Project 3

FYS4150 Computational Physics

Even S. Håland

Abstract

The motion of the planets in the solar system is governed by gravitational forces between the planets and the sun, and between the planets themselves. This can be expressed mathematically as a set of coupled differential equation. In this project we solve these equations, mainly by using the velocity Verlet algorithm. When doing so we use an object oriented approach, and two classes are developed; a *planet* class and a *solver* class. We find that the problem can be solved in a quite nice and compact way, and we get quite good estimates of the planetary orbits.

1 Introduction

2 Modelling planetary motion

Newtons law of gravitation states that the gravitational force between two objects, with mass M and m respectively, is given by

$$F_G = \frac{GMm}{r^2}, \tag{1}$$

where G is the gravitational constant and r is the distance between the two objects.

Earth-Sun system

We start by considering a system with only two objects, namely the Sun (with mass $M_\odot$) and the Earth (with mass M_{Earth}), and we assume that the Sun is fixed, so the only motion we have to care about is that of the Earth. We also assume that the motion of the Earth is co-planar, and take this to be the xy-plane, with the Sun in the origin. When writing the programs and implementing the algorithm we actually work in $3D$, but extending from two to three dimensions is quite trivial.

The forces acting on the Earth in the x- and y-direction are then given by

$$F_{G,x} = -F_G \cos\theta = -\frac{GM_\odot M_{\text{Earth}}}{r^2}\cos\theta = -\frac{GM_\odot M_{\text{Earth}}}{r^3}x$$

and

$$F_{G,y} = -F_G \sin\theta = -\frac{GM_\odot M_{\text{Earth}}}{r^2}\sin\theta = -\frac{GM_\odot M_{\text{Earth}}}{r^3}y,$$

where we have used the relations $x = r\cos\theta$ and $y = r\sin\theta$. From Newtons second law we know that the accelerations, a_x and a_y, are given as

$$a_x = \frac{d^2x}{dt^2} = \frac{F_{G,x}}{M_{\text{Earth}}} \quad \text{and} \quad a_y = \frac{d^2y}{dt^2} = \frac{F_{G,y}}{M_{\text{Earth}}}.$$

We also know that the acceleration is the time derivative of the velocity, v, which again is the time derivative of the position, meaning that we can express the equations of motion as two coupled first order differential equations in each dimension:

$$v_x = \frac{dx}{dt}, \quad v_y = \frac{dy}{dt}, \quad a_x = \frac{dv_x}{dt} = -\frac{GM_\odot x}{r^3}, \quad a_y = \frac{dv_y}{dt} = -\frac{GM_\odot y}{r^3}. \tag{2}$$

When extending to three dimensions we simply add two more equations like those above, simply replacing x or y by z.

Scaling of equations

The next thing we want to do is to scale the equations appropriately. When working on an astronomical scale we prefer to work with years (yr) as the time unit and

AU[1] as the length unit. This means that we need to find some way of scaling the gravitational constant, G, to these units.

If we assume the orbit of the Earth to be circular (which is very close to the truth), the acceleration is given as

$$a = \frac{v^2}{r} = \frac{F_G}{M_{\text{Earth}}} = \frac{GM_\odot}{r^2} \quad \Rightarrow \quad v^2 r = GM_\odot.$$

where $r = 1$ AU, while the velocity is

$$v = 2\pi \text{ AU/yr},$$

which means that

$$GM_\odot = 4\pi^2 \text{ AU}^3/\text{yr}^2,$$

which can be inserted in the equations of motion. In addition to this it is also convenient to scale the mass of the earth (and other planets) to the solar mass, i.e. we put $M_\odot = 1$, and scale other masses accordingly. We do this to decrease the chance of loosing numerical precision through round-off errors, as the planetary masses are quite large (see Table 1).

The solar system

Once we know the form of the gravitational interaction it is quite easy to extend our system to include other planets, and eventually the full solar system. The general forces in the xy-planet between two planets with mass M_a and M_b are given by

$$F_{G,x} = \frac{GM_aM_b}{r^3}\Delta x \quad \text{and} \quad F_{G,y} = \frac{GM_aM_b}{r^3}\Delta y,$$

where Δx and Δy are the distances between the planets in x- and y-direction, and r the absolute distance between them. When we want to model a system with several planets we simply add up the forces acting on each planet, and calculate the acceleration of a specific planet as the total force acting on it divided by its mass. The planets of the solar system are listed in Table 1, along with their masses and distances to the Sun.

Before we start looking at how we should study this numerically there is one more thing that needs to be mentioned in this somewhat theoretical introduction. Although Newtonian mechanics is able to describe the planetary orbits quite well, it does not provide a perfect approximation. This is most evident when observing the perihelion precession of Mercury, which is found to be off by $43''$ ($\sim 0.012°$) per century (which is a very small deviation!) compared to the predictions from the Newtonian theory. However, in Einstein's general theory of relativity the gravitational force gets a small correction, and can be written as

$$F_G = \frac{GM_\odot M_{\text{Mercury}}}{r^2}\left[1 + \frac{3l^2}{r^2c^2}\right],$$

[1] Astronomical units; 1 AU is defined as the mean distance between the Earth and the Sun.

Table 1: Mass and distance to the Sun for the planets in the solar system. (Numbers taken from the project description.)

