

FYS3150 Computational Physics - Project 3

Nicholas Karlsen

Looking at how n-coupled differential equations can be solved numerically using the Velocity-Verlet and Forward Euler algorithms for systems governed by Newtonian gravity. Found that the Forward Euler method falls short compared to the Velocity-Verlet method as it is non-conservative and requires a greater number of integration points to converge in the given system. There was also an unsuccessful attempt at reproducing the perihelion precession of mercury using a modified model in a 2-body system.

INTRODUCTION

In this report we will take a look at two methods for solving Ordinary differential equations numerically, which is of great interest in physics as a lot of physical systems are governed by such equations, most of which lack closed form 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 well as looking at a slightly altered model, which accounts for relativistic effects, this time restricting ourselves to a 2-body system.

Therefore, 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 their separation, or more precisely, the force acting on some body with mass m due to a mass m' is

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

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

Choosing 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

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

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

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

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

This force also forms a vector field, and the associated potential energy due it is

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

The derivation of which is given in [3], and elsewhere. From this, we can form an equation for the escape velocity of a body "trapped" in a gravitational field. Which is the velocity in which the Kinetic energy is greater or equal to the potential energy, that is

$$\frac{1}{2}mv^2 = G \frac{mm'}{r} \quad (5)$$

Which leads to the expression

$$v_{escape} = \sqrt{\frac{2Gm'}{r}} \quad (6)$$

A special solution to the differential equation 1 is for the perfectly circular orbit, for which we can use the equations of circular motion and get exact analytical predictions very easily.

$$\frac{v^2}{r} = G \frac{m'}{r^2} \quad (7)$$
$$v = \sqrt{\frac{Gm'}{r}}$$

where in this case, the speed v , as well as r are expected to be constant quantities.

Relativistic Correction

The aforementioned model of gravity fails to predict the perihelion (see fig. 1) precession of mercury, which is observed to be 43" (arc seconds) per century [2].

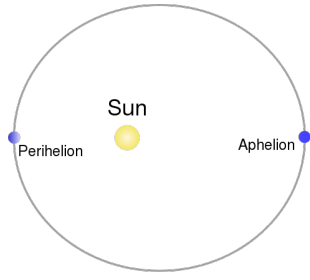


FIG. 1: In an elliptic orbit, the closest and farthest points in the orbit is defined as the Perihelion and Aphelion respectively [Image source]

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 [2]

$$|\mathbf{F}| = G \frac{mm'}{r^2} \left[1 + \frac{3l^2}{r^2 c^2} \right] \quad (8)$$

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 $f(t + \delta t)$

$$f(t + \delta t) = f(t) + f'(t)\delta t + \frac{1}{2}f''(t)\delta t^2 + \dots + \frac{1}{n!}f^{(n)}(t)\delta t^n \quad (9)$$

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

$$f(t + \delta t) = f(t) + f'(t)\delta t + \mathcal{O}(\delta t^2) \quad (10)$$

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

If $f(t)$, $f'(t)$ 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 $\mathbf{v}(0) \simeq \mathbf{v}_0$, $\mathbf{r}(0) \simeq \mathbf{r}_0$.

Forward Euler Algorithm (4N FLOPS)

```

1  for  $i = 0, \dots, N - 1$ 
2       $\mathbf{v}_{i+1} = \mathbf{v}_i + \mathbf{a}_i \Delta t$ 
3       $\mathbf{r}_{i+1} = \mathbf{r}_i + \mathbf{v}_i \Delta t$ 
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, but is listed below for the sake of comparison.

Euler-Cromer Algorithm (4N FLOPS)

```

1  for  $i = 0, \dots, N - 1$ 
2       $\mathbf{v}_{i+1} = \mathbf{v}_i + \mathbf{a}_i \Delta t$ 
3       $\mathbf{r}_{i+1} = \mathbf{r}_i + \mathbf{v}_{i+1} \Delta t$ 
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 [1] for the full derivation and simply state the result

$$\begin{aligned}
 f(t + \delta t) &= f(t) + f'(t)\delta t + \frac{f''(t)\delta t^2}{2} + \mathcal{O}(\delta t^3) \\
 f'(t + \delta t) &= f'(t) + \frac{(f''(t + \delta t) + f''(t))\delta t}{2} + \mathcal{O}(\delta t^3)
 \end{aligned} \quad (11)$$

