets are located.

In addition, Figure 8 explains the oscillations obtained in Figure 7. We note that there are some points in time where our simulation and NASA's data are very close, whereas some other points in time show a bigger difference in position. This pattern seems to be repeating itself with more and more difference as time goes. This is exactly what was observed in Figure 7, and the same reasoning can be applied to other planets' accuracy plots.

5.2.3 Slope Analysis

As observed in Figure 7, difference ratios can be approximated by a linear function of time despite the oscillations. We used linear regression methods to calculate the slopes characterizing the difference ratios increase over time. In other words, the slope is an indicator of the simulations' accuracies and stability over time. A bigger slope means that the simulation becomes highly inaccurate in a short period of time, while a lower slope means that the simulation results are close to NASA's data with an error that increases slowly with time. We performed the slope analysis for simulations with different time-steps and integration methods and plotted

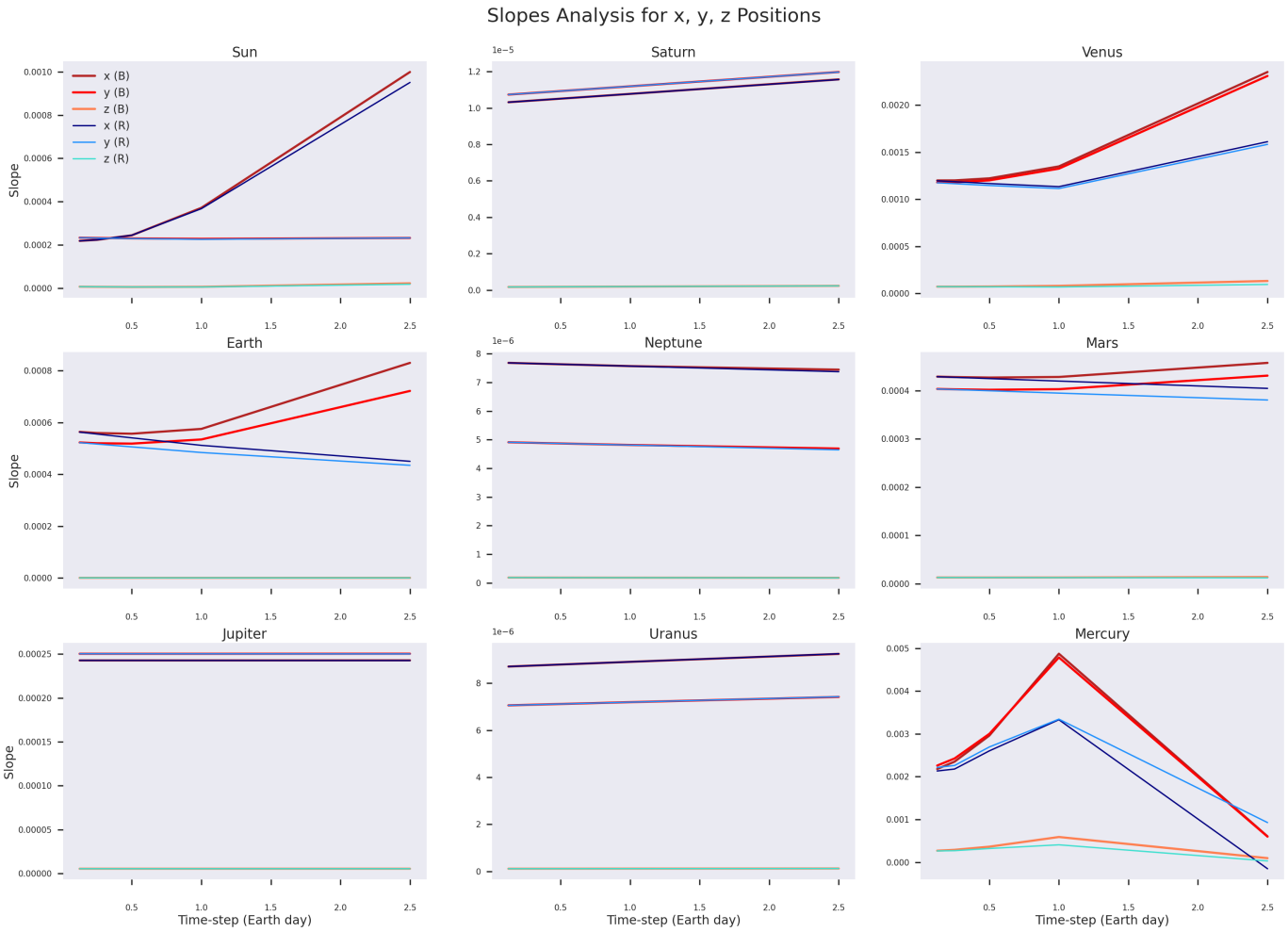

Figure 9: Slope analysis as described in section 5.2.3. In the legend, x(B) refers to the slope characterizing the accuracy of the x position component for a simulation performed with Bulirsch-Stoer integration method. x(R) refers to the same data but for a simulation performed with Runge-Kutta integration method.

the results in Figure 9.

