# Project 3, FYS 3150 / 4150, fall 2013

Odd Petter Sand and Nathalie Bonatout

October 28, 2013

All our source code can be found at our GitHub repository: https://github.com/OPSand/Project3/

## 1 Introduction

The aim for this third project is to create an algorithm which should be able to simulation the solar system. To make things easier, we will begin by working on a system composed by only two planets, the Sun and the Earth, and supposed that since the mass of the Sun is larger that the mass of the Earth, the motion of the sun can be neglect. Then, we will progressively add more and more planets to our system. The main mathematical tool we will use to study this system is the Runge Kutta 4th order method. One of the main points here is to study the stability of our algorithm, thanks to the initial conditions that we will apply to our system.

## 2 Theory and Technicalities

### 2.1 System Sun - Earth

Newton's law of gravitation gives us the following expression:

$$F_G = \frac{G.M_{Earth}.M_{Sun}}{r^2}$$

where$M_{Sun}$ and $M_{Earth}$ are, as we can expect it, the masses of the Sun and the Earth, $r$ is the distance separating them, and $G$ the gravitational constant.Thanks to this relation and to Newton's second law of motion, we can get:

$$\frac{d^2x}{dt^2} = \frac{F_{G,x}}{M_{Earth}}$$

$$\frac{d^2y}{dt^2} = \frac{F_{G,y}}{M_{Earth}}$$

where $F_{G,y}$ and $F_{G,x}$ are the components on y and x of the gravitational force (reminder: we are working in a plane, which is why we don't take into account the z component).

We want now to transform the two above equations into a set of coupled first order differential equations.

We can express the components of $F$ by doing a projection as

$$F_{G,x} = -\frac{G.M_{Earth}.M_{Sun}}{r^2}.cos(\theta) = -\frac{G.M_{Earth}.M_{Sun}}{r^3}.x$$

$$F_{G,y} = -\frac{G.M_{Earth}.M_{Sun}}{r^2}.sin(\theta) = -\frac{G.M_{Earth}.M_{Sun}}{r^3}.y$$

where $x = rcos(\theta)$ and $y = rsin(\theta)$.

So we know that, by definition,

$$\frac{dv}{dt} = a$$

$$\frac{dx}{dt} = v$$

Applied here, we got

$$\frac{dv_x}{dt} = -\frac{G.M_{Sun}}{r^3}.x \quad \frac{dv_y}{dt} = -\frac{G.M_{Sun}}{r^3}.y$$

$$\frac{dx}{dt} = v_x \quad \frac{dy}{dt} = v_y$$

By doing this, we translated our two order differential equations into one order differential equations. We want now to use interesting set of dimensions.

It is relevant here to look a bit to the International System. We usually process things by using second as the unit of time variable, but here, we are dealing with celestial bodies. Thus, we will process our data with year as time unit. Furthermore, we will use the A.U. (i.e. Astronomical unit, the distance Sun-Earth) as the unit for the distances. So in terms of units, we have here -penser à ajouter l'analyse dimensionnelle ici–

So we have here $1AU = 1,5.10^{11}m$ and $1year = 3,2.10^7s$. The motion is supposed to be circular, thus, we have

$$M_{Earth}.a_{Earth} = F_G = G.M_{Earth}.M_{Sun} \quad \Leftrightarrow \quad M_{Earth}.\frac{v_{Earth}}{r_{Sun-Earth}}^2 = G.\frac{M_{Earth}.M_{Sun}}{r^2_{Sun-Earth}}$$

Thus :

$$v^2_{Earth} = G.\frac{M_{Sun}}{r_{Sun-Earth}}$$

Here, $r$ is, as we said it, the distance between the Sun and the Earth. Thus, equal to 1AU, according to the definition of the unit.

Thus, we can rewrite this equation as

$$v_{Earth}^2 = 4\pi^2.[\frac{(A.U)^2}{year^2}] \quad \Leftrightarrow \quad v_{Earth} = 4\pi.[\frac{AU}{year}]$$

We will now look at the way to discretize it. We are working on a continuous system, but want to make it into a discretized one. We will do the demonstration with the variable x, and assume that it works in the same way for y.

We know that

$$v_x = \frac{dx}{dt}$$

The step intervalle looks like $[t_0; t_{max}]$. We will work on $n_{Steps}$. Let's define h as $h = \frac{t_{max}-t_0}{n_{Steps}}$. Thus we have:

$$t_i = t_0 + i.h \text{ for each i belonging to } [0, n_{Steps}],$$

We can now define

$$x_i = x(t_i)$$

The general formulae of Simpson's rule is: $\int_{t_i}^{t_i+1} f(t,x)dt = \frac{1}{6}h(f(t_i, x_i) + 2f(t_{i+\frac{1}{2}}, x_{i+\frac{1}{2}}) + f(t_{i+1}, x_{i+1}))$.