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

Velocity-Verlet Algorithm (10N FLOPS)

```

1  for  $i = 0, \dots, N - 1$ 
2       $\mathbf{r}_{i+1} = \mathbf{r}_i + \mathbf{v}_i \Delta t + \frac{1}{2}\mathbf{a}_i(\Delta t)^2$ 
3       $\mathbf{v}_{i+1} = \mathbf{v}_i + \frac{1}{2}(\mathbf{a}_{i+1} + \mathbf{a}_i)\Delta t$ 
4
```

Take note that the number of FLOPS given for each algorithm does is not account for any computation required to compute the acceleration. For n-body systems in particular, the Velocity-Verlet algorithm can become quite expensive, as the acceleration would have to be computed twice per loop, compared to the Euler methods which only requires this calculation to be done once.

RESULTS AND DISCUSSIONS

Earth-Sun System

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 of that sort, which seems to be a pitfall¹ 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 3². 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`³. 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.

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.

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 particular system with a fixed sun and one orbiting planet, as calculating the acceleration on more than one planets adds to the number of FLOPS.

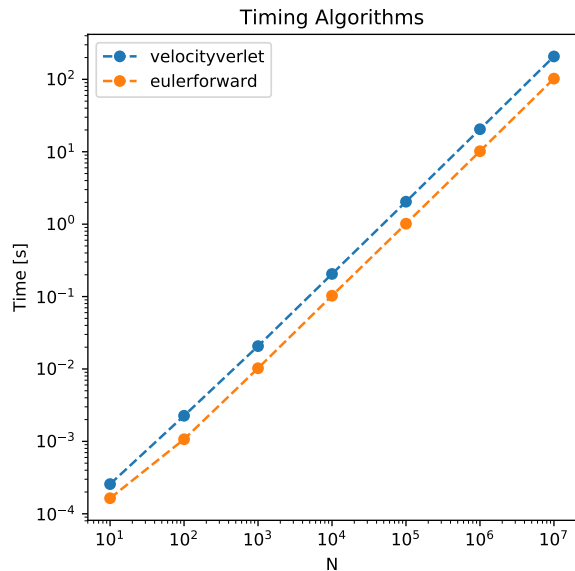


FIG. 5: Time taken to solve the Earth-Sun system for different number of integration points for $t_n = 10 Yr$

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 con-

¹ Within the context of scientific computation

² Technically, it would work for any higher dimensions as well, provided the shape of the input arrays are compatible.