Planet	Mass (kg)	Distance to Sun (AU)
Earth	6×10^{24}	1
Jupiter	1.9×10^{27}	5.20
Mars	6.6×10^{23}	1.52
Venus	4.9×10^{24}	0.72
Saturn	5.5×10^{26}	9.54
Mercury	3.3×10^{23}	0.39
Uranus	8.8×10^{25}	19.19
Neptune	1.03×10^{26}	30.06
Pluto	1.31×10^{22}	39.53

where $l = |\mathbf{r} \times \mathbf{v}|$ is the magnitude of the orbital angular momentum of Mercury and c is the speed of light. Towards the end of the project we will see (spoiler alert!) that this correction actually is able to explain the observed perihelion precession of Mercury!

3 Discretization and algorithms

The main objective is to write a code that calculate updated positions for system of planets as time passes. In order to do so we must, as usual, start by making a discrete approach to the problem. The discretization will only be shown for one dimension (x), since the procedure is completely equivalent for the other dimensions. Time and position are discretized as

$$t \to t_i = t_0 + ih$$
$$x(t) \to x(t_i) = x_i,$$

with the time step, h, given by

$$h = \frac{t_f - t_0}{n}.$$

Here t_0 and t_f is the initial and final time respectively, n is total number of time steps and i runs from 1 to n.

Position and velocity after some time $t_i + h$ is given by Taylor expansion as

$$x_{i+1} = x_i + hx_i' + \frac{h^2}{2}x_i'' + O(h^3) \tag{3}$$
$$= x_i + hv_i + \frac{h^2}{2}a_i + O(h^3) \tag{4}$$

and

$$v_{i+1} = v_i + hv_i' + \frac{h^2}{2}v_i'' + O(h^3) \tag{5}$$

$$= v_i + ha_i + \frac{h^2}{2}v_i'' + O(h^3), \tag{6}$$

where a_i is the acceleration, which for the Earth-Sun system is given in discrete form as

$$a_i = \frac{F(x_i, t_i)}{M_{\text{Earth}}} = -\frac{GM_\odot x_i}{r_i^3}.$$

Based on these equations we will consider two methods for approximating positions and velocities.

The first one is the forward Euler method, which we get directly from the above equations, by only including terms from eqs. (4) and (6) up to $O(h^2)$:

$$x_{i+1} \approx x_i + hv_i$$
$$v_{i+1} \approx v_i + ha_i$$

The second one is the velocity Verlet method, where we keep terms up to $O(h^3)$. This means that we are stuck with a second derivative of the velocity, which we want to get rid of. This is done by Eulers formula, so

$$v_i'' \approx \frac{v_{i+1}' + v_i'}{h} = \frac{a_{i+1}' + a_i'}{h},$$

which leaves us with the following approximations for position and velocity:

$$x_{i+1} \approx x_i + hv_i + \frac{h^2}{2}a_i$$
$$v_{i+1} \approx v_i + \frac{h}{2}\left[a_{i+1} + a_i\right]$$

Both of these methods will be implemented in our code. However, velocity Verlet will be proven to work somewhat better than forward Euler, so throughout most of the project we will stick to using the velocity Verlet method.

Notice that in order to get the algorithms started we need some initial values for position and velocity, i.e. x_0 and v_0, hence these kinds of problems are referred to as *initial value problems*.

4 Code

All code written for this project can be found in the following git repository:

https://github.com/evensha/FYS4150/tree/master/Project3/Programs

The most important files in this repository are:

- `planet.cpp/planet.h`
- `solver.cpp/solver.h`
- `main.cpp`
- `Plot_planets.py`

Before going into the details of the code we should have a quick look at the main structure and purpose of the different programs and classes.

Firstly we have two classes called **planet** (implemented in `planet.cpp` and `planet.h`) and **solver** (implemented in `solver.cpp` and `solver.h`). The main idea is that we let each planet we want to consider be an object of the planet class, which has properties like position, velocity, mass, and functions that can calculate other quantities for the planet. We then make an object of the solver class, and put our planets in to this object. The solver class contain functions that will solve our problem, i.e. by using forward Euler or velocity Verlet, in addition to other functions that could be useful. The main program (`main.cpp`) is used to initialize the necessary objects, and run the solver functions, while the python script `Plot_planets.py` is used for plotting the results. The output from the programs is stored in the "Output" repository. However, one might not find all produced output files in the mentioned git repository, as some of the produced output files are quite big.

In addition to the above described programs it is also worth mentioning that the git repository contains a `makefile`, used for compiling it all, and a file called `Planet_data.txt`. The latter file contains necessary information about all the planets, i.e. mass and the positions and velocities we will use as initial values.

4.1 The planet class

The planet class has four public variables:

- `mass` (double)
- `position` (double, 3D vector)
- `velocity` (double, 3D vector)
- `name` (string)

An object of this class can be initialized with a default initialization that sets all the variables to zero, and the name to "Planet". Alternatively it can be initialized with mass, positions, velocities and name. Positions and velocities can be given in either two or three dimension.

Further the class contains the following functions (which all return a double):

- `Distance(planet otherPlanet)`: Take an other object of the planet class as input argument, and calculates the distance to this planet.

- `PotentialEnergy(double Gconst)`: Take an other object of the planet class as input, and calculates the planets potential energy with respect to the other planet.
- `xMomentum()`: Calculates the planets momentum in the x-direction.
- `yMomentum()`: Calculates the planets momentum in the y-direction.
- `AngularMomentum()`: Calculates the magnitude of the orbital angular momentum per unit mass of the planet.

4.2 The solver class

4.3 The main program and plotting

4.4 Testing the code

5 Results

6 Summary and conclusions