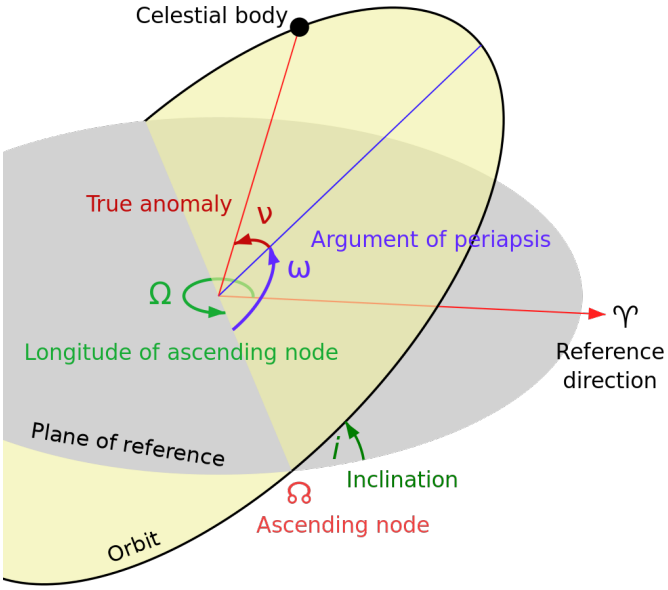
Numerical Simulation
- Project Report -
2023 - 2024

The Kozai-Lidov resonances in stellar dynamics

Cédric Accard & **Adrien Hoornaert**

Under the supervision of : **Christian Boily**



Observatoire astronomique de Strasbourg | ObAS

Table of contents

1 Introduction

In this numerical simulation report, we will explore the stellar dynamics of a triple star system to put light on a complex phenomenon : the Kozai-Lidov effect. Let's start with a brief overview of the phenomenon.

In physics, everything is well described when considering a two-body problem, but when adding a third body things gets tough. The system becomes chaotic and really sensible to initial conditions, but also some resonance phenomena can manifest. One of them being the Kozai-Lidov resonance. It was discovered separately first by Mikhail Lidov in 1961 while looking at orbits of satellites around planets and then again separately by Yoshihide Koza in 1962 who worked on orbits of asteroids perturbed by Jupiter. Since then, the interest over this effect has raised up a lot in the astrophysics community as it can have an impact on the long-term evolution of triple systems.

In fact the Kozai-Lidov effect explains that in a three body problem composed of an inner binary system and a third outer body, the orbit of the third body on an inclined plane will induce a change in the eccentricity of the inner system while it's semi major axis will remain the same. This can make a quasi-circular orbit in equilibrium break when going through very high eccentricity.

In this report we will present the method we used to make a code in Python reproducing the Kozai-Lidov resonance in a three body problem from the very start of building the code to testing the limits cases playing parameters such as initial eccentricity, plane inclination and also look at how the system behave on different time scales.

For that we will go as the following :

- Building the code : creating the parts of code to be used later such as the objects/classes but also the different integrators and the optimal time step to stabilize the binary system before adding the third object through different steps and correcting the problem it causes
- Try to recover the results of the example given before hand so that we can confirm that our code works fine
- Quantifying the impact of various parameters, such as eccentricities and inclinations, on the resonance behavior.
- Investigating the consequences of Kozai-Lidov resonances on the stability and orbital evolution of triple systems.

The possibility of a parallelization of the code will also be addressed after that. Lastly, we will finish with a brief conclusion on the work and the possibilities that could have been explored if we had more time.

2 Building the code & preliminary results

2.1 Conception

To start off the project we needed some basic elements to create a coherent and easy to use code. After loading the only three packages that we'll need (numpy, matplotlib and time), we defined easy access units so we know what we are working with later on :

- G = 6.67E-11 : Gravitational constant in N.m²/kg²
- au = 149597870700 : Mean distance Earth-Sun in meters
- msol = 2e30 : Mass of the sun in kg
- earth = 3e-6 : Mass ratio of the earth compared to the one of the sun
- hour = 3600, day = hour*24 and year = day*365.25636 : Duration of an hour/day/year in seconds

We chose to represent vectors (positions, velocities and acceleration) with `numpy` arrays. The final result of the simulation will thus consists of two arrays, one for the positions of each object at each time step, and one for their velocities. Each one of this large arrays is then of size $T \times N \times 3$, where T is the number of steps of integration, N the number of bodies and 3 the three dimensions of space.

2.1.1 Representation of bodies

To describe the celestial objects and their dynamics, we went for an object orientated programming. We represent each star by an object of a `body` class, which attributes will be updated at each time step of the simulation. An instance of the class `body` consists of :

- a name,
- a mass,
- two arrays for initial position and velocity,
- two arrays for current position and velocity,
- additional arrays (derivatives of acceleration) used for the computation with Hermite integrator.

As a starting point, we considered that each object requires its initial position and velocities (given in a rest frame) and mass to be created. This would later evolve since that in the context of this project, every body is considered to be part of a binary system, and thus initial conditions would be derived from the constraints we would put on the orbit of that system. For the sake of simplification, each quantity will be converted in the Center of Mass (COM) frame.

In that class, we also defined some basic methods to compute the different properties of the objects :

- `vec_position_in_COM`, `vec_velocity_in_COM` and `setup_coord_in_COM` to convert the coordinates of the object in the COM frame,
- `reset` to reassign the initial coordinates to the objects so we can start the simulation again,
- `display_infos` to display the name, position and velocity of the object so we can have some checks during the process,
- Some others that will be described in details later as they are useful for the integration process (`grad_potential`, `a_dot`) or inclining the orbit (`rotate_velocity`).

A major function for the problem, `initialize_body`, computes the positions and velocities of each body of a list of bodies in the COM frame and updates their initial conditions as a consequence. This function has to be called before starting any simulation.

We also created other functions outside of that class for simplicity of computation of certain properties in some other functions, we will not talk about them as they're just intermediate steps.

2.1.2 Representation of a binary system

To handle the parameters of a binary system, we created a second class called `binary_system` that will naturally contain two instance of the `body` class. The aim with this second class is mainly to compute the initial conditions of its components (two stars) to satisfy the parameters of the orbit of the system. An object of type `binary_system` consists of the following attributes :

- a list of two `bodies`,
- the position of the center of mass (in a rest frame),
- its total mass,
- parameters of the orbit (eccentricity, semi-major axis, inclination angle, and coordinates of the perihelion and aphelion).

A `binary_system` is created with two bodies. We then define the orbit with the two parameters :

- the semi-major axis : a,
- the eccentricity : e.

The initial velocities and positions of the two bodies are computed in a function `initialize_system` such as they satisfy the orbit. We also implemented a parameter `phi` to choose for the inclination of the plane, but for our binary alone it's zero for now.

2.1.3 Computation of initial conditions

To do the initialization of a system we first focused on creating a binary system to see if we could reproduced some properties of orbits such as a Earth-Sun system. To do that, we placed ourselves as said previously in the COM frame, making appear the notions of total and reduced masses. Working with those is easier to get access to initial conditions as we can define a semi-major axis a and an eccentricity e in that frame (so the reduced mass orbits around the COM with those properties) for the motion of the two bodies and the code will compute the initial conditions for us in the absolute frame before putting them in the COM. To simplify the treatment we also chose to get orthogonal initial conditions : the two bodies orbits in the same plane with opposite direction velocities (and with amplitude that scales with the body mass the constraints on a and e are full-filled) making us dealing with only one coordinate instead of three.

With that in mind we defined the `compute_positions` and `compute_velocities` functions that compute the initial position/velocities to give to the objects in COM with orthogonal initial conditions so the orbit is respecting the semi-major axis and eccentricity asked and we choose to define those initial conditions at the apoapsis of the orbit.

Therefore the positions for the bodies 1 and 2 are defined as the following :

$$x_1 = a(1+e) \times \frac{m_2}{m_1+m_2} \quad \& \quad x_2 = -a(1+e) \times \frac{m_1}{m_1+m_2} \quad \Rightarrow \quad \vec{r}_1 = \begin{pmatrix} x_1 \\ 0 \\ 0 \end{pmatrix} \quad \vec{r}_2 = \begin{pmatrix} x_2 \\ 0 \\ 0 \end{pmatrix}$$

On the other side the velocities are defined as :

$$v_1 = v_\mu \times \frac{m_2}{m_1+m_2} \quad \& \quad v_2 = -v_\mu \times \frac{m_1}{m_1+m_2} \quad \Rightarrow \quad \vec{v}_1 = \begin{pmatrix} 0 \\ v_1 \\ 0 \end{pmatrix} \quad \vec{v}_2 = \begin{pmatrix} 0 \\ v_2 \\ 0 \end{pmatrix}$$

where $v_\mu = \sqrt{(1-e) \times \frac{G(m_1+m_2)}{x_1 - x_2}}$ is the reduced mass velocity of the system.

With all of that we have now a fully initialized system with the correct condition to produce a precise motion so we should be able to look at the motion itself but we're missing one thing still : an integrator to advance in time.

2.2 Computation of motion

To compute the motion of the bodies, we define a function for each of the two previous integrators that requires a list of bodies, a time step and the duration of the simulation. At the end of computation, the functions return three arrays : the array of positions (size $T \times N \times 3$), the array of velocities (size $T \times N \times 3$) and the array of time (size T). The whole process works as follows :

1. Takes the list of initialized bodies in entry as well as a time step and a simulation time
2. Initialize the three result arrays as blank arrays,
3. for each time step :
    - (if it does not require to compute the potential) : update the new position and velocities of each body depending on previous values,
    - compute the gravitational potential seen by each body,
    - (if intermediate steps) : compute intermediate quantities using gravitational potential for each body,
    - compute and update the final positions and velocities of each body and save them in arrays (thus for one time step),
    - append the arrays of positions and velocities for this time step to the result arrays,
4. Return the three results arrays.