Runge-Kutta's method defines four quantities, to predict and correct the value of $x_{i+1}$:

$$k_1 = f(t_i, x_i)$$

$$k_2 = f(t_{i+\frac{1}{2}}, x_{i+\frac{1}{2}}) \qquad x_{i+\frac{1}{2}} = x_i + \frac{h}{2}k_1$$

$$k_3 = f(t_{i+\frac{1}{2}}, x^*_{i+\frac{1}{2}}) \qquad y_{i+\frac{1}{2}} = x_i + \frac{h}{2}k_2$$

$$k4 = f(t_{i+1}, x_{i+1}) \qquad x_{i+1} = x_i + hk_3$$

In our case, we will have to compute this results four times by time step: once for the x and for the y positions, and once for the x and y velocities. And because x, y and their velocities are functions from the distance $r$, we will have to compute after each step the new distance between each planet to keep on working with a coherent algorithm.

This Runge-Kutta algorithm is especially interesting since we only need initial conditions to unfold it.

## 2.2 Adding more elements to our system

We saw how to compute our equations in a two elements system, but what happens if we want to add more elements?

Obviously, the expression of the forces will change. Let's assume that we now are considering a system composed by n bodies. The new expression of the force on the x axis exerced by the other planets on, for example, the Earth, will look like

$$F_{Earth,x} = \sum_{Planet!=Earth} G * \frac{m_{Planet} * m_{Earth*}(x_{Earth} - x_{Planet})}{r^3_{Earth-Planet}}$$

Thus, the expression of the derivative of the velocity will change into:

$$\frac{dv_{Earth,x}}{dt} = \sum_{Planet!=Earth} G * \frac{m_{Planet} * (x_{Earth} - x_{Planet})}{r^3_{Earth-Planet}}$$

This change will also have to be applied for y velocity.

## 2.3 Structure and tools of the code

To solve this problem, we wrote a Runge Kutta of 4 order, in the object-oriented style. Thus, we created two main classes:

- CelestialBody, the class describing every objects of the solar system. A celestialBody contains characteristics such as a name, a mass, or vectors defining the position, the velocity and the force exerced by other objects, and a boolean, to decide wether the current object should be seen by our algorithm as fixed, or not. This feature will be very useful at first, to test our algorithm on a two bodies system.

- SolarSystem, which is basically an array of CelestialBodies. Thus, this class will be used to determine the force exerced by other objects, body after body, and will also be used to plot our system.

Another class, constants, is, as we can expect it, a container of all the astrophysic constants that were useful for this project.

And to finish about program flow, all these classes were used in a main class, which was our controller.

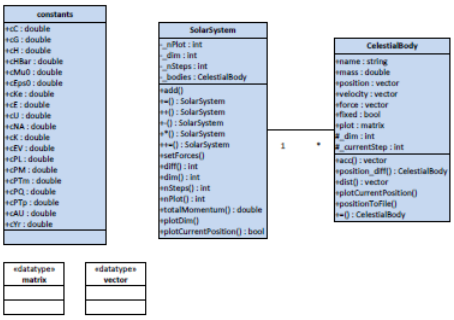
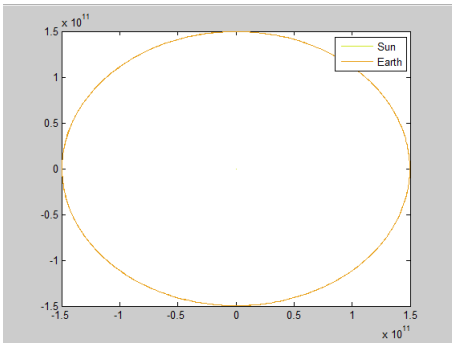



Figure 1: Class Diagram

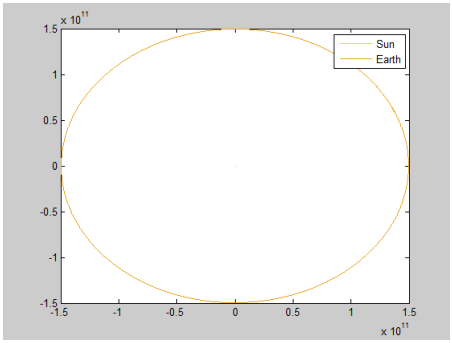
# 3 Results

## 3.1 Earth - Sun system

### 3.1.1 Circular Orbit




(a) Time step : every days




(b) Time step : every minutes

Figure 2: Plots for different time steps - length 2 years

```
Ek before: 2.67685e+033
Ep before: -8.92283e+008
L before: 2.68123e+040

Runge-Kutta...

Finished plotting 730 of 730 steps (Runge-Kutta)!

Earth circular factor (1.0 means perfect circle): 1

Ek after: 2.67685e+033
Ep after: -8.92283e+008
L after: 2.68123e+040
```

