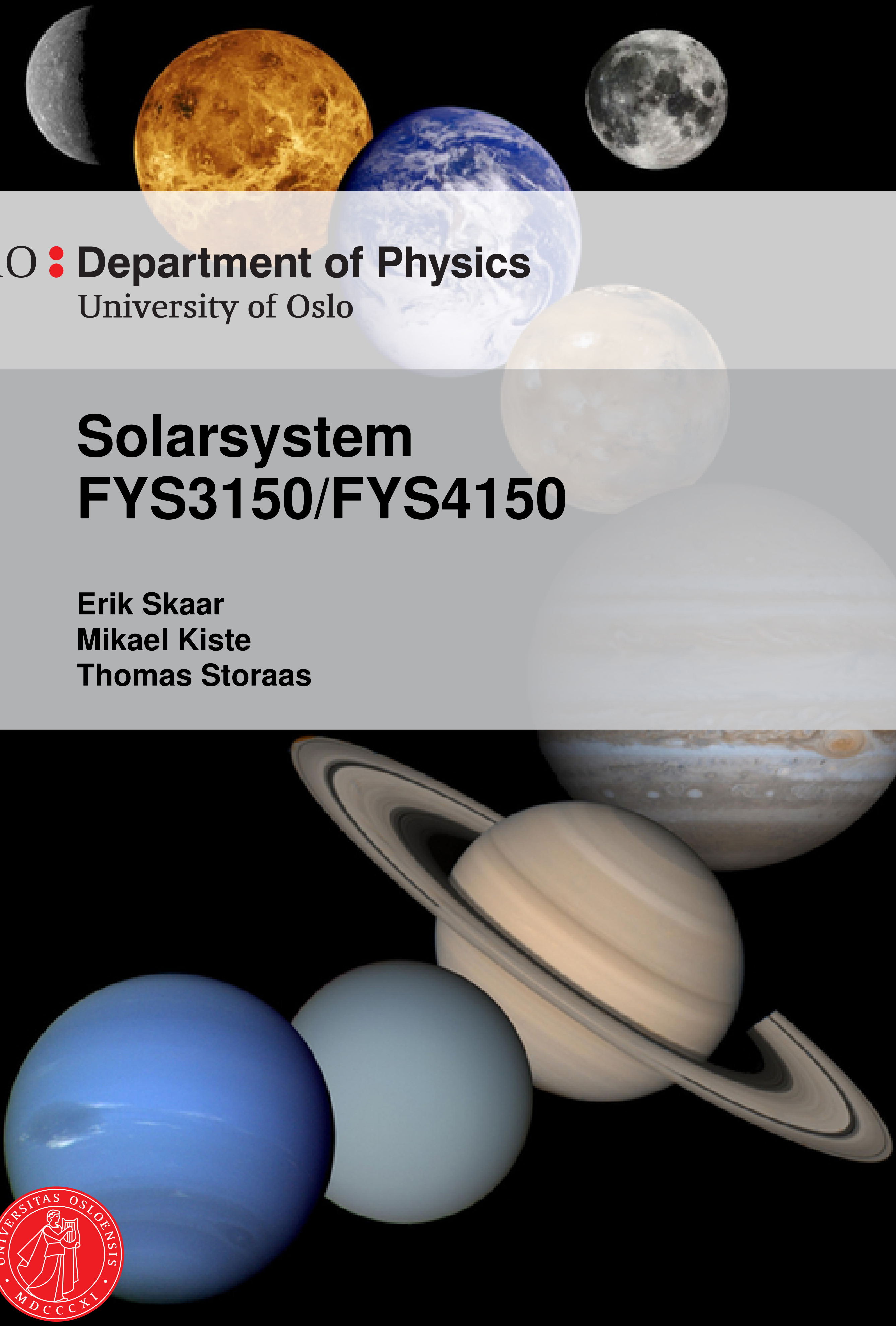
UiO : **Department of Physics**

University of Oslo

Solarsystem FYS3150/FYS4150

Erik Skaar
Mikael Kiste
Thomas Storaas



Innhold

Abstract

1 Introduction

For thousand of years humankind have looked up to the beyond and wondered. Specifically our race has wondered about the motion of our solar system. Finally after Newton the mystery was solved. Newton developed the gravitational law, which made it possible to predict the motion of the planets. After a couple of centuries Einstein came with the thoery of general relativity and made a small refinement to the law of motion that Newton proposed.

These laws are not enough to solve the motion of the planets. The laws makes differential equation which are not trivial or even impossible to solve. This is where a our computational physics course comes in . With the tools developed in this course we can make a prediction to the motion of the planets in our solarsystem.

REF KURSETS HJEMMESIDE

In this project we will make an object oriented code to solve the solar system with Forward Eulers method and Velocity Verlet. Both method will be derived, discussed, implemented and benchmarked for the Earth-Sun system.

WRITE ABOUT THEIR PERFORMANCE

Finally, a relativistic correction was made.......

