Alessandro Sciarra:sciarra@th.physik.uni-frankfurt.de

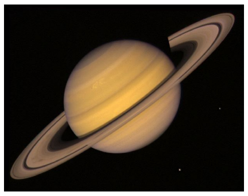
Final Project: Two dancing moons of Saturn

«*The study of gravity has led to many revolutions in science, from Newtonian Dynamics to General Relativity to Quantum Field Theory. Despite these advances, however, gravity continues to puzzle scientists. In the* 17th *century, Sir Isaac Newton was able to successfully model the gravitational force on a macroscopic scale, but in the succeeding 400 years physicists have made little progress in understanding the precise nature of this elusive force*[1]*. Determining the gravitational force exerted by an object is fairly trivial; however, trying to model gravitational interactions between multiple objects quickly becomes nearly impossible. The simplest case, the two-body problem, is the most complicated gravitational interaction that can be fully understood without some basic and unrealistic assumptions*»[2].

Janus and Epimetheus are moons of Saturn, located at orbital radii of 151 472 km and 151 422 km respectively. Janus has a mass of around 1.98×10^{18} kg and a diameter of approximately 175 km, while Epimetheus' mass and diameter are about 5.5×10^{17} kg and 105 km. Because the orbits of Janus and Epimetheus are only 50 km apart, smaller than the radii of the moons, a naïf analysis would suggest that the two moons would eventually collide. However, the gravitational interactions prevent Janus and Epimetheus from colliding with each other[3]. The final goal of this work is to use numerical integration to determine (and in theory animate) the trajectories of the moons in time and measure how often they exchange their orbits.

1 Preliminary overview

The system we want to study (Epimetheus and Janus moving around Saturn) is actually part of the Solar System. To approach this problem in the most general and realistic way is nearly unfeasable. Trying to take into account just only the motion of the other planets would mean to develop a really complicated program, probably too slow to obtain result in a reasonable interval of time (and quite certainly not so more precise than that we will write). The circular restricted three-body problem is a fairly good approximation to the Epimetheus – Janus – Saturn system. It is namely possible to consider the moon orbits as circular (the eccentricity of the orbits of Epimetheus and Janus are, respectively, 0.0098 and 0.0068) and, if needed, the two satellites massless if compared to the planet[4] ($M_{Saturn} = 5.688 \times 10^{26}$ kg). Also the angle between the orbital planes can be approximated to zero (it is 0.198°). Then, we end up with a quite simple problem, which is mathematically described by a system of 6 second order (coupled) differential equations (they are *six* because we tackle the problem in a plane, hence only *two* components of $\vec{x}_i$ survive):



$$\begin{cases} m_0\,\ddot{\vec{x}}_0(t) = \vec{F}_0\big(\dot{\vec{x}}_0(t),\,\dot{\vec{x}}_1(t),\,\dot{\vec{x}}_2(t),\,\vec{x}_0(t),\,\vec{x}_1(t),\,\vec{x}_2(t),\,t\big) \\ m_1\,\ddot{\vec{x}}_1(t) = \vec{F}_1\big(\dot{\vec{x}}_0(t),\,\dot{\vec{x}}_1(t),\,\dot{\vec{x}}_2(t),\,\vec{x}_0(t),\,\vec{x}_1(t),\,\vec{x}_2(t),\,t\big) \\ m_2\,\ddot{\vec{x}}_2(t) = \vec{F}_2\big(\dot{\vec{x}}_0(t),\,\dot{\vec{x}}_1(t),\,\dot{\vec{x}}_2(t),\,\vec{x}_0(t),\,\vec{x}_1(t),\,\vec{x}_2(t),\,t\big) \end{cases} \quad , \tag{1}$$

where the index on each variables identifies the body ($\vec{F}_i$ is the force acting on the i-th body). If we now decide to consider the velocity of each body as an independent unknown, we can reformulate the above system as one with 12

[1] *Current theories of gravity are extremely varied: Newtonian theories predict that gravity is an attractive force; General Relativity predicts that gravity is the curvature of space-time due to the presence of mass; and Quantum Field Theory says that gravity is due to the exchange of elementary particles called gravitons. Each of these theories has its own problems. The most likely candidate, General Relativity, cannot yet be reconciled with Quantum Mechanics.*

[2] Charli Sakari – `The Saturn-Janus-Epimetheus system`.

[3] This is true also in the so called *circular restricted three body problem*, approximation that we will do later on.

[4] Obviously, from now on we will neglect all the other bodies of the Solar System.

first order (coupled) differential equations:

$$\begin{cases} \dot{\vec{x}}_0(t) = \vec{v}_0(t) \\ m_0\,\dot{\vec{v}}_0(t) = \vec{F}_0\big(\vec{v}_0(t),\,\vec{v}_1(t),\,\vec{v}_2(t),\,\vec{x}_0(t),\,\vec{x}_1(t),\,\vec{x}_2(t),\,t\big) \\ \dot{\vec{x}}_1(t) = \vec{v}_1(t) \\ m_1\,\dot{\vec{v}}_1(t) = \vec{F}_1\big(\vec{v}_0(t),\,\vec{v}_1(t),\,\vec{v}_2(t),\,\vec{x}_0(t),\,\vec{x}_1(t),\,\vec{x}_2(t),\,t\big) \\ \dot{\vec{x}}_2(t) = \vec{v}_2(t) \\ m_2\,\dot{\vec{v}}_2(t) = \vec{F}_2\big(\vec{v}_0(t),\,\vec{v}_1(t),\,\vec{v}_2(t),\,\vec{x}_0(t),\,\vec{x}_1(t),\,\vec{x}_2(t),\,t\big) \end{cases} , \tag{2}$$

