FYS2085 project work: planetary motion

Mika Mäki

December 22, 2021

Contents

Introduction

Newtonian gravity simulations are a cornerstone of modern astrophysics. They can be used to model various astrophysical systems spanning several orders of magnitude all the way from an individual solar system to the movement and merging of galaxies.[1]

This project work provides an introduction to such n-body simulations by modeling the Solar System and investingating the effects of different simulation parameters on the results. Both velocity Verlet and Runge-Kutta 4 integration algorithms are used.

[1] As a side note they can also be highly entertaining and beautiful to watch but simultaneously rigorously rooted in physics.

1 Methods

The primary n-body simulator of this project is based on the velocity Verlet integration algorithm. Therefore its mathematical basis starts from the familiar Newton's second law

$$\vec{F} = m\vec{a}, \tag{1}$$

which relates the force F to the resulting acceleration a and the mass of the object m. Using it the acceleration caused by the net effect of multiple forces is

$$\vec{a} = \frac{1}{m}\sum \vec{F}_i. \tag{2}$$

Now if we approximate the acceleration to be constant over the time step of interest we have for the velocity and position

$$v = v_i + a_i\Delta t, \tag{3}$$

$$x = x_i + v_i\Delta t + \frac{1}{2}a\Delta t^2. \tag{4}$$

Since these equations are highly prone to divergence due to slight numerical errors, adjustments are needed for better results. This can be accomplished by averaging the accelerations of the current and previous timesteps, leading us to the Velocity Verlet algorithm

$$x_{i+1} = x_i + v_i\Delta t + \frac{1}{2}a_i\Delta t^2, \tag{5}$$

$$a_{i+1} = \frac{F(x_{i+1})}{m}, \tag{6}$$

$$v_{i+1} = v_i + \frac{1}{2}(a_i + a_{i+1})\Delta t. \tag{7}$$

As this is a Newtonian gravity simulation, the forces between the objects are governed by the Newton's law of gravitation

$$F = G\frac{m_1 m_2}{r^2}, \tag{8}$$

where G is the gravitational constant, m_1 and m_2 are the masses of the objects and r is the distance between them.

The second algorithm chosen for this project is Runge-Kutta 4, as it's a very commonly used and versatile integrator with good accuracy. However, it should be noted that neither the velocity Verlet nor the Runge-Kutta 4 are symplectic integrators. Therefore they both experience a drift in the total energy of the simulated system over time. The Runge-Kutta 4 algorithm is designed to solve a system of first-order differential equations. Therefore we have to expand our second-order differential equation to two first-order equations. These can be written as

$$\dot{\vec{r}} = \vec{v} \tag{9}$$

$$\dot{\vec{v}} = \vec{a}. \tag{10}$$

Now we have two first-order differential equations that can be solved with the algorithm. However, it should be noted that when computing the intermediate

slopes $\vec{k}$, we need to use the intermediate results from the other differential equation that correspond to the intermediate step we are computing. Therefore we can write the intermediate slopes for the position as

$$\vec{k}_{1,r_{i+1}} = \vec{v}_i \tag{11}$$

$$\vec{k}_{2,r_{i+1}} = \vec{v}_i + \vec{k}_{1,v_{i+1}} \frac{h}{2} \tag{12}$$

$$\vec{k}_{3,r_{i+1}} = \vec{v}_i + \vec{k}_{2,v_{i+1}} \frac{h}{2} \tag{13}$$

$$\vec{k}_{4,r_{i+1}} = \vec{v}_i + \vec{k}_{3,v_{i+1}} h, \tag{14}$$

where we have denoted the timestep Δt as h to be consistent with the usual notation for the algorithm. The corresponding definitions for velocity are

$$\vec{k}_{1,v_{i+1}} = \vec{a}(\vec{r}_i) \tag{15}$$

$$\vec{k}_{2,v_{i+1}} = \vec{a}(\vec{r}_i + \vec{k}_{1,r_{i+1}} \frac{h}{2}) \tag{16}$$

$$\vec{k}_{3,v_{i+1}} = \vec{a}(\vec{r}_i + \vec{k}_{2,r_{i+1}} \frac{h}{2}) \tag{17}$$

