# FYS3150 Computational Physics - Project 3

Nicholas Karlsen

This is an abstract

## INTRODUCTION

In this report we will take a look at two methods for solving Ordinary differential equations (ODE) numerically, which is of great interest to physicists, as a lot of physical systems are governed by such equations, most of which lack analytical solutions. In particular, we will look closely at how these numerical methods can be used to solve multi-body systems governed by Newton's law of gravity, which has no exact analytical solutions for $n \geq 3$ bodies.

As such, i have developed a flexible object oriented class in python which solves systems of coupled differential equations using the Velocity-Verlet algorithm and the Forward Euler method, the source code of which can be found on my Github page at: https://github.com/nicholaskarlsen/FYS3150.

## THEORY, ALGORITHMS AND METHODS

### Newton's law of universal gravitation

Between every body, there is a force of attraction inversely proportional to the square of the separation, or more precisely, the force acting on some body with mass $m$ due to a mass $m'$ is

$$\mathbf{F} = -G\frac{mm'}{|\mathbf{r}-\mathbf{r}'|^2}\hat{\mathbf{u}}_{\mathbf{r}-\mathbf{r}'}, \quad \hat{\mathbf{u}}_{\mathbf{r}-\mathbf{r}'} = \frac{\mathbf{r}-\mathbf{r}'}{|\mathbf{r}-\mathbf{r}'|} \tag{1}$$

Where $G$ is the gravitational constant and $\mathbf{r}, \mathbf{r}'$ denote the position vectors of bodies with mass $m, m'$ respectively.

Chosing the the 2D cartesian coordinate system, let $\mathbf{r}-\mathbf{r}' = (x_r, y_r)$, where the $r$ suffix denote that these coordinates are to be understood as the relative coordinates between bodies $m, m'$. Further, $|\mathbf{r}-\mathbf{r}'| = \sqrt{x_r^2+y_r^2} = r$ is the distance between the two bodies.

By Newtons second law, the acceleration on body 1 due to the gravitational pull of body 2 can then be written as

$$\mathbf{a} = \frac{1}{m}\mathbf{F} = -G\frac{m'}{r^2}\frac{(\mathbf{r}-\mathbf{r}')}{r} = -G\frac{m'}{r^2}\frac{(x_r, y_r)}{r} \tag{2}$$

Written out component-wise in terms of the positions, we get the set of coupled differential equations

$$\frac{\partial^2 x}{\partial t^2} = -G\frac{m'}{r^2}\frac{x_r}{r}, \quad \frac{\partial^2 y}{\partial t^2} = -G\frac{m'}{r^2}\frac{y_r}{r} \tag{3}$$

Similar, for 3 dimensions where $\mathbf{r}-\mathbf{r}' = (x_r, y_r, z_r), r = \sqrt{x_r^2+y_r^2+z_r^2}$.

The potential due to the gravitational field can be shown to be [4]

$$E_p(\mathbf{r}) = -G\frac{mm'}{r} \tag{4}$$

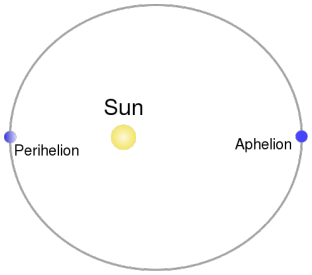



FIG. 1: In an eliptic orbit, the closest and farthest points in the orbit is defined as the Perhelion and Aphelion respectively [Image source]

### *Relativistic Correction*

The aforementioned model of gravity fails to predict the perihelion (see fig. 1) precession of mercury, which is observed to be 43" (arcseconds) per century [3].

That is, the closed, uniform elliptical orbits predicted by the Newtonian model for gravity does not match with observations in Astronomy, where the perihelion of Mercury seems to shift its location over time. In fact, the perihelion precession of mercury was the first experimental confirmation of General relativity, the model of gravity developed by Albert Einstein which accurately predicts this phenomena.

And so, a correcting factor accounting for the relativistic effects is added to Newtons model, and the magnitude of the gravitational force becomes [3]

$$|\mathbf{F}| = G\frac{mm'}{r^2}\left[1+\frac{3l^2}{r^2c^2}\right] \tag{5}$$

Where $l$ denotes the magnitude of the angular momentum of the orbiting body and $c$ the speed of light.

