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. This is the "corner stone" of all calculations done in this project.

2.1 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.

2.2 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).

2.3 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

[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

Newtonian theory. However, in Einstein's general theory of relativity the gravitational force gets a small correction, and can be written as

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

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

$$\begin{aligned} t &\to t_i = t_0 + ih \\ x(t) &\to x(t_i) = x_i, \end{aligned}$$

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

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

and

$$\begin{aligned} v_{i+1} &= v_i + hv_i' + \frac{h^2}{2}v_i'' + O(h^3) \\ &= v_i + ha_i + \frac{h^2}{2}v_i'' + O(h^3), \end{aligned} \tag{4}$$

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 (FE) method, which we get directly from the above equations, by only including terms from eqs. (3) and (4) up to $O(h^2)$:

$$\begin{aligned} x_{i+1} &\approx x_i + hv_i \\ v_{i+1} &\approx v_i + ha_i \end{aligned}$$

The second one is the velocity Verlet (VV) 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:

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

Notice that the VV algorithm require some more floating point operations (FLOPS) than FE. If we pre-calculate $\frac{h}{2}$ and $\frac{h^2}{2}$ we need 7 FLOPS for VV against 7 for FE. However, we also need to calculate the forces acting on the planet in order to calculate the acceleration. This must be done twice for each time step in the VV loop and only once per time step in the FE loop.

Notice also 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*.

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.

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. The positions and velocities are taken from ref. [1].

4.1 The planet class

The planet class has four public variables:

- `mass`: The mass of the planet.
- `position`: Three dimensional double containing the coordinates of the planets position.
- `velocity`: Three dimensional double containing the velocities of the planet in each dimension.
- `name`: The name of the planet, given as a 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

The solver class has the following public variables:

- `mass`: Mass of the system you are studying.
- `G`: Gravitational constant, $4\pi^2$ by default.
- `beta`: Power of r in the denominator of the gravitational force, i.e. $F_G \propto 1/r^\beta$. By default $\beta = 2$, but we will also study some variations of this gravitational force.
- `RelCorr`: Integer that indicates whether or not you want to add the previously discussed relativistic correction to the gravitational force:
  - `0`: without relativistic correction (default).
  - `1`: with relativistic correction.
- `total_planets`: Integer denoting the total number of planets in your system.
- `all_planets`: Vector that contains the "planet objects" you have added to the solver class.

The solver class is initialized either with a default initialization, or by specifying `RelCorr` or `beta` discussed above. The class contains the following functions:

- `addPlanet(planet newplanet)`: Takes an object of the planet class as input, and adds this object to the solver object, updating the total mass and number of planets in the system.

- `ForwardEuler(int integration_points, double time)`: Takes number of integration points and the time period (in years) you want to consider as input, and updates positions and velocities of the planets in your system according to the forward Euler algorithm. An output file is made, but nothing is printed to this file unless you call one of the "print"-functions (see later).
- `VelocityVerlet(int integration_points, double final_time, int withOutput)`: Similar in structure to the forward Euler function, only this one is using the velocity Verlet method, and you can also specify whether or not you want to produce output[2]. The case where we don't want to produce output is in this project typically related to the perihelion precession of Mercury, as this requires *a lot* of time steps, which leads to very big output files. Because of this, when no output is produced, we instead chose to locate the perihelion, update its position as time passes, and print out the final position in the end.
- `GravitationalForce(planet &Planet, planet &other, double &F_x, double &F_y, double &F_z, double beta, int RelCorr)`: Calculates the gravitational forces between two specified planets in three dimensions.
- `PrintPositions()`: Prints the current positions to the output file.
- `PrintNames()`: Prints the names of all your planet objects to the output file.

4.3 The main program and plotting

The purpose of the main program is to initialize the need planet objects and the solver, put the planets into the solver, and then run the relevant functions. In order to initialize the planets we read in the relevant planet data from the `Planet_data.txt`-file, and store them in maps related to each planet, which are then used in the initialization. Different solver objects are initialized for different parts of the project, depending on what kind of problem we are studying.

The python script `Plot_planets.py` are then used to read the output files produced by the solver functions and make the required plots.