NEED TO IMPLEMENT THIS AND GET FACTS

2 Theory

2.1 Gravitation

We will simulate the solar system with only the gravitational force affecting the planets. Newton's gravitational law is stated in equation (1). Where G is the gravitational constant ($6.67 \cdot 10^{-11}\mathrm{Nm}^2/\mathrm{s}^2$), r is the distance between the planets, m is the mass of the object and M is the mass of the other object.

$$\vec{F_G} = \frac{GmM}{|\vec{r}|^3}\vec{r} \tag{1}$$

Thankfully for us the force is an attractive force. It is worth noting that if the sun is at origo distance is simply the norm position vector of the planet. From now on we will denoted the different masses with $M_{planetname}$ except for the sun that has the special symbol $M_\odot$.

If you have n object attracting each other the total gravitational force, F_k, for each object is:

$$F_k = \sum_{i=1}^{N} \vec{F_i} = \frac{Gm_k m_i}{|\vec{r_k - r_i}|^3}(\vec{r_k} - \vec{r_i})(1 - \delta_{k,i}) \tag{2}$$

Where $\delta_{k,i}$ is the function:

$$\delta_{k,i} = \begin{cases} 1 & \text{if} \quad k = i \\ 0 & \text{if} \quad k \neq i \end{cases}$$

We can use Newton's second law to determine the accelration of the planet. Newton's second law state $F = m\vec{a}$. Which translate to:

$$a_k = \sum_{i=1}^{N} \frac{Gm_i}{|\vec{r_k} - \vec{r_i}|^3} (\vec{r_k} - \vec{r_i})(1 - \delta_{k,i}) \tag{3}$$

2.2 Units

As a famous person once said, Ïf you use seconds and meters you will not finish within the deadline. I think our professor, Morten Hjorth-Jensen, is on to something. Seconds and meters are impractical in this project. Therefore we use more suitable units. The distance between the earth and sun, 149 597 870 691 meter, is defined as one astronomical unit (au). For time one year seems reasonable. These are the units that will be used in this project.

We can now express the gravitational constant with these units. To do this we use the formula for the acceleration for a circular orbit is $\frac{v^2}{r}$. Combine this with the gravitational law, we get:

$$\frac{v^2}{r} = \frac{GM_\odot}{|\vec{r}|^3}\vec{r} \implies G = \frac{v^2 r}{M_\odot} = \frac{\text{au}(2\pi \cdot \text{au/year})^2}{M_\odot} = \frac{4\pi^2}{M_\odot}\text{au}^3/\text{year}^2$$

Applying this to equation (3), we get:

$$a_k = \sum_{i=1}^{N} 4\pi \frac{m_i}{M_\odot} \frac{(\vec{r_k} - \vec{r_i})}{|\vec{r_k} - \vec{r_i}|^3} (1 - \delta_{k,i}) \text{au}^3/\text{year}^2 \tag{4}$$

2.2.1 Escape velocity

Later in the project we are asked to find the escape velocity by trial and error. This question is basicly, when does the earth have enough kinetic energy to escape the potential. The potential is a integral from 0 to ∞ F_G over dr. For a given planet with only the sun in the solar system the integral is from r to ∞. This is shown in equation (6).

$$E_k = \int_r^\infty \frac{GmM}{|\vec{r}|^2} d\vec{r} \tag{5}$$

$$\tag{6}$$

Let's solve it:

$$E_k = \int_r^\infty -\frac{GmM_\odot}{|\vec{r}|^2} d\vec{r}$$
$$\frac{1}{2}mv^2 = GmM_\odot \int_r^\infty -\frac{1}{|\vec{r}|^2} d\vec{r}$$

G = 1, see section 2.2 for explanation.

$$\frac{1}{2}v^2 = M_\odot \int_r^\infty -\frac{1}{|\vec{r}|^2} d\vec{r}$$
$$\frac{1}{2}v^2 = M_\odot \left[\frac{1}{|\vec{r}|}\right]_r^\infty$$
$$v^2 = 2M_\odot \frac{1}{|\vec{r}|}$$

r = 1, see section 2.2 for explanation.

$$v = \sqrt{8\pi^2}$$
$$v \approx 8.8857\text{au/year}$$

2.3 prehelion

since this is not implemented in the code yet nothing is writen

2.4 Numerical methods

Equation (4) initial conditions for velocity and position determines the orbit of the planets. These equations are for x,y,z direction and when writen out you get a coupled set of differential equations. This set is near impossible to solve analytically. It might be impossible. We will use numerical methods to solve this set. More specifically we will use the Forward Euler method and the Velocity-Verlet method. A given t_i is equal to $t_0 + ih$, where $h = (t_0 + t_n)/n$.