Note that it is crucial to segment the implementation of the integrators : one must be careful not to update the positions of the object until the gravitational potential has been computed for every body at the current time step (not to mix old values and updated ones).

Once a binary system is correctly initialized, meaning each one of its components has initial positions and velocities defined in the COM frame and satisfying the given orbit, the procedure to compute the motion is straight forward.

2.2.1 Verlet's or Leapfrog Integrator

For the Verlet's integrator, the method consists in adjusting independently the positions and velocities moving by half-time step between the updates. If we start with (**x**,**v**) for a given body, denoted with the index $\star$, at time t_0, we first shift the spatial coordinates **x** to time $t' = t_0 + \delta t/2$. where $\delta t/2$ is the said half-time step :

$$\mathbf{x}' = \mathbf{x} + \mathbf{v}\delta t/2$$

Then we adjust velocity **v** at time $t" = t_0 + \delta t$:

$$\mathbf{v}" = \mathbf{v}(t) - \nabla\phi(\mathbf{x}', t')\delta t \; = \; \mathbf{v}(t) - \sum_{i \neq \star} G\frac{m_i}{r_{i,\star}^2}\hat{\mathbf{r}}_{i,\star}\delta t$$

$-\nabla\phi(\mathbf{x}', t')$ being the derivative of the gravitational potential computed via our function `grad_potential`.

Then we repeat again the first step to go from $\mathbf{x}'$ to $\mathbf{x}"$. So in the end both of the coordinates are expressed at a time $t + \delta t$ keeping both quantities synchronous. The big advantage of this one is that the computations are not too complex to do and we thus preserve the integrals of motion up to the 2nd order. However, it necessitates small time steps δt for optimal accuracy.

2.2.2 Hermite's Integrator

The second integrator used is a bit more complex but is more accurate in the evolution of our system and this for bigger time steps. It does in fact conserves the integrals of motion up to the fourth order. To implement it we must consider not only the position and the velocity of the body but also the acceleration $\mathbf{a}$ and it's derivative $\dot{\mathbf{a}}$ so we can apply corrections to reach the high order accuracy. Starting again at ($\mathbf{x}$,$\mathbf{v}$) for the body $\star$, we first compute the set of predicted coordinates at $t' = t + \delta t$:

$$\mathbf{x}_p = \mathbf{x} + \delta t\, \mathbf{v} + \frac{\delta t^2}{2}\mathbf{a} + \frac{\delta t^3}{6}\dot{\mathbf{a}}$$

$$\mathbf{v}_p = \mathbf{v} + \delta t\, \mathbf{a} + \frac{\delta t^2}{2}\dot{\mathbf{a}}$$

where, for the $\star$ body, we have (via the function `a_dot`) :

$$\dot{\mathbf{a}}_\star = -\sum_{i\neq\star} \frac{Gm_i}{r_{i,\star}^3}\left[\mathbf{v}_{i,\star} - 3\frac{\mathbf{v}_{i,\star}\cdot\mathbf{r}_{i,\star}}{r_{i,\star}}\hat{\mathbf{r}}_{i,\star}\right]$$

Then we compute the corrections mentioned earlier :

$$\mathbf{a}^{(2)} = -\frac{6(\mathbf{a}-\mathbf{a}') + \delta t(4\dot{\mathbf{a}} + 2\dot{\mathbf{a}}')}{\delta t^2}$$

$$\mathbf{a}^{(3)} = -\frac{12(\mathbf{a}-\mathbf{a}') + \delta t(6\dot{\mathbf{a}} + 2\dot{\mathbf{a}}')}{\delta t^3}$$

And obtain the final corrected coordinates at $t + \delta t$:

$$\mathbf{x}' = \mathbf{x}_p + \frac{\delta t^4}{24}\mathbf{a}^{(2)} + \frac{\delta t^5}{120}\mathbf{a}^{(3)}$$

$$\mathbf{v}' = \mathbf{v}_p + \frac{\delta t^3}{6}\mathbf{a}^{(2)} + \frac{\delta t^4}{24}\mathbf{a}^{(3)}$$

We should also mention than when dealing with more than one body, the order of the different processes is important : each step has to be done for the three bodies before going for the next one otherwise the corrections will not be relevant and the integrator won't work as intended.

2.3 Additional computations

The computation of motion done by the integration process described before give us positions and velocities of each object, but it would be interesting to derive other quantities. For instance, at the end of the computation, we build the lists of the following quantities :

- the arrays of distances and velocity differences (vector and scalar) between the two bodies of a system (size $T \times N$),
- the array of the eccentricity of the orbit (size T),
- the array of the axis of the orbit (size T).

We also create the function `energy_from_trajectory` to compute the list of mechanical energy and angular momentum of each bodies along the duration of the simulation (sizes $T \times N$). This function also computes the theoretical energy and compares it to the one obtained by the simulation.

Note that we explicitly decided not to compute these quantities during the integration not to add blocking computation time, so we can save the motion and later do further calculations.

2.4 Comparing the integrators

Now that we had introduced our two integrators, we were able to do a comparison. As we said earlier, one should be more accurate for larger time steps whereas the other should be faster to do integration, so let's plot those two properties as a function of the time steps. For that, the function `energy_from_trajectory` allows us to recover the relative error from the completion of an orbit of a binary system. As we had in mind to reproduce a figure given to us, which we'll see later, we took as a configuration two star of a mass of $1M_\odot$ at a distance of $a = 1A.U.$ and an eccentricity of $e = 0$. This configuration does give us a period for 1 orbit of around 9 months so the comparison on relative error will be done on that duration. We want the precision to be at maximum $\delta E/E = 10^{-5}$ for the relative error comparison, and for the time of simulation we stop the comparison when one of the two has reached the required precision, so we only have simulation time for both integrator on the same time steps.

To reach the limit we built a function (`find_integration_step_method`) that, for a given simulation time, an integrator and a desired precision will find the optimal time step starting for $\delta t = 1\ yr$ and dividing by two until it reaches the wanted value of $\delta E/E$. We also implemented a timer in it to compute the time it takes to process the integration.

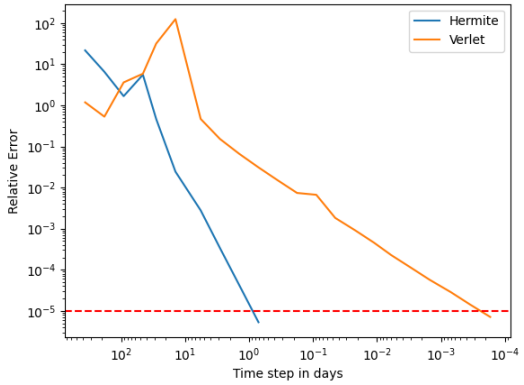


Figure 1: Relative Error $\delta E/E$ on energy for 1 orbit as a function of the time step. The red dashed line correspond to the desired limit of $\delta E/E = 10^{-5}$

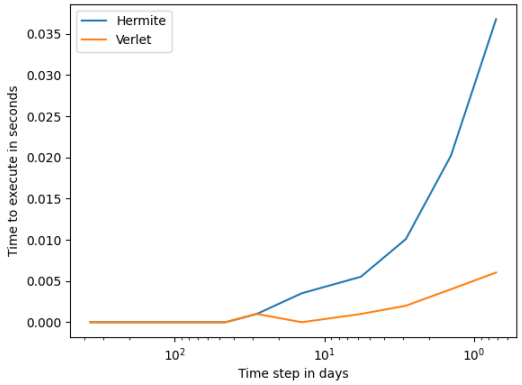


Figure 2: Time to simulate one orbit as a function of time step. Comparison stopped when one of the two reached the $\delta E/E = 10^{-5}$ limit

As we can see from the first plot, the Hermite integrator is better to obtain good precision as it only needs a time step of 0.713 days to reach to desired limit whereas the Verlet's one needs a 0.00017 day time step. If we look at the simulation time we can also see clearly that the Verlet is faster for the same time step but as we mentioned, it takes smaller time steps to reach the same precision so in the end it takes 31s to reach same precision as Hermite does in 0.037s ! That's a clear evidence that we should use the Hermite's integrator if we want to reach high precision in short amount of time.

Let's have a quick word on how we did to get a relative error measurement. To obtain such a value, we need two quantities : Theoretical and Observed energy. The first one is intrinsic of the configuration : for two bodies of masses m_1 and m_2 orbiting around each other at a distance a, E_{th} is given by :

$$E_{th} = \frac{-Gm_1m_2}{2a}$$

As we didn't implement any dissipative forces that would reduce the total energy, the value of a is always the one of the initial configuration and therefore E_{th} remains constant.

On the other side, using the gravitational potential energy and the kinetic energy for our system in the COM rest frame to compute the Hamiltonian, we get at any instant t :

$$H = \sum_i \frac{1}{2} m_i v_i^2 - \sum_{i,j\neq i} G\frac{m_i m_j}{r_{ij}} \qquad \text{giving the relative error :} \qquad \frac{\delta E}{E} = \frac{H - E_{th}}{E_{th}}$$

As in the `energy_from_trajectory`, we compute the angular momentum, we can also have a look at the deviation of $\mathbf{L}$ during one orbit for the same time step (0.713*day) with respect to the initial value : Both integrator have a linear deviation from the intial angular momentum but the values are very different 1.4% of deviation after 1 orbit for the Leapfrog integrator (Fig.3) compared to 0.00025% for the Hermite integrator (Fig.4).