$$\vec{k}_{4,v_{i+1}} = \vec{a}(\vec{r}_i + \vec{k}_{3,r_{i+1}} h). \tag{18}$$

With these values we can compute the velocity and position for the next step as

$$\vec{r}_{i+1} = \vec{r}_i + \frac{h}{6} \left(\vec{k}_{1,r_{i+1}} + \vec{k}_{2,r_{i+1}} + \vec{k}_{3,r_{i+1}} + \vec{k}_{4,r_{i+1}} \right) \tag{19}$$

$$\vec{v}_{i+1} = \vec{v}_i + \frac{h}{6} \left(\vec{k}_{1,v_{i+1}} + \vec{k}_{2,v_{i+1}} + \vec{k}_{3,v_{i+1}} + \vec{k}_{4,v_{i+1}} \right) \tag{20}$$

It should be noted that the source used for this algorithm has a typo in the definitions of $\vec{k}_{n,r_{i+1}}$. in its equation 10. The product between the velocity and the previous $\vec{k}$ should be a sum. [1]

2 Implementation

The program consists of two parts. Fortran is used for the computation, and Python for the configuration of the simulation and plotting. The interface between Fortran and Python is facilitated by F2PY library, which is nowadays a part of Numpy. This program also uses several other libraries for plotting. Matplotlib is used for exporting static images, and PyQtGraph is used for 3D graphics and animations. PyQtGraph in turn is built upon the PySide6 Qt bindings, and also uses PyOpenGL and PyOpenGL-accelerate to improve the performance of its 3D graphics.

The program is used through a Python file that contains the specification of the celestial bodies and the selection of tools to analyze the simulation with. The `main.py` serves this purpose in this project. The file `sim.py` contains the actual simulation wrapper that interacts with the Fortran-based simulation `core.f90`.

The `core.f90` Fortran module consists of the primary simulation loop `iterate` and the function `force` that it uses to compute the forces excerted on an object by the other celestial bodies. It also contains some printing functions for arrays. In addition to the Python bindings this module can be called from the Fortran main program `main.f90`. This main program utilizes also the modules `cmd_line.f90` and `utils.f90` that provide various utility functions.

The Fortran-based main program can be configured by modifying the ini-like configuration file `config.txt` in the `run` folder and passing it as a command-line argument. The Python-based main program in turn is configured by modifying the values directly within `main.py`.

Instructions for building and running the program are available in the readme files in the repository. Copies of those files are included here for convenience in appendix A.

3 Results

At first the software was tested by simulating the binary system of the Sun and Jupiter. The period is estimated from the average time between the change of sign for the x-coordinate of the orbit. This gives the results of figure 1, and comparing these to the true period gives figure 3 (parts 1. a) and c)). In these figures we can see that the simulation starts to rapidly diverge if the timestep is above 0.06 years, which corresponds to about 22 days. Below this timestep the results are fairly consistent, but with timesteps around 0.05 years it seems that the numerical errors cancel the systematic errors with some timestep values. Below these timesteps we are left with a systematic difference to the true period. This could be caused by the integration, but also by differences in the input values and the fact that this does not take into account the effects of general relativity and the gravitational wells of other planets.