2.4.1 Forward Euler

Both method use a Taylor polynomial approximation to solve the set of differential equation. The Forward Euler use the first order Taylor polynomial. With $r'(t) = v(t)$ and $v'(t) = a(t)$ the Forward Euler method result in:

$$\vec{r_i}(t+h) \approx \vec{r_i}(t) + h\vec{v_i}(t) \tag{7}$$
$$\vec{v_i}(t+h) \approx \vec{v_i}(t) + h\vec{a_i}(t) \tag{8}$$

Discretized version. i is the object number and j is the time step:

$$r_{i,\vec{j}+1} \approx r_{\vec{i},j} + h\vec{v}_{i,j}$$
$$v_{i,\vec{j}+1} \approx v_{\vec{i},j} + h\vec{a}_{i,j}$$

The error for a first order taylor polynomial goes as $\mathcal{O}(h^2)$ [**compphys**]. This is the error for each step. The error is accumulated for each step and will thereby be proportional to h. FIX REF

2.4.2 Velocity-Verlet

You guessed it, this is the second order Taylor polynomial. . As I said the Velocity-Verlet method is based on a Taylor polynomial approximation. And this is the second order.

$$\vec{r_i}(t+h) \approx \vec{r_i}(t) + h\vec{v_i}(t) + \frac{1}{2}h^2 a(t) \tag{9}$$
$$\vec{v_i}(t+h) \approx \vec{v_i}(t) + h\vec{a}(t) + \frac{1}{2}h^2 \vec{a}'(t) \tag{10}$$

Since we have no formula for the derivative of a, we use the approximation:

$$\vec{a}'(t) \approx \frac{\vec{a}(t+h) - \vec{a}(t)}{h}$$

We simply update the position first. We keep the old acceleration and calculate the acceleration at the new position. Using this equation (10) we get:

$$\vec{v_i}(t+h) \approx \vec{v_i}(t) + h\vec{a}(t) + \frac{1}{2}h(\vec{a}(t+h) - \vec{a}(t))$$
$$\vec{v_i}(t+h) \approx \vec{v_i}(t) + \frac{1}{2}h(\vec{a}(t+h) + \vec{a}(t))$$

Discretized version. i is the object number and j is the time step:

$$\vec{r}_{i,j+1} \approx \vec{r}_{i,j} + h\vec{v}_{i,j} + \frac{1}{2}h^2 a_{i,j}$$
$$\vec{v}_{i,j+1} \approx \vec{v}_{i,j} + \frac{1}{2}h(\vec{a}_{i,j+1} + \vec{a}_{i,j})$$

The error for a first order taylor polynomial goes as $\mathcal{O}(h^2)$ [**compphys**]. This is the error for each step. The error is accumulated for each step and will thereby be proportional to h^2. This is one order of magnitude better, then the Forward Euler method. FIX REF

3 Method

3.1 Implementation of code

The implementation is writen in c++ and is object oriented. We found it natural to divide the project in to two classes and one main program.

We made one class for the planet (planet.cpp and planet.h). This class has the variables for the position, velocity and mass for the specific planet. As well as a filename to write the output to. The class has methods for updating position, velocity and acceleration as well as methods for writing to a file.

The second class is for the solarsystem (solarsystem.cpp and solsystem.h). The solar system contain a list with all the planets and has methods to make the planets move and react to each other. You can look at the solar system as a class with for-loops for the planets.

The main.cpp program is where all the inputs to the solar system is given. Here all the initial condition for the planets are stated and organized for proper input to the solarsystem class.

All these programs are combined with makefile. When this is done the solsys.exe will be made. Takes in three arguments. The first argument is the number of planets that you would like to simulate. This is a preset list of the bodies in the solarsystem. Where the Sun is the first element, Earth is the second and Jupiter is the third. After that they ascend based on radius. The second argument is the end time measured in years from now. The start time is set to 19th of october 2017. This is the time that the intial values were obtained from NASA. Finally the third argument is the number of steps, n.

Finally, in our github repository there is directories were earlier versions can be found. For instance the Project3 directory is for where all the code is centered. In this directory there are many directories with different results files and python scripts for making figures. There are also a coupled of other main.cpp variants. These are for different assisgments in this project. It should be pretty self explanatory.

3.2 FLOPs for the algorithms

Let's talk FLOPs. The counting of the FLOPs are in the section 6, a.k.a. appendix. There we show the code and have comments that says how many flops each line are.

Common for both algorithms are calculation of the acceleration. The acceleration itself is 19 FLOPs. But for both algorithm this is called $n(n-1)$ times. After that the Forward Euler method calls functions to set position and velocity. This is done for all the planets. And for each planet it costs 6 FLOPs for the position and 6 more for the velocity. This gives a total of:

$$19n(n-1) + 12n = 19n^2 - 7n$$

For the Verlet-Velocity it is 21 FLOPs for the position and 12 FLOPs for the velocity (nothing is precalculated). This is of course done for all the planets and give us:

