Contents

1 Abstract . . . 1
2 Introduction . . . 1
3 Physical theory . . . 1
3.1 Gravitation . . . 1
3.2 Choice of units . . . 2
4 Mathematical theory . . . 2
4.1 Forward Euler . . . 2
4.2 Velocity-Verlet . . . 3
5 Implementation . . . 4
5.1 Number of floating point operations (FLOPs) . . . 4
6 Simulations and results . . . 6
6.1 Earth-sun-system . . . 6
6.2 Three-body-problem with fixed sun and varying masses . . . 9
6.3 Realistic simulations . . . 9

1 Abstract

2 Introduction

3 Physical theory

3.1 Gravitation

In this project, a solar system will be studied. By solar system, I mean a system where only gravitational forces affect the bodies. Newtons gravitational law states that the gravitational force on a body with mass m from another body with mass M and relative position $\vec{r}$ is given by

$$\vec{F}_{\mathrm{G}} = -\frac{GmM}{\|\vec{r}\|^2}\vec{r}$$

where G is the gravitational constant, $6.67 \cdot 10^{-11}\ \mathrm{N\,m^2/s^2}$. The direction of the force is given by the fact that gravity is an attractive force. If one of the objects is the sun, the mass is denoted by $M_\odot$. With the sun placed in origo, r is simply the norm of the position vector of the planet with mass m.

If there are n planets in the solar system, in addition to the sun, the sum of the forces on planet i with mass m_i is

$$\sum \vec{F}_i = \sum_{\substack{k=0 \\ k\neq i}}^{n} \frac{G m_i m_k}{\|\vec{r}_k - \vec{r}_i\|^3}(\vec{r}_k - \vec{r}_i)$$

with $m_0 = M_\odot$ and $\vec{r}_0 = \vec{0}$. From Newton's second law, we know that $\sum \vec{F}_i = m_i \vec{a}_i$, so the acceleration of planet i is given by

$$\vec{a}_i = \sum_{\substack{k=0 \\ k\neq i}}^{n} \frac{G m_k}{\|\vec{r}_k - \vec{r}_i\|^3}(\vec{r}_k - \vec{r}_i) \tag{1}$$

As the sun has been fixed to origo, $\vec{a}_0$ is set to $\vec{0}$.

3.2 Choice of units

In the solar system, seconds and meters are unpractical, as planets are millions of kilometers apart and take years to do one lap around the sun. As such, it is common to use so-called astronomical units, where 1 ua is the mean distance between the sun and the earth, and time is measured in years. To express the gravitational constant in these units, we can use that if the earth were moving in a circle around the sun, the acceleration in Newton's second law would be given by the sentripetal acceleration:

$$\frac{mv^2}{r} = \frac{GmM_\odot}{r^2} \implies G = \frac{r}{M_\odot} v^2 = \frac{1 \text{ ua}}{M_\odot} \cdot \left(\frac{2\pi \cdot 1 \text{ ua}}{1 \text{ yr}}\right)^2 = \frac{4\pi^2}{M_\odot} \cdot 1 \text{ ua}^3/\text{yr}^2$$

With this change of units, equation (1) can be written as

$$\vec{a}_i = \sum_{\substack{k=0 \\ k\neq i}}^{n} 4\pi^2 \frac{m_k}{M_\odot} \frac{\vec{r}_k - \vec{r}_i}{\|\vec{r}_k - \vec{r}_i\|^3} \cdot 1 \text{ ua}^3/\text{yr}^2 \tag{2}$$

4 Mathematical theory

Equation (2), together with some initial conditions, determines the motion of the bodies in the solar system. When written out in components, the equation gives a coupled set of differential equations. This set of equations is difficult, if at all possible, to solve analytically, so numerical work is required. As per usual, the time is discretised as $t_i = t_0 + ih$, where h is the time step, $h = (t_n - t_0)/n$. Acceleration, velocity and position are discretised correspondingly.