### Solving ODEs numerically

*Forward Euler*

Consider a function $f(t)$, which derivative, $f'(t)$ is known and we want to find $f(t+\delta t)$. Take the taylor expansion of $f(t+\delta t)$

$$f(t+\delta t) = f(t) + f'(t)\delta t + \frac{1}{2}f''(t)\delta t^2 + \cdots + \frac{1}{n!}f^{(n)}(t)\delta t^n \tag{6}$$

If we then truncate this series after the second term we get

$$f(t+\delta t) = f(t) + f'(t)\delta t + \mathcal{O}(\delta t^2) \tag{7}$$

where the term $\mathcal{O}(\delta t^2)$ contains the numerical error associated by truncating the series early.

Since $f(t), f'(t)$ are known, this allows us to find $f(t+\delta t)$, which is the basis of the Forward Euler algorithm. Following, i have written out in pseudocode how this algorithm can be used to solve a system governed by some known force $\mathbf{F}$ and initial conditions $\mathbf{v}(0) \simeq v_0, \mathbf{r}(0) \simeq r_0$.

Forward Euler Algorithm

```
1  for i = 0, ..., N − 1
2      v_{i+1} = v_i + a_i Δt
3      r_{i+1} = r_i + v_i Δt
4
```

Notice that the value of $\mathbf{v}_{i+1}$ is known when computing $\mathbf{r}_{i+1}$. By using this newly calculated velocity in the computation instead, we get the Euler-Cromer method, which wont be discussed further in this report.

Euler-Cromer Algorithm

```
1  for i = 0, ..., N − 1
2      v_{i+1} = v_i + a_i Δt
3      r_{i+1} = r_i + v_{i+1} Δt
4
```

*Velocity-Verlet*

The Velocity-Verlet method is another method for solving ODE's numerically, which just like Forward Euler is derived from the manipulation of Taylor expansions. However, this time i refer you to [2] for the full derivation and simply state the result

$$\begin{aligned} f(t+\delta t) &= f(t) + f'(t)\delta t + \frac{f''(t)\delta t}{2} + \mathcal{O}(\delta t^3) \\ f'(t+\delta t) &= f'(t) + \frac{\left(f''(t+\delta t) + f''(t)\right)\delta t}{2} + \mathcal{O}(\delta t^3) \end{aligned} \tag{8}$$

Which again, can be applied to solve a Physical system where $\mathbf{F}$ is known. The algorithm for doing this is written out below

Velocity-Verlet Algorithm

```
1  for i = 0, ..., N − 1
2      r_{i+1} = r_i + v_i Δt + (1/2) a_i (Δt)^2
3      v_{i+1} = v_i + (1/2)(a_{i+1} + a_i) Δt
4
```