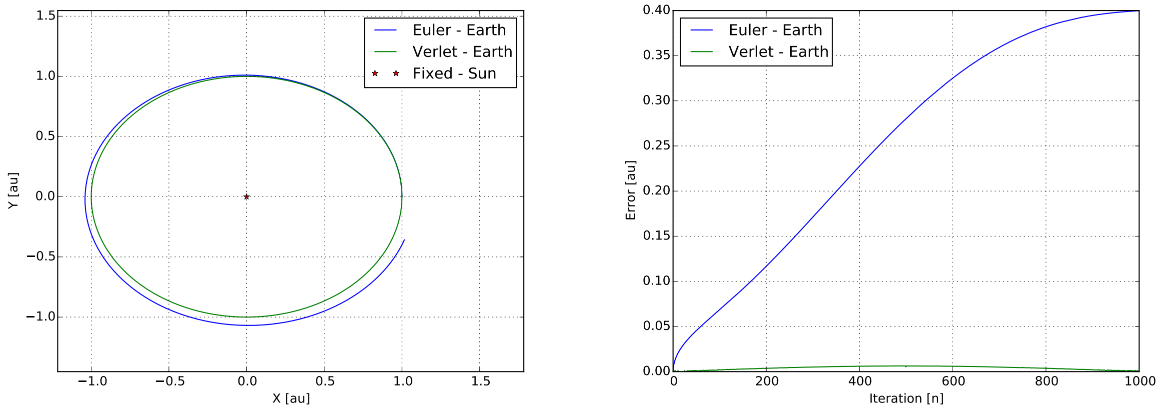
$$19n(n-1) + 33n = 19n^2 + 14n$$

So, this means that for 9 planets the verlet method has about 10 % more FLOPs, which is not much considering that much of the time are used on writing to file and getting variables and so on. The difference between the two algorithms should not be much.

4 Result & Discussion

4.1 Earth-Sun system

4.1.1 Stability



(a) (b)

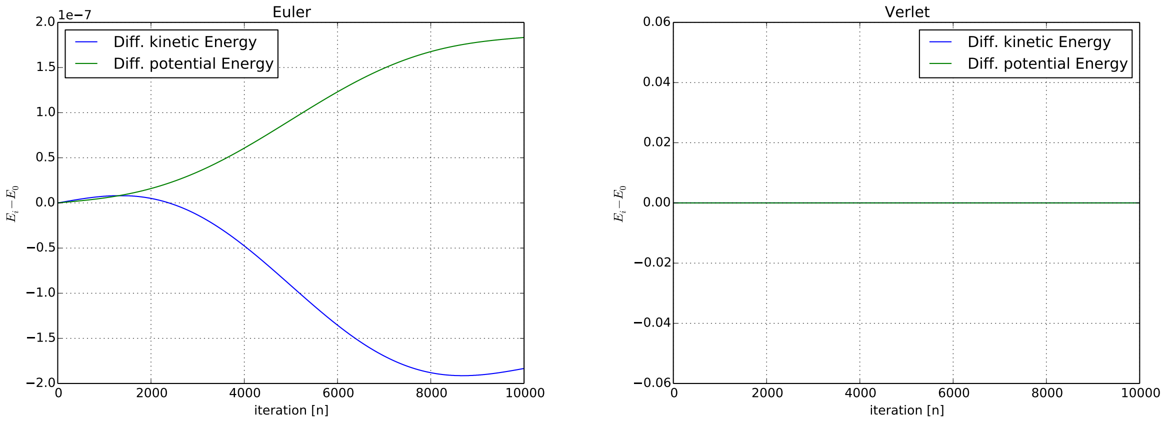
Figur 1: a) show the orbit of earth around the sound. The intial velocity is set to 2π in y direction and the start position to 1 au in x direction. b) shows how the error behaves. The intial values should give a perfect circular motion. So the error is calculated by $r_i - r_0$. It is clear that Verlet-Velocity method is superior. This simulation was with 1000 points with the end time of 1 year. Both simulations was produced by plot_earth_sun.py

Tabell 1: Time table for the different algorithms. The algorithms use nearly the same time. This is not a shocker since the number of FLOPs for the algorithms are similar. Time grows very linear as expected from section ??. Disclaimer: this is only the result from one test, but several was done. Both algorithms were very close and it seem to be random which is fastest.

n	Forward-Euler	Verlet-Velocity	fastest	$\frac{slowest}{fastest}$
10	0.000136	0.000148	Euler	1.08823529412
100	0.000208	0.000179	Verlet	1.16201117318
1000	0.000392	0.000389	Verlet	1.00771208226
10000	0.002427	0.002426	Verlet	1.00041220115
100000	0.022931	0.022293	Verlet	1.02861884897
1000000	0.167022	0.175944	Euler	1.05341811258
10000000	1.58721	1.52666	Verlet	1.03966174525
100000000	15.1786	15.1176	Verlet	1.00403503202

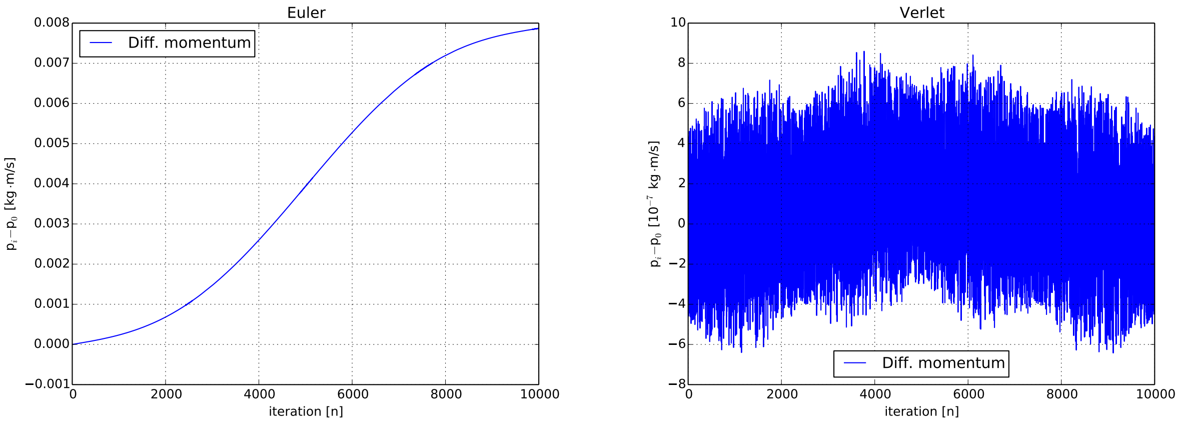
4.1.2 Conserved quantities

All the figures in this section was made from the data and python script in the directory conserved-values.