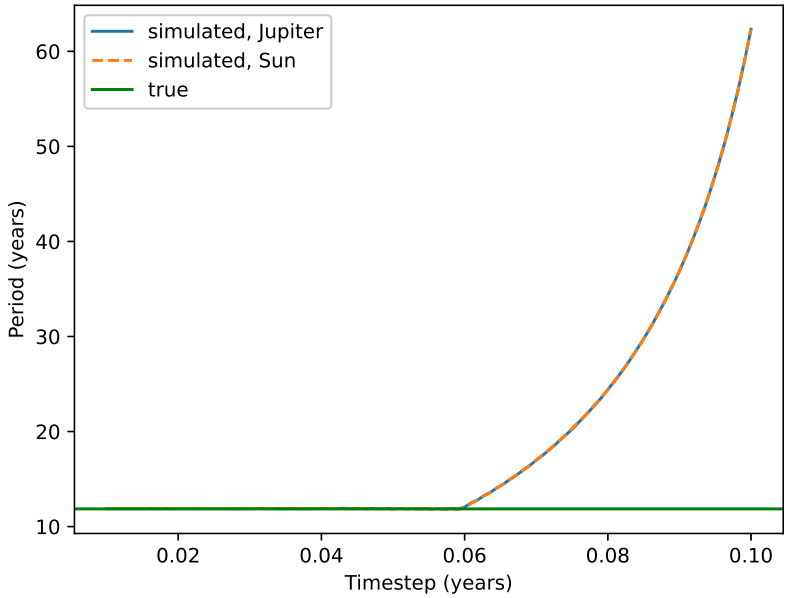

Figure 1: Period of Jupiter's orbit as a function of the simulation timestep

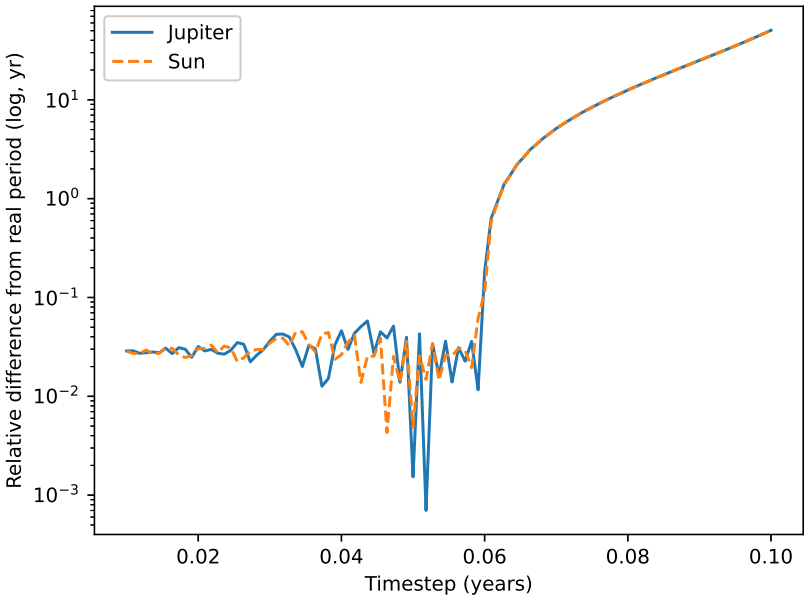

Figure 2: Difference between the simulated and true orbits

The radius of the orbit of the Sun was also investigated and found to be of the order of 744000 km, which is slightly higher than its radius, 696340 km. The error bars denote standard deviation over the course of the simulation. For the period of the motion of the Sun, please see figures 1 and 3.

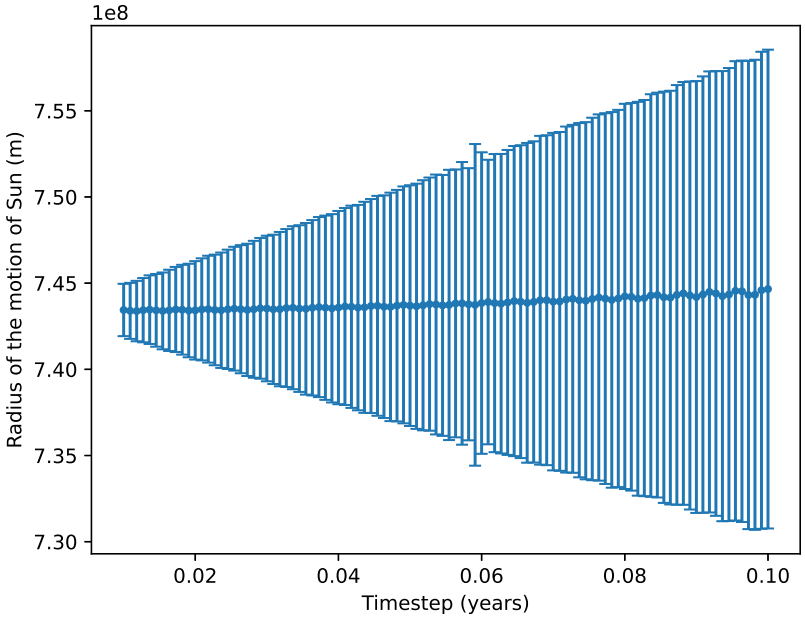

Figure 3: Radius of the orbit of the Sun

Then the motion of Jupiter was simulated for one orbital period (part 1. b)). In this case the reference value for the orbital period was calculated form the initial radius and velocity of its orbit. According to the results of figures 4 and 5 the movement seems to be too slow compared to the analytical value, but the error is reduced to the order of 1 deg with 10^3 steps. However, increasing the number of steps does not reduce the error significantly.