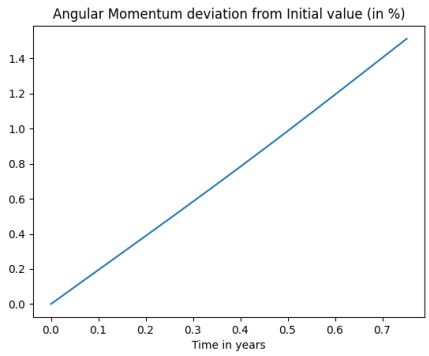


Figure 3: Deviation of Angular momentum for the Leapfrog integrator

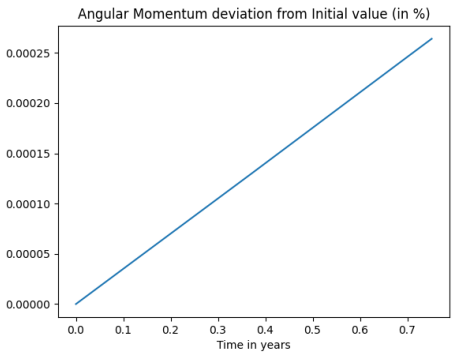


Figure 4: Deviation of Angular momentum for the Leapfrog integrator

2.5 Playing with the parameters

If we plot the system using 2 bodies of the same mass (1 $M_\odot$) on a orbit of 0 eccentricity simulated on half of the time it takes to do a full orbit (so 4 months instead of 9), we get :

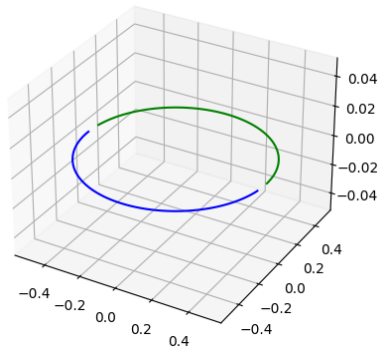


Figure 5: Simulation Time = 4 months, e=0, $m_1 = m_2 = 1\ M_\odot$

Note that the axes in Fig.5 are in A.U. and it will be the same for all the remaining figures of that type in this report

Testing eccentricity critical values : If we now try to increase the eccentricity of the orbit, we can try to see if something happens when it gets too big. Let's use our same two bodies but this time on a 5 year simulation so the system has the time to complete multiple orbits.

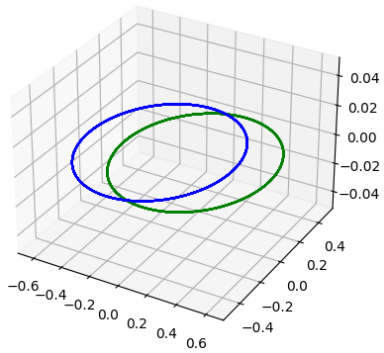


Figure 6: eccentricity=0.25

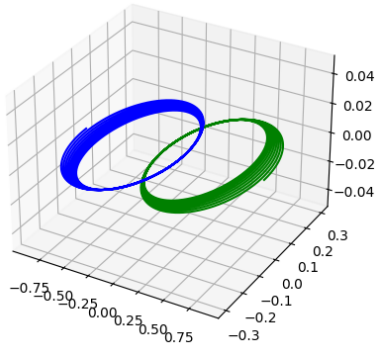


Figure 7: eccentricity=0.8

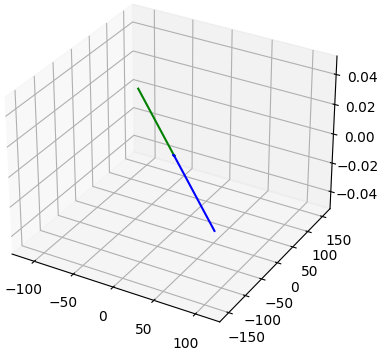


Figure 8: eccentricity=0.9

As we can see when increasing e slightly (first case), the system does stay in equilibrium, the two binaries staying at the same distance. But when going higher in e (second case), the two binaries progressively fall on each-other before at one point eject themselves of the system, here it's not happening as the simulation time isn't long enough. But when going even higher in e (third case), the system is not just stable anymore and breaks in even shorter time scales.

Recreating the Earth-Sun system : An other thing we could try to look at to evaluate the coherence of our simulation, is to simulate a Earth-Sun system to see if we can recover the orbital velocity of the Earth while the Sun shouldn't move much on his side.

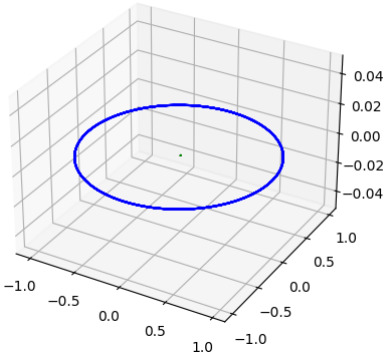


Figure 9: Simulation Time = 1 year, e=0.0167,
$m_1 = 1\ M_\odot$, $m_2 = 3 \times 10^{-6}\ M_\odot$

Looking at the visual on Fig.9, we can notice that for 1 year the Earth does in fact complete one full orbit and that the trajectory that looks circular (the eccentricity being 0.0167, the ellipticity can't really be seen with the naked eye) and at a distance of 1 A.U. of the sun on average. On the other hand, the Sun barely moves : only a dot visible at the center. If we now look at the value of the velocity of the earth at the initialization, we find ~29.37 km/s while the true value at the aphelion (where we do initialize the system) is 29.29 km/s, which gives us a relative error of 0.273%

2.6 Adding a third Object

Now that we built the code that can generate a binary system in (quasi-)equilibrium, we are able to create the system that will allow the Kozai-Lidov effect to appear. This effect requires the addition of a third body on an outer and inclined orbit. To do so, we consider the binary system as one object seen from afar by the third body. The situation is now being equivalent to a new binary system (for the point of view of the perturbator). To do so, a new method (`system_to_body`) is defined to convert a binary system to an equivalent body of mass $M = m_1 + m_2$. This pseudo-body will be used in a new binary system with the perturbator. Note that the integration of the motion will take into account the three bodies of the problem since it operates on a list of N bodies (the aim of the binary system class is to put a frame to compute initial conditions).

2.6.1 Mass-less body

To build up the proper initial condition, the easier is to start with mass-less third body. By doing so, inner binary shouldn't be affected while the outer body should be at perfectly equilibrium. In fact if the third body is mass-less, it can't affect the inner binary system but will just orbit around it with the initial condition imposed.

Using the previously mentioned assumption, it lets the outer body as one of reduced mass $\mu = \frac{M \times 0}{M+0}$ positioned at a given distance of the center occupied by the "mass M". This allowed to use the `binary_system` class once again to define the initial condition only using two parameters a_{out} and e_{out}. But that class does the initialization for two bodies only, so the inner binary won't be initialized properly without due attention being paid. To solve the problem, two initialization have to be done, the first one for the inner binary and the second one for the inner-COM and outer body system. One last thing to do is to correct the positions of the two inner stars : the values of positions returned by the first initialization are added to the ones of the second one corresponding to the center of mass of the binary. For $e_{out} = 0$ and the same configuration as before for the inner system, the final situation is shown on Fig.10. As it can be seen, the COM of the binary is a rest as $m_{out} = 0$ while the two inner bodies and the third one have velocities that allow the orbit to respect the value of semi-major axis and eccentricity we did input beforehand for both dual systems.

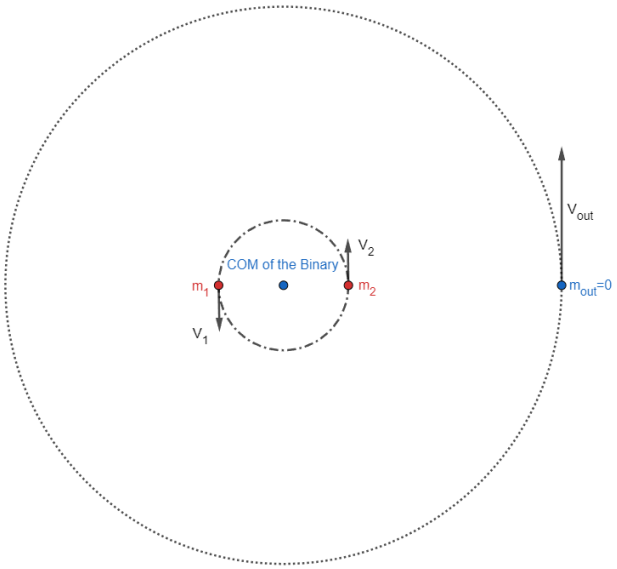


Figure 10: Schema of the initial configuration when the third body mass is $m_{out} = 0$ and $e_{out} = 0$ (vectors and distances not necessarily to scale)

2.6.2 Adding a mass : velocity of the COM correction

The second step was to add mass to the third body, but it doesn't resume as just changing one parameter. In fact if the third body is massive, it will induce effects on the binary system therefore it's COM won't at rest anymore. To correct for that, one must update the velocities of the two inner star. As the `binary_system` class is used once again, we already know what the velocity of the binary COM will be so we just have to add that to the velocities of the two individual stars : we place them in the new COM of the 3-body system. If we plot the results of a simulation for 10 years with the two cases we just saw we obtain the following :