together with the initial conditions $\vec{x}_i(t_0)$ and $\vec{v}_i(t_0)$. This is the system we want to integrate numerically. Actually, we have not yet specialized it to our problem, i.e. we have not yet explicitly written the force acting on each body. To do that, it is enough to use the Newton's law of universal gravitation ($\vec{F}_{ij}$ is the force applied on the i-th body due to the j-th body):

$$\vec{F}_{ij} = -G\,\frac{m_i\,m_j}{|\vec{r}_i - \vec{r}_j|^2}\,\hat{r}_{ij}\ , \tag{3}$$

where the constant G is equal to $6.673\,84(80) \times 10^{-11}\,\mathrm{N\,m^2\,s^{-2}}$, while $\hat{r}_{ij}$ is the unit vector from the j-th body to the i-th body. The initial conditions are such that, for example, Saturn is in the origin of the coordinates and the two satellites are on proper circular orbits. Let us do now a not so short excursus in order to introduce some basic numeric integrator that will be used for our simulation.

2 Integrating Ordinary Differential Equations

The first big step of the project will be to get in touch with algorithms to solve numerically (systems of) differential equations. It is highly recommended to study this topic before starting the project[5].

First Order Differential Equations

Just to help you, here there are summarized the mainly used numerical methods. To fix the notation, we will consider one first order differential equation

$$\dot{x}(t) = f\big(x(t),\,t\big)$$

with the initial condition $x(t_0) = x_0$. The general procedure is to discretize the t variable in steps of size h and to build iteratively the solution at the *times* $(t_0 + h), (t_0 + 2h), \ldots, (t_0 + \Delta t)$. In general, having the solution for $t = t_n$ (we will use the shorter notation x_n instead of $x(t_n)$, but do not be confused with the body index used in § 1), the solution at the time $t = t_n + h$ can be derived (again, $x(t_n + h) \equiv x_{n+1}$).

Euler's method

$$x_{n+1} = x_n + \dot{x}(t_n)\,h + \mathcal{O}(h^2)$$
$$\Downarrow$$
$$\boxed{x_{n+1} \approx x_n + f(x_n, t_n)\,h}$$

Unfortunately, it can be shown that the error at the end of the interval (after N steps), the so called *final global error*, is $\mathcal{O}(h)$, namely terms of order h. A too small step is required to have good precision and, hence, simulations slow drastically down.

[5]A good starting point is the Euler's method in http://en.wikipedia.org/wiki/Numerical_ordinary_differential_equations and then http://en.wikipedia.org/wiki/Runge-Kutta_method. Also the Charli Sakari article `The Saturn-Janus-Epimetheus system`, § 7 is very good.

Introductory exercise (1): Let us start to solve a really simple problem, quite scorrelated from the main task of the project but that will make you more familiar with numerical integration of differential equations. Consider the following differential equation

$$\dot{x}(t) = -\alpha\, x(t)\ ,$$

where α is a real positive constant. Write a program that solves it with the Euler's method. Note that, so far, we are working in one dimension and, then, you have only one variable to deal with. It should be neither difficult nor long. The best way to check if your code is correct is to compare the resulting data with the analytc solution. Fixing $\alpha = 1\,\mathrm{s}^{-1}$ and $x(0) = 1\,\mathrm{m}$ the analytic solution is

$$x(t) = e^{-\alpha\, t}\ .$$

You can make your program prepare an output file for gnuplot with two columns: time and position. Then use the following commands to check the correctness of your code (`data` is the data filename):

```
plot "data" u 1:2
replot exp(-x) lc 3
```

In order to give you a idea of how important is the numeric precision, try to integrate the differential equation between $t_0 = 0\,\mathrm{s}$ and $t_f = 20\,\mathrm{s}$, first with $h = 0.5\,\mathrm{s}$, then with $h = 0.1\,\mathrm{s}$ and lastly with $h = 0.01\,\mathrm{s}$ (save data to file not more often than 0.05 s, otherwise the plot will be not so *nice*).

Second order Runge-Kutta

$$x_{n+1} = x_n + \dot{x}(t_n)\, h + \frac{1}{2}\,\ddot{x}(t_n)\, h^2 + \mathcal{O}(h^3)$$

How to obtain $\ddot{x}(t_n)$?

$$\dot{x}\Big(t_n + \frac{h}{2}\Big) \approx \dot{x}(t_n) + \ddot{x}(t_n)\,\frac{h}{2} \quad\Rightarrow\quad \ddot{x}(t_n) \approx \frac{2}{h}\Big[\dot{x}\Big(t_n + \frac{h}{2}\Big) - \dot{x}(t_n)\Big]$$

Therefore

$$x_{n+1} = x_n + \dot{x}\Big(t_n + \frac{h}{2}\Big)\, h + \mathcal{O}(h^3)\ .$$

The algorithm can be then easily summarized.