## RESULTS AND DISCUSSIONS

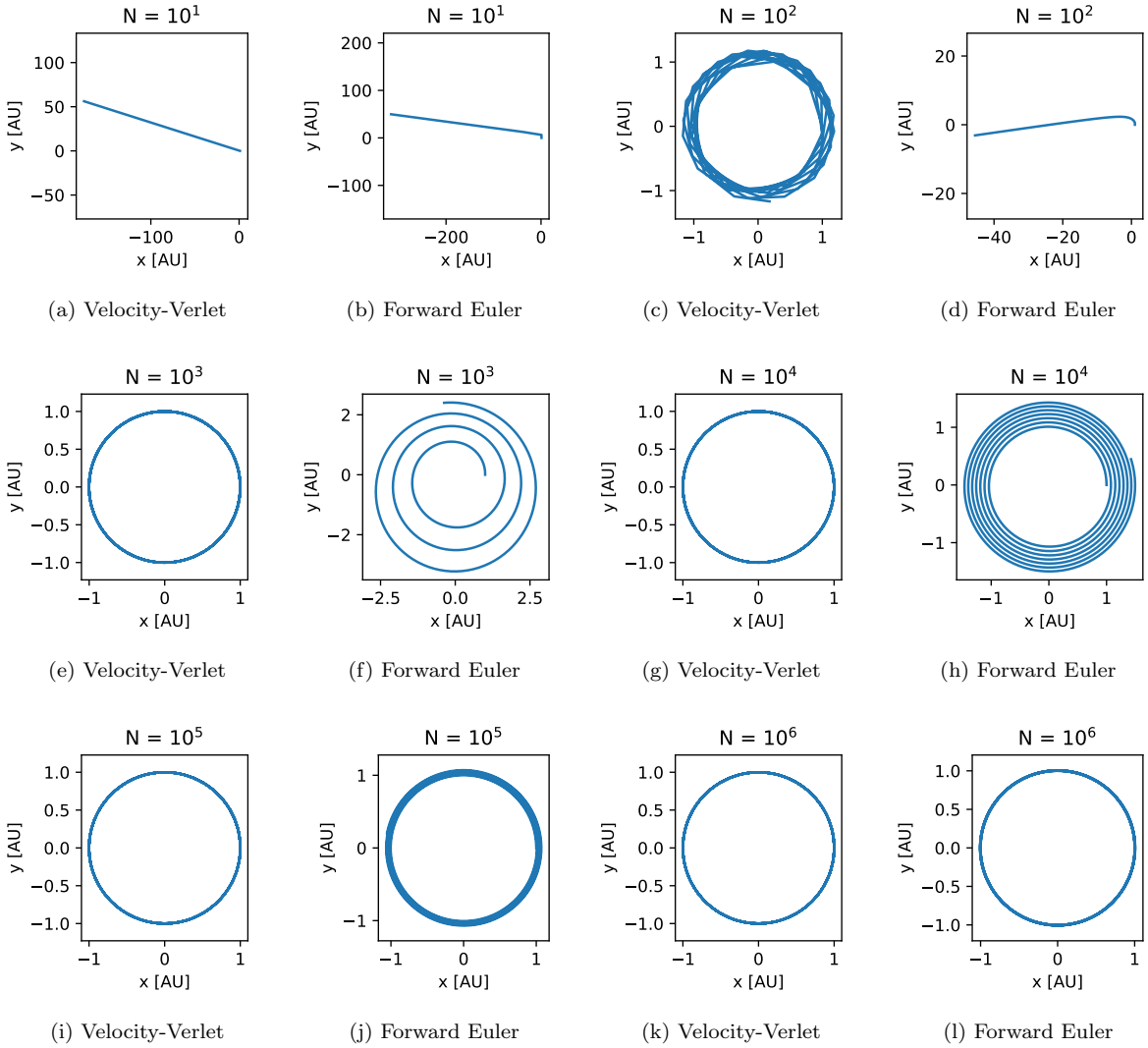
### Object orientation

When designing the class n_solver, i wanted to strike a balance between utilizing the benefits, and expandability you get from object orientation whilst also not abstracting the data too much. As such, the class simply manipulates arrays of a particular format in a particular way. Not abstracting the process by creating objects for each planet (or something else of that nature), which seems to be a pitfall of object orientation as i perceive it.

As such, the class can easily be expanded by adding new solvers and differential equations, to solve similar multi-body problems governed by a different set of coupled differential equations.

A particular benefit in the way i wrote my code, is that there is no difference if the input data is in 2 Dimensions or 3. The code will work just the same either way by making full use of the way Numpy arrays work.

I also made an attempt to utilize just-in-time (JIT) compilation by using jitclass from the Numba package, however, jitclass is poorly documented in its current state and imposes many restrictions as to what is permitted compared to regular python, or even the standard jit flag that is used for regular python functions. Eventually, i reached the point where i would have to sacrifice the modularity of my program in order to run it using jitclass, which obviously was not an option. So i gave up on the endeavor.