³ Which does not work for classes

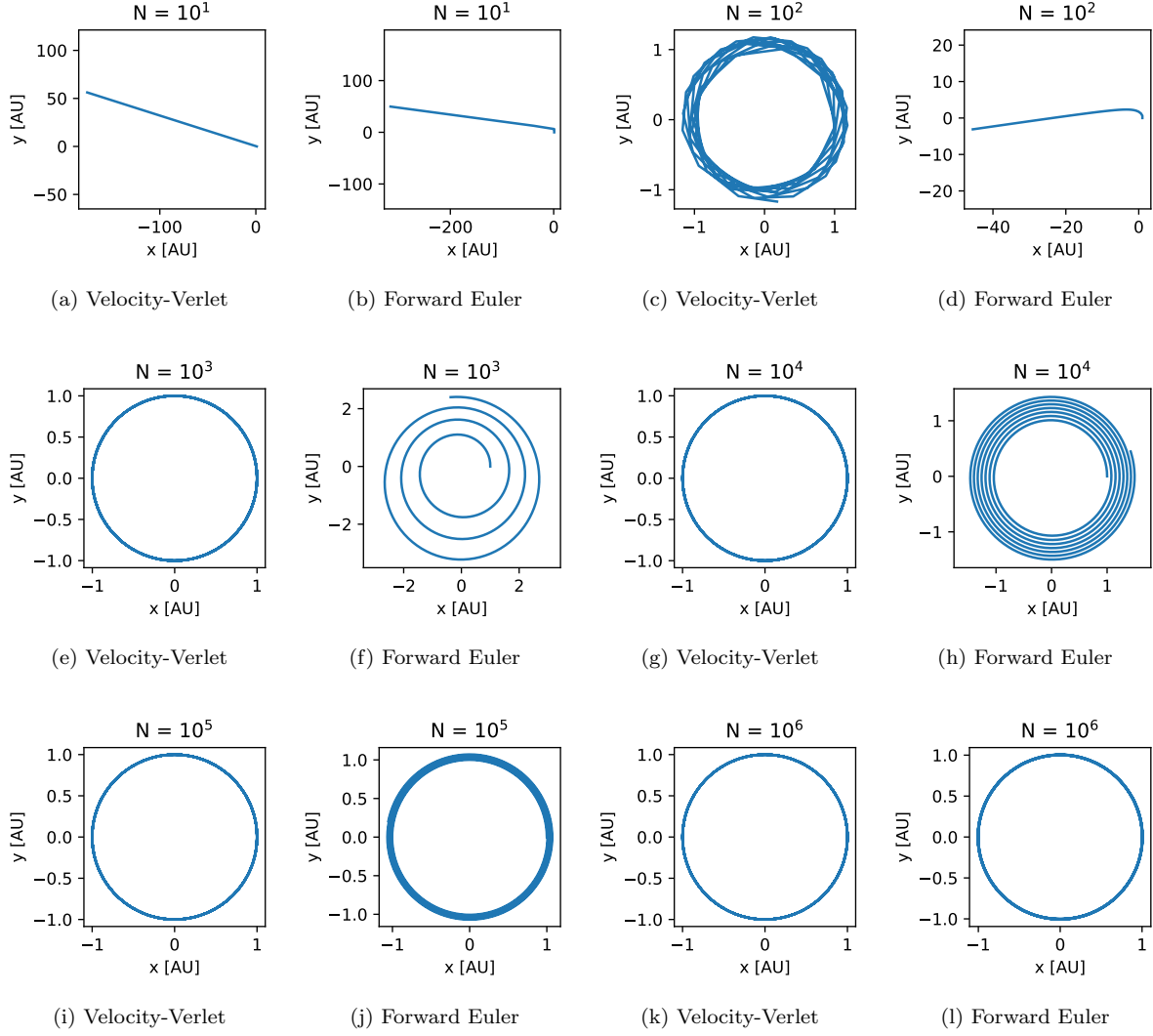


FIG. 2: The orbit of earth around a stationary sun for a timeperiod of 10 Years with simulated with a varying number of integration points N (see fig titles) using the Velocity-Verlet and Forward Euler algorithms

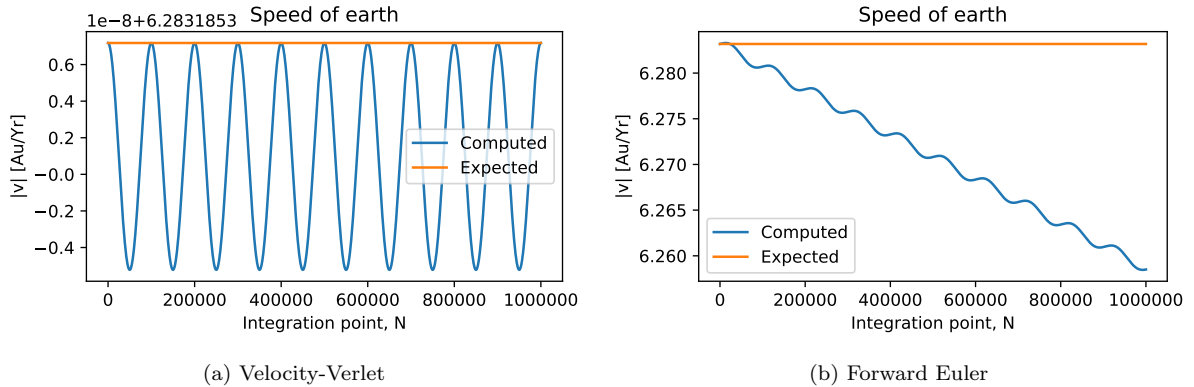


FIG. 3: The speed of Earth in the Earth-Sun system for solutions using the Velocity-Verlet algorithm (a) and the Forward Euler algorithm (b) in a 10 Year simulation and $N = 10^6$ integration points