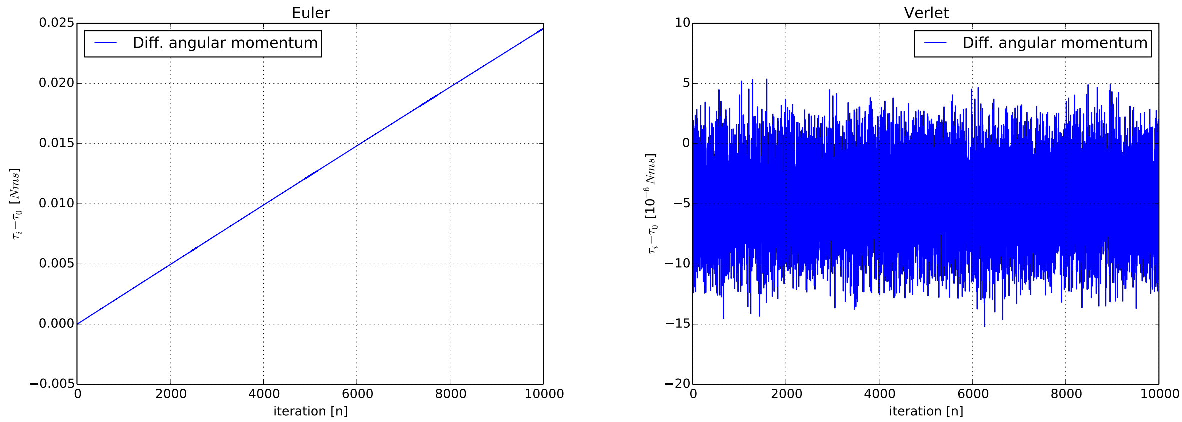
(a) (b)

Figur 2: Both are figures are graphs of the kinetic energy and potential energy and how it differ from they intial value. a) is the Forward Euler method and b) is the Verlet-Velocity method. As expected the energies are not conserved in the Forward Euler method, but is conserved in the Verlet-Velocity.



(a) (b)

Figur 3: Both are figures are graphs of the momentum and how it differ from they intial value. a) is the Forward Euler method and b) is the Verlet-Velocity method. It should come as no suprise that momentum is not conserved for the Forward Euler method as the kinetic energy was not conserved, as the mass is a constant. Once again the Verlet-velocity method conserve the quantity.

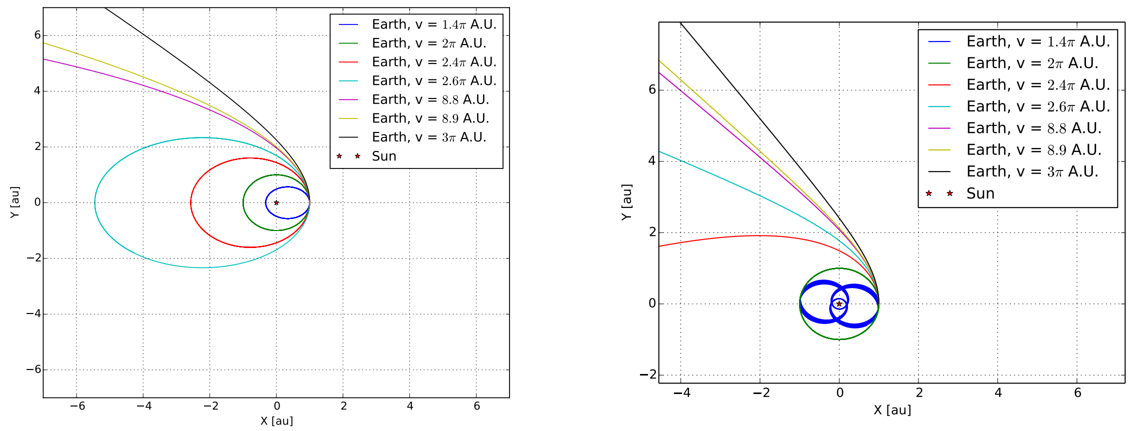


(a) (b)

Figur 4: Both are figures are graphs of the angular momentum and how it differ from they intial value. a) is the Forward Euler method and b) is the Verlet-Velocity method. Forward Euler is once again not capable of conserving the value, but luckily for us the Verlet-Velocity method is.

