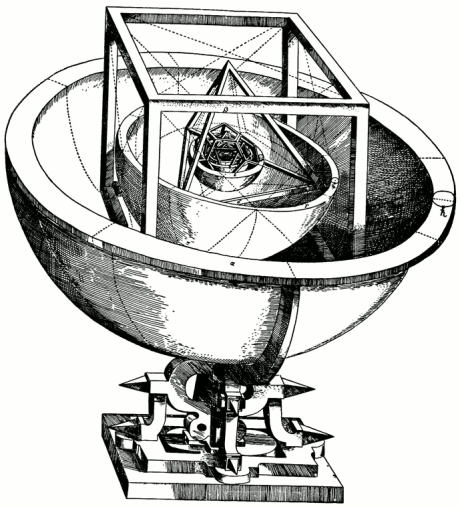
The Leap-Frog and the Solar System
A Stable Orbit in the 3-Body Problem

Practical Numerical Simulations
Assignment 1

Michælmas Term 2011

Fionn Fitzmaurice
fionnf@maths.tcd.ie



1 Introduction

In this code, the leap-frog integrator was employed in order to solve the equations of motion for an n-body system interacting gravitationally. This was used to simulate the motion of celestial bodies in a simple solar system, consisting of a sun, planet and moon.

2 Symplectomorphisms & The Leap-Frog Integrator

Simple integrators such as the Euler method diverge for oscillatory motion, hence we must choose a method that is more suited to solving our problem.

Since we are dealing with classical mechanics, it seems appropriate to consider Hamilton's equations. The symplectomorphic time evolution of these equations preserves the symplectic 2-form $\mathrm{d}p \wedge \mathrm{d}q$. An integrator that also preserves this form is called a symplectic integrator, and is particularly suitable for problems which can be formulated in Hamiltonian mechanics as it clearly prevents energy drift. The leap-frog integrator falls within this class of integrators. It also has the advantages of being time-reversible and second-order, compared to the first-order Euler method, despite requiring the same number of evaluations per step.

The method updates position and momentum as follows:

$$x_{k+\frac{1}{2}} = x_k + \frac{h}{2} p_k,$$

$$p_{k+1} = p_k + h f(x_{k+\frac{1}{2}}, t_{\frac{h}{2}}),$$

$$x_{k+1} = x_{k+\frac{1}{2}} + \frac{h}{2} p_{k+1},$$

where $f = \dot{p}$.

3 Conservation of Energy

The function `print_energy` was written to calculate the total energy of the system. It did this by evaluating the kinetic and potential energies of the system and summing them.

If we live in a Newtonian world, we know that, for the i^{th} particle, the kinetic and potential energies are, respectively,

$$T_i = \frac{1}{2} m_i \mathbf{v}_i^2, \qquad V_{ij} = -G \frac{m_i m_j}{r_{ij}}, \; i \neq j,$$

where m_i is the mass of the i^{th} particle, $\mathbf{v}_i$ is the velocity of the i^{th} particle, r_{ij} is the distance between the the i^{th} and j^{th} particles and G is the gravitational constant. Then

$$T = \sum_i T_i, \qquad V = \sum_{i>j} V_{ij},$$

$$E = T + V.$$

At this stage, our program considers the number of "planets" to be 2. I use quotation marks here as we will see later the term "planet" will eventually generalise to refer to a sun and a moon too.

3.1 Energy Evolution Over Time

As we expect, the total energy of the system is conserved. We can see this by checking the energy after each update of the leap-frog integrator.

If we label each iteration of the `update` function a, then we see a variation in the energy for small values of a which quickly converges to a constant. This value is also dependant on the step-size h. For three values of h, the (convergent value of the) energy is tabulated in table 1.

Step-size	0.5	0.1	0.01
Energy	136.586	58.8776	486.427

Table 1: Variation of energy with step-size.

We can equally tabulate the time it takes for the energy to converge , as seen in table 2.

Step-size	0.5	0.1	0.01
Iterations	11	53	40

Table 2: The number of iterations required for the energy to converge.

The energy as a function of simulation time is now plotted below in figure 1, for our three values of h.