4.1 Forward Euler

The Forward Euler method, also called the Explicit Euler method, and hereafter called simply the Euler method, uses a first order Taylor polynomial to approximate a solution to the

diffrential equation. With $x'(t) = v(t)$ and $v'(t) = a(t)$, we have that

$$\vec{r}_i(t+h) \approx \vec{r}_i(t) + h\vec{v}_i(t)$$
$$\vec{v}_i(t+h) \approx \vec{v}_i(t) + h\vec{a}_i(t)$$

Discretised version:

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

To clarify the indices: $\vec{a}_{i,j}$ is the acceleration of planet i at time step j. This is calculated from equation (2) on the preceding page.

From Taylor's formula, the error for a first order Taylor polynomial goes as $O(h^2)$. This is the error made in each step — the error is accumulated, so the total error will be proportional h.

4.2 Velocity-Verlet

The Velocity-Verlet method, hereafter called the Verlet method, is based on a second order Taylor polynomial.

$$\vec{r}_i(t+h) \approx \vec{r}_i(t) + h\vec{v}_i(t) + \tfrac{1}{2}h^2\vec{a}_i(t)$$
$$\vec{v}_i(t+h) \approx \vec{v}_i(t) + h\vec{a}_i(t) + \tfrac{1}{2}h^2\vec{a}_i'(t)$$

There is no explicit expression for $\vec{a}'(t)$, however it can be approximated using the good old formula

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

Since the acceleration is independent of the velocity, the newly updated position, $\vec{a}(t+h)$, can be calculated using $\vec{r}(t+h)$. Inserting this into the expression for $\vec{v}(t+h)$, we get

$$\begin{aligned}\vec{v}_i(t+h) &\approx \vec{v}_i(t) + h\vec{a}_i(t) + \tfrac{1}{2}h(\vec{a}_i(t+h) - \vec{a}_i(t)) \\ &= \vec{v}_i(t) + \tfrac{1}{2}h(\vec{a}_i(t) + \vec{a}_i(t+h))\end{aligned}$$

The discretised version then becomes

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

The error of a second order Taylor polynomial is given as $O(h^3)$. The approximation for $\vec{a}'(t)$ has an error proportional to h, but this error is multiplied with h^2 when inserted into the expression for $\vec{v}_i(t+h)$. As such, the error for each step is proportional to h^3. The error is again accumulated for each step, so the total error will be proportional to h^2. This is one order better than the Euler method.

5 Implementation

The implementation is heavily object oriented and modular by design. The `Planet` class (planet.h and planet.cpp) represents a body, with methods for calculating the acceleration according to equation (2) as well as updating position and velocity for both algorithms. The `SolarSystem` class (solarsystem.h and solarsystem.cpp) administrates the planets, and contains a method for each algorithm. cpp_ui.cpp sets up the solar system according to the command line arguments, and calls the solver method specified. python_ui.py is a user-friendly interface to cpp_ui.cpp, see README.md.

conservation.py is a Python program that reads a data file and checks that momentum, energy and angular momentum are conserved. As there are no external forces (or torques) acting on the solar system, these quantities should always be conserved. Note that this program is not intended to be read.

5.1 Number of floating point operations (FLOPs)

To calculate the acceleration (common for both algorithms):

- For each of the n planets:
  - For each of the $n-1$ other planets:
    1. Subtract the x, y and z components (3 FLOPs).
    2. Square the differences in the x, y and z components (3 FLOPs).
    3. Add these differences (2 FLOPs).
    4. Take the square root (1 FLOP[1]).
    5. Calculate the cube of the norm (2 FLOPs).
    6. Divide $4\pi^2$ times the mass ratio (precalculated) by the cube of the norm (1 FLOP).
    7. Multiply this factor by the differences in x, y and z positions (3 FLOPs).
    8. Add these to the x, y and z components of the acceleration (3 FLOP).

$18n(n-1)$ floating point operations in total.

For the Euler method:

- For each of the n planets:
  1. Multiply the x, y and z components of the acceleration with the time step (3 FLOPs).
  2. Add these to the x, y and z components of the velocity (3 FLOPs).
  3. Multiply the x, y and z components of the velocity with the timestep (3 FLOPs).

[1] Or possibly more.