- $k_1 = h\, f(x_n,\, t_n)$
- $k_2 = h\, f\Big(x_n + \frac{k_1}{2},\, t_n + \frac{h}{2}\Big)$
- $x_{n+1} \approx x_n + k_2$

Fourth order Runge-Kutta

- $k_1 = h\, f(x_n,\, t_n)$
- $k_2 = h\, f\Big(x_n + \frac{k_1}{2},\, t_n + \frac{h}{2}\Big)$
- $k_3 = h\, f\Big(x_n + \frac{k_2}{2},\, t_n + \frac{h}{2}\Big)$
- $k_4 = h\, f(x_n + k_3,\, t_n + h)$
- $x_{n+1} \approx x_n + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6}$

The main advantage to the Runge-Kutta Methods is that they have $\mathcal{O}(h^N)$ as *final global error*, meaning that we can choose a specific order N to minimize the error; generally a common choice is $N = 4$. This provides four derivatives, which will help to make the approximation more accurate; however, it does not use more derivatives than necessary so as to save computing time.

Second Order Differential Equations

Apparently, the above methods are suitable for first order differential equations. Nevertheless, as already done in § 1, one can always consider the first derivative as unknown and then map a second order differential equation in a system of *two coupled* first order differential equations.

$$\ddot{x}(t) = f\big(\dot{x}(t), x(t), t\big) \quad \Rightarrow \quad \begin{cases} \dot{x}(t) = v(t) \\ \dot{v}(t) = f\big(v(t), x(t), t\big) \end{cases}$$

In this way, one can generalize all the above methods to a second order differential equation and, in principle, also to higher order equations. Since in our project you will be asked to implement the Fourth Order Runge-Kutta method, let us now generalize it to the system of equations above (the generalization of the second order method is not here reported).

- $k_1 = h\, v_n$
- $w_1 = h\, f(v_n, x_n, t_n)$
- $k_2 = h\left(v_n + \dfrac{w_1}{2}\right)$
- $w_2 = h\, f\left(v_n + \dfrac{w_1}{2}, x_n + \dfrac{k_1}{2}, t_n + \dfrac{h}{2}\right)$
- $k_3 = h\left(v_n + \dfrac{w_2}{2}\right)$
- $w_3 = h\, f\left(v_n + \dfrac{w_2}{2}, x_n + \dfrac{k_2}{2}, t_n + \dfrac{h}{2}\right)$
- $k_4 = h\,(v_n + w_3)$
- $w_4 = h\, f(v_n + w_3, x_n + k_3, t_n + h)$
- $x_{n+1} \approx x_n + \dfrac{k_1}{6} + \dfrac{k_2}{3} + \dfrac{k_3}{3} + \dfrac{k_4}{6}$
- $v_{n+1} \approx v_n + \dfrac{w_1}{6} + \dfrac{w_2}{3} + \dfrac{w_3}{3} + \dfrac{w_4}{6}$

Introductory exercise (2): Again, a short pause to deal with something simple is important. Let us solve another simple problem, that will make you deal with the numerical integration of a second order differential equations. Consider the harmonic oscillator differential equation

$$\ddot{x}(t) = -\omega^2\, x(t) \equiv f\big(v(t), x(t), t\big) \,,$$

where ω is a real positive constant. Write a program that solves it with the Euler's method:

$$\begin{cases} \dot{x}(t) = v(t) \\ \dot{v}(t) = f\big(v(t), x(t), t\big) \end{cases} \quad \Rightarrow \quad \begin{cases} x_{n+1} = x_n + h\, v_n \\ v_{n+1} = v_n + h\, f\big(v(t), x(t), t\big) \end{cases}$$

Note that, also in this case, we are working in one dimension and then it should be neither difficult nor long. Test the correctness of your code comparing the resulting data with the analytc solution. Fixing $\omega = 1\,\mathrm{s}^{-1}$, $x(0) = 0\,\mathrm{m}$ and $v(0) = 1\,\mathrm{m/s}$ the analytic solution is

$$x(t) = \sin(\omega\, t)\ .$$

You can make your program prepare an output file for gnuplot with three columns: time, position and velocity. Then use the following commands to check the correctness of your code (`data` is the data filename):

```
plot "data" u 1:2
replot sin(x) lc 3
```

to compare the position as function of time, or

```
plot "data" u 1:3
replot cos(x) lc 3
```

to compare the velocity as function of time. If you plot velocity as function of position,

```
set size square
plot "data" u 2:3
```

you should obtain a circle of radius 1. In order to give you a idea of how important is the numeric precision, try to integrate the differential equation between $t_0 = 0\,\mathrm{s}$ and $t_f = 20\,\mathrm{s}$, first with $h = 0.1\,\mathrm{s}$, then with $h = 0.01\,\mathrm{s}$ and lastly with $h = 0.001\,\mathrm{s}$ (save data to file not more often than $0.05\,\mathrm{s}$, otherwise the plot will be not so *nice*).