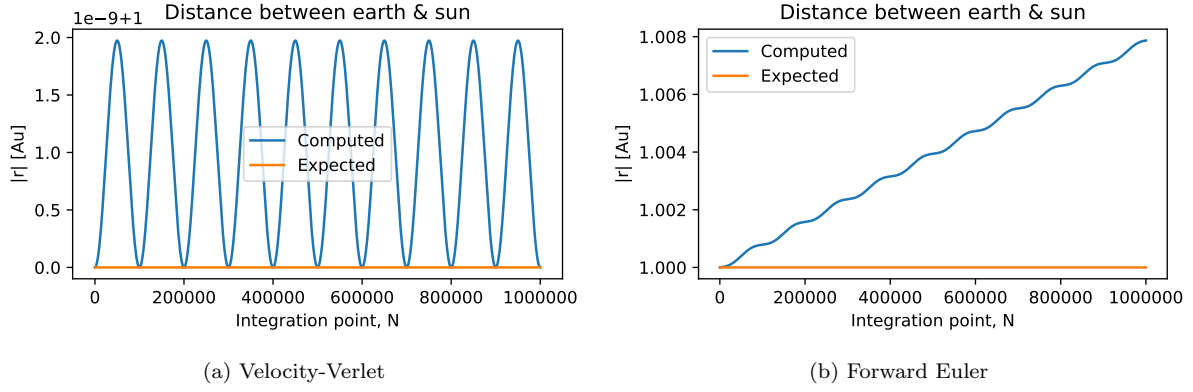


FIG. 4: The radius of Earth in the Earth-Sun system for solutions using the Velocity-Verlet algorithm (a) and the Forward Euler algorithm (b) in a 10 Year simulation and $N = 10^6$ integration points

served⁴. 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.

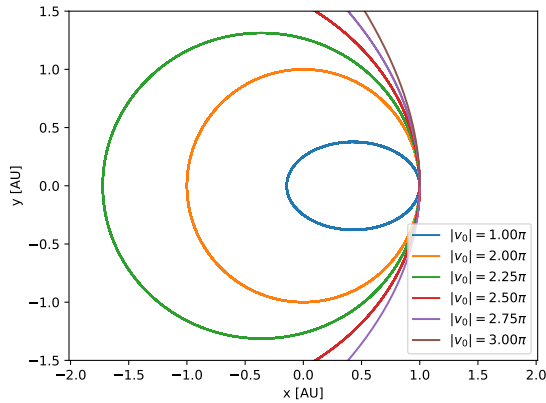


FIG. 6: The trajectories of Earth in the Earth-Sun system for different initial velocities and $N = 5 \cdot 10^5$ integration points over a 50 year simulation.

Escape Velocity

In Fig. 6 we see the trajectories resultant of different initial speeds, $|v_0|$. See that the only orbit which isn't closed is that for $|v_0| = 3\pi$, meaning the escape velocity is

in the interval $[2.75\pi, 3\pi]$. From Eqn. 6 we find that the theoretical escape velocity is $\sqrt{8}\pi$, matching our result. Of course this is rather trivial, so i now point you to Fig. 7, in which we see what happens to the same problem in a universe where the force of gravity is proportional to $r^{-\beta}$, $\beta \in [2, 3]$, instead of $|\mathbf{F}| \propto r^{-2}$. What is very interesting to note, is that $|v_0| = 2\pi$ seemingly remains a stable orbit in the selected values for which i simulated, whilst all other 'surrounding' change drastically, taking particular note of the interesting pattern that emerges from $\beta = 2.5$, $|v_0| = 1\pi$.

3-Body problem

Fetching initial conditions from the Horizons⁶ system hosted by JPL at NASA, i simulated the trajectory of Earth and Jupiter orbiting around a fixed sun. As the 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 to the way in which the code is called. The trajectory for a 30 Year simulation with $N = 10^6$ integration points is presented in Fig 8a. The resultant orbits are seemingly uniform, and stable within the time frame of the solution.