Figure 3: Conservation of the kinetic and potential energies, and of the angular momentum

We can notive that our algorithm is rather stable for our time steps. We notice too that the kinetic and potential energies are constant, as the angular mo-

mentum: once the rotation of our planet has begun, the motion can be seen as uniform: the speed is constant.Thus, the kinetic energy is a constant, since $E_c =$. $E_p = -G * \frac{m_{Earth} * m_{Sun}}{r_{Sun-Earth}}$, thus the potential energy is a constant too.

About the angular momentum: it is expressed as

$$\hat{L} = \hat{r}.\hat{p}$$

where r is position vector of our planet, and p its linear momentum. p is defined as:$\hat{p} = m * \hat{v}$.

Since the velocity is constant, and the distant between the Sun and the Earth won't change, we will have a constant angular momentum too.

### 3.1.2 Escape velocity

We now consider a planet which begins at a distance of 1 AU from the Sun. We found that an initial velocity for this planet equal to 1.72464 m.s-1 results in its escape from the Sun.

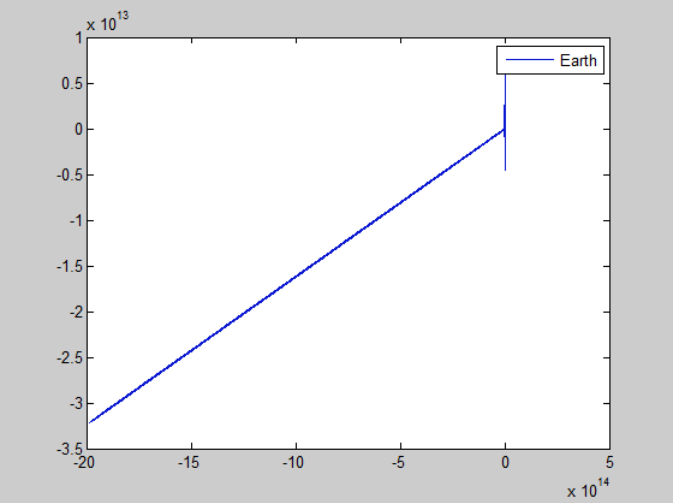



Figure 4: Escape from the Solar system

## 3.2 Earth - Jupiter - Sun system

### 3.2.1 Adding Jupiter

Among all the planets of the solar system, we chose Jupiter since it was the planet which has the biggest mass. Indeed, its mass is only 1000 times smaller than the mass of the Sun.

The main point here is to study the stability of the results after adding Jupiter into our system. We will change the mass of Jupiter, by multiplying it at first by a factor of 10, then by a factor of 1000. We expect that the change

will be really small with a factor of 10, but that our system won't be stable with the 1000 factor.