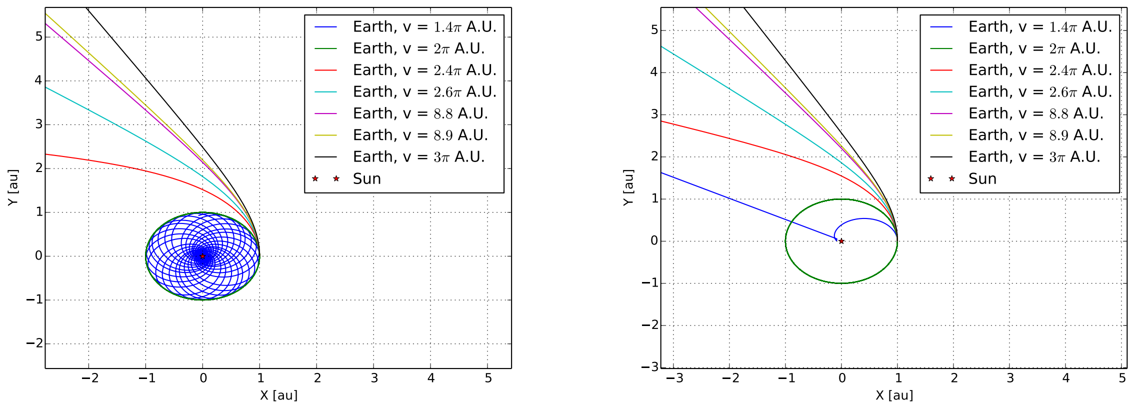
4.1.3 Escape velocity

The assignment was to find the escape velocity for the earth by trial and error. Fortunate for us that we know some math and can calculate it. See section 2.2.1 for this. But we started guessing ¨randomly¨(winking Face emoji). Figure (5) a) shows these guesses. Where we can see that the velocities around 8.8 au/year shots out and never returns. The algorithms only runs for 15 years and will thereby not see the 8.8 au/year return to orbit even tho it should. The plots were made by the data and python scripts in the directory escape-velocity.



(a) (b)

Figur 5: a) Show how the orbits of earths with different initial velocity are. b) Shows the same as a) but this time the dependency of r in the denominator in equation (1) is set to 3.5.



(a) (b)

Figur 6: a) Shows the same as figure (5) but this time the dependency of r in the denominator in equation (1) is set to 3.75. b) Shows the same as a) but this time the dependency of r in the denominator in equation (1) is set to 4.

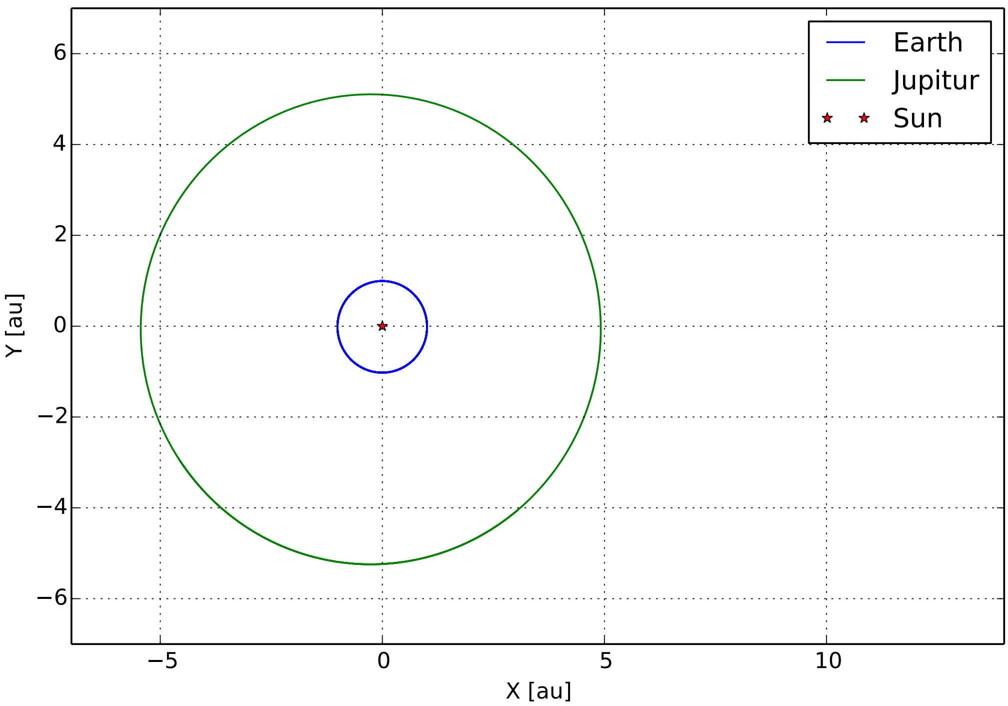
Personally I feel extremly lucky for living in a universe with a r dependency of 2, but then again i probably would not exist if the dependency was different. All the other dependencies are very unstable for even the slightest change in velocity from a perfect circle.

4.2 Three body system

All the figures in this section has been made from the directory mass jupitur. In this directory there are different directories for the r dependency and a python script to generate graphs. The assignment said to try with mass multiplied with 10 and 1000. It was fun, so we did 10,100,1000 and 1100. Hope you enjoy the results.