5 Results

In this section all the results are presented. As there are quite a few results to present I found it beneficial to structure this section in a somewhat similar way as the different tasks are given in the project description.

5.1 Testing the algorithms (Earth-Sun system)

When testing the programs and algorithms we consider a binary system with the Sun (fixed in the origin) and the Earth in a circular orbit, and we only consider

[2]By output it is here meant a file containing the positions we are calculating.

two dimensions. In order for the earth to get a circular orbit the magnitude of the velocity must be $v = 2\pi$ AU/year (given $r = 1$ AU), so we initialize the system by putting $v_x = 0$ and $v_y = 2\pi$ AU/year.

Figure 1 shows comparisons of the orbits we get by using FE and VV over a period of 10 years for. In the upper plot we have chosen $10,000$ time steps, while in the lower plot we have $100,000$ time steps. We see from these plots that the orbits obtained by the VV looks very nice and stable in both cases, while the FE orbits are spiralling outwards. The spirals gets closer when we increase the number of time steps, but we see that the effect is still there.

Another way of testing the algorithms is to consider conservation of some kinematic quantities. Since the orbits (in this test case) should be perfect circles the kinetic and potential energy should be separately conserved, since the velocity and distance to the sun should stay the same. These two quantities are given as

$$E_k = \frac{1}{2}mv^2$$

and

$$E_p = -\frac{GMm}{r}$$

respectively.Also, since the sun is fixed there should be no transfer of angular momentum, meaning that also this quantity should be conserved. We will consider the magnitude of the angular momentum per unit mass, which is given as

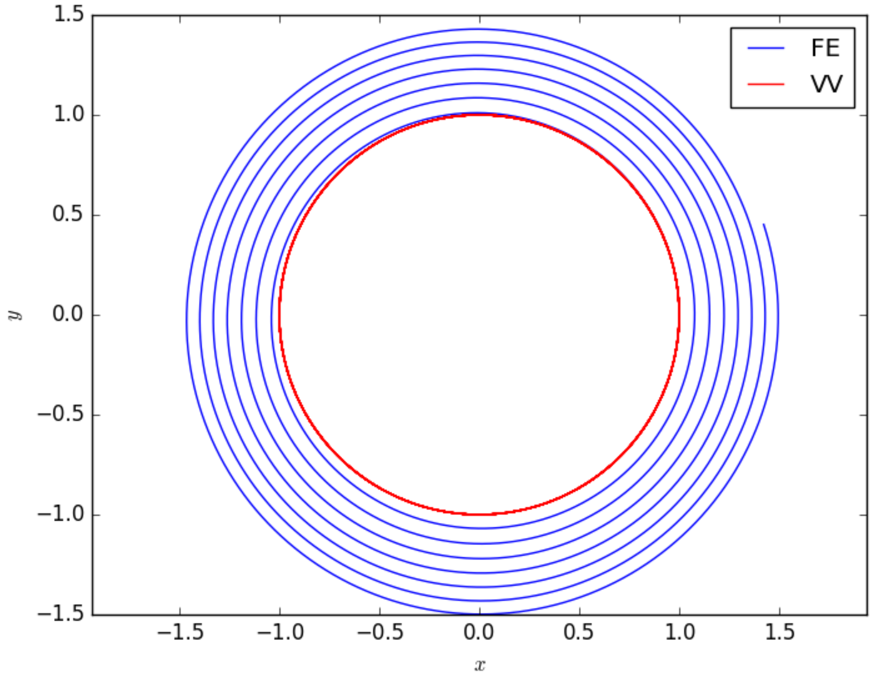
$$l = |\mathbf{r} \times \mathbf{v}|.$$

In Table 2 these quantities are given for the initial state of the Earth, and after a 10 years simulation with $10,000$ time steps for the two algorithms. Notice that the calculations are done in the units we used for scaling the equations (i.e. AU, years and mass per $M_\odot$.) We see here that with the VV algorithm all of these are perfectly conserved, while non of them are conserved using the FE algorithm.