⁴ Because $E_k \propto v^2$ and $E_p \propto -\frac{1}{r}$

⁵ Which is found after the main text, because figure placement isn't always the easiest when producing a document in revtex

⁶ <https://ssd.jpl.nasa.gov/horizons.cgi>, for which astroquery provides an easy way to import data into python, further simplified by my script `get_initconds.py`

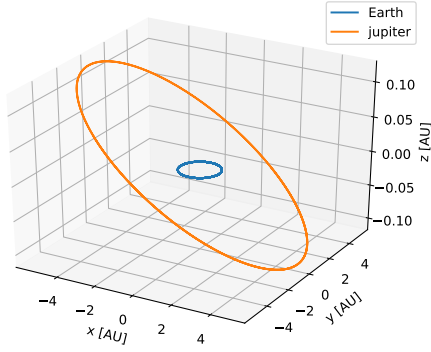
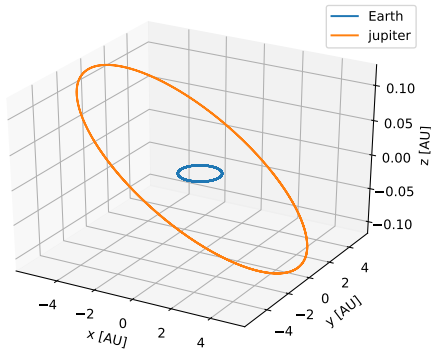
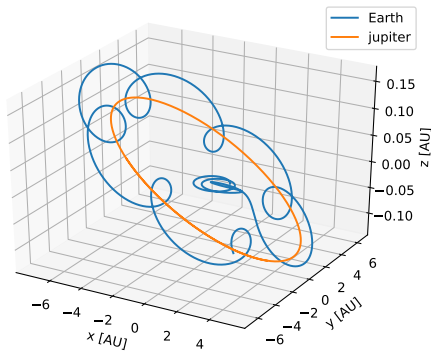
(a) $M_{jupiter}$, $t = 30$ Yr(b) $10M_{jupiter}$, $t = 30$ Yr(c) $10^3 M_{jupiter}$, $t = 15$ Yr

FIG. 8: 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 (b) and 10^3 (c) solved with $N = 10^6$ integration points

Now, we will look at what the system would look like if the mass of Jupiter was increased by a factor 10, and 1000, also presented in Fig. 8. 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. 8a, 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 ⁷Fig. 10. 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 gravity. A more realistic outcome of this scenario would be that earth crashes into either the sun and Jupiter in a horrid, and violent fashion.

Simulating the Solar system

As a final example of simulations using Newtonian gravity, i present a simulation of all the major bodies in the solar system in motion about the barycenter⁸ in Fig 9, once again having fetched the initial conditions from the Horizons system. Pluto, and other smaller celestial objects could very easily be added thanks to the script i wrote, `get_initconds.py`, which fetches initial conditions using the Astroquery python package. However, i opted not to, simply because it would add further "clutter" to a figure already lacking in any significant detail.

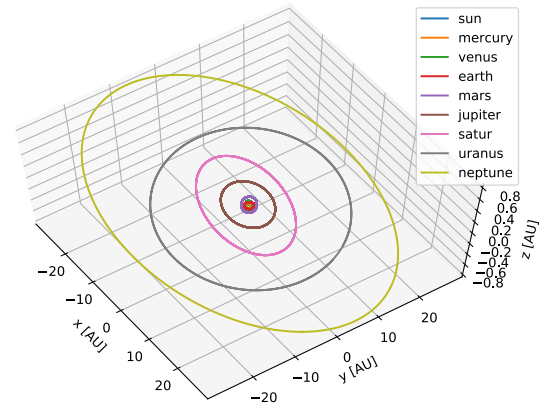


FIG. 9: Simulation of the major bodies within the solar system for 165 Years and $N = 10^6$ integration points using initial conditions from Horizons

⁷ Again, after the main text has ended

⁸ Center of mass in an n-body system

The perihelion precession of Mercury

In order to test the relativistic correction proposed in Eqn. 8 i extended `n_solver` by adding a method which solves 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 Yr^{-1} [2]. I simulated the system for 100 years with $N = 10^8$ integration points, which took my computer 7131.14 s, just under 2 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 be due to either 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

We have seen how the choice of numerical method when solving ODE's is of great importance when analyzing numerical results, as one might draw incorrect conclusions when inferring about the time evolution of the

energy of a system when using non-conservative methods, which the Forward Euler algorithm turned out to be in this problem. Further, whilst more computationally expensive per iteration, the Velocity-Verlet proves to be the cheaper method when accounting for its quick convergence when compared to the Forward Euler method.