4. Add these to the x, y and z components of the position (3 FLOPs).

- Calculate the acceleration ($18n(n-1)$ FLOPs).

This gives a total of

$$12n + 18n(n-1) = 18n^2 - 6n \tag{3}$$

floating point operations per time step.

For the Verlet method:

- For each of the n planets:
  1. Multiply the time step with the x, y and z components of the velocity (3 FLOPs).
  2. Multiply half the square of the time step (precalculated) with the x, y and z components of the acceleration (3 FLOPs).
  3. Add these to the x, y and z components of the position (6 FLOPs).
- Calculate the new acceleration ($18n(n-1)$ FLOPs).
- For each of the n planets:
  1. Add the x, y and z components of the new and old accelerations (3 FLOPs).
  2. Multiply these by half the time step (precalculated) (3 FLOPs).
  3. Add the results to the x, y and z components of the velocity (3 FLOPs).
  4. Set the new acceleration to be the old acceleration, and set the new acceleration to $\vec{0}$ (0 FLOPs).

This gives a total of

$$12n + 18n(n-1) + 9n = 18n^2 + 3n \tag{4}$$

floating point operations per time step.

6 Simulations and results

6.1 Earth-sun-system

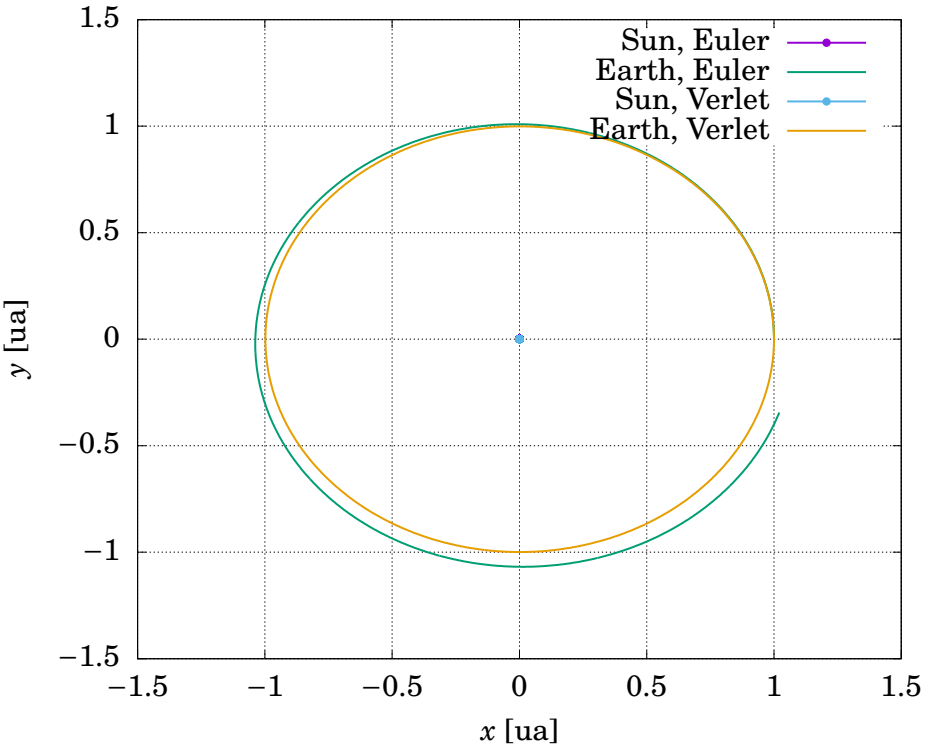

Figure 1: Simulation of the earth-sun-system with both the Forward Euler and the Velocity-Verlet algorithms. The simulation was run for 1 year with 1000 integration points, which clearly is insufficient for the Forward Euler method. The simulation was run by earthsun.sh.

6.1.1 Visual error analysis

When given a tangential velocity of 2π ua/yr, the earth should return to its initial position after one year.