4 Conservation of Angular Momentum

Angular momentum is given by

$$\mathbf{L} = \mathbf{x} \times \mathbf{p}.$$

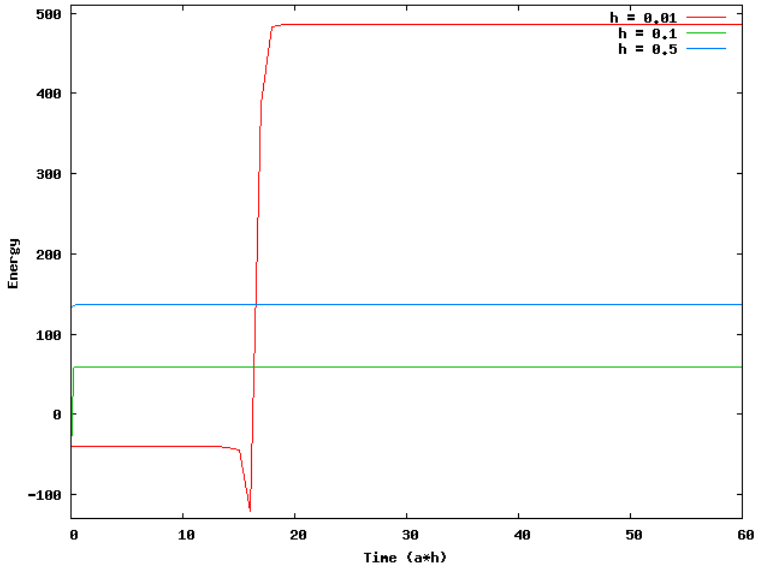


Figure 1: Energy as a function of simulation time for three different step-sizes.

If we consider the motion of the two planets to be in the xy-plane, then the angular momentum vector will have a z-component only. This allows us to compute the cross product relatively simply. For a single particle of mass m, its angular momentum is thus

$$\mathbf{L} = \begin{vmatrix} 0 & 0 & \hat{\mathbf{z}} \\ x & y & 0 \\ p_x & p_y & 0 \end{vmatrix} = m \begin{vmatrix} 0 & 0 & \hat{\mathbf{z}} \\ x & y & 0 \\ v_x & v_y & 0 \end{vmatrix} = m(xv_y - yv_x)\hat{\mathbf{z}},$$

where $\hat{\mathbf{z}}$ is the unit vector in the z-direction. So $L = \|\mathbf{L}\| = m(xv_y - yv_x)$. Of course, $\mathbf{x} = (x, y, 0)$ is meaningful only relative to some reference point. We define this point to be the origin, $(0, 0, 0)$, described in the program as the array `origin[]`.

4.1 Evolution of Angular Momentum Over Time

The value of L does not evolve in time. This is exactly as expected, as we again know from Noether's theorem that it is a conserved quantity.

For sufficiently small iterations, the angular momentum is not prone to variation, unlike the energy as we saw above. Additionally, the angular momentum is extremely insensitive to changes in the step-size h. This is tabulated in table 3.

Step-size	10	0.1	0.0000001
Angular momentum	2.34	2.34	2.34

Table 3: Variation of angular momentum with step-size.

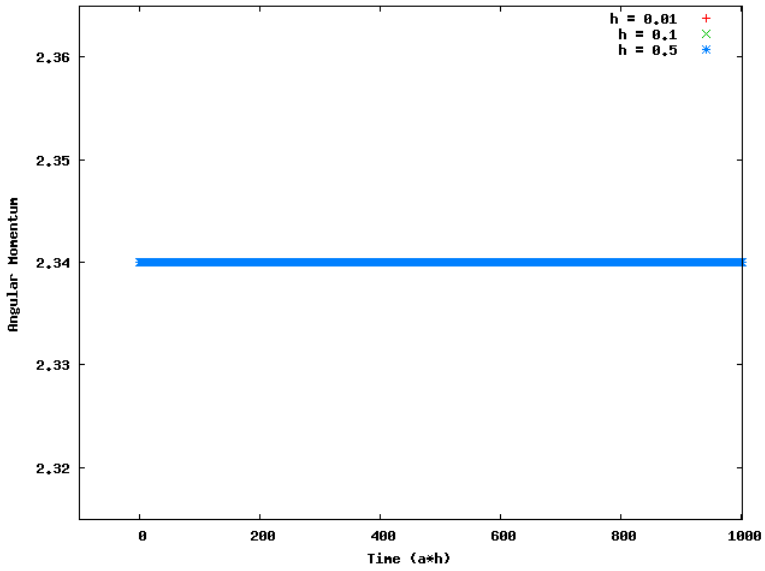


Figure 2: Angular momentum as an uninteresting function of simulation time.