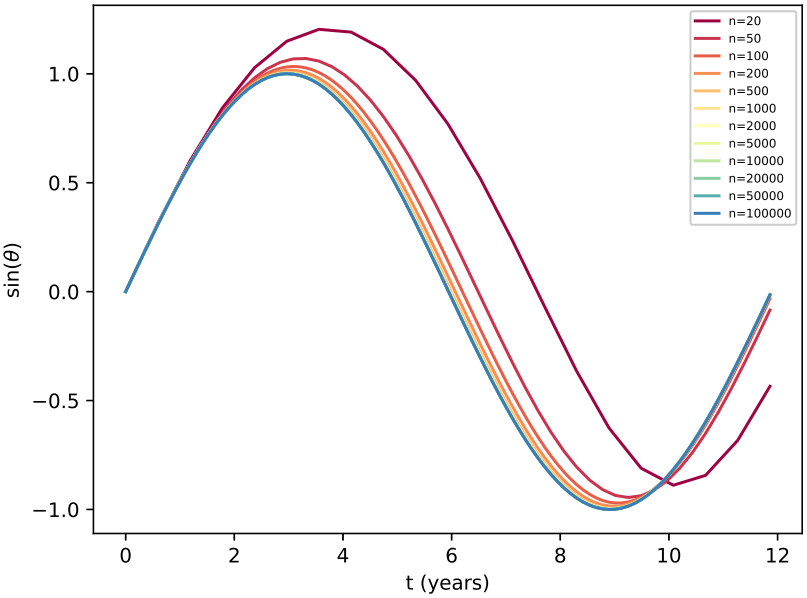

Figure 4: Sine of the rotation angle of Jupiter's orbit

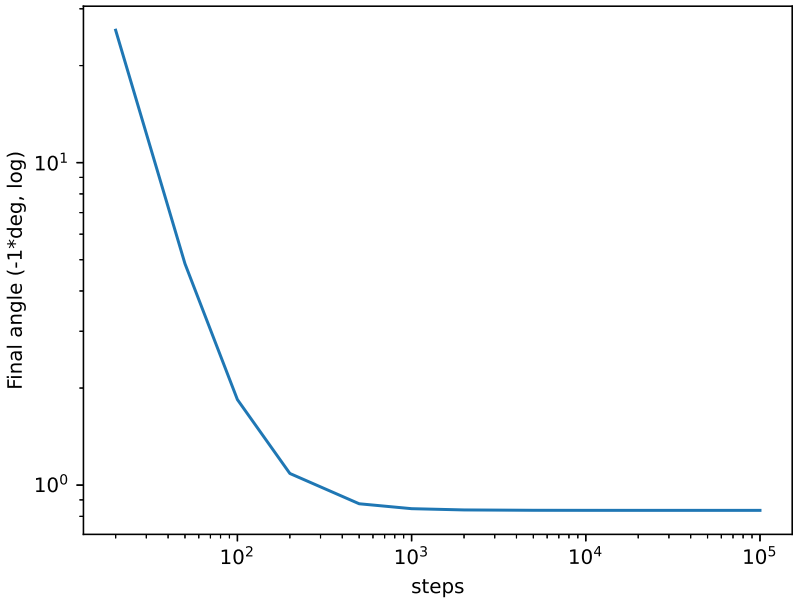

Figure 5: Final angle of Jupiter after simulating for one period

Other binary pairs were also tested (part 2). Simulating the length of a year with the Sun and Earth gave the results of figure 6, and similarly simulating a month with the Earth and the Moon gave the results of figure 7. Both of these have a clear point separating a relatively stable solution from a diverging one. For the simulation of a year this is approximately $dt = 0.05$ yr and for a month $dt = 0.0004$ yr $= 0.15$ d. Interestingly above these values the difference exhibits oscillatory behaviour.