4.2.1 Fixed mass for jupitur

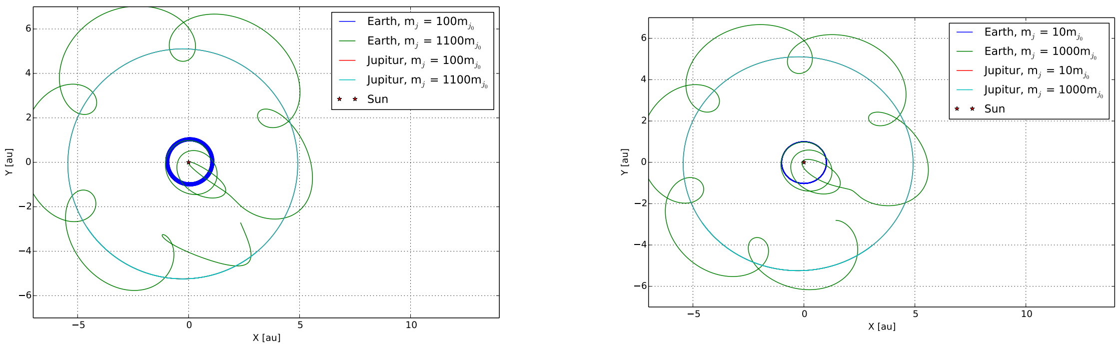
For a simulation with jupitur original mass 100000 points over 15 years is sufficient to calculate the orbits of the planets. The figure (7) is quit smooth and you should not expect to get any major change in the result even with many more points per year.



Figur 7: Plot of the three body system with Earth, Jupitur and the sun. The graph is discussed in the paragraph above.

4.2.2 Varying mass for jupitur

For varying mass it is mostly the same as for the original mass. For the mass multiplied with 10 and 100 the orbit is n̈ormalẅith some slight changes and you should not expect any major differences with higher amount of points per year. For the biggest masses the results vary much more on the number of steps per year. I found a couple of hundred thousand steps in total gave a good approximation. A bit higher step count makes earth disappear and a way bigger step count makes the earth come back to orbit like shown below. This is basicly the best of both worlds. The graphs is few point, so easy for python to plot, and has a pretty good approximation.



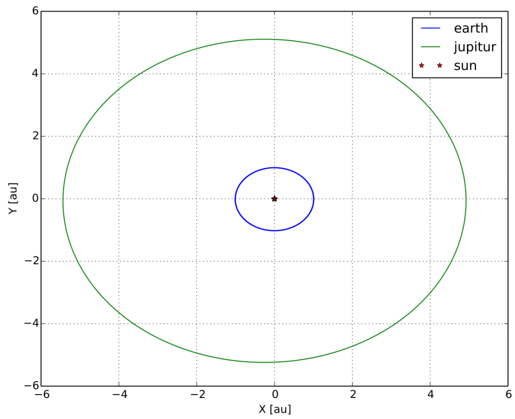
(a) (b)

Figur 8: Both plot has the sun as a fixed point. What the graphs represent is discussed above and legend should be pretty self explanatory.

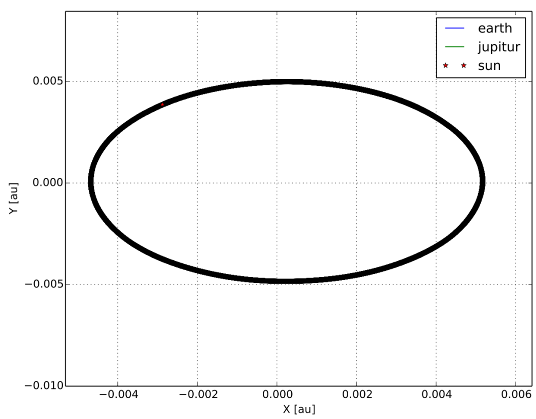
4.3 Solar system

All the figures in this section was made from the results and python scripts in the directory full-solarsystem.

4.3.1 Three planets and all moving



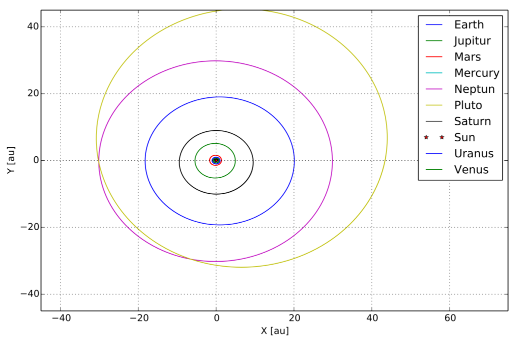
(a)



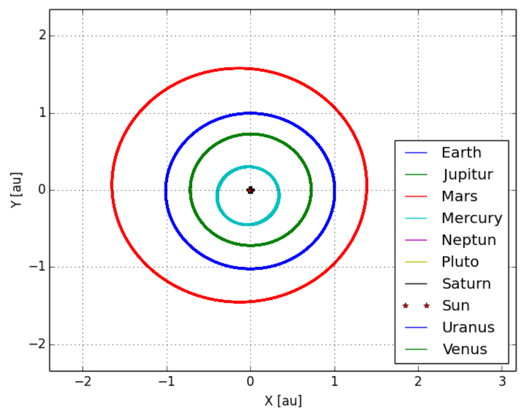
(b)

Figur 9: a) Shows the three body system solved with a moving sun. It has a n=10^5 and ran for 15 years. If this would be used for any tests I would recommend using more points. To use more points here would be stupid. You can't see the difference. b) is a zoom of the middle part of a). This is for verifying that the sun moves around the center of mass and is not drifting.