The angular momentum as a function of simulation time is now plotted in figure 2, for our three values of h. The value of the angular momentum remains constant over all time, so the graph is somewhat redundant.

5 That's No Moon

We now introduce a third "planet", and simulate a simple earth-moon-sun solar system. This done by creating a new input file containing all the data necessary to determine the solutions of the equations of motion for 3 bodies. See §A for more details on this operation.

5.1 Collisions

As our planets are point-like particles, they can get arbitrarily close to each other. This is what I will refer to as a "collision". Due to the inverse square law of

the gravitational force $-\nabla V$, these particles receive an enormous energy in a collision and this jettisons them far from the safety of the solar system we created for them. Hence, our initial conditions must be finely tuned in order to avoid such collisions. This is the problem of creating a stable orbit.

5.2 A Stable Orbit

Through what I attribute to luck, rather than a keen physical intuition, a small number of adjustments to the planetary initial positions and initial momenta, a stable orbit was achieved. The values used were those in `test3.dat`, expanded upon in §A.2

"Stability" here is used in a loose sense. It is taken to mean that the solar system survives for at least 10 years, rather than an infinite number of years. To prove this strong case of stability, it would suffice to show that the solar system returns exactly to its initial conditions periodically. This condition does not hold for our weak stability. The solar-system is particularly fragile and sensitive to small variations in the initial conditions. This behaviour is to be expected – indeed, our own solar system is a chaotic system.

An orbit using the initial conditions outlined in §A.1 was found to be stable over 15000 iterations and a step-size of 0.01. This orbit is depicted in figures 3–6.

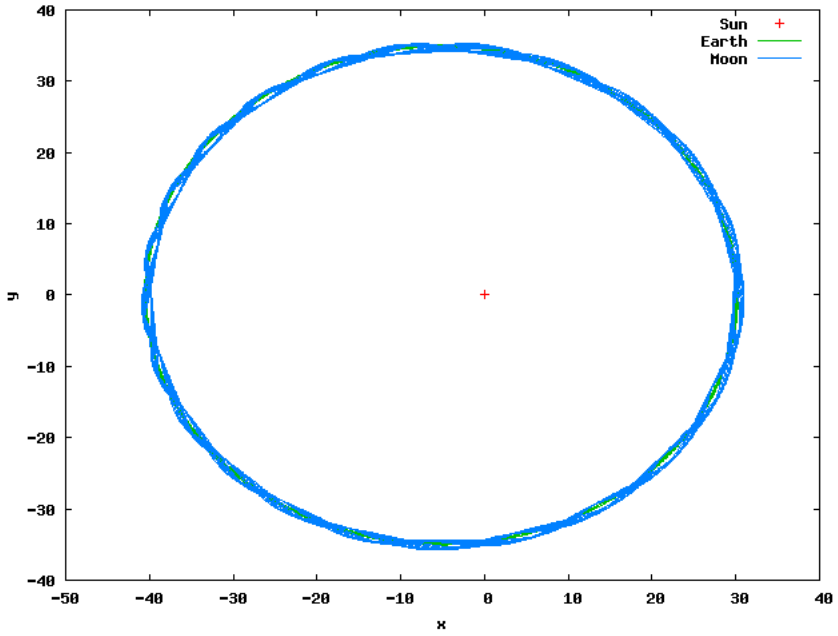


Figure 3: 11 long years.