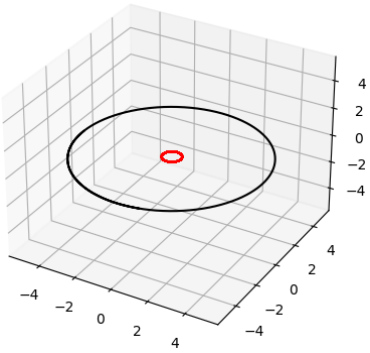


Figure 11: Mass-less third body : no correction needed at first sight

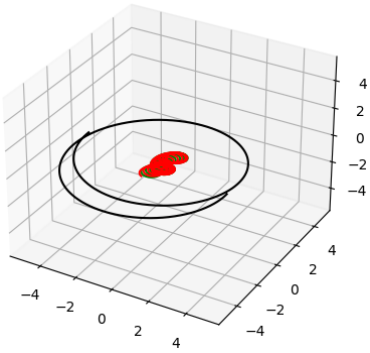


Figure 12: Third body with mass of 0.1 $M_\odot$ but no velocity correction

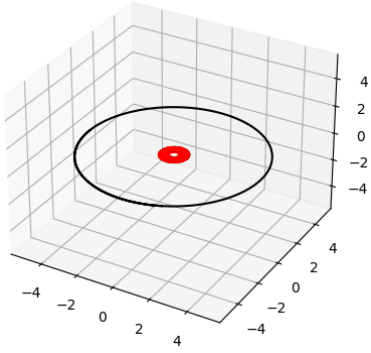


Figure 13: Third body with mass of 0.1 $M_\odot$ with velocity correction

2.6.3 Inclined orbit : correcting constant drifting

The last step to get the wanted configuration to produce the Kozai-Lidov effect is to put the third body on an inclined orbit. But this doesn't come without difficulties either. On a first glance we could say that we only need to incline the initial velocity of the third body to the wanted angle. But in fact it will create a constant drift of the COM which of course won't be at rest anymore. As we want to represent the motion in that rest frame, we have to deal with this drift. First, we use the function `rotate_velocity` to create that inclined rotation via a simple vector rotation of angle `phi`. Then to correct for the drift, we have to note that the change of inclination will make appear new components in the angular momentum on L_y and reduce the value in L_z (as we define the intial conditions only on y so far).

So for an angle ϕ the change in momentum translate to :

$$dpy = (1-\cos\phi) \times m_{out} \times \frac{\|\vec{v_{out}}\|}{(m_1+m_2+m_{out})} \quad \text{and} \quad dpz = -\sin\phi \times m_{out} \times \frac{\|\vec{v_{out}}\|}{(m_1+m_2+m_{out})}$$

Those two quantities need to be subtracted to the initial velocities of our three objects so the COM is at rest again. If we plot the result in both cases for a 100yrs simulation with the binary being 2 bodies of $1M_\odot$ at 1 A.U. and the third one being a $0.1M_\odot$ body at 5 A.U. on a $\phi = 40°$ inclined orbit, with both eccentricity at zero, we obtain Fig.14 and Fig.15. We also plotted the projections in 2D to better see what we meant by "constant" drift, the result is seen on Fig.16 and Fig.17.

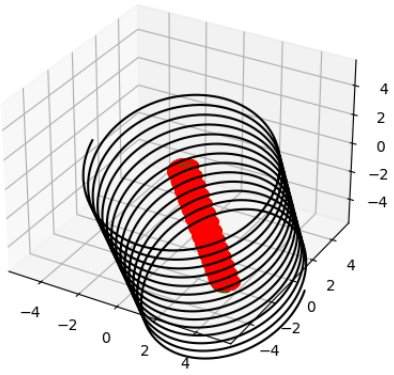


Figure 14: No correction applied

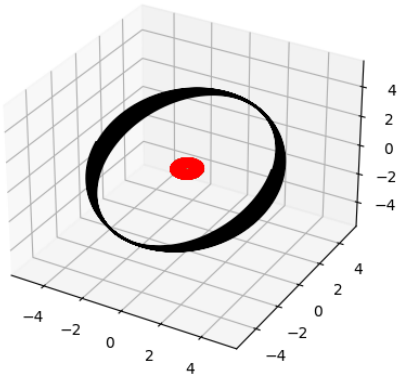


Figure 15: Correction applied

Note that the oscillations on the third object orbit (Fig.15) is not related to the Kozai-Lidov effect. In fact for a mass-less body, the same can be observed (see Fig.18). This is due to the coupling of it's motion with the one of the binary system. The only difference is that the inner system remains on the same plane compared to the massive third object case.

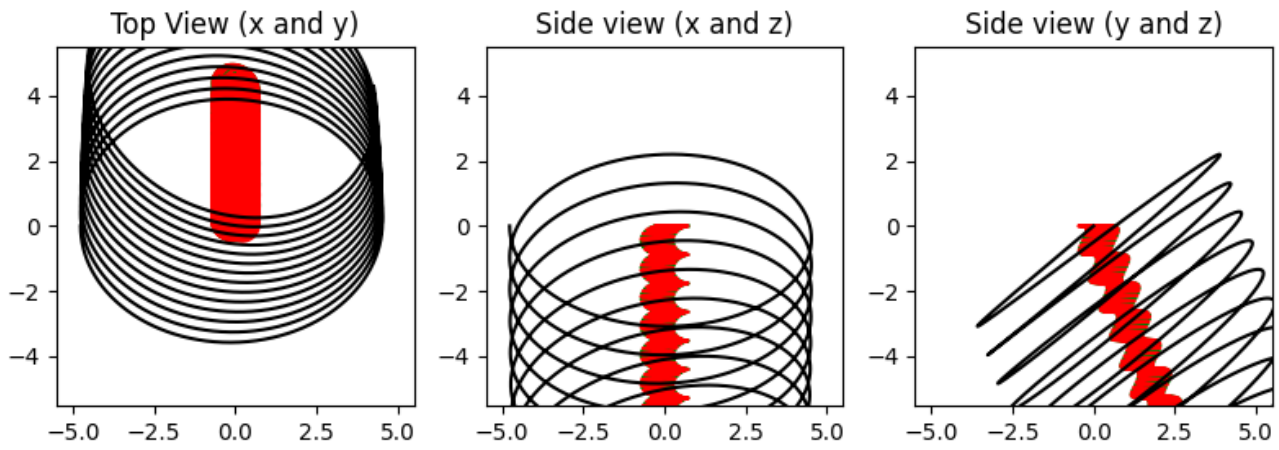


Figure 16: 2D projections when no correction applied

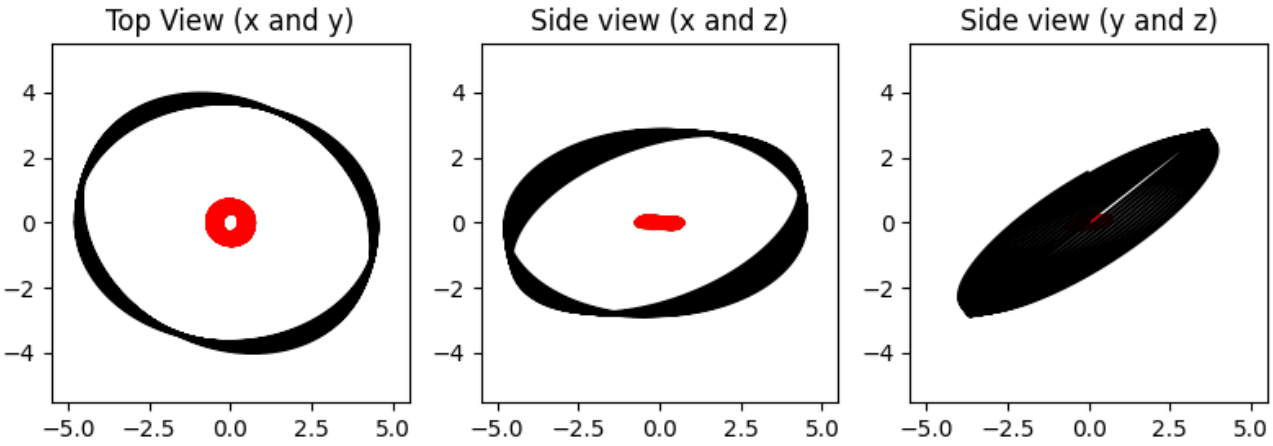


Figure 17: 2D projections when correction applied

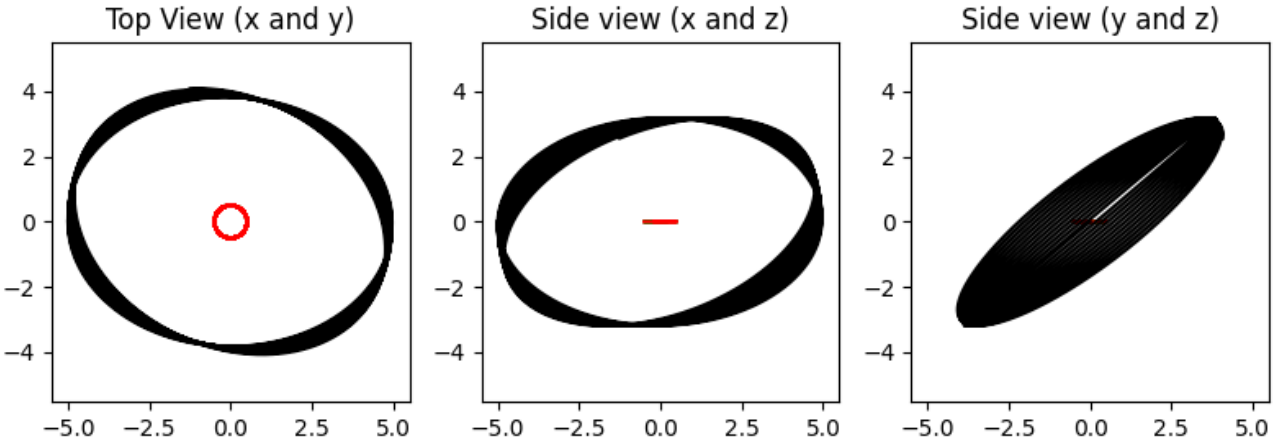


Figure 18: 2D projections with $m_{out} = 0$ (correction not needed)

3 The Kozai-Lidov effect : Results

3.1 Finding back the main result from the paper

As a preamble to this work, we were given a sheet with instructions to help us understand the project. In this paper, a figure was appearing representing the semi-major axis and the eccentricity of the inner binary of a "Kozai-Lidov compatible system" as a function of time. Our goal was then to try to reproduce it, but for that we still need two more expressions to compute the wanted parameters.