We observe that for the Sun, Venus, Earth, and Mars, choosing smaller time-steps gives a smaller slope, hence a higher accuracy and more stable trajectories over time. Moreover, slopes are usually smaller for the Runge-Kutta method but decrease more rapidly with time-step when Using Bulirsch-Stoer is used. Regarding Neptune, Jupiter, Saturn, and Uranus, there does not seem to be any big change with time-step or integration method. This might come from the fact that they are bigger planets hence less affected by smaller perturbations. Finally, the case of Mercury is particular. Figure 10 displays the accuracy plot of this planet for a simulation with a 2.5 day time-step, with calculated slopes. We observe that the slope is very small, but it is because the accuracy over 250 years does not follow a linear growth. Instead, it is periodic. Hence, the slope analysis for this planet does not make sense and no interpretation should be induced from this plot.

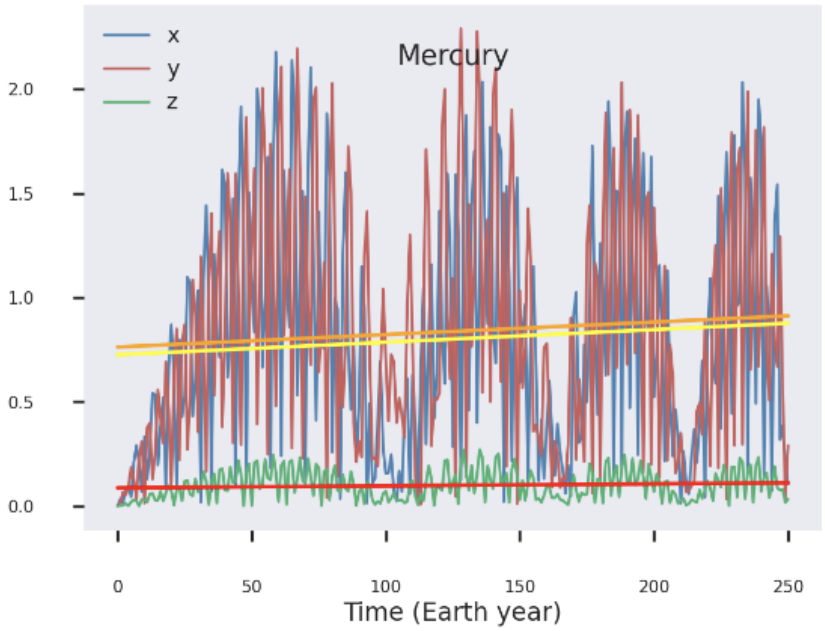

Figure 10: Accuracy plot for Mercury

6 Conclusion

We successfully simulated the solar system. Starting from January 1st, 1750, we calculated approximations of the Sun and 8 other planets' trajectories across 250 years. Matplotlib and Blender visualizations showed that the 9 bodies gravitationally interact as expected, with orbits close to being circular and roughly on the same plane. In addition, NASA's JPL Horizons Database allowed us to measure the accuracy and stability of these predictions over time. Overall, the difference ratios' general trend (inversely proportional to accuracy and stability over time) can be approximated with a linear increase over time. We have shown that using smaller time-steps can help reduce errors in the predictions for the Sun, Earth, Venus, and Mars.