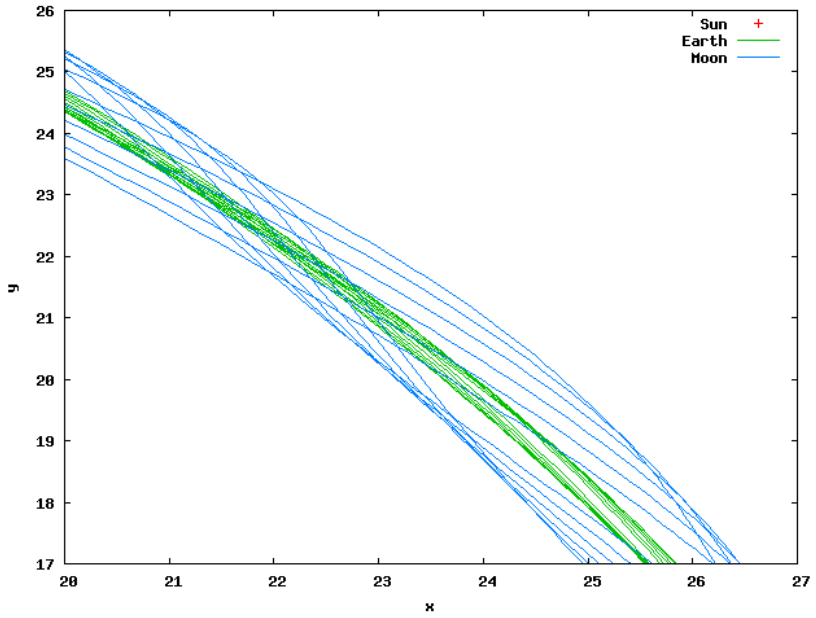


Figure 4: A section of the orbit.

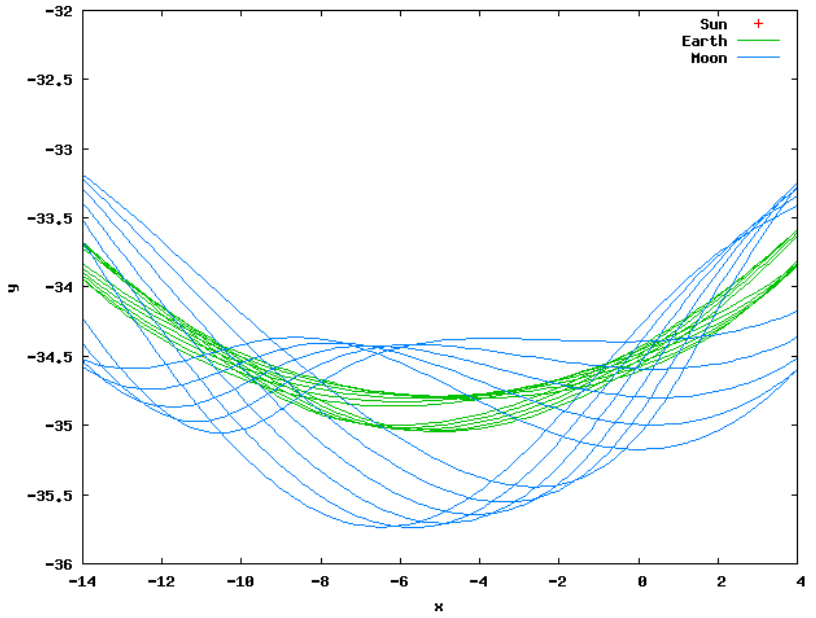


Figure 5: Would you want to live here?

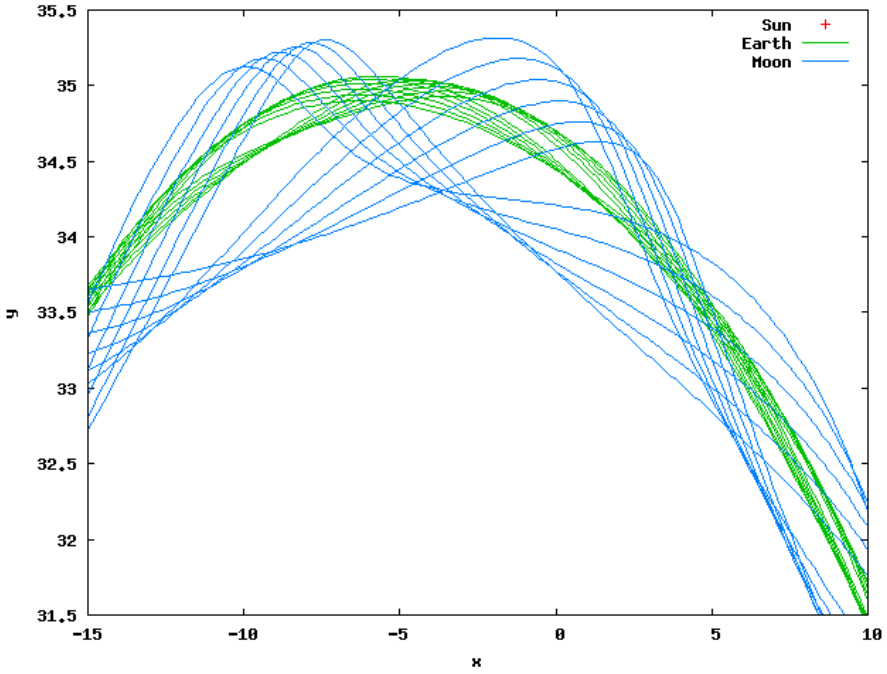


Figure 6: The celestial dance.