Knowing that our simulation is giving us the complete list of positions and velocities at each moment visited, we can, with the appropriate expressions, go back to the values of a_{in} and e_{in} as a function of time. If we note with indices 1 and 2 the two inner bodies of the system and by using the definition of the eccentricity vector, we obtain :

$$\vec{e}(t) = \frac{\vec{v} \times \vec{h}}{\mu} - \frac{\vec{r}}{\|\vec{r}\|} \quad \text{where} \quad \vec{v}(t) = \vec{v}_1(t) - \vec{v}_2(t) \ ,$$

$$\vec{r}(t) = \vec{r}_1(t) - \vec{r}_2(t) \ , \ \vec{h}(t) = \vec{r}(t) \times \vec{v}(t) \ \text{ and } \ \mu = G \times (m_1 + m_2)$$

$$\Longrightarrow e(t) = \|\vec{e}(t)\| \ \text{ and } \ a(t) = \frac{\|\vec{h}\|^2}{\mu \times (1 - e^2)}$$

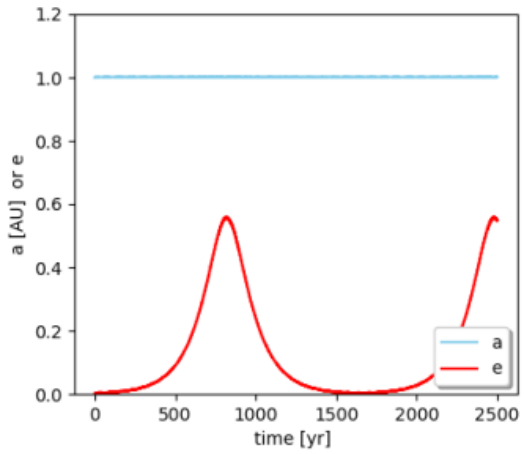


Figure 19: Time evolution of a_{in} and e_{in} that we want to reproduce. Parameters : $m_1 = m_2 = 1M_\odot$, $a_{in} = 1A.U.$, $e_{in} = 0$, $m_3 = 0.1M_\odot$, $a_{out} = 5A.U.$, $e_{out} = 0.25$, $\phi = 80°$

We now have all the tools need to represent the evolution of a_{in} and e_{in} with time. Taking the parameters suggested in Fig.19, we obtain on our side :

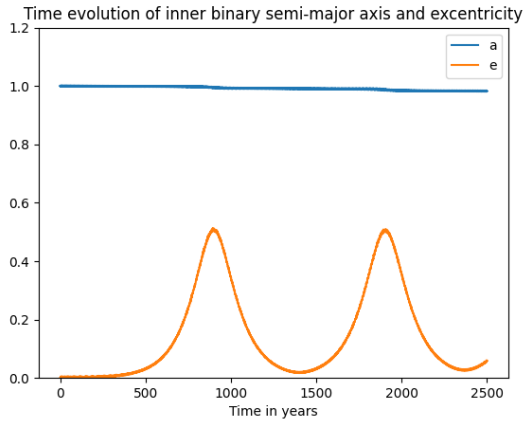


Figure 20: Time evolution of a_{in} and e_{in} using our code

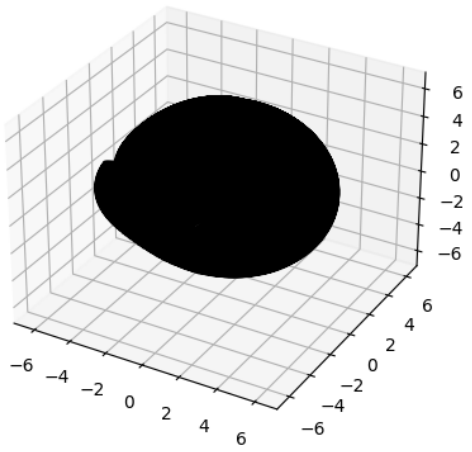


Figure 21: 3D Plot of the result

As we can see from the 3D plot, the different effect we showed earlier are still there and amplified making it impossible to see anything. So, for the remaining of the discussion, we won't show the 3D or 2D plots of the simulations unless something striking is showing up. On the other side, the time evolution plot is showing interesting results if we compare with the one of Fig.19. In fact, we can see three main differences :

- The semi-major axis seems to be slowly decreasing with time
- The eccentricity doesn't go back perfectly to 0
- The oscillation period seems to be shorter on our side (first minima at t~1400 yrs compared to t~1650yrs in the example)

To better understand the differences, we went back to a simpler configuration with a coplanar third body and we noticed a repeating pattern in both a (Fig.22) and e (Fig.23) that we think could be due to integration artifacts and thus maybe a bad implementation of the equations of motions. This could explain both points 1 and 2 in the differences we have with the example. In fact, if the motion presents instabilities, they might play a role on large time scales preventing the system to reach the initial configuration after the first oscillation.

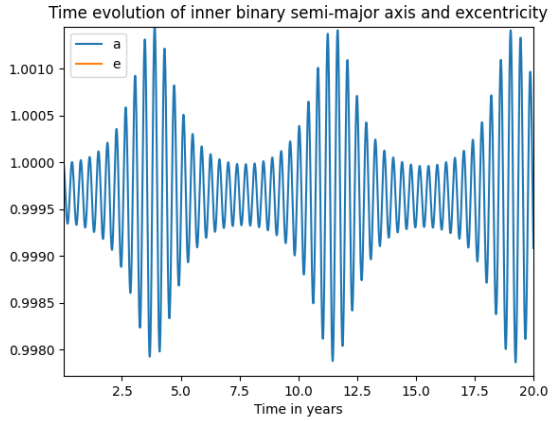


Figure 22: Oscillatory pattern seen in the behavior of a for a coplanar third body

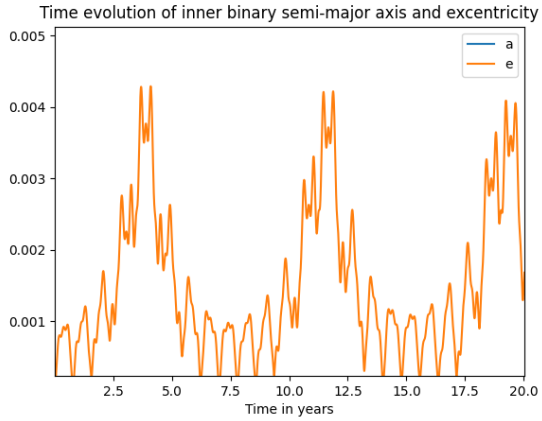


Figure 23: Oscillatory pattern seen in the behavior of e for a coplanar third body

We still have to explain the period difference. For that we will use the formula giving the Kozai-Lidov timescale T_{KL} for these cycles :

$$T_{KL} = \frac{P_{out}^2}{P_{in}} \left(\frac{1+q_{out}}{q_{out}} \right) (1-e_{out}^2)^{3/2} g(e,\phi) \quad \text{with} \quad q_{out} = \frac{m_{out}}{m_1+m_2}$$

For our problem, we have $P_{out} = 7.2yrs$ and $P_{in} = \frac{9}{12}yr$ and $g(e,\phi)$ is assumed to be one as the central binary as an eccentricity of $e_{in} = 0$ in the unperturbed state. This gives us a final value of $T_{KL} \approx 1318yrs$: this is the value of at which the variation in e_{in} should reach zero for the first time. Therefore, despite our noticed instabilities our value of T_{KL} observed is closer to the theoretical one than the example is even though the three values are within the same order of magnitude. We then ask ourselves if maybe the parameters given in the examples aren't necessarily the ones actually used to obtain the figures. But we do keep in mind that the $g \approx 1$ assumption could not be true and that the formula might give only a idea of the period and not an exact value.

3.2 Playing with the parameters

Now that we discussed the reproduction of the example, we decided to go a step further and actually play with the parameters of the problem to check the stability of the Kozai-Lidov effect under different intial conditions

3.2.1 Longer time

The first thing we wanted to check was how that slow change in a value observed in reproducing the example was going to develop with time, so we pushed the simulation time at the maximum achievable at that time for our computer before running out of memory : 10 000 yrs.