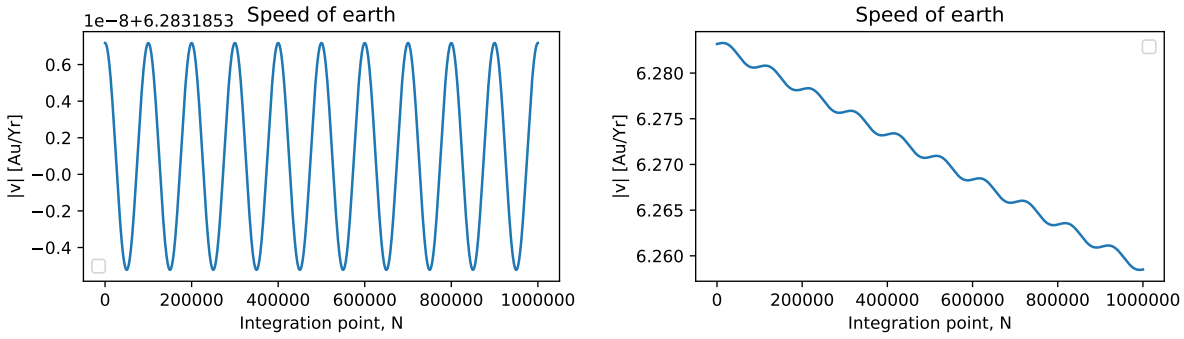



(a) Velocity-Verlet (b) Forward Euler (c) Velocity-Verlet (d) Forward Euler

(e) Velocity-Verlet (f) Forward Euler (g) Velocity-Verlet (h) Forward Euler

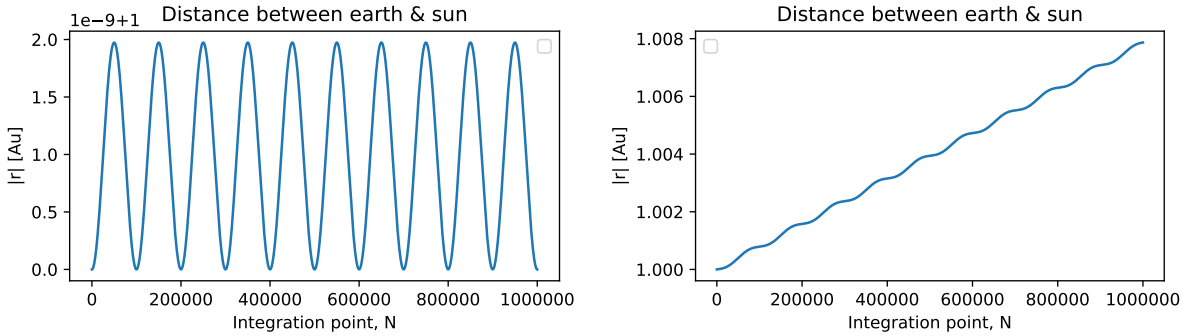
(i) Velocity-Verlet (j) Forward Euler (k) Velocity-Verlet (l) Forward Euler

FIG. 2: The orbit of earth around a stationary sun for a timeperiod of 10 Years with simulated with a varying number of integration points N (see fig titles) using the Velocity-Verlet and Forward Euler algorithms




(a) Velocity-Verlet (b) Forward Euler

FIG. 3: The speed of Earth in the Earth-Sun system for solutions using the Velocity-Verlet algorithm (a) and the Forward Euler algorithm (b) in a 10 Year simulation and $N = 10^6$ integration points




(a) Velocity-Verlet

(b) Forward Euler

FIG. 4: The radius of Earth in the Earth-Sun system for solutions using the Velocity-Verlet algorithm (a) and the Forward Euler algorithm (b) in a 10 Year simulation and $N = 10^6$ integration points

### Earth-Sun System

Working in units $M_\odot, AU, Years$ for mass, length and time respectively, i initialized a 2D system using the n_solver class where the sun is fixed at the origin and the earth has initial velocity $\mathbf{v}_0 = (0, 2\pi)$ at position $\mathbf{r}_0 = (1, 0)$, corresponding to an orbit of eccentricity, $e = 0$, a perfect circle.