Further research needs to be done in order to raise the quality of the simulations. First, we need to improve the implementation of the Bulirsch-Stoer method by choosing a polynomial of a higher degree as extrapolation function, and by using the modified midpoint method with smaller intervals. In addition, the difference in accuracy between the small terrestrial planets and the other gas planets needs to be investigated. Furthermore, we could extend the system to bigger time-spans and include moons. Finally, regarding Blender visualizations, we need to automate the process of adding textures for the planets.

References

[1] SJ Aarseth. Numerical experiments on the n-body problem. In *International Astronomical Union Colloquium*, volume 10, pages 20–34. Cambridge University Press, 1971.

[2] Leslie Greengard. The numerical solution of the n-body problem. *Computers in physics*, 4(2):142–152, 1990.

Appendix

```python
def runga_kutta_3d_optimized(position, velocity, timestep, Fx, Fy, Fz, args_x, args_y, args_z):
    x_values = [None] * 4
    y_values = [None] * 4
    z_values = [None] * 4

    Vx_values = [None] * 4
    Vy_values = [None] * 4
    Vz_values = [None] * 4

    Fx_values = [None] * 4
    Fy_values = [None] * 4
    Fz_values = [None] * 4

    # step 1
    x_values[0] = position[0]
    y_values[0] = position[1]
    z_values[0] = position[2]

    Vx_values[0] = velocity[0]
    Vy_values[0] = velocity[1]
    Vz_values[0] = velocity[2]

    Fx_values[0] = Fx(args_x, Vx_values[0], Vy_values[0], Vz_values[0], x_values[0], y_values[0], z_values[0])
    Fy_values[0] = Fy(args_y, Vx_values[0], Vy_values[0], Vz_values[0], x_values[0], y_values[0], z_values[0])
    Fz_values[0] = Fz(args_z, Vx_values[0], Vy_values[0], Vz_values[0], x_values[0], y_values[0], z_values[0])

    # step 2
    x_values[1] = x_values[0] + Vx_values[0] * timestep/2
    y_values[1] = y_values[0] + Vy_values[0] * timestep/2
    z_values[1] = z_values[0] + Vz_values[0] * timestep/2

    Vx_values[1] = Vx_values[0] + Fx_values[0] * timestep/2
    Vy_values[1] = Vy_values[0] + Fy_values[0] * timestep/2
    Vz_values[1] = Vz_values[0] + Fz_values[0] * timestep/2

    Fx_values[1] = Fx(args_x, Vx_values[1], Vy_values[1], Vz_values[1], x_values[1], y_values[1], z_values[1])
    Fy_values[1] = Fy(args_y, Vx_values[1], Vy_values[1], Vz_values[1], x_values[1], y_values[1], z_values[1])
    Fz_values[1] = Fz(args_z, Vx_values[1], Vy_values[1], Vz_values[1], x_values[1], y_values[1], z_values[1])
```