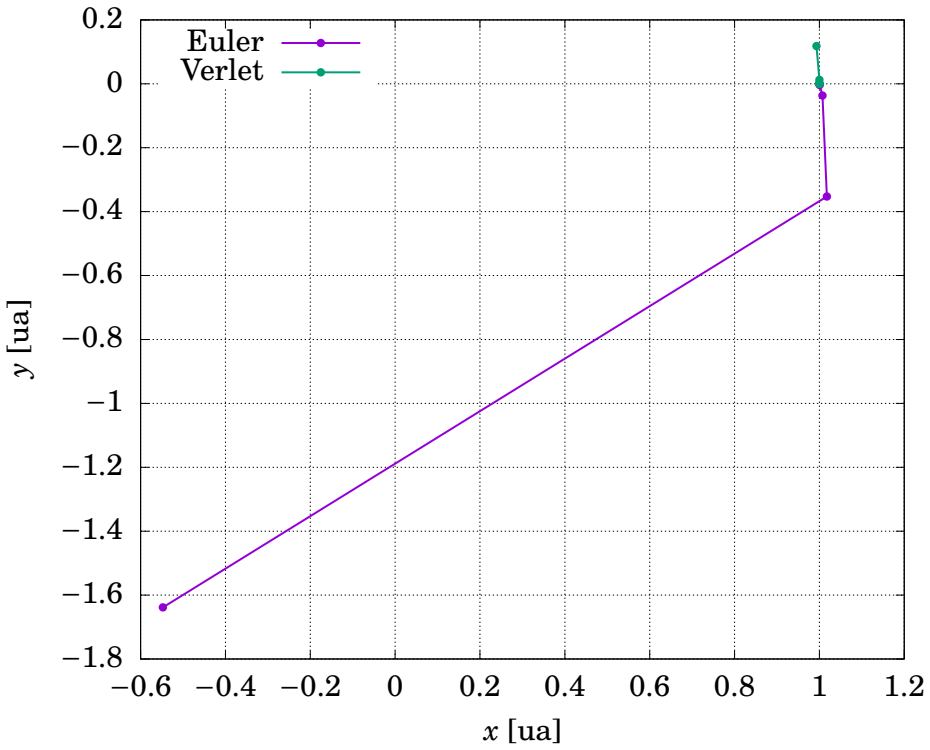

Figure 2: The end point of the earth after simulating for one year, with 10^n timesteps for $n = 2, 3, \ldots$. It is clear that the Verlet method converges much more quickly than Euler's method. The data is generated by errorearthsun.sh.

6.1.2 Benchmarks

Table 1: Benchmarks for both algorithms for the earth-sun-system. Generated by timetable.py.

Number of time steps	Euler time	Verlet time	Ratio
100	$2 \cdot 10^{-5}$ s	$2.6 \cdot 10^{-5}$ s	1.3
1000	$7.2 \cdot 10^{-5}$ s	$8.5 \cdot 10^{-5}$ s	1.18
10000	0.000571 s	0.000943 s	1.65
100000	0.0055 s	0.00661 s	1.2
1000000	0.0532 s	0.066 s	1.24
10000000	0.525 s	0.676 s	1.29
100000000	5.29 s	6.78 s	1.28

The time grows very linearly as a function of the number of time steps, as expected.

With $n = 2$ (the number of planets), equation (3) and equation (4) predict that the ratio between the times should be

$$\frac{18 \cdot 2^2 + 3 \cdot 2}{18 \cdot 2^2 - 6 \cdot 2} = \frac{78}{60} = 1.3$$

This fits reasonably well with the tabulated data. The slightly lower observed ratio is probably caused by the fact that floating point operations do not take up all of the time — some time is also spent on function calls, if-tests etc. Additionally, the square root might require more than one floating point operation.