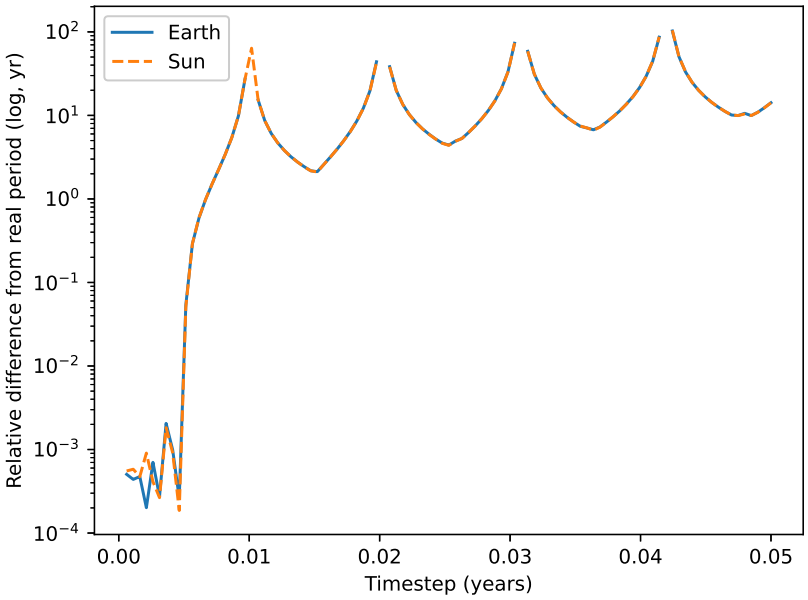

Figure 6: Relative difference of a simulated year from the true value

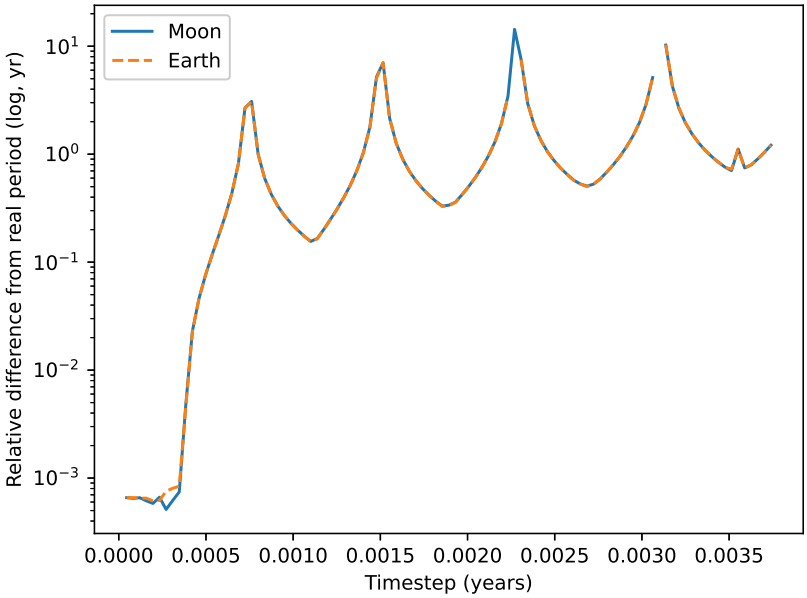

Figure 7: Relative difference of a simulated month from the true value

As a final test the entire solar system was simulated (part 3), giving the results of figure 8. The vertical line corresponds to the $dt = 0.06$ yr, which was the point of divergence for Jupiter in the previous simulations. It can be seen that the periods of other inner planets are already significantly diverging at these values. The period of Mercury could not even be computed with the chosen algorithm. To see where they diverge, we can have a look at the zoomed in version of this plot, figure 9. The periods for other planets seem to be relatively stable at around $2 \cdot 10^{-2}$ yr, but for Mercury we need 10^{-2} yr for it to be relatively stable. However, if we look at the zoomed in plot we can see that a step of about 10^{-3} yr is needed until the period of Mercury is no longer affected by the choice of timestep. It should be noted that the lines for the outer planets don't extend all the way to the shortest timesteps, as with those timesteps the simulation ended before they could complete their orbits.