As we can see, the instabilities noticed on a short time scales are actually causing the system to go out of equilibrium and this more and more with time. The minima are further and further away with each oscillations (which are losing in amplitude also) as the two center objects gets closer and closer

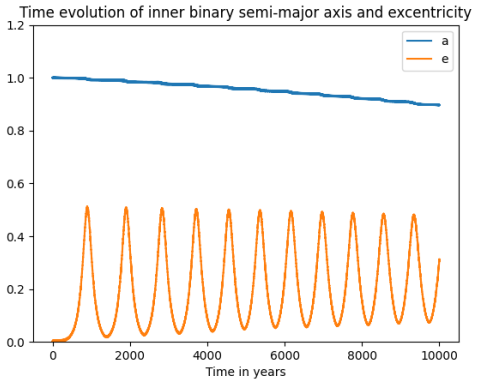


Figure 24: Time evolution of a_{in} and e_{in} with a simulation time of 10 000 yrs

3.2.2 Eccentricity effect

Now let's have a look over something we already studied in the binary system only : the effect of eccentricity on the stability of the system. This time we look at the one of the outer body without touching the inner system. We made a simulation for 2 new cases : $e_{out=0.5}$ and $e_{out=0.8}$

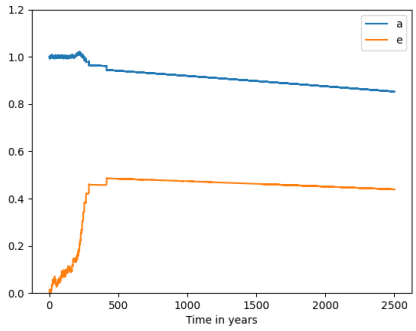


Figure 25: Time evolution of the parameters for $e_{out=0.5}$

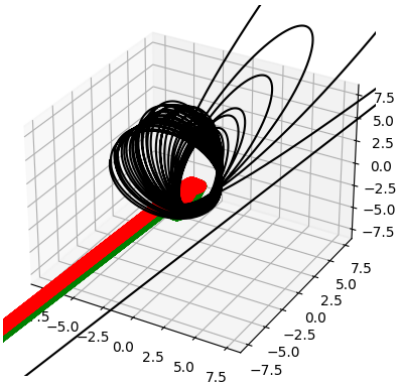


Figure 26: 3D plot for $e_{out=0.5}$

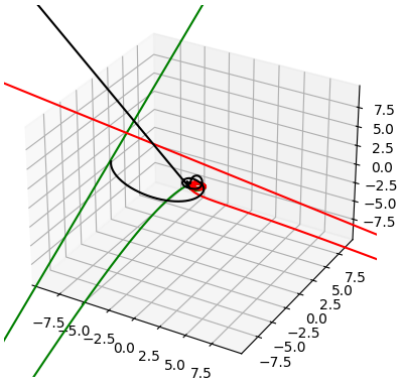


Figure 27: 3D plot for $e_{out=0.8}$

For the $e_{out=0.5}$, we can see that at first the system is unstable as the third body highly perturbs the inner system before being ejected. Once the third body isn't there anymore, the binary comes back to a quasi-equilibrium situation where the eccentricity and the distance between them slowly decrease with time. In the $e_{out=0.8}$ case, the third body comes so close to the binary that the whole system breaks in less than $1000yrs$ with all objects flying in different directions. The system is therefore highly sensitive to the initial e_{out} value.

3.2.3 Inclination effect

As we have seen in section 2.6.2, when the third body is co-planar with the binary, there's no change in the inclination and so no Kozai-Lidov resonance can happen. Therefore, there should be a critical value above which the phenomenon appears. This value is given by the formula :

$$\phi_{crit} = \text{acos}\left(\sqrt{\frac{3}{5}}\right) \approx 39.23°$$

We thus searched if that value was in fact a critical value for our system. We first made a simulation with an inclination of $\phi = 40°$ (Fig.28) which is just above the critical value. But on that simulation we can see no resonance appear within the first $2500yrs$, and this doesn't comes from the geometry of the system as the T_{KL} doesn't change that much with ϕ as $g(e,\phi) \approx 1$ because of $e = 0$. So instead of lowering the inclination to precise the critical value, we rose it up to $\phi = 60°$ where the first signs of change in e appeared (Fig.29). This value is 1.5 times greater than the critical value in theory, that is a significant gap. Maybe this could come from our instabilities pattern saw in Fig.22 and Fig.23 : as the motion of the third body isn't perfectly represented, the inclination has to be significantly higher than the critical value to produce the Kozai-Lidov resonances. A more precise expression of $g(e,\phi)$ could be use to see if the ≈ 1 approximation is really valid for any angle.

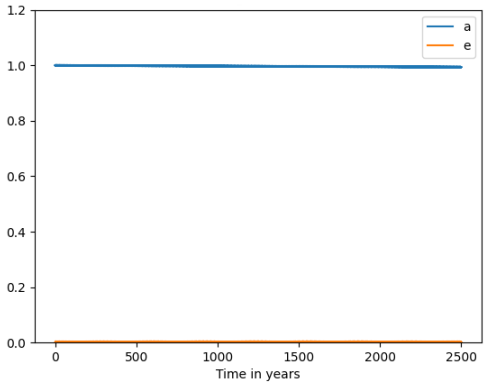


Figure 28: Parameters from the example but with $\phi = 40°$

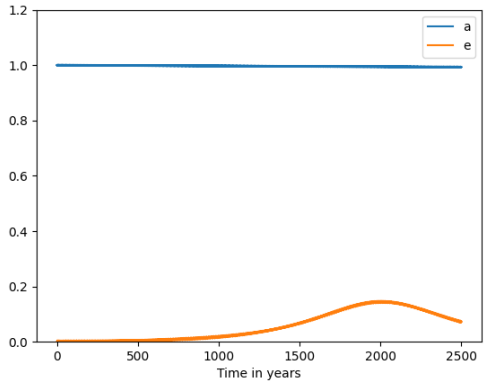


Figure 29: Parameters from the example but with $\phi = 60°$

4 A word about parallelization

The problem of the Kozai-Lidov effect is a specific kind of N-body problem. The computation of the motion of every single body resides in the resolution of the differential equation equating motion to gravitational potential. Because the gravitational potential seen by one body depends on the position of the $N-1$ others, the differential equations changes at each step of integration, and we thus used sequential numerical methods to integrate step by step.

The program we developed can be divided into four main parts :

1. The creation of objects and initialization of positions and velocities,
2. The computation of the motion (at each time step),
3. The calculation of several quantities from the lists of positions and velocities (at each time step),
4. The visualisation of the results.

In this section, we will discuss to what extent we can parallelize our code, taking a look at each of these four blocks. We define the number of bodies as N and the number of time steps for the integration as T. The problem is third dimensional (3 dimensions of space). Note that in practical, $N = 2,3$ but the program is developed such as it can compute a general N-body problem.

As a first remark, note that these four parts must run sequentially, as they require the results of the previous ones.

4.1 Initialization

To define the problem, we create 3 objects, instances of the class `body`. This creation does not require any calculation. We then create a first binary system (with two bodies) defined by the eccentricity and axis of the orbit. There is then a phase of computation to determine the initial conditions of each objects to satisfy the orbit. In practice, there are only a few calculations :

1. compute the position of the center of mass,
2. compute the reduced mass,
3. compute the initial velocity of the reduced mass to satisfy the orbit,
4. compute, from the velocity of the reduced mass, the initial velocities of the two bodies.

These steps do not need to be parallelized, as they represent a very negligible time of execution. If we still wanted to parallelize it, we would distribute the fourth step between several processes that would treat one body each.

After a binary system is created, we have to convert it into an equivalent body and create a second binary system consisting of the equivalent body and a third object. The discussion is the same, noting that this step is necessarily done after the previous one : we could not distribute the creation of the two systems to two processes that would run simultaneously (the one treating the second would wait the first).

In theory, we could imagine that our code treats N bodies (which is possible) and therefore need to compute $N/2$ binary systems. Here, a parallelization as discussed would be interesting : distribute on several machines (or cores) each system and for each system, distribute again the computation of initial velocities for each body on two threads. We would then wait and proceed for the second round of binary systems.

4.2 Computation of motion

The one part that requires lots of resources (both in time and memory) is the resolution of the equation of motion. The computation detailed in a previous section can be summed up as follows :

- For each time step :
  - For each body : Compute intermediate quantities
  - For each body : Compute the gravitational potential from the $N-1$ other bodies
  - For each body : Compute the updated velocity and position

If we consider the calculation of the potential as one operation of same order of the computation of the updated velocities and positions, the number of operations of this algorithm is $T \times (N \times (N-1) + N)$. The complexity is then $C = \Theta(TN^2)$. As experimented, it quickly requires large running times, even in our case when we consider only 3 bodies. To distribute the tasks, we can at first glance naturally imagine several possibilities :

- Distribute on the T time steps,
- Distribute on the N bodies,
- Distribute on the 3 sub-steps (gravitational potential computation and update of quantities).

- Distribute on the N bodies within a sub-step,
- Distribute on the 3 dimensions of space.

Naturally, we could also imagine sub-distributing tasks (like distributing on the bodies between N cores and sub-dividing into 3 dimensions). Unfortunately, many of these possibilities are not conceivable. In fact, we can not parallelize on time steps since the computation of the equation of motion at time t requires the result of the previous calculation, it has to be done sequentially.

Moreover, we can not distribute globally and independently between the N bodies since we need to have access to the positions of the other ones at each time step, requiring thus to communicate at each time step between the processes and waiting the others to be at the same time step. For that reasons, parallelizing would make the code even slower.

The 3 sub-steps have to be done sequentially, since the first uses the previous results, and the third uses the value of the potential. Distributing the tasks between the three dimensions of space looks interesting, but not reasonable either. We actually need the distances to each objects at each time step and thus retrieve the three coordinates. Again, this would mean to communicate with the two other processes and wait for them to be at the same step.

Finally, one thing that could be done is the remaining suggestion. We could define P processes each treating approximately N/P bodies each as follows : at the beginning of a time step, we distribute the computation of the intermediate quantities. Then we communicate to every one of them the results. We redistribute to compute the potential and again communicate the results. We repeat for the last sub-step where we update the positions and velocities. In the case of a large value of N, this would be interesting, as each sub-steps would be N times faster (neglecting the communications at the end of each sub-step and the waiting time). The issue with this approach is that in our case, the number of bodies is not critical, and it looks like the time spent for communications would compensate the little gain on these very small individual operations. It could nevertheless be worth to give it a try.

4.3 Additional calculations

Calculations of other physical quantities, like angular momentum over time, energy over time or eccentricity of the orbit over time all consists in scanning the results arrays and performing an operation to derive the wanted quantity for each time step. Therefore, to compute the list of values of a quantity, we perform T operations, where each operation needs to use the three components of position (or velocity, or both) of each body (meaning $3N$ or $6N$ values). In this case, we can only distribute the tasks on the T time steps and this is in fact very relevant, since the data already has been computed. These additional calculations do take a non negligible time in the execution of our program (but still way faster than the actual computation of the motion), and it would be interesting to parallelize this part like follows :

- Distribute the different quantities to be computed between P processes (like machines or cores),
- Distribute, for each quantity (meaning each machine or each core), the T calculations between K threads (each thread would compute approximately T/K operations).