Systems of Differential Equations

Let us now do a further generalization, that will lead us back to the physical problem. Whenever we are interested in studying a N-body system, we will have to solve the system

$$m_i\,\ddot{\vec{x}}_i(t) = \vec{F}_i\big(\dot{\vec{x}}_0(t),\,\ldots,\,\dot{\vec{x}}_N(t),\,\vec{x}_0(t),\,\ldots,\,\vec{x}_N(t),\,t\big)\,,$$

nothing but the prototype of Eq. (1). This can be rewritten as

$$\begin{cases}\quad\dot{\vec{x}}_i(t) = \vec{v}_i(t)\\[2ex] m_i\,\dot{\vec{v}}_i(t) = \vec{F}_i\big(\vec{v}_0(t),\,\ldots,\,\vec{v}_N(t),\,\vec{x}_0(t),\,\ldots,\,\vec{x}_N(t),\,t\big)\end{cases}\,,$$

nothing but the prototype of Eq. (2). Having N bodies and solving the problem in D dimensions means to deal with a $2\times D\times N$ coupled first order differential equation system. Again, we can write the 4^{th} Runge-Kutta method for such a system. It is quite trivial, on condition that the quantities $\vec{k}_\alpha$ and $\vec{w}_\alpha$ (now D-dimensional vectors) are **all** updated, before updating $\vec{k}_{\alpha+1}$ and $\vec{w}_{\alpha+1}$ ($\vec{k}_\alpha$ and $\vec{w}_\alpha$ go from $\vec{k}_1$ and $\vec{w}_1$ to $\vec{k}_4$ and $\vec{w}_4$, respectively).

$$\texttt{I}\begin{cases}\vec{k}_{1i} = h\,\vec{v}_i(t_n)\\[2ex] \vec{w}_{1i} = h\,f\big(\vec{v}_j(t_n),\,\vec{x}_j(t_n),\,t_n\big)\end{cases}\qquad \texttt{II}\begin{cases}\vec{k}_{2i} = h\left(\vec{v}_i(t_n) + \dfrac{\vec{w}_{1i}}{2}\right)\\[2ex] \vec{w}_{2i} = h\,f\left(\vec{v}_j(t_n) + \dfrac{\vec{w}_{1j}}{2},\,\vec{x}_j(t_n) + \dfrac{\vec{k}_{1j}}{2},\,t_n + \dfrac{h}{2}\right)\end{cases}$$

$$\texttt{III}\begin{cases}\vec{k}_{3i} = h\left(\vec{v}_i(t_n) + \dfrac{\vec{w}_{2i}}{2}\right)\\[2ex] \vec{w}_{3i} = h\,f\left(\vec{v}_j(t_n) + \dfrac{\vec{w}_{2j}}{2},\,\vec{x}_j(t_n) + \dfrac{\vec{k}_{2j}}{2},\,t_n + \dfrac{h}{2}\right)\end{cases}\qquad \texttt{IV}\begin{cases}\vec{k}_{4i} = h\left(\vec{v}_i(t_n) + \vec{w}_{3i}\right)\\[2ex] \vec{w}_{4i} = h\,f\big(\vec{v}_j(t_n) + \vec{w}_{3j},\,\vec{x}_j(t_n) + \vec{k}_{3j},\,t_n + h\big)\end{cases}$$

$$\texttt{V}\begin{cases}\vec{x}_{n+1} \approx x_n + \dfrac{k_1}{6} + \dfrac{k_2}{3} + \dfrac{k_3}{3} + \dfrac{k_4}{6}\\[2ex] \vec{v}_{n+1} \approx v_n + \dfrac{w_1}{6} + \dfrac{w_2}{3} + \dfrac{w_3}{3} + \dfrac{w_4}{6}\end{cases}$$

In the above equations, the index i indentifies the body while the index j has be used as shortcut to mean that f depends in principle on positions and velocities of all bodies:

$$f\big(\vec{v}_j(t_n),\,\vec{x}_j(t_n),\,t_n\big) \equiv f\big(\vec{v}_1(t_n),\ldots,\,\vec{v}_N(t_n),\,\vec{x}_1(t_n),\ldots,\,\vec{x}_N(t_n),\,t_n\big)\,.$$

Note: The red quantities are to be used as arguments of f. It is then better to calculate them in previous step.

3 The precision of the algorithms and (semi) Step–Adaptive procedure

As already mentioned, the smaller the step of the explained methods the bigger the precision of the algorithms. Obviously, a smaller step makes the algorithm slower. A good compromise between precision and performance is mostly important when solving differential equations. In particular, one would like to have bigger step when position and velocity are almost constant, while smaller steps if they vary quickly. In other words, it would be optimal to be able to change the step size depending on the error we are going to introduce.

Let us suppose for a moment to know the error Δ that we are introducing on the quantity A between the times t_i and $t_i + h$. If $\varepsilon_{\rm abs}$ and $\varepsilon_{\rm rel}$ are the absolute and relative errors I can admit, the step size can be adapted in the following way:

- $\Delta \leqslant \Delta_{\rm max}$ the step is increased,
- $\Delta > \Delta_{\rm max}$ the step is decreased (and the quantities are calculated again).

Above, $\Delta_{\max} = \max(A\,\varepsilon_{\text{rel}}, \varepsilon_{\text{abs}})$. Supposing $\Delta \propto h^n$, the criterion to change the step is

$$h_{\text{new}} = h_{\text{old}} \left| \frac{\Delta_{\max}}{\Delta} \right|^{\frac{1}{n}}$$

or, more easily, $h_{\text{new}} = 2\,h_{\text{old}}$ or $h_{\text{new}} = 0.5\,h_{\text{old}}$. But how do we find a good estimation for the error we are going to introduce? There are really a lot of different strategies. Here you find summarized those which you are asked to use in your project.

Energy conservation

If the system is such that the total energy is conserved, one can use it as the quantity A above. Once the maximum variation of energy allowed has been established, the step size is increased or decreased comparing the total energy before and after the step. Of course, this is not a general method and the energy calculation has to be rewritten changing problem.