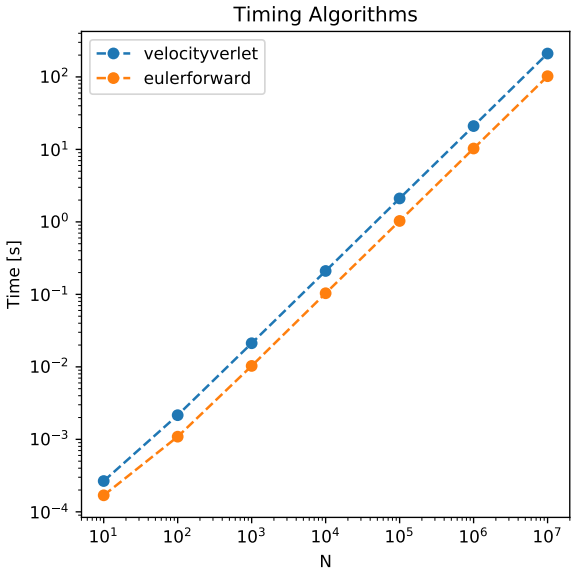



FIG. 5: Time taken to solve the Earth-Sun system for different number of integration points for $t_n = 10\,Yr$

The system was then simulated for a time period of 10 years using a different number of integration points for both the Velocity-Verlet and Forward Euler algorithms. The resulting orbits shown in Fig. 2, shows that the Velocity-Verlet solution yields a reasonable approximate orbit for $N = 10^3$ integration points, whilst the Forward Euler algorithm doesn't produce a comparable orbit until $N = 10^6$ integration points. In Fig. 5 we see the time it took for my program to solve this system for a selection of $N$. These numbers apply of course, only to the system with a fixed sun with one planet orbiting, as calculating the acceleration on more than one planets adds to the number of FLOPS.

In Figures 3, 4 we see how the speed and radius evolves throughout the integration points for $N = 10^6$, and most notably, how they change. As mentioned prior, for the set of initial conditions used, we expect a perfectly circular orbit, but also constant speed. Looking at Figures 3b, 4b, corresponding to the Forward Euler solution, that the earth is slightly drifting away from the sun, whilst losing speed, which further implies that both the potential and kinetic energy in the system has not been conserved[1]. Now we turn to Figures 3a, 4a, corresponding to the Velocity-Verlet solution, where we see a periodic, but seemingly constant behavior of both the radius and speed, meaning that on average the energy of the system is conserved. Since the angular momentum is proportional to the product of the speed and radius, it follows that it is conserved and not conserved for the Verlet and Euler algorithms respectively.

*Escape Velocity*

### 3-Body problem

Fetching initial conditions from the Horizons[2] system hosted by JPL at NASA, i simulated the trajectory of Earth and Jupiter orbiting around a fixed sun. As the

[1] Because $E_k \propto v^2$ and $E_p \propto -\frac{1}{r}$

[2] https://ssd.jpl.nasa.gov/horizons.cgi

data is in 3 dimensions, this also proved a great time to test flexibility of n_solver, which as expected worked just as well when fed 3D data, without having to make any adjustments. The trajectory for a 30 Year simulation with $N = 10^6$ integration points is presented in Fig 6. Pluto, and other smaller celestial objects could very easily be added thanks to a script i wrote, get_initconds.py, which fetches initial conditions using the Astroquery python package. However, i opted not to simply because it adds clutter to figure.

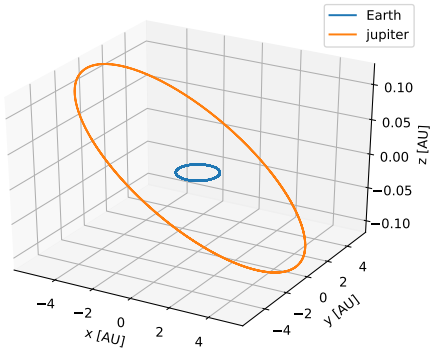


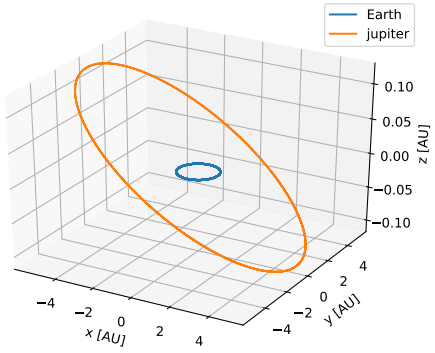

FIG. 6: Trajectory of Earth and Jupiter with the sun fixed at the center for 30 years with $N = 10^6$ integration points.

The resultant orbits are seemingly uniform, and stable within the time frame of the solution.