4.3.2 Solar system all moving



(a)

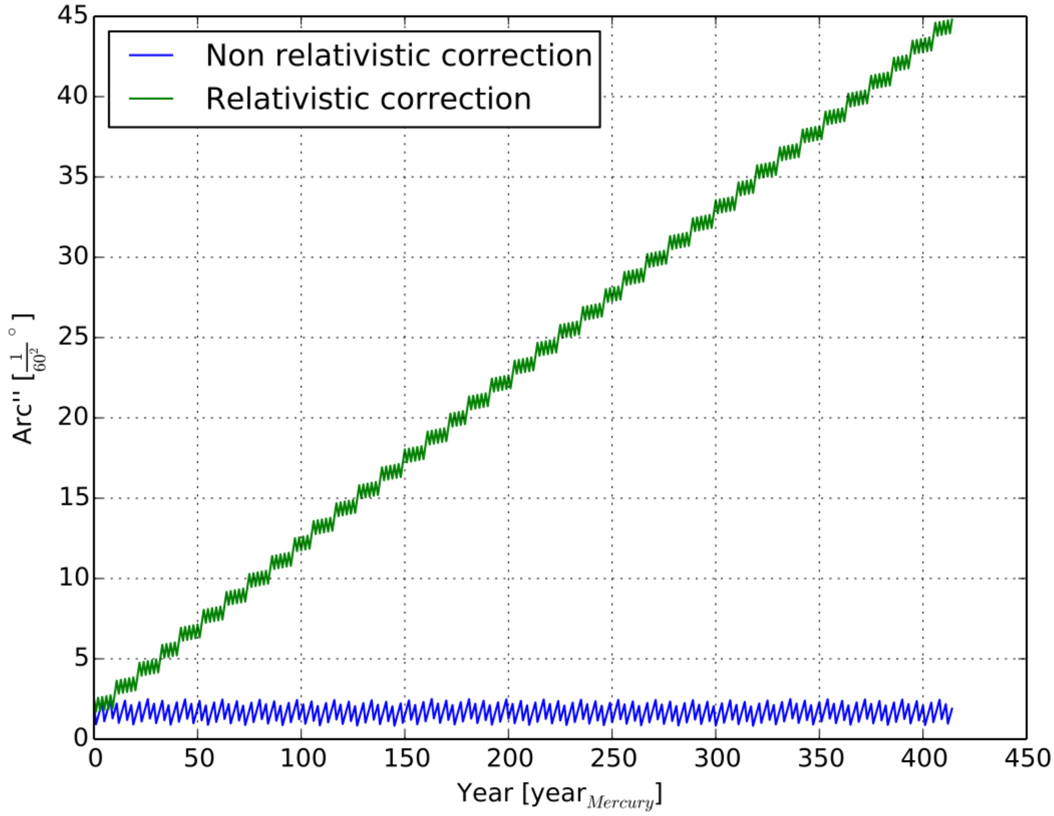


(b)

Figur 10: a) Shows the hole solar system solved with a moving sun. It has a n=10^6 and ran for 300 years. If this would be used for any tests I would recommend using more points, but this would be stupid for this figure, since you can't see the difference. b) is a zoom of the middle part of a). This is so you can see that they also move in orbits.

4.4 The perihelion precession of Mercury

This has been discussed in section 2.3. The figure in this section was made from the results and python script in the directory mercury-perihelion.



Figur 11: The assignment stated that after 100 years mercury perihelion should move about 43". From the graphs we can see that it should move about 45". This is probably duo to numerical inaccuracy. The slope's inaccuracy is proably caused by the numerical solver precision. In other words more points would increase the accuracy. The jumping up and down along the linear curve is probably duo to the computer's accuracy with numbers. This will also improve by more steps, but one of them might stop to improve before the other.

5 Conclusion

6 Appendix

```
//FLOPs FOR ACCELERATION
// 1 FLOP * 3 directions
dx = x1 - x2
// 3 FLOPs
double r = sqrt(dx*dx + dy*dy + dz*dz)
// 7 FLOPs
double a = - (Gconst*m*M/(r*r)) / (m*r)
// 2 FLOPs * 3 directions
a = a + a*(x1-x2);
//TOTAL FLOPs = 19 FLOPs


//FLOPs FOR POSITION :: EULER
// 2 FLOPs * 3 directions
x = x + t_step*Vx
//TOTAL FLOPs = 6 FLOPs


//FLOPs FOR VELOCITY :: EULER
// 2 FLOPs * 3 directions
Vx = Vx + t_step*ax
//TOTAL FLOPs = 6 FLOPs


//FLOPs FOR POSITION :: Verlet
// 6 FLOPs * 3 directions
x = x + t_step*Vx + (0.5*t_step*t_step*a);
//TOTAL FLOPs = 21 FLOPs


//FLOPs FOR VELOCITY :: Verlet
// 4 FLOPs * 3 directions
Vx = Vx + (0.5*t_step*(Ax+Ax_old));
//TOTAL FLOPs = 12 FLOPs
```