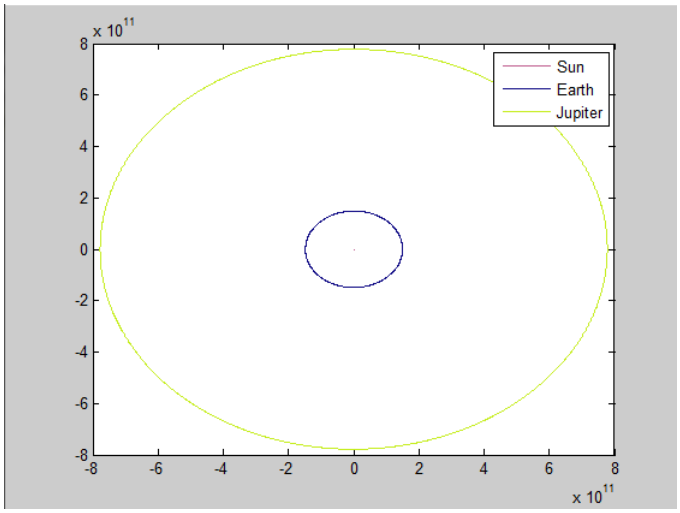



Figure 5: With the real mass of Jupiter

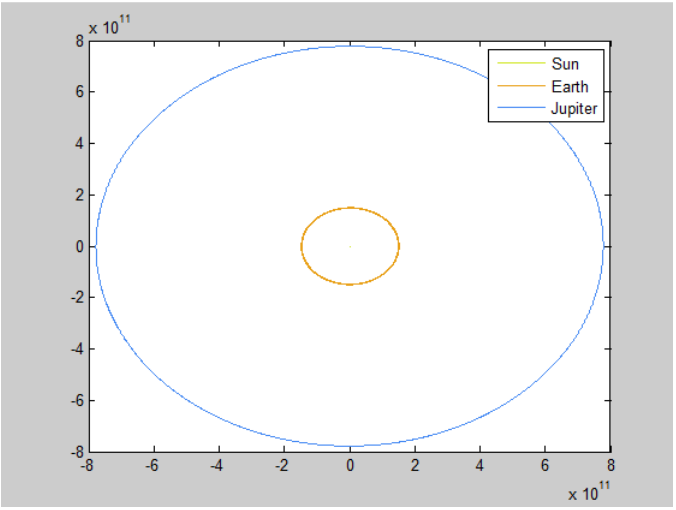



Figure 6: With the 10* mass of Jupiter

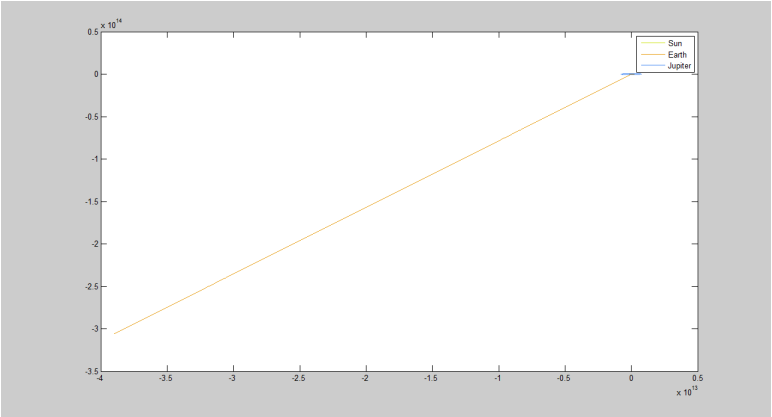



Figure 7: With the 1000* mass of Jupiter

Thanks to the plots, we can check that indeed, once that the mass of Jupiter becomes equal to the mass of the Sun, the stability of our algorithm becomes questionable: since Jupiter has a strong gravitational influence on the solar system, if we increase its mass, we completely modify our planets orbits. Thud, we can see that on the last plot, the Earth escapes the solar system.

### 3.2.2 Moving Sun

To get closer to the real system, we now consider that the Sun is moving. To do so, we will take the center of mass of our system not on the sun anymore, but on the origin of our system. Furthermore, we want the total momentum of the system to be 0, thus, we will evaluate the initial velocity of the sun as follow:

$$v_{Sun} = -\frac{\sum m_{Planet} * v_{Planet}}{m_{Sun}}$$

We found an initial velocity for the Sun of 12.40e3 m.s-1

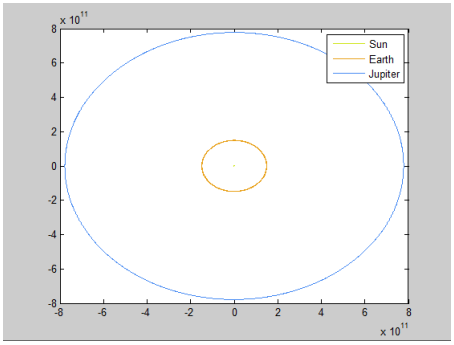



Figure 8: Sun - Jupiter - Earth in motion

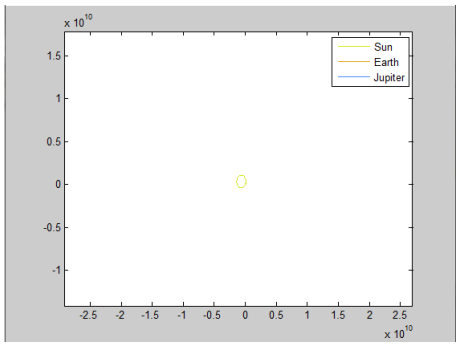



Figure 9: Sun - Jupiter - Earth in motion - Sun's motion

We can see the influence of our two planets (mainly Jupiter) on the orbit of the Sun.