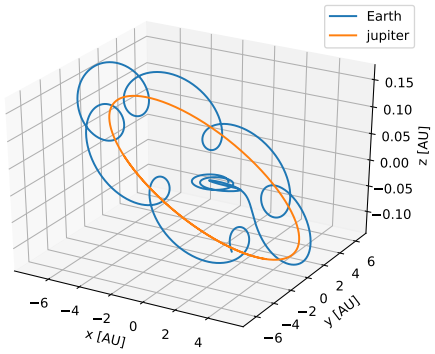
Now, we will look at what the system would look like if the mass of Jupiter was increased by a factor 10, and 1000, presented in Fig. 7. We see that the resultant trajectory with the mass of Jupiter increased by a factor 10 looks much the same as what we saw in Fig. 6, but increasing the mass by a factor 1000 throws the system out of control and eventually slingshots the earth far away from the system, which can be seen in Fig. 9, after the main text has ended in the appendix. Whilst a fun exercise, it is important to note that this simulation is not necessarily physically realistic, not because of the instantaneous mass increase of Jupiter, but because the planets are treated as point particles, which only interact via their gravitational attractions. A more realistic outcome of this scenario would be that earth crashes into either the sun and Jupiter in a terribly violent fashion.

### Simulating the Solar system

As a final example of simulations using Newtonian gravity, i present a simulation of all the major bodies




(a) $10M_{jupiter}$, $t = 30$ Yr




(b) $10^3 M_{jupiter}$, $t = 15$ Yr

FIG. 7: Trajectories for earth and jupiter in the 3-body system with fixed sun at the origin, with the mass of jupiter multiplied by a factor 10 (a) and $10^3$ (b) solved with $N = 10^6$ integration points

in the solar system in motion about the barycenter[3] in Fig 8, once again having fetched the initial conditions from the Horizons system.

### The perihelion precession of Mercury

In order to test the relativistic correction proposed in Eqn. 5 i extended n_solver by adding a method that solved the 2-body problem with a stationary sun. I then initialized the system with mercury at its perihelion, 0.3075 AU away from the sun with speed 12.44 AU $\mathrm{Yr}^{-1}$ [3]. I simulated the system for 100 years with $N = 10^8$ integration points, which took my computer 7131.14 s, just under 2

[3] Center of mass in an n-body system

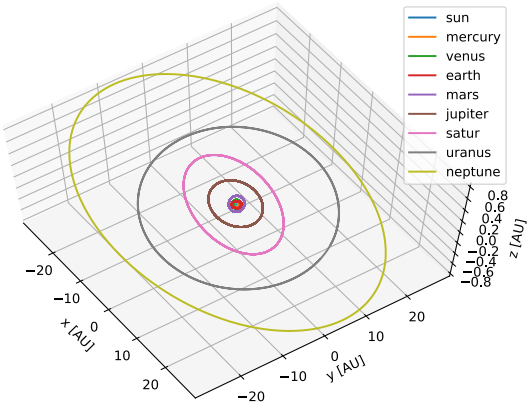



FIG. 8: Simulation of the major bodies within the solar system for 165 Years and $N = 10^6$ integration points using initial conditions from Horizons

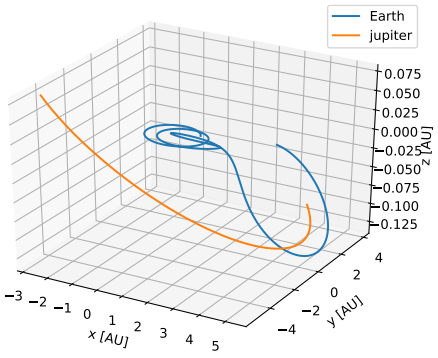
hours to solve. I then found the radius between the sun and mercury for all $t$ and looped backwards, finding the first minima. Having found the index in which the mercury was previously at its perihelion I then found that the angle of precession to be 5.09 rad, or 10.51", significantly lower than what is observed experimentally.

This could either be due to the simplification in the model, not including the other planets in the system, or it could simply be an error in my implementation of the system. As this simulation takes such a significant amount of time to run on my computer, it simply isn't feasible for me to explore this problem much further.