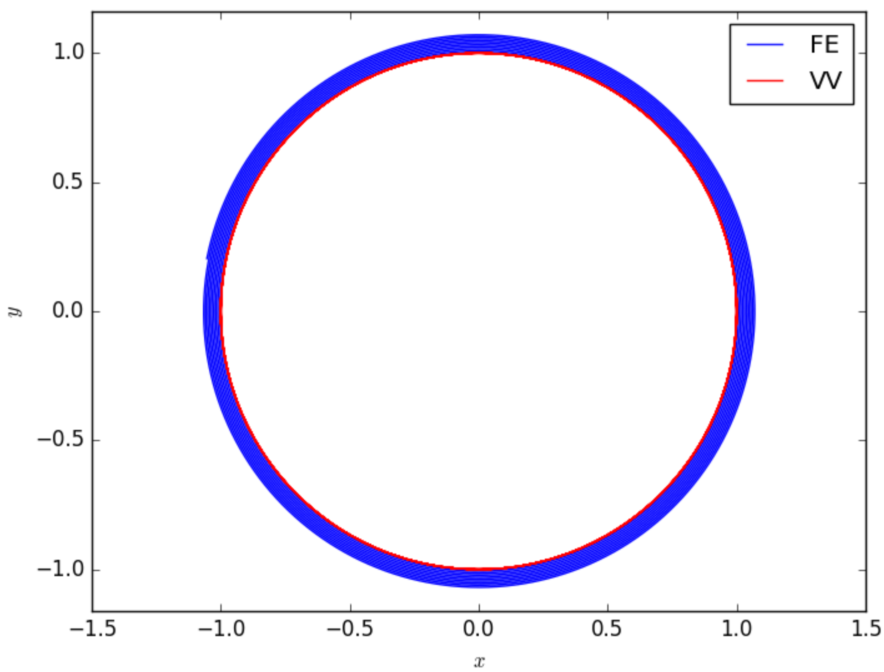
Table 2: Kinetic and potential energy, and angular momentum (per unit mass), calculated before we run the algorithms and after 10 years and $10,000$ time steps of forward Euler and velocity Verlet. The quantities are calculated using the units discussed earlier.

	Kinetic energy	Potential energy	Angular momentum
Initial value	$5.92 \cdot 10^{-5}$	$-1.18 \cdot 10^{-4}$	6.28
Forward Euler	$3.98 \cdot 10^{-5}$	$-7.93 \cdot 10^{-5}$	7.69
Velocity Verlet	$5.92 \cdot 10^{-5}$	$-1.18 \cdot 10^{-4}$	6.28

The last thing we want to test is the performance of the two algorithms in terms of CPU time. Results for various number of time steps are given in Table 3. As discussed previously the VV algorithm requires more FLOPS than FE, so it is



(a) $10,000$ time steps.



(b) $100,000$ time steps.

Figure 1: Orbits obtained by using forward Euler and velocity Verlet, when running over 10 years.

not surprising to see that VV is about twice as time consuming as FE. However, simulating 10 million time steps in 10 second is still not that bad. So based on this, and the previous observations about stability and conservation of kinematic quantities, we will through the rest of the project use the VV algorithm for our simulations.

Table 3: CPU time consumption for different number of time steps with forward Euler and velocity Verlet.

	CPU time	
n	Forward Euler	Velocity Verlet
10^4	0.012 s	0.016 s
10^5	0.083 s	0.106 s
10^6	0.521 s	1.012 s
10^7	4.822 s	10.73 s

5.2 Escape velocity and modification of gravitational force

Since we have established the VV algorithm as our preferred method we can now play around with the program, for example by trying to find the escape velocity of a planet, and by doing some modifications of the gravitational force. When doing this we will stick to using the same system as for the testing above, only with the necessary modifications.

In Figure 2 planet trajectories are shown for four different values of the initial velocity, i.e 2.4π, 2.6π, 2.8π and 3π. (Remember that the initial velocity for the circular orbit was 2π.) We see that for $v = 2.4\pi$ the orbit is still relatively circular, but becomes gradually more elliptical as we increases v, while at $v = 3\pi$ the planet manages to escape, suggesting that the critical velocity, v_c for escape lies between 2.8π and 3π.