6.1.3 Conservation tests

Running the command `python conservation.py earthsunverlet.dat` yields

```
Momentum conserved!
Total energy conserved!
Angular momentum conserved!
```

while `python conservation.py earthsuneuler.dat` yields

```
Momentum not conserved!
Initial: [  0.00000000e+00   1.88400000e-05   0.00000000e+00]
Final: [  5.77573800e-06   1.71795840e-05   0.00000000e+00]
Total energy not conserved!
Initial: -5.92776528131e-05
Final: -5.07987047801e-05
Angular momentum not conserved!
Initial: [  0.00000000e+00   0.00000000e+00   1.88400000e-05]
Final: [  0.00000000e+00   0.00000000e+00   1.95221041e-05]
```

6.2 Three-body-problem with fixed sun and varying masses

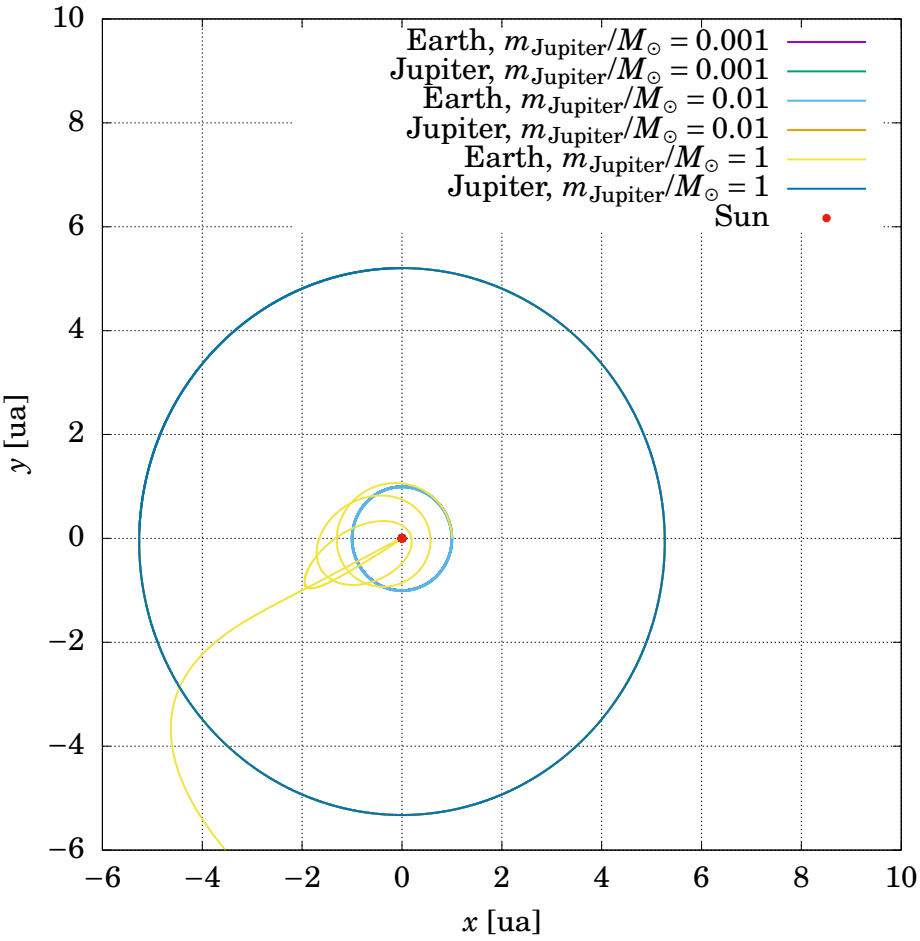

Figure 3: Simulation of the three-body-problem for 15 years with different masses for Jupiter, with 10^8 integration steps. Generated by threebody.sh. Initial conditions are set from NASA (mean distance and velocity). The sun was fixed at origo.

For the two smaller Jupiter masses, both planets move in stable, circular motion. When the mass of Jupiter is set equal to that of the sun, however, the motion of the earth becomes unstable. With 10^8 integration steps, even the Verlet method is insufficient, and the earth suddenly decides to leave the solar system. With 10^9 integration points, this does not occur.

6.3 Realistic simulations

In these simulations, real data from NASA was used - see nasadata.csv.