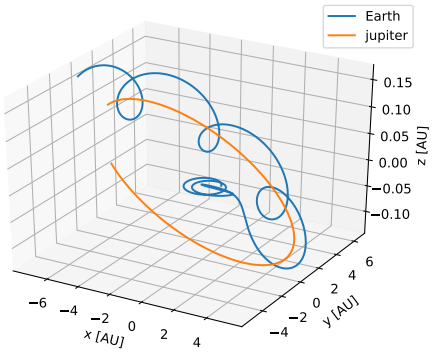
## CONCLUSIONS

[1] M. Hjorth-Jensen, Computational Physics - Lecture Notes 2015, (2015).
[2] M. Hjorth-Jensen, Ordinary differential equations - Computational Physics Lectures (2017)
[3] M. Hjorth-Jensen, Building a model for the solar system using ordinary differ- ential equations - Project 3 (2018)
[4] A. Malthe-Sørenssen, Elementary Mechanics Using Python (2015)

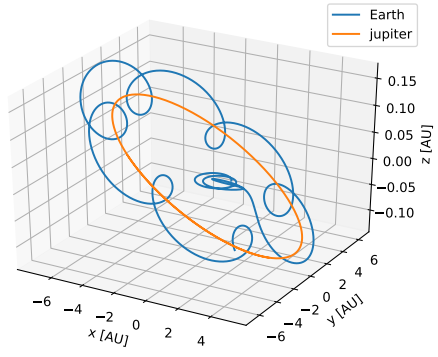
### Earth goes on a journey



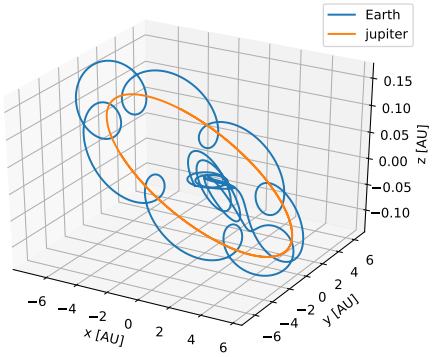

(a) t=5 years



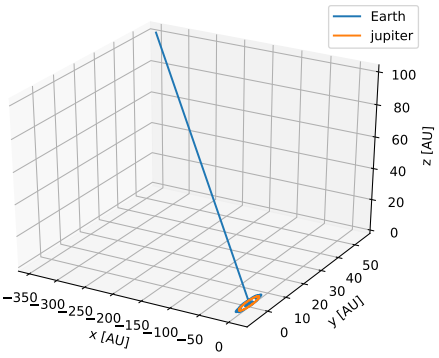

(b) t=10 years



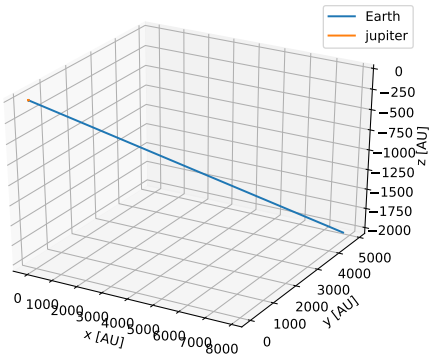

(c) t=15 years




(d) t=20 years




(e) t=20 years




(f) t=20 years

FIG. 9: Trajectories of Earth and Jupiter with the sun fixed at the center and jupiters mass multiplied by $\times 1000$ at different times, simulated with $N = 10^6$ integration points

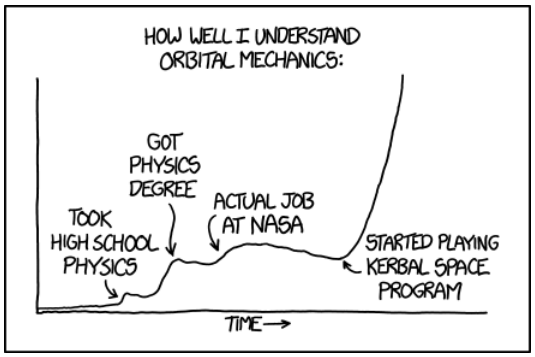



FIG. 10: To wrap things up i present to you a highly relevant xkcd comic