Lastly, in regards to object orientation, i find that many of the supposed 'benefits' of this type of development simply does not apply to numerical computation, regardless of its usefulness in other scenarios. Abstracting what is otherwise simple, numerical computations which we as physicists are very much familiar is nonsensical in my opinion. If i were to define my own, special data-types to be operated on by numerical algorithms, that would only serve to decrease the flexibility of my algorithms by requiring to define new, similar data types which could be operated on in a similar fashion if i were to utilize the program to another problem.

Of course, the aforementioned problems does not strictly apply to my class⁹, `n_solver`, which should work effortlessly for different systems, given new methods for the governing forces.

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

⁹ which just as easily could have been implemented in a strictly

functional approach

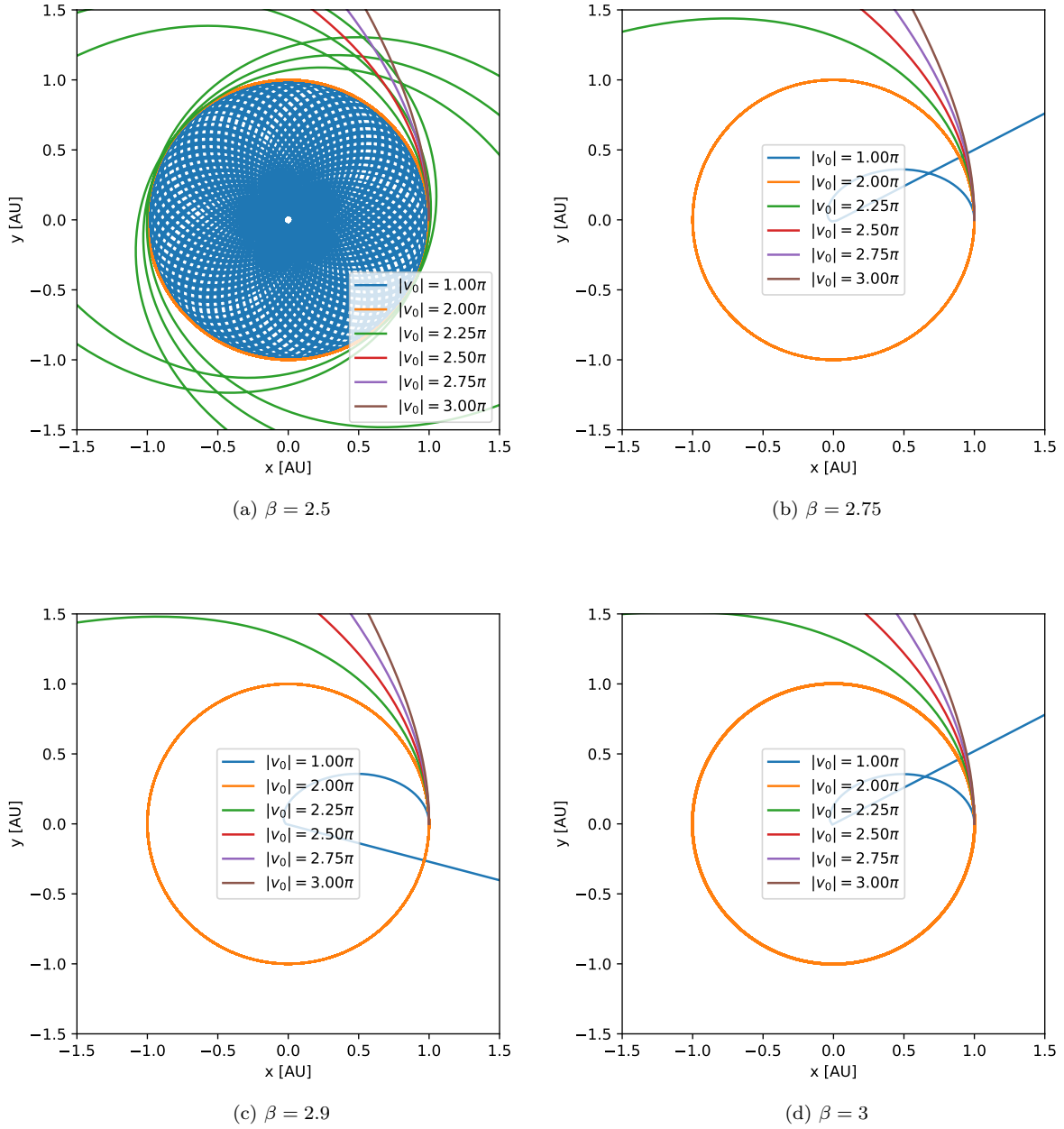


FIG. 7: Trajectories in the earth-sun system initialized 1 AU away from the sun with modified laws of gravity such that $|\mathbf{F}| \propto r^{-\beta}$