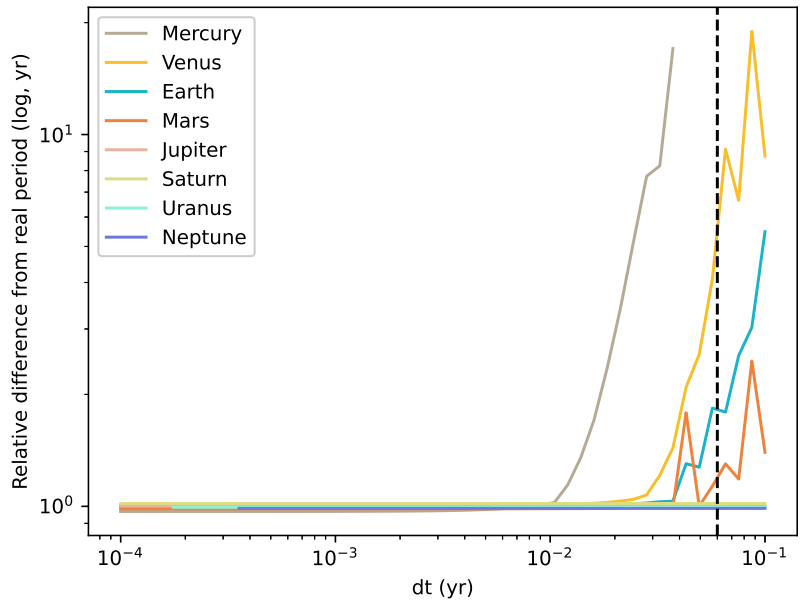

Figure 8: Relative differences of the orbits compared to their true values

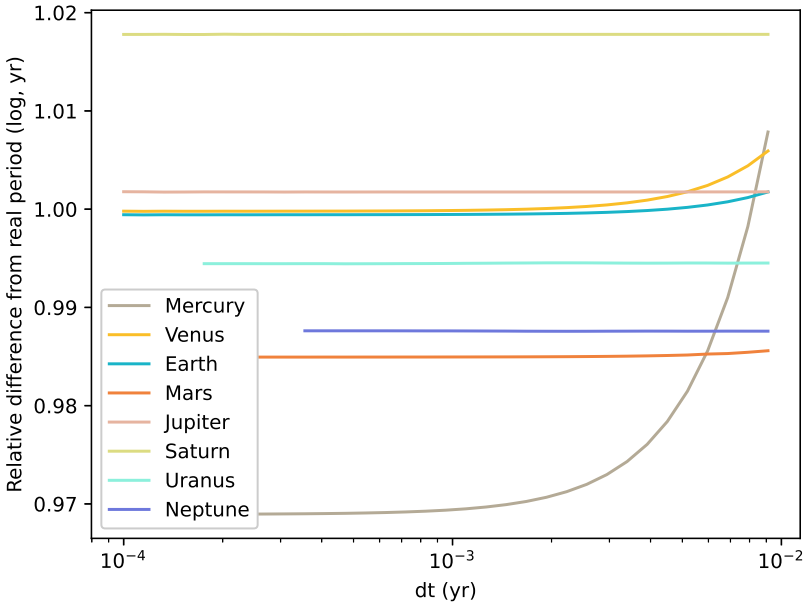

Figure 9: Relative differences of the orbits compared to their true values, zoomed

This simulation was also run with Runge-Kutta 4. Interestingly with this integrator the development of the divergences was not monotonous, but instead with increasing timestep the periods first decrease and then explode. The overall scale of the timesteps necessary for accurate simulation is similar, but there are notable differences as well. For Mercury the value explodes at a similar timestep, 10^{-2} yr, but when looking at the zoomed plot we can see that the value stabilizes earlier. This might tempt towards the use of Runge-Kutta 4 over velocity Verlet. However, Runge-Kutta 4 requires the computing and storage of more intermediate values and seems therefore computationally more demanding. As is usually the case, each algorithm has its pros and cons.