Figure 11: Python code for Runge-Kutta integration method (part 1)

```
    # step 3
    x_values[2] = x_values[0] + Vx_values[1] * timestep/2
    y_values[2] = y_values[0] + Vy_values[1] * timestep/2
    z_values[2] = z_values[0] + Vz_values[1] * timestep/2

    Vx_values[2] = Vx_values[0] + Fx_values[1] * timestep/2
    Vy_values[2] = Vy_values[0] + Fy_values[1] * timestep/2
    Vz_values[2] = Vz_values[0] + Fz_values[1] * timestep/2

    Fx_values[2] = Fx(args_x, Vx_values[2], Vy_values[2], Vz_values[2], x_values[2], y_values[2], z_values[2])
    Fy_values[2] = Fy(args_y, Vx_values[2], Vy_values[2], Vz_values[2], x_values[2], y_values[2], z_values[2])
    Fz_values[2] = Fz(args_z, Vx_values[2], Vy_values[2], Vz_values[2], x_values[2], y_values[2], z_values[2])

    # step 4
    x_values[3] = x_values[0] + Vx_values[2] * timestep
    y_values[3] = y_values[0] + Vy_values[2] * timestep
    z_values[3] = z_values[0] + Vz_values[2] * timestep

    Vx_values[3] = Vx_values[0] + Fx_values[2] * timestep
    Vy_values[3] = Vy_values[0] + Fy_values[2] * timestep
    Vz_values[3] = Vz_values[0] + Fz_values[2] * timestep

    Fx_values[3] = Fx(args_x, Vx_values[3], Vy_values[3], Vz_values[3], x_values[3], y_values[3], z_values[3])
    Fy_values[3] = Fy(args_y, Vx_values[3], Vy_values[3], Vz_values[3], x_values[3], y_values[3], z_values[3])
    Fz_values[3] = Fz(args_z, Vx_values[3], Vy_values[3], Vz_values[3], x_values[3], y_values[3], z_values[3])

    # step 5
    x_final = x_values[0] + ((Vx_values[0] + 2*Vx_values[1] + 2*Vx_values[2] + Vx_values[3])*timestep)/6
    y_final = y_values[0] + ((Vy_values[0] + 2*Vy_values[1] + 2*Vy_values[2] + Vy_values[3])*timestep)/6
    z_final = z_values[0] + ((Vz_values[0] + 2*Vz_values[1] + 2*Vz_values[2] + Vz_values[3])*timestep)/6

    Vx_final = Vx_values[0] + ((Fx_values[0] +2*Fx_values[1] +2*Fx_values[2] + Fx_values[3])*timestep)/6
    Vy_final = Vy_values[0] + ((Fy_values[0] +2*Fy_values[1] +2*Fy_values[2] + Fy_values[3])*timestep)/6
    Vz_final = Vz_values[0] + ((Fz_values[0] +2*Fz_values[1] +2*Fz_values[2] + Fz_values[3])*timestep)/6

    return (x_final, y_final, z_final), (Vx_final, Vy_final, Vz_final)
```

Figure 12: Python code for Runge-Kutta integration method (part 2)

```
def bulirsch_stoer_3d_optimized(position, velocity, timestep, Fx, Fy, Fz, args_x, args_y, args_z, N, h, q, p):
    # step 1: modified midpoint method with step size h
    x_values1, y_values1, z_values1, Vx_values1, Vy_values1, Vz_values1 =
        modified_midpoint_method_optimized(position, velocity, Fx, Fy, Fz, args_x, args_y, args_z, N, h)

    # step 2: modified midpoint method with step size h/q
    N_2 = int((timestep * q)/h)
    h_2 = h/q

    x_values2, y_values2, z_values2, Vx_values2, Vy_values2, Vz_values2 =
        modified_midpoint_method_optimized(position, velocity, Fx, Fy, Fz, args_x, args_y, args_z, N_2, h_2)

    # step 3: Richardson Extrapolation
    x_final = Richardson_extrapolation(x_values1, x_values2, q, p)
    y_final = Richardson_extrapolation(y_values1, y_values2, q, p)
    z_final = Richardson_extrapolation(z_values1, z_values2, q, p)

    Vx_final = Richardson_extrapolation(Vx_values1, Vx_values2, q, p)
    Vy_final = Richardson_extrapolation(Vy_values1, Vy_values2, q, p)
    Vz_final = Richardson_extrapolation(Vz_values1, Vz_values2, q, p)

    return (x_final, y_final, z_final), (Vx_final, Vy_final, Vz_final)


def Richardson_extrapolation(f_h, f_hq, q, p):
    return f_h + (f_h - f_hq)/((q**(-p)) -1)
```