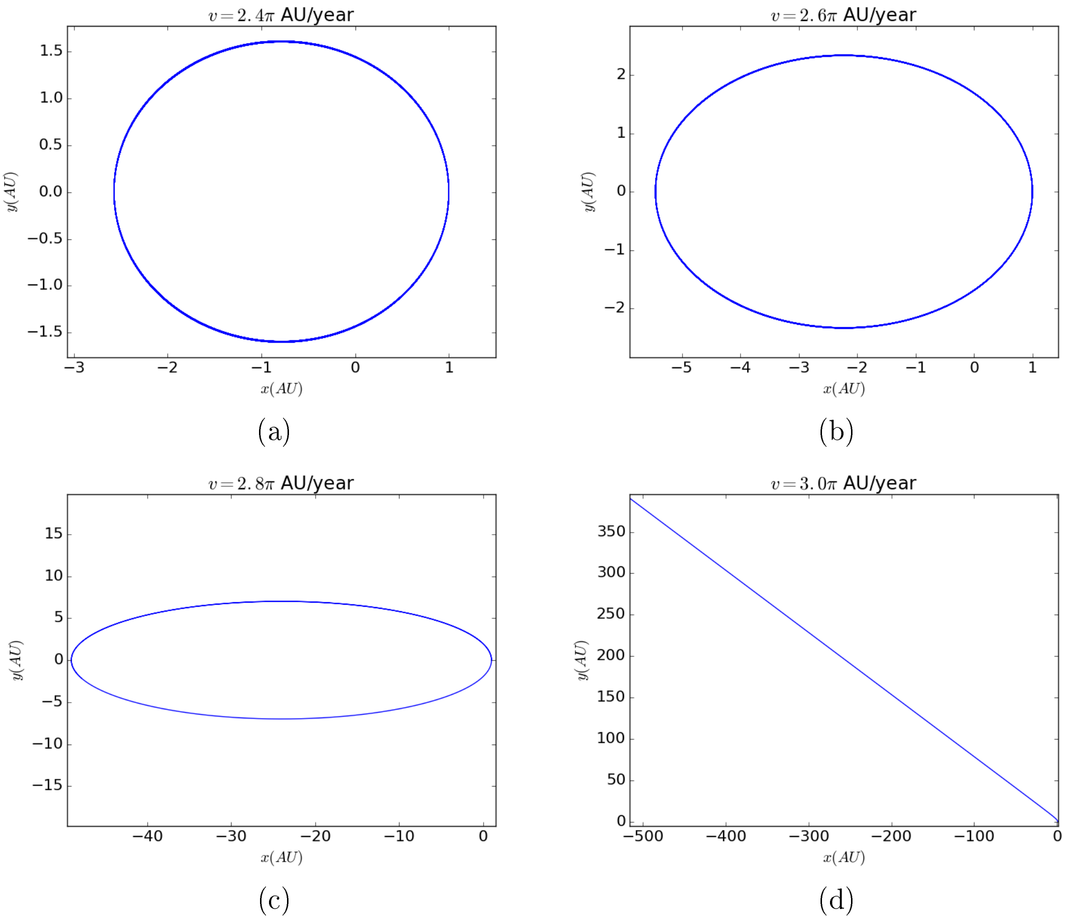
The criteria for escape is that $E_k > -E_p$, so the critical velocity is found when $E_k = -E_p$, i.e

$$\frac{1}{2}mv_c^2 = \frac{GM_\odot m}{r},$$

which, by solving for v_c leads to

$$v_c = \sqrt{\frac{2GM_\odot}{r}} = \sqrt{2}2\pi \approx 2.83\pi,$$

where we have used $GM_\odot = 4\pi^2$ and $r = 1$.



(a) (b) (c) (d)

Figure 2: Planet trajectories for different values of the initial velocity, simulated over 200 years with 100,000 time steps. The initial velocity is indicated above each plot.

5.3 Three-body problem

5.4 The solar system

5.5 Perihelion precession of Mercury

6 Summary and conclusions

References

[1] Solar System Dynamics (HORIZONS Web-Interface), NASA. https://ssd.jpl.nasa.gov/horizons.cgi (16/10-2017)