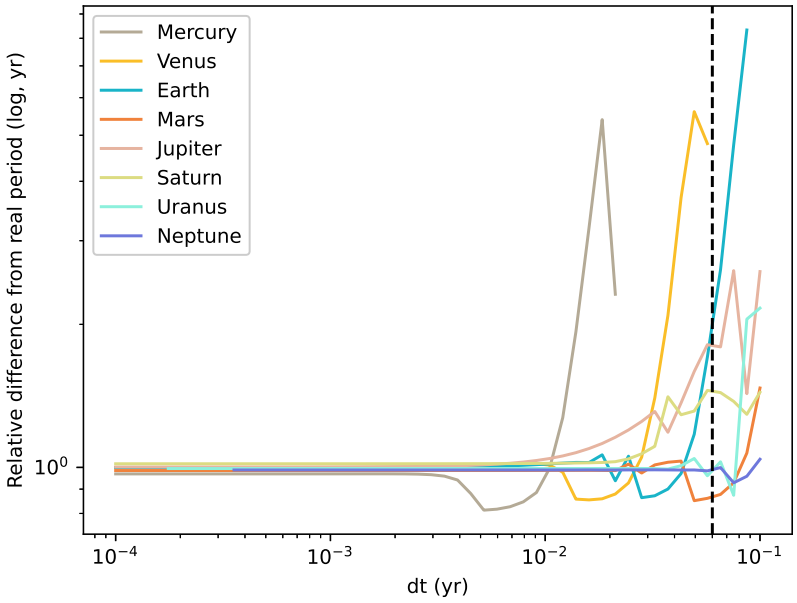

Figure 10: Relative differences of the orbits compared to their true values with Runge-Kutta

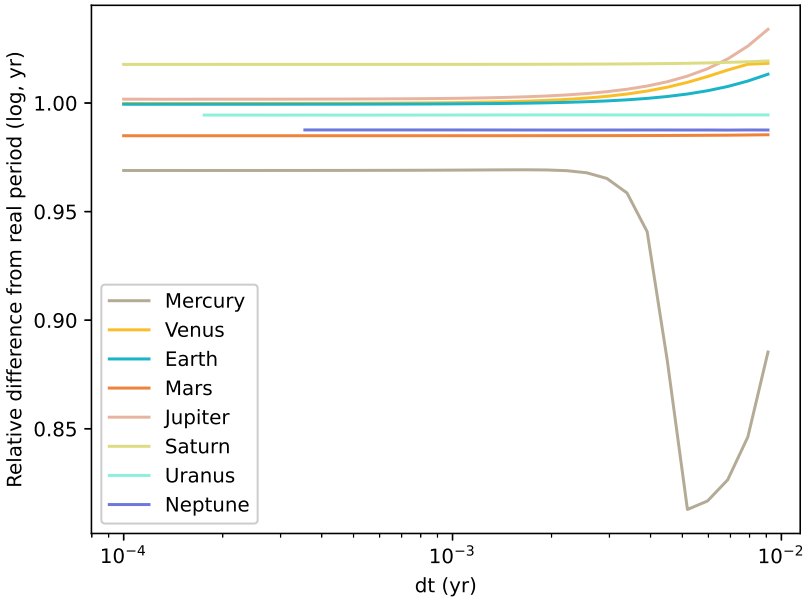

Figure 11: Relative differences of the orbits compared to their true values with Runge-Kutta, zoomed

The software also contains tools for interactive 3D visualization. Please have a look at the commented-out 3D visualizations and n-body simulation tests. Figure 12 provides an example of these visualizations.