With this approach, we distribute the computation between PK threads. We can then expect a total computation up to PK faster (but in practice, the acceleration and efficiency never scale linearly with the number of processes).

4.4 Exploitation

The part of exploitation of the obtained data consists of plotting results or saving data on the disk. If the data to save were to be considerably heavy, we could imagine to distribute the writing on the disk but this is absolutely not our case.

4.5 Conclusion

We discussed the efficiency of parallelizing each section of our code. It appeared that the nature of the problem, and especially the equation of motion being obviously time dependent and indivisible between different bodies, is not easily compatible with a parallel computation approach. However, in the most sensible part of the calculation, it is still theoretically possible to distribute the tasks of a precise region to gain in execution time. It is then to be determined if in our case of a 3 body problem it would still be relevant.

Moreover, we found out that all the computations done after the main calculation were very easy to parellelize, which would be a massive gain of execution time relatively to this part. To illustrate what we discussed, we could distribute the different calculations between several processes on different machines with MPI and subdivide by distributing the operations between the cores of a same machine with for instance OpenMP.

Finally, some secondary parts of the code done sequentially could in theory get done in parallel, but that would not be relevant in our case.

5 Conclusion and opening

This numerical simulation project allowed us to implement and experiment the Kozai-Lidov resonances in a three body problem. Starting from a simple binary system, we were able, via the addition of a third body and the more complex situations associated, to test what were the conditions for that effect to appear and what were the consequences. We had to deal with some difficulties such as the center of mass being perturbed by the presence of a third body thus creating a constant drifting from it's rest position. By achieving the needed corrections, it allowed us to get a better understanding of the forces in place and the implications of multi-body interactions.

We also tried to reproduce a time series of which parameters were given to see if our code was realist. We spotted some differences such as a non perfect oscillation of the binary eccentricity as the two bodies were slowly coming together. We also noted a difference on the period of oscillation that seemed bigger than what it supposed to be in the given example where as our version was closer to the theoretical value even though the order of magnitude was the same. We were able to question the accuracy of the given parameters.

We finished by addressing a word on the possibility of a parallelization of our code dealing with each sections separately. It appeared that the nature of the problem, is not easily compatible with a parallel computation approach. Still some "post-motion computation" quantities could be computed via a parallelization process or also some secondary parts of the code but it's less relevant in terms of actual time gain.

If we had more time, we could have explored some more complex cases to reflect even more realist systems found in our cosmic neighborhood. We could have for example replace the third body by another binary system, but also we could have implemented some dissipative forces that would cause the system to lose angular momentum on shorter time scales than those we found with our integration uncertainties. That integration problem could be solved if we were to search for even smaller values of $\delta E/E$ or even implementing even more accurate integrators. This type of treatment could also be used to look at predicting the evolution of the solar system as proposed by the Nice Model (formation and evolution of the solar system) requiring a N-body simulation implying complex interactions between each of them and changes in the orbital parameter of the different objects.

You will find our code in the appendix as well as using the following link :

https://github.com/Cedric-Accard/Kozai-Lidov-Resonances.git

Appendix A : Kozai-Lidov Resonances Code

Note that the cells that appeared as commented at the very end are the ones we used to do the precision test and comparison or to create the Earth-Sun system. As they're not relevant for the Kozai-Lidov effect (3 bodies) we commented them out so we still have access to them if needed

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib widget
import time

#Units and constants
G      = 6.67E-11          #Gravitational constant in N.m²/kg²
au     = 149597870700      #Mean distance Earth-Sun in m
msol   = 2e30              #Mass of the sun in kg
earth  = 3e-6              #Mass of the earth as a fraction of Msol
hour   = 3600              #Duration of an hour in seconds
day    = hour*24           #Duration of a day in seconds
year   = day*365.25636     #Duration of an year in seconds
```

[2]:
```
#Some basic but useful functions definitions

def norm_vec(vec):
    return np.sqrt(vec[0]**2 + vec[1]**2 + vec[2]**2)

def COM_pos(body_list):
    vec = np.array([0., 0., 0.], dtype = np.float64)
    mtot = 0
    for body in body_list:
        vec += body.mass * body.position
        mtot += body.mass
    return vec/mtot

def COM_vel(body_list):
    vec = np.array([0., 0., 0.], dtype = np.float64)
    M = 0
    for body in body_list:
        vec += body.mass * body.velocity
        M += body.mass
    return vec/M

def total_mass(body_list):
    M = 0
    for c in body_list:
        M += c.mass
    return M

def velocity_reduced_mass(body_list, e):
    r = norm_vec(body_list[0].position-body_list[1].position)
    return np.sqrt((1-e)*(G*total_mass(body_list))/r)

def compute_positions(body_list, excentricity, axis):
    aph = axis*(1+excentricity)
    pos1 = aph*body_list[1].mass/(body_list[0].mass+body_list[1].mass)
    pos2 = aph*body_list[0].mass/(body_list[0].mass+body_list[1].mass)
    pos_c1 = np.array([pos1, 0., 0.], dtype = np.float64)
    pos_c2 = np.array([-pos2, 0., 0.], dtype = np.float64)
    return pos_c1, pos_c2
```

```
def compute_velocities(body_list, e):
    v_mu = velocity_reduced_mass(body_list, e)
    v1 = v_mu*body_list[1].mass/(body_list[0].mass+body_list[1].mass)
    v2 = v_mu*body_list[0].mass/(body_list[0].mass+body_list[1].mass)
    v_c1 = np.array([0., v1, 0.], dtype = np.float64)
    v_c2 = np.array([0., -v2, 0.], dtype = np.float64)
    return v_c1, v_c2

def initialize_body(body_list, e, axis):
    p_c1, p_c2 = compute_positions(body_list, e, axis)
    body_list[0].position = p_c1
    body_list[1].position = p_c2
    body_list[0].initial_position = p_c1
    body_list[1].initial_position = p_c2

    v_c1, v_c2 = compute_velocities(body_list, e)
    body_list[0].velocity = v_c1
    body_list[1].velocity = v_c2
    body_list[0].initial_velocity = v_c1
    body_list[1].initial_velocity = v_c2

    for body in body_list:
        body.setup_coord_in_COM(body_list)
        # body.display_infos()

def reset_body(body_list):
    for body in body_list:
        body.reset()

def update_velpos_bs1_to_bs2_com(bs1,bs2,phi) :
    bs1.body_list[0].position = bs1.body_list[0].position + bs2.
 ↪body_list[0].position
    bs1.body_list[1].position = bs1.body_list[1].position + bs2.
 ↪body_list[0].position

    triple = [bs1.body_list[0], bs1.body_list[1], bs2.body_list[1]]

    dpy=(1-np.cos(phi))*bs2.body_list[1].mass*norm_vec(bs2.body_list[1].
 ↪velocity)/total_mass(triple)
    dpz=-(np.sin(phi))*bs2.body_list[1].mass*norm_vec(bs2.body_list[1].
 ↪velocity)/total_mass(triple)

    bs1.body_list[0].velocity = bs1.body_list[0].velocity + bs2.
 ↪body_list[0].velocity - [0,dpy,dpz]
    bs1.body_list[1].velocity = bs1.body_list[1].velocity + bs2.
 ↪body_list[0].velocity - [0,dpy,dpz]
    bs2.body_list[1].velocity = bs2.body_list[1].velocity - [0,dpy,dpz]

    triple = [bs1.body_list[0], bs1.body_list[1], bs2.body_list[1]]

    return triple
```

[3]:
```
#Creating a class that contains all the properties of a body (mass, name,
 ↪position in different coordinates systems, velocity,...)

class body():
    def __init__(self, name, mass):
        self.name = name
        self.mass = mass
        self.initial_position = np.array([0., 0., 0.], dtype = np.float64)
        self.initial_velocity = np.array([0., 0., 0.], dtype = np.float64)
        self.position = np.array([0., 0., 0.], dtype = np.float64)
        self.velocity = np.array([0., 0., 0.], dtype = np.float64)
        self.a = np.array([0.,0.,0.], dtype = np.float64)    #Acceleration
        self.a_pt = np.array([0.,0.,0.], dtype = np.float64) #Jerk
        self.a_prime = np.array([0.,0.,0.], dtype = np.float64)
        self.a_pt_prime = np.array([0.,0.,0.], dtype = np.float64)
        self.a_2 = np.array([0.,0.,0.], dtype = np.float64)
        self.a_3 = np.array([0.,0.,0.], dtype = np.float64)

    def vec_position_in_COM(self, body_list):
        return self.position - COM_pos(body_list)

    def vec_velocity_in_COM(self, body_list):
        return self.velocity - COM_vel(body_list)

    def setup_coord_in_COM(self, body_list):
        self.position = self.vec_position_in_COM(body_list)
        self.velocity = self.vec_velocity_in_COM(body_list)

    def rotate_velocity(self, phi):
        self.velocity[2] = self.velocity[1]*np.sin(phi)
        self.velocity[1] = self.velocity[1]*np.cos(phi)

    def reset(self):
        self.position = self.initial_position

    def display_infos(self):
        print(f'Objet : {self.name}, x : {self.position[0]}, y : {self.
 ↪position[1]}, vx : {self.velocity[0]}, vy : {self.velocity[1]}, vz :
 ↪{self.velocity[2]}')
    def grad_potential(self, body_list, indice):
        res = np.array([0., 0., 0.])
        rCOM = self.position
        for (i, body) in enumerate(body_list):
            if i != indice:
                rCOMi = body.position
                res+=(rCOM-rCOMi)*body.mass/(pow(norm_vec(rCOM-rCOMi), 3))
        return G*res