6.3.1 Three-body-problem

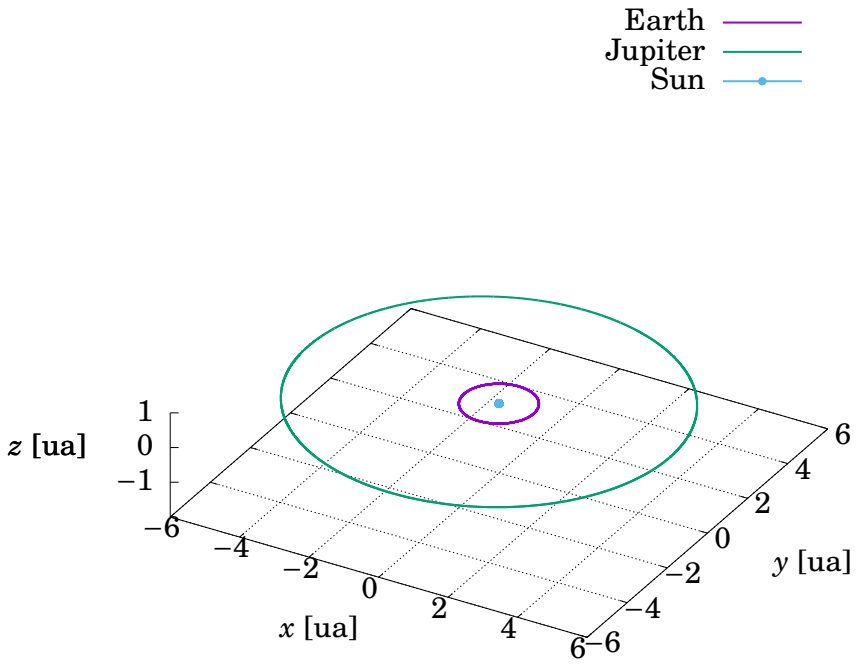

Figure 4: Realistic simulation of the sun, earth and jupiter for 20 years with 10^8 integration steps. Generated by realthreebody.sh.

From the figure, it is clear that the sun barely moves. We also see that the planets now move in elliptic orbits.

6.3.2 All planets

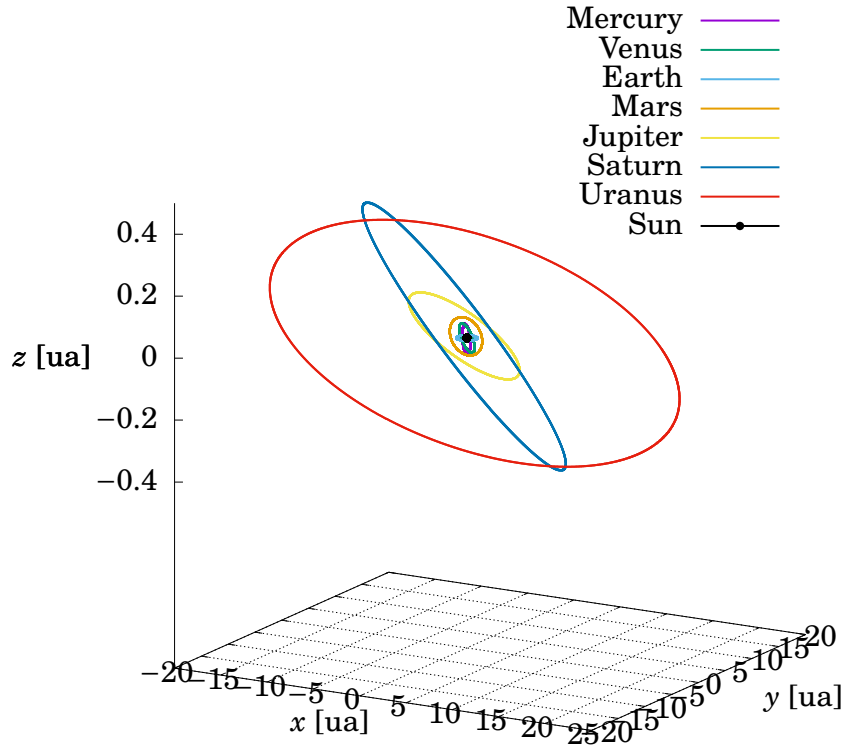

Figure 5: Realistic simulation of all planets in the solar system for 20 years with 10^8 integration steps Note that the tilting of the orbits is exaggerated. Generated by everything.sh.