Figure 13: Python code for Bulirsch-Stoer integration method (part 1)

```
def modified_midpoint_method_optimized(position, velocity, Fx, Fy, Fz, args_x, args_y, args_z, N, h):
    x_values = [None] * (N + 1)
    y_values = [None] * (N + 1)
    z_values = [None] * (N + 1)

    Vx_values = [None] * (N + 1)
    Vy_values = [None] * (N + 1)
    Vz_values = [None] * (N + 1)

    Fx_values = [None] * (N + 1)
    Fy_values = [None] * (N + 1)
    Fz_values = [None] * (N + 1)

    # prep
    x_values[0] = position[0]
    y_values[0] = position[1]
    z_values[0] = position[2]

    Vx_values[0] = velocity[0]
    Vy_values[0] = velocity[1]
    Vz_values[0] = velocity[2]

    Fx_values[0] = Fx(args_x, Vx_values[0], Vy_values[0], Vz_values[0], x_values[0], y_values[0], z_values[0])
    Fy_values[0] = Fy(args_y, Vx_values[0], Vy_values[0], Vz_values[0], x_values[0], y_values[0], z_values[0])
    Fz_values[0] = Fz(args_z, Vx_values[0], Vy_values[0], Vz_values[0], x_values[0], y_values[0], z_values[0])

    # step 1
    # print("step:", 1)
    x_values[1] = x_values[0] + Vx_values[0] * h
    y_values[1] = y_values[0] + Vy_values[0] * h
    z_values[1] = z_values[0] + Vz_values[0] * h

    Vx_values[1] = Vx_values[0] + Fx_values[0] * h
    Vy_values[1] = Vy_values[0] + Fy_values[0] * h
    Vz_values[1] = Vz_values[0] + Fz_values[0] * h

    Fx_values[1] = Fx(args_x, Vx_values[1], Vy_values[1], Vz_values[1], x_values[1], y_values[1], z_values[1])
    Fy_values[1] = Fy(args_y, Vx_values[1], Vy_values[1], Vz_values[1], x_values[1], y_values[1], z_values[1])
    Fz_values[1] = Fz(args_z, Vx_values[1], Vy_values[1], Vz_values[1], x_values[1], y_values[1], z_values[1])

    # steps 2 +
    for n in range(2, N+1):
        # print("step:", n)
        x_values[n] = x_values[n-2] + Vx_values[n-1] * 2 * h
        y_values[n] = y_values[n-2] + Vy_values[n-1] * 2 * h
        z_values[n] = z_values[n-2] + Vz_values[n-1] * 2 * h

        Vx_values[n] = Vx_values[n-2] + Fx_values[n-1] * 2 * h
        Vy_values[n] = Vy_values[n-2] + Fy_values[n-1] * 2 * h
        Vz_values[n] = Vz_values[n-2] + Fz_values[n-1] * 2 * h

        Fx_values[n] = Fx(args_x, Vx_values[n], Vy_values[n], Vz_values[n], x_values[n], y_values[n], z_values[n])
        Fy_values[n] = Fy(args_y, Vx_values[n], Vy_values[n], Vz_values[n], x_values[n], y_values[n], z_values[n])
        Fz_values[n] = Fz(args_z, Vx_values[n], Vy_values[n], Vz_values[n], x_values[n], y_values[n], z_values[n])

    # final step
    x_values = 1/2 * (x_values[N] + x_values[N-1] + (h * Vx_values[N]))
    y_values = 1/2 * (y_values[N] + y_values[N-1] + (h * Vy_values[N]))
    z_values = 1/2 * (z_values[N] + z_values[N-1] + (h * Vz_values[N]))

    Vx_values = 1/2 * (Vx_values[N] + Vx_values[N-1] + (h * Fx_values[N]))
    Vy_values = 1/2 * (Vy_values[N] + Vy_values[N-1] + (h * Fy_values[N]))
    Vz_values = 1/2 * (Vz_values[N] + Vz_values[N-1] + (h * Fz_values[N]))

    return x_values, y_values, z_values, Vx_values, Vy_values, Vz_values
```

Figure 14: Python code for Bulirsch-Stoer integration method (part 2)