1 step VS 2 steps

Starting from the *same* configuration, one step of size h and two of size $h/2$ are done. The estimation of the error will be the maximum difference between the positions after one big step and those after two small steps. If this difference is below threshold, the new configuration is saved and the step size doubled (or adapted using the general criterion); otherwise the configuration before the step is restored and the step halved (or, again, adapted using the general criterion).

4 Epimetheus – Janus – Saturn: The Code Design

Since this is probably the first big project of your life, you will be almost guided through it. It is always important to think about the structure we want to give to our program, in order to start coding with well defined ideas. Let us divide our design in two part: everything related to the integration method and the study of the problem itself.

The 4^{th} Order Runge-Kutta method for a 3 body system in 2 dimensions

The first thing to think about is how to store data into variables. We have basically two possibilities.

- One array $2 \times N \times D$. It is not so easy to be handled (a sort of super-index is required), but in general is a good choice if you want to extend your code to differential equations of order higher than second.
- Two matrices $N \times D$. This is the more physical approach but its generalization is less straightforward.

In Figure 1 and Figure 2 you can see a sketch of what has been said. Choose that you like more.

Now let us focus on how to implement the numerical integration. The first thing we can do is to list all the probable variables and functions we will need. Supposing to work with two matrices for position and velocity, we could need:

```c
#define N 3
#define D 2
//variables
double t, h, prec;
double x[N][D], v[N][D], m[N];
double k1[N][D], w1[N][D];
double k2[N][D], w2[N][D];
double k3[N][D], w3[N][D];
double k4[N][D], w4[N][D];
double xtmp[N][D], vtmp[N][D];
//functions prototypes
double fInternal(int i, int j, int d, double t, double x[N][D], double v[N][D]);
double fExternal(int i, int d, double t, double x[N][D], double v[N][D]);
double Energy(double t, double m[N], double x[N][D], double v[N][D]);
double acceleration(int i, int d, double t, double m[N], double x[N][D], double v[N][D]);
```

Figure 1: *One array* $2 \times N \times D$

```
#define N 3
#define D 2
//...
//Array declaration
double x[N*D*2];
//Loop
for(int n=0; n<N; n++){
  for(int d=0; d<D; d++){
    x[    2*n*D+d] = ... // pos.
    x[(2*n+1)*D+d] = ... // vel.
  }
}
```

Typical piece of code

`x[0]`	$\Leftrightarrow$	Component 1 of $\vec{x}_1$
`x[1]`	$\Leftrightarrow$	Component 2 of $\vec{x}_1$
`x[2]`	$\Leftrightarrow$	Component 1 of $\vec{v}_1$
`x[3]`	$\Leftrightarrow$	Component 2 of $\vec{v}_1$
`x[4]`	$\Leftrightarrow$	Component 1 of $\vec{x}_2$
`x[5]`	$\Leftrightarrow$	Component 2 of $\vec{x}_2$
`x[6]`	$\Leftrightarrow$	Component 1 of $\vec{v}_2$
`x[7]`	$\Leftrightarrow$	Component 2 of $\vec{v}_2$
...		

Memory structure

Figure 2: *Two matrices* $N \times D$

```
#define N 3
#define D 2
//...
//Matrices declaration
double x[N][D];
double v[N][D];
//Loop
for(int n=0; n<N; n++){
  for(int d=0; d<D; d++){
    x[n][d] = ... // pos.
    v[n][d] = ... // vel.
  }
}
```

Typical piece of code

`x[0][0]`	$\Leftrightarrow$	Component 1 of $\vec{x}_1$
`x[0][1]`	$\Leftrightarrow$	Component 2 of $\vec{x}_1$
`v[0][0]`	$\Leftrightarrow$	Component 1 of $\vec{v}_1$
`v[0][1]`	$\Leftrightarrow$	Component 2 of $\vec{v}_1$
`x[1][0]`	$\Leftrightarrow$	Component 1 of $\vec{x}_2$
`x[1][1]`	$\Leftrightarrow$	Component 2 of $\vec{x}_2$
`v[1][0]`	$\Leftrightarrow$	Component 1 of $\vec{v}_2$
`v[1][1]`	$\Leftrightarrow$	Component 2 of $\vec{v}_2$
...		

Memory structure

```
void Next(double *t, double h, double m[N], double x[N][D], double v[N][D]);
void NextError(double *t, double *h, double m[N], short energy_check);
```

The meaning of the above code is quite intuitive, in any case let us give a brief description of each part.

- `double t, h, prec;`
  These are, respectively, the *time* of the simulation (that is not the CPU time...), the *step size* of the numerical integration and the *precision* of the integration.

- `double x[N][D], v[N][D], m[N];`
  These are, basically, the data of the bodies: *positions*, *velocities* and *masses*.

- `double k_[N][D], w_[N][D];`
  These 8 variables will be used to store the quantities $\vec{k}_i$ and $\vec{w}_i$ (see last part of § 2).

- `double xtmp[N][D], vtmp[N][D];`
  These are needed to calculate the red quantities of the 4th Runge Kutta methods (see last part of § 2).

- `double fInternal(int i, int j, int d, double t, double x[N][D], double v[N][D]);`
  This function must return the `d`-th component of the force acting on the `i`-th body due to the `j`-th body (at time `t`). In principle, it can depend on both the positions and the velocities of all bodies.

- `double fExternal(int i, int d, double t, double x[N][D], double v[N][D]);`
  This function must return the `d`-th component of the *total* force acting on the `i`-th body due to exteral causes (at time `t`). In principle, again, it can depend on both the positions and the velocities of all bodies.