6 Simulated Calendar

The output data for this section is contained in the files `month.out`, `year.out` and `monthandyear.out`, attached in the tarball.

6.1 A Good Year

The planet initially makes its way around the sun starting on the x-axis, so its y coordinate is zero. This formulation affords us a natural way to count the years, as it is equivalent to counting the number of times the planet crosses the x-axis in the inertial reference frame of the sun. This is done by detecting a sign change in the y coordinate.

This prescription would double-count, however, as the planet crosses the axis twice in a single revolution. The code corrects for this by requiring that the y coordinate moves from negative to positive or, equivalently, that the planet makes the transition between the fourth and first quadrant. It is without loss of generality that we ascribe a specific direction, as the system has parity symmetry and our initial positions are arbitrarily chosen.

My fragile planet traces out the year, diligently recorded by the program.

Here is the output:

```
 1  Iteration   Simulation time Orbit
 2  1315        13.15          1 earth orbits
 3  2631        26.31          2 earth orbits
 4  3947        39.47          3 earth orbits
 5  5263        52.63          4 earth orbits
 6  6579        65.79          5 earth orbits
 7  7894        78.94          6 earth orbits
 8  9211        92.11          7 earth orbits
 9  10526       105.26         8 earth orbits
10  11842       118.42         9 earth orbits
11  13158       131.58         10 earth orbits
12  14474       144.74         11 earth orbits
```

The number in the first column is the iteration a in which the transition takes place. The simulation time is, as usual, the iteration by the temporal step-size, $a * h$.

This averages to give a year corresponding to a simulation time of 13.1581 time steps.

6.2 Synodic Periods

One might expect the calculation of a month to be a painful process. Perhaps the mind wanders to ideas about boosting into the planet's reference frame and repeating the prescription used above. However, the method employed in this code involves more trickery.

We know that at the beginning of time, the sun-earth vector and the earth-moon vector are aligned along the x-axis with zero y component. This means that their cross product vanishes. One synodic month is defined as the time taken for the moon to reach the same point again with reference to the sun and earth. This configuration will again satisfy the condition of a vanishing cross product.

It is tempting, therefore, to introduce a scheme which counts a month as the interval in time between one vanishing cross product and another. This would also count the instances where the moon is between the earth and the sun, the half-month.

In order to avoid this double-counting, one could simply divide by two. Once again, rather than taking the simple route, we will employ some trickery. By calculating the distances between the moon and sun and earth and sun, $d(s, m)$ and $d(s, e)$ respectively, we can require that a month is considered to have passed only if the cross product vanishes and $d(s, m) > d(s, e)$.

There then arises the problem of determining when the cross product goes to zero. One could say that the cross product, X, vanishes if it satisfies $|X| \leq \varepsilon$.

This is not sufficient, as one cannot determine an appropriately small value of ε such that the value of X will lie within $[-\varepsilon, \varepsilon]$ when it approaches zero, nor can one conclude that it will not fall into this range more than once.

The solution is to set ε relatively large and prime the counter such that it counts when X enters the range, but only once. The counter was primed when X becomes negative and "unprimed" when the month is counted, meaning that the program only wants to count the month when X is in the lower half-plane.

For a step-size of $h = 0.01$ and running over $a = 15000$ iterations, the planet experiences 126 months.

The result is a month of average simulation time 1.1890. Therefore, an average year consists of 11.0665 average months.

A Input Data

The program was written to accept a data file from the command line using `<fstring>` and `<sstring>` in the `SolarSystem` class. This data file contained all the initial conditions for an arbitrary number of planets.

The format of the input data is illustrated in the following sample.

```
1 Number of planets
2 Mass of planet
3 x    y    v_x  v_y
```

`x`, `y`, `v_x` and `v_y` are the components of the initial position and velocity, respectively. Lines 2 and 3 are repeated for each additional planet.

A.1 2-Body Data

```
1 2
2 1.23
3 -0.5 0.0  0.0  -1.0
4 3.45
5 0.5  0.0  0.0  1.0
```

A.2 3-Body Data

```
1 3
2 1000.0
3 0.0  0.0  0.0  0.0
4 1.0
5 30.0 0.0  0.0  20.0
6 0.2
7 31.0 0.0  0.0  17.0019
```