### 3.3 Extending the algorithm to all the planets of the Solar system

Since our algorithm was written with the goal of being used for the entire Solar system, adding the other planets was only a matter of increasing the number of CelestialBodies. As initial values for the velocity, we took the following values:[1]

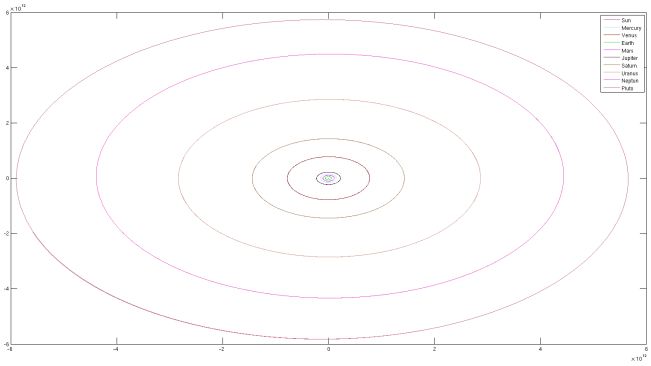
| Mercury | Venus | Earth | Mars | Jupiter | Saturn | Uranus | Neptun | Pluto |
|---|---|---|---|---|---|---|---|---|
| 47.9e3 | 35.0e3 | 29.8e3 | 24.1e3 | 13.1e3 | 9.7e3 | 6.8e3 | 5.4e3 | 4.7e3 |

Table 1: Initial velocities (m.s-1)

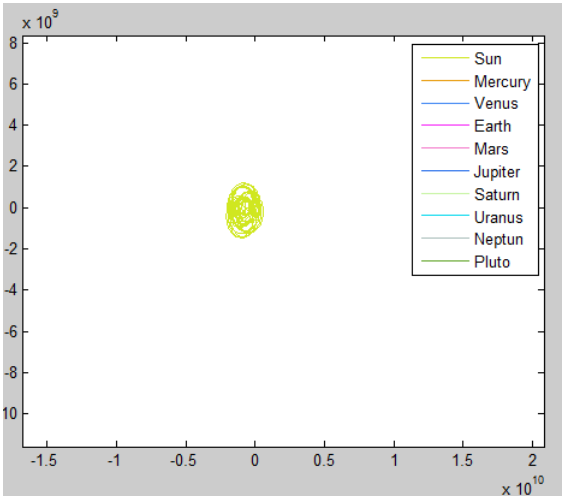
The velocity of the Sun is still evalutated in the process, to keep the total momentum of our system equal to 0.

If we want to plot entire trajectories for every planet, we now have to pay attention to the time length used to compute our algorithm. Indeed, Pluto, for example, which is the farthest planet of the Sun, has an orbital period of 90588 (i.e. 255 years) days whereas Jupiter, which was our farthest planet until now, only has an orbital period of 4331 days.

[1] Source : http://nssdc.gsfc.nasa.gov/planetary/factsheet/



(a) Entire system




(b) Sun's motion

Figure 10: Results with Runge-Kutta's method

We too implemented the Euler-Cromer method, to check our result against those of another algorithm. This is the results that we got:

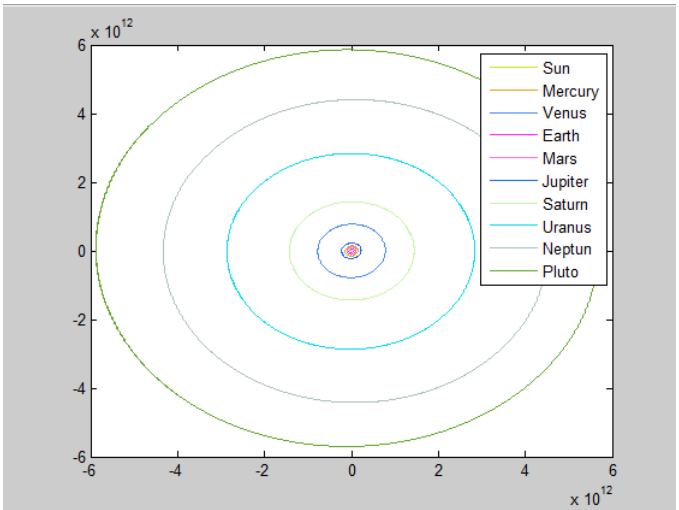



Figure 11: Results with Euler-Cromer's method

We can see that even once the all system is added, our algorithm is rather stable, and the results seem pretty coherent.

We can too notice that the motion of the Sun changed a little after we added the other planets, but that everything still behave as we expect it to.

# 4 Conclusion

From this project, we learned

- that writing an object-oriented code is easier is time is taken beforehand to look at the entire project. Indeed, we avoided rewriting classes again and again, after reflecting on the structure of our code before beginning to write it.
- the importance of checking the stability of our algorithm with our initial conditionss
- some technical specificities of C++, such as what are deep and shallow copies