- `double Energy(double t, double m[N], double x[N][D], double v[N][D]);`
  This is the total energy of the system at time `t`. The masses, the positions and the velocities are the ingredients to calculate it.

- `double acceleration(int i, int d, double t, double m[N], double x[N][D], double v[N][D]);`
  This function must return the `d`-th component of the acceleration of the `i`-th body (at time `t`). In principle, it can depend on both the positions and the velocities of all bodies.

- `void Next(double *t, double h, double m[N], double x[N][D], double v[N][D]);`
  This function will update once all quantities (`x[N][D`, `v[N][D]` and `t`), integrating the differential equations using a step size `h`.

- `void NextError(double *t, double *h, double m[N], short energy_check);`
  Function analogous to the previous one. The update will be checked against the precision `prec`. If `energy_check` is equal to `0` then the *1 step VS 2 steps* criterion will be used, otherwise the *Energy conservation* one is adopted.

Of course *not* all the variables above are to be declared inside the `main`, since they are needed only in some functions (or only in one). And maybe there could be something that can be done differently.

[More advanced – though general – approach] A good way to keep your code ordered and readable is to define a *structure* inside which all the relevant quantities will be stored. If you are not so familiar with the `struct` declaration and the `typedef` keyword, you are highly encouraged to study them[6]. Inside structures, pointers to functions can be declared. This can be useful to include in our structure those functions that are not in common to every problems. Think, for example, of the force acting on a body: in our case it will be the sum of contributions like Eq. (3), but in general this is not true. If you want to study an anharmonic oscillator, or whatever you want, the force expression will change. Also the total energy of the system has an expression that depends on the system itself. What about the following new type?

```
#define N 3
#define D 2
//our new type
typedef struct {
 short energy_check;
 double t, h, prec;
 double x[N][D], v[N][D], m[N];
 double (*fInternal)(int, int, int, double, double[N][D], double[N][D]);
 double (*fExternal)(int, int, double, double[N][D], double[N][D]);
 double (*Energy)(double, double[N], double[N][D], double[N][D]);
} RungeKutta4;
```

Of course then some prototypes will change,

```
//functions prototypes
double _fInternal(int i, int j, int d, double t, double x[N][D], double v[N][D]);
double _fExternal(int i, int d, double t, double x[N][D], double v[N][D]);
double _Energy(double t, double m[N], double x[N][D], double v[N][D]);
double acceleration(int i, int d, double t, double x[N][D], double v[N][D]);
void Next(RungeKutta4 *rk);
void NextError(RungeKutta4 *rk);
```

and you will have one function to initialize your new type:

```
void Initialize_RK4(RungeKutta4 *rk, ...);
```

or

```
void Initialize_RK4(RungeKutta4 *rk, char *filename);
```

if you would like to use an input file.

[6]Refer to standard C books, to the lectures or to http://en.wikipedia.org/wiki/Struct_(C_programming_language) (as starting point).

The physical system: Epimetheus – Janus – Saturn

Once the numerical integrator is ready, you can start writing your simulation. Do not worry, the biggest part is done. If you want to see what actually happens to the two satellites, you will have to solve Eq. (2). But this is now quite trivial. After have declare all the needed variables in the `main`, you will call the function `Next(...)` or `NextError(...)` until the desired time interval of the simulation has been covered[7]. Of course you will have to save on an output file all the data that you will plot later on with `gnuplot`. Maybe some of the following functions could help you:

```
double From_Degree_to_Radians(double alpha);
double From_Radians_to_Degree(double alpha);
void From_Polar_to_Cartesian(double r, double theta, double *x, double *y);
void From_Cartesian_to_Polar(double x, double y, double *r, double *theta);
void Get_Velocity_Components(double v, double theta, double *vx, double *vy, short clockwise);
double From_Seconds_to_Days(double t);
double From_Days_to_Seconds(double t);
void Get_Distances_from_Center_of_Mass(double m[N], double x[N][D], double r[N]);
void Save_data_to_file(char *filename, ... );
```

Just to enlighten you about the element above, consider that:

- The angle `theta` can be either in degree or in radians. It is not important what you choose but it is crucial to be coherent. One suggestion could be to insert every angle in degree, since it is easier and then convert it in radians where needed.
- The variable `clockwise` tells the function `Get_Velocity_Components` whether the satellite is moving around Saturn clockwise (`clockwise!=0`) or counterclockwise (`clockwise==0`).

If you chose the more advanced approach, the last but one function can be modified as follows:

```
void Get_Distances_from_Center_of_Mass(RungeKutta EGS, double r[N]);
```

and, when you define it, you can get the masses and the positions of the bodies using the structure in the standard way.

```
EGS.m[i];    //to get the mass of the i-th body
EGS.x[i][d] //to get the d-th component of the position of the i-th body
```

5 Your tasks

After all the previous information, you should be able to cover the project quite independently. Nevertheless, here in the following you can find a short summary (with some further suggestion) of what you are supposed to do.