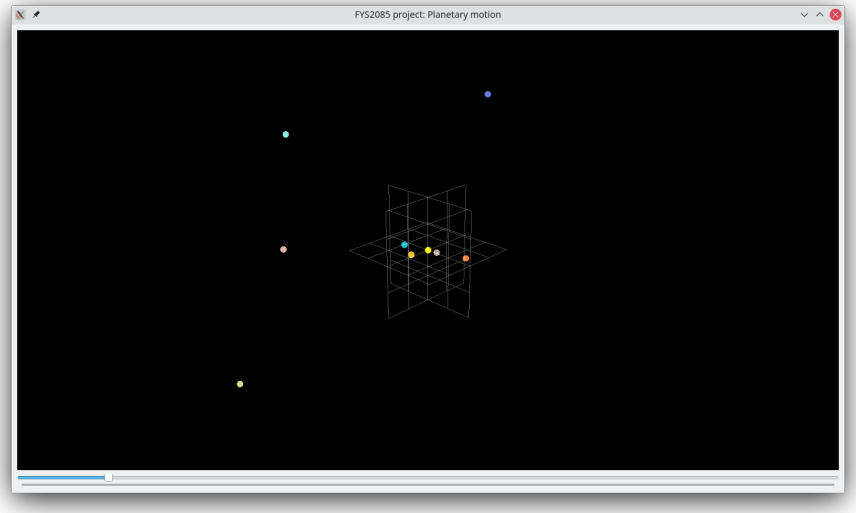

Figure 12: A screenshot of the 3D visualization

4 Conclusions

This project has provided a quick overview to the fascinating world of n-body simulations in astrophysics and a useful template for creating such a simulation. However, it should be highlighted that neither of the tested algorithms is sympletic and they both will therefore eventually exhibit divergence even with extremely small time steps. Therefore the most logical continuation to this project would be to implement a sympletic integrator.

Real-world use cases are also extremely large compared to these almost trivial examples. Therefore for practical applications the performance of the code should be improved significantly, especially with parallelization, as modern supercomputers and even personal computers are nowadays highly parallel systems thanks to the rapid advances in CPU core counts and GPU computing.

References

[1] C J Voesenek. “Implementing a Fourth Order Runge-Kutta Method for Orbit Simulation”. In: (June 14, 2008), p. 3. URL: http://spiff.rit.edu/richmond/nbody/OrbitRungeKutta4.pdf.

A Readme files

A.1 Build and install instructions

<!– This is a Markdown file and is best viewed with a suitable program such as Okular. These comments should not be visible when viewing this file with a proper program. –>

Building the Fortran part of this project requires a Fortran compiler such as gfortran[2].

Building the Python part of this project requires Python 3 (tested with Python 3.9) and some packages from PyPI[3]. The recommended way to install these packages is by using pip[4] in a virtualenv[5]. You can create a virtualenv to the current folder by running `python3 -m venv venv`. Then you can activate it with `source ./venv/bin/activate` on Linux and by running the corresponding PowerShell script on Windows. Now both the commands `python` and `python3` should point to the Python 3 installation of the virtualenv in the current shell session. When the virtualenv has been set up, the packages can be installed by running `pip install -r requirements.txt` when in the src folder. Alternatively you can use an IDE such as PyCharm[6], which will take care of the virtualenv and libraries for you. However, since the requirements.txt is not at the root of this project, you may have to set its location manually in the settings at Tools → Python Integrated Tools → Packaging → Package requirements file.

When the all the dependencies have been installed, the project can be built by running `make all`. (Personally I would prefer to use a bash script instead, but the use of `make` is said to give bonus points.) The GitHub Actions[7] workflow files[8] can be used as an additional template on how to build the project.

There are also precompiled binaries available on GitHub[9], but running the Python code requires the installation of the necessary libraries regardless.

<!– By the way, in my opinion it would be the best to put both build and usage documentation in one README file in the root of the repository, since this is a rather small project. –>

[2] <https://gcc.gnu.org/wiki/GFortran>
[3] <https://pypi.org/>
[4] <https://pip.pypa.io/en/stable/>
[5] <https://docs.python.org/3/library/venv.html>
[6] <https://www.jetbrains.com/pycharm/>
[7] <https://github.com/features/actions>
[8] <../.github/workflows>
[9] <https://github.com/AgenttiX/planetary-motion/actions>

A.2 Run instructions

<!– This is a Markdown file and is best viewed with a suitable program such as Okular These comments should not be visible when viewing this file with a proper program. –>

When you have compiled the program using the build instructions[10], you can run it from the src folder[11] without arguments to use the default configuration. Alternatively you can specify two command-line arguments: the configuration file path and the output folder path. `./planetary-motion /path/to/config.txt /path/to/output_folder`

The program can also be directly run from the Python script to get the advanced plotting functionality. The Python script will also take care of the compilation, provided that you have installed all the necessary libraries. `python3 main.py`

The first line of the configuration file should be `planetary-motion configuration`. There should be two sections: `[scalars]` and `[arrays]`. Scalars should be specified by `name = value` and arrays should have their names on separate rows with data directly on the following rows. Empty lines and lines starting with # and ; are ignored. Please see the example config.txt[12] in this folder for a detailed listing of the parameters.

<!– By the way, in my opinion it would be the best to put both build and usage documentation in one README file in the root of the repository, since this is a rather small project. –>

[10] <../src/README.md>
[11] <../src>
[12] <./config.txt>