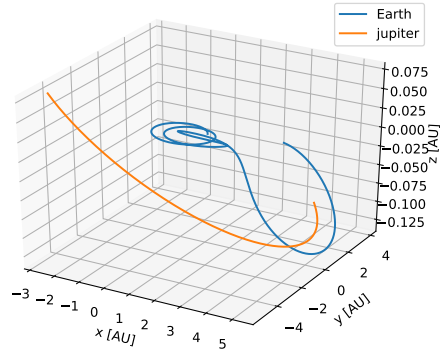
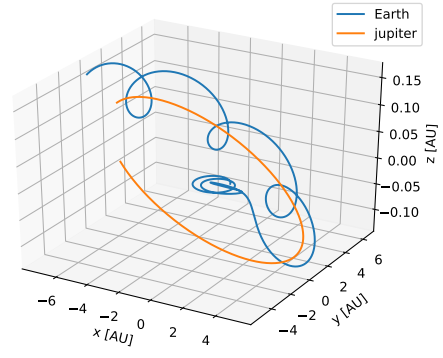
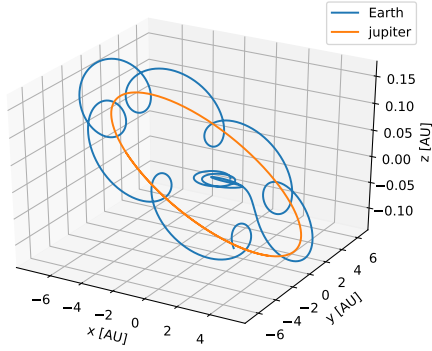
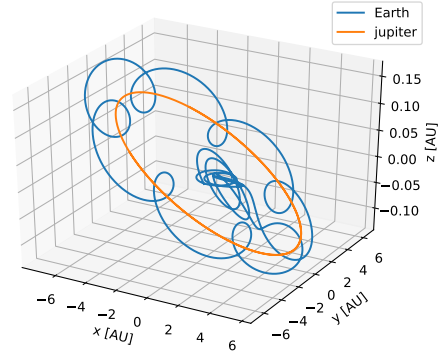
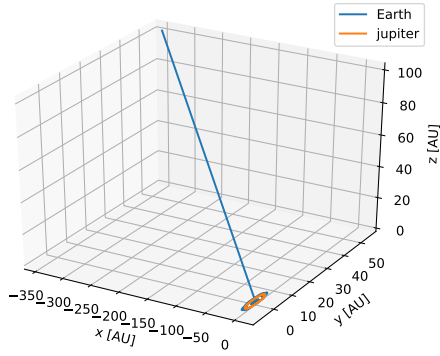
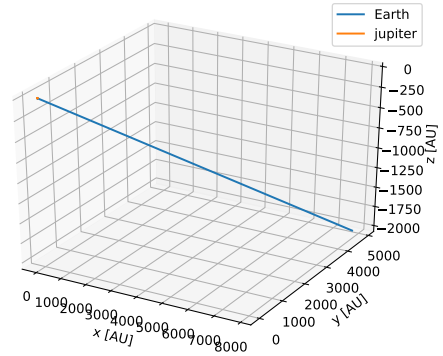
(a) $t=5$ years(b) $t=10$ years(c) $t=15$ years(d) $t=20$ years(e) $t=20$ years(f) $t=20$ years

FIG. 10: 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

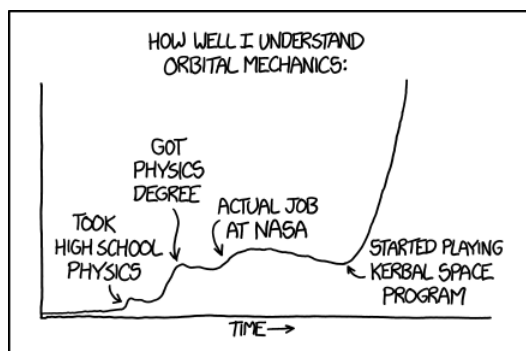


FIG. 11: To wrap things up i present to you a highly relevant xkcd comic