- First of all read and *study* all the material you have received: lectures note, this text and the references given, etc. Use some books if you feel that not everything is clear to you. Also a quick review of the tutorial could be useful.
- The first step is to deal with the two introductory exercises. Write them in different `.c` files and use the parameters given in § 2.
- Once your first programs are working, start to implement the Runge Kutta algorithm, following mostly the code scheme of § 4. If you did not really understand the algorithm, this is a good moment to deepen into it. For those who are more self-confident the more advanced approach is suggested. Nevertheless, if you prefer to write a simpler program, you can use the easier procedure (I will not care about this choice). In any case, if you use the new type `RungeKutta4`, remember that in the function `Initialize_RK4` you will have to assign the function pointers to the corresponding functions:

[7] If you followed the more advanced approach, now the game is even easier: you will declare in your main file a `RungeKutta4 rk` variable, you will initialize it thanks to `Initialize_RK4(&rk)` and you will call the function `Next(&rk)` or `NextError(&rk)` until the desired time interval of the simulation has been covered.

```
rk->fInternal = _fInternal;
rk->fExternal = _fExternal;
rk->Energy    = _Energy;
```

or, maybe, `rk->Energy = NULL;` if the variable `energy_check` is zero. In our simulation no external forces will be considered and then you will have

```
double _fExternal(int i, int d, double t, double x[N][D], double v[N][D]){
  return 0.0;
}
```

(this let your code be able to tackle more general problems).

- Once your integrator is ready, go on writing the functions that are listed at the end of the previous section. Think if you could need something else and, in case, develop it.

- Then start writing your `main` function. If you do not have yet thought about it, reflect on which physical data you need as input parameters for your program. Everything you need is inside this text, hence, if it seems to you that some data is missing, think about it again. Just some recall. The initial position of the two satellites must be at the right distance from Saturn, no matter where on the orbit they are put (you can choose to make them start at antipodal points). Saturn can be placed in the origin of the coordinate frame. The velocities must be such to put Epimetheus and Janus on a circular orbit around Saturn: it should not be difficult to calculate such a velocity (notice that in the program you will have to set all the velocity components correctly).

- Think about what, how and how often to save data on an output file. Remember that if you save data too often the program becomes slow and you do not gain so much information. Just to give you an idea: you should simulate the system behaviour for some years[8] (more than 5 is recommended), then do not save data every second or minute! Remember that, in the output file, lines beginning by `#` are ignored by `gnuplot` and then you can create a good heading for your data. For example:

```
#time{d}  r_cdm_Epim{m}  theta_Epim{rad}   r_cdm_Jan{m}   theta_Jan{rad}   r_cdm_Sat{m}
```

- To simplify your life: when you are sure that your program works (the first runs must be not physical, but just to check that the code does what we expect), run the first simulation with t_0 =0 d and t_f =3651 d, with `prec=0.1` and using the `NextError` function (`energy_check==0`). Using a semi–adaptive step size, it is not important the value of the variable `h`, set it to 60 s.

- Your final result should be a plot of the distance of Epimetheus and Janus from the center of mass as a function of time and/or of the angle θ around Saturn. You can also make your program find when the two satellites are exchanging their orbits, save this times either on a new file or on temporary variables and then calculate every how many days or years the exchange happens. In order to do this, you can monitor the difference of the distances of the two satellites from the center of mass and look when this changes sign. You could wonder why it is better to take the distance from the center of mass and not from Saturn. The point is that, if your simulation is not enough precise, Saturn is not more motionless (see Figure 3).

- Test your code! Do it whenever you can. In principle you should test each single function you write. And, especially, test your functions one by one: do not start writing the following if you have not tested the previous! Probably it sounds annoying to you, but this is the only way to save a lot of works later on. Even if this is a rough procedure and even if there are more sophisticated ways to do that, you can use a test file, where to copy each function to be tested (use the `main` of this new file to test it). If you want to do it in a quite elegant way, in this new file write a new `short` function for each one you want to test and make it return 0 if the test failed, something different from 0 if it successed. Then, in the `main` you can call the test functions and also build up a quite beautiful report.

```
double From_Degree_to_Radians(double alpha);      //Function to be tested
short From_Degree_to_Radians_TEST(double alpha); //Function test

//at some point in the main
short success = From_Degree_to_Radians_TEST(test_angle);
```

[8]This time is referred to the system time. It does not mean that you have *to run a program for some CPU years...* ☺

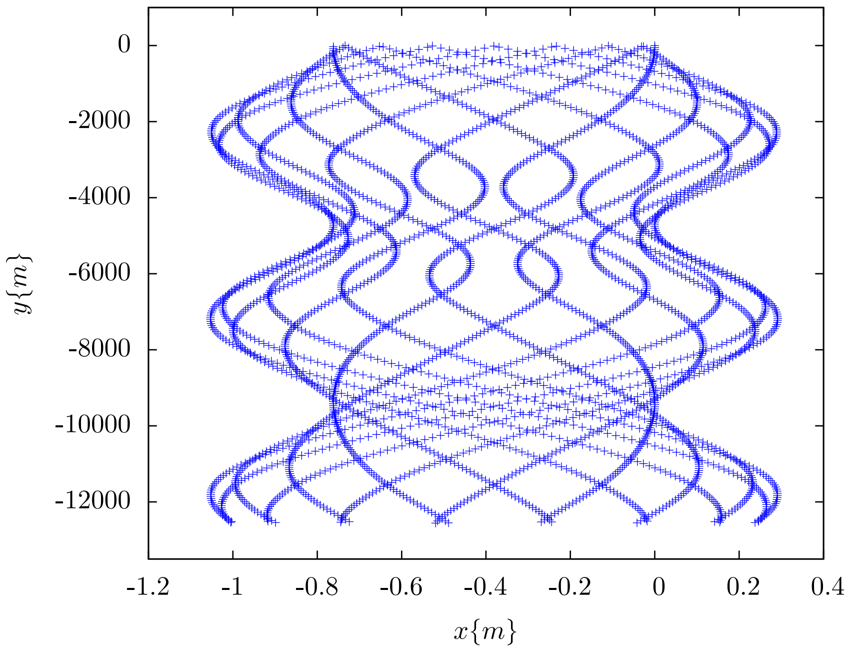


Figure 3: *The movement of Saturn in ten years of simulations. Observe that its maximum displacement is of about 14 km.*

- As final test of your code, just produce a file data `output_coord_cartesian` with

  ```
  #time{d}   x_Epim{m}   y_Epim{m}   x_Jan{m}   y_Jan{m}   x_Sat{m}   y_Sat{m}
  ```

  and plot the orbits of the two satellites with gnuplot in this way:

  ```
  set size square
  #To plot Epimetheus and Saturn
  plot "output_coord_cartesian" u ($2/1e6):($3/1e6) lc 1 title "Epimetheus"
  replot "output_coord_cartesian" u ($6/1e6):($7/1e6) lc 3 title "Saturn"
  #To plot Janus and Saturn
  plot "output_coord_cartesian" u ($4/1e6):($5/1e6) lc 5 title "Janus"
  replot "output_coord_cartesian" u ($6/1e6):($7/1e6) lc 3 title "Saturn"
  ```

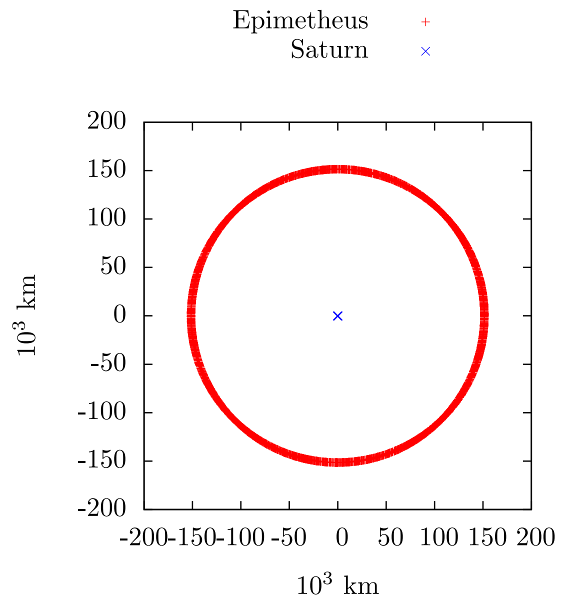
  You shoud obtain something similar to Figure 4.

- Some last suggestions. Make your code as readable as possible: choose the name of the variables and of the functions such that is obvious what they are or they do, comment your code where needed[9] and use English instead of German. Whenever you are in trouble, feel free to ask the tutor: his mail is on the first page. Naturally think about your problem, but remember *no question is stupid, only answers can be such.*

- To conclude, I would like to tell you how to call the files you are going to hand in (put them in a folder with the name `Final_Project_<surname1>_<surname2>` without angular brackets <>). This will help me a lot.

  ```
  Intr_ex_1.c              //The 1st introductory exercise
  Intr_ex_2.c              //The 2nd introductory exercise
  Runge_Kutta_4.h          //The declaration of the Runge Kutta algorithm functions
  Runge_Kutta_4.c          //The definition of the Runge Kutta algorithm functions
  Utilities.h              //Declaration of additional functions (those in the end of § 4)
  Utilities.c              //Definition of additional functions
  EJS.c                    //The simulation main program
  input_physical_data      //Physical data of the system (masses and distances of the 3 bodies)
  input_parameters         //Parameters of the simulation (t0, tf, h, prec, ...)
  output_coord_cartesian   //1st output file to check that the code is working
  output_distances         //2nd output file to see the dance of the two moons
  exchanges_periods        //3rd output file to save the periods between consecutive exchanges
  ```

[9]To add the comment `//This function sums two real numbers` is unnecessary if the function is called `Sum_two_real_numbers`.



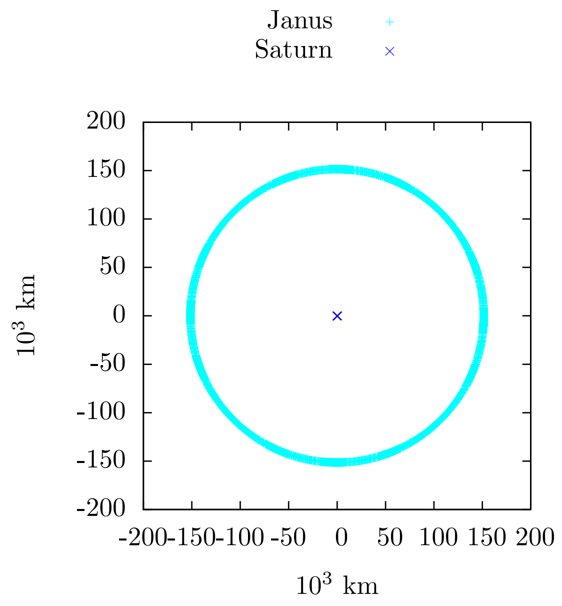


Figure 4: *The orbits of Epimetheus and Janus around Saturn in ten years of simulations.*