    def a_dot(self, body_list, indice):
        res = np.array([0., 0., 0.])
        rCOM = self.position
        vCOM = self.velocity
        for (i, body) in enumerate(body_list):
            if i != indice:
                rCOMi = body.position
                vCOMi = body.velocity
                res += (body.mass/(pow(norm_vec(rCOM - rCOMi), 3))) *
 ↪(((vCOM-vCOMi) - 3*np.dot((vCOM - vCOMi),(rCOM - rCOMi))*(rCOM -
 ↪rCOMi))/(pow(norm_vec(rCOM - rCOMi), 2)))
        return -G*res
```

[4]:
```
#Class to handle two objects as a binary system (Initialization and Orbit properties)

class binary_system():
    def __init__(self, c1, c2, rotation_angle = 0):
        self.body_list = [c1, c2]
        self.COM_postion = COM_pos([c1, c2])
        self.mass = total_mass([c1, c2])
        self.excentricity = 0
        self.perihelion = None
        self.aphelion = None
        self.axis = None
        self.phi = rotation_angle

    def initialize_system(self, e, axis,info=None):
        initialize_body(self.body_list, e, axis)
        self.excentricity = e
        if(e < 1):
            self.axis = axis
            self.perihelion = self.axis*(1-self.excentricity)
            self.aphelion = self.axis*(1+self.excentricity)
        self.body_list[1].rotate_velocity(self.phi)
        if info !=None :
            self.display_infos()

    def reset_system(self):
        print(f'\nFinal total momentum : {self.total_momentum()}')
        print(f'Final total angular momentum : {self.total_angular_momentum()}')
        reset_body(self.body_list)

    def display_infos(self):
        print(f'Axis (in A.U.) : {self.axis/au}, Excentricity : {self.excentricity}, Initial Total momentum : {self.total_momentum()}, Initial Tot Ang Mom : {self.total_angular_momentum()}')

    def total_momentum(self):
        return self.body_list[0].velocity*self.body_list[0].mass + self.body_list[1].velocity*self.body_list[1].mass

    def total_angular_momentum(self):
        L = np.array([0.,0.,0.], dtype = np.float64)
        for body in self.body_list:
            L += np.cross(body.position,body.velocity*body.mass)
        return L

    def compute_motion(self, integrator, step, duration):
        I = integrator(self.body_list, step, duration)
        return I

    def system_to_body(self):
        body_eq = body("system", self.mass)
        return body_eq
```

[5]:
```
#Verlet/Leapfrog Integrator :

def leapfrog_integrator(body_list, step, duration):

    t = 0
    time_list = [0]
    positions_t0 = []
    velocities_t0 = []

    for body in body_list:
        positions_t0.append(body.position)
        velocities_t0.append(body.velocity)

    positions_list = [positions_t0]
    velocities_list = [velocities_t0]

    while t < duration:
        t += step
        if ((t/(3600*24*365*(0.1*duration/year))).is_integer()==True):
            print(int(t/(365*day)),'/', duration/year, ' years' )
        time_list.append(t)
        positions_t = []
        velocities_t = []
        p = [0., 0., 0.]

        #Calcul de x' pour chaque body
        for body in body_list:
            body.position = body.position + body.velocity * (step/2)
        #Calcul de v'' et x'' pour chaque body

        for (i, body) in enumerate(body_list):
            body.velocity = body.velocity - step*body.
↪grad_potential(body_list, i)
            body.position = body.position + body.velocity * (step/2)
            positions_t.append(body.position)
            velocities_t.append(body.velocity)
            p += body.velocity*body.mass

        positions_list.append(positions_t)
        velocities_list.append(velocities_t)

    return positions_list, velocities_list, time_list
```

[6]:
```
# Hermite Scheme :

def hermite_integrator(body_list, step, duration):

    t = 0
    time_list = [0]
    positions_t0 = []
    velocities_t0 = []

    for body in body_list:
        positions_t0.append(body.position)
        velocities_t0.append(body.velocity)

    positions_list = [positions_t0]
    velocities_list = [velocities_t0]

    while t < duration:
        t += step
        if ((t/(3600*24*365*(0.1*duration/year))).is_integer()==True):
            print(int(t/(365*day)),'/', duration/year, ' years' )
        time_list.append(t)
        positions_t = []
        velocities_t = []
        p = [0., 0., 0.]

        #Compute the acceleration and jerk of each body
        for (i, body) in enumerate(body_list):
            body.a = -body.grad_potential(body_list, i)
            body.a_pt = body.a_dot(body_list, i)
        #Compute xp & vp for each body
        for (i, body) in enumerate(body_list):
            body.position = body.position + body.velocity*(step) + 0.
↪5*(step**2)*body.a + (1/6)*(step**3)*body.a_pt
            body.velocity = body.velocity + step*body.a + 0.
↪5*(step**2)*body.a_pt
        #Compute x' & v' for each body
        for (i, body) in enumerate(body_list):
            body.a_prime = -body.grad_potential(body_list,i)
            body.a_pt_prime = body.a_dot(body_list,i)
            body.a_2 = -(6*(body.a-body.a_prime) + step*(4*body.a_pt +
↪2*body.a_pt_prime))/step**2
            body.a_3 = (12*(body.a-body.a_prime) + 6*step*(body.a_pt +
↪body.a_pt_prime))/step**3
        #Final, corrected coordinates are now :
        for (i, body) in enumerate(body_list):
            body.position = body.position + ((step**4)/24)*body.a_2 +
↪((step**5)/120)*body.a_3
            body.velocity = body.velocity + (step**3/6)*body.a_2 +
↪(step**4/24)*body.a_3
        #Add the position/velocity and momentum at time t+dt to their
↪respective list
            positions_t.append(body.position)
            velocities_t.append(body.velocity)
            p += body.velocity*body.mass

        positions_list.append(positions_t)
        velocities_list.append(velocities_t)

    return positions_list, velocities_list, time_list
```

[7]:
```
# Total Energy, Mechanical Energy and Angular Momentum

def energy_from_trajectory(b_s, P, V, T, plot=None):
    nb_time = len(T)
    #Total Energy
    H = np.zeros(nb_time)
    for (i,bodyi) in enumerate(b_s.body_list) :
        H += 0.5 * bodyi.mass*(np.linalg.norm(V[:,i,:],axis=1))**2
        for j in range(i+1, len(b_s.body_list)):
            bodyj = b_s.body_list[j]
            H += -G*(bodyi.mass*bodyj.mass)/np.linalg.norm((P[:,i,:]-P[:
 ↪,j,:]),axis=1)
    #Angular Momentum
    L = np.zeros((nb_time, 3))
    for (i,body) in enumerate(b_s.body_list) :
        L += body.mass * np.cross(P[:,i,:],V[:,i,:])
    #Mecanical Energy
    E = -G/(2*b_s.axis) * np.ones(nb_time)
    for (i,bodyj) in enumerate(b_s.body_list) :
        E *= bodyj.mass
    Ecart=np.absolute((H-E)/E)
    max_ecart = np.max(Ecart)
    distance_list_c1_c2 = np.array([norm_vec(P[k,0]-P[k,1]) for k in
 ↪range(len(T))])
    #Graphs
    if plot!=None :
        plt.figure()
        plt.plot(T, 100 - 100*L[:,2]/L[0,2])
        plt.title('Angular Momentum deviation from Initial value (in %)')
        plt.xlabel('Time in years')
        plt.figure()
        plt.plot(T,Ecart)
        plt.title('Relative error on energy')
        plt.xlabel('Time in years')
        plt.figure()
        plt.plot(T, distance_list_c1_c2/au) #(t, distance de corps 0 à 1)
        plt.title("Distance corps0-corps1 (in A.U.)")
        plt.xlabel('Time in years')

    return max_ecart
```

[8]:
```
def find_integration_step_method(fonction, C, precision, step0, duration):
    erreur = 1
    step = step0
    while erreur > precision:
        B_S = binary_system(C[0], C[1])
        B_S.initialize_system(e_bin, a_bin)
        start = time.time()
        P, V, T = fonction(B_S.body_list, step, duration)
        end = time.time()
        print('Time : ', (end-start), ' seconds to execute')
        P = np.array(P)
        V = np.array(V)
        T = np.array(T)/year
        erreur = energy_from_trajectory(B_S, P, V, T)
        print('\u03B4E/E =', "{:.2E}".format(erreur) , 'for \u0394t =',
 ↪"{:.3f}".format(step/day), 'days')
        step = step/2

    return 2*step
```

[9]:
```
binary1 = body("B1", msol)
binary2 = body("B2", msol)
binary3 = body("B3", 0.1*msol)

#Parameters
e_bin=0.                    #Excentricity of the binary
e_out=0.25                    #Excentricity of the outer body
a_bin = 1*au                #Semi-major axis of the binary
a_out = 5*au                #Semi-major axis of the outer body
phi=80*np.pi/180             #Initial inclination of the third body plane
simu_time=2500*year          #Total time of the simulation
step = day/2

systeme_binaire = binary_system(binary1, binary2)
systeme_binaire.initialize_system(e_bin, a_bin)

body_eq = systeme_binaire.system_to_body()

systeme_binaire2 = binary_system(body_eq, binary3, phi)
systeme_binaire2.initialize_system(e_out, a_out)

triple_system =
 ↪update_velpos_bs1_to_bs2_com(systeme_binaire,systeme_binaire2,phi)

positions_list3, velocities_list3, time_list3 =
 ↪hermite_integrator(triple_system, step, simu_time)

positions_list3 = np.array(positions_list3, dtype=np.float64)
velocities_list3 = np.array(velocities_list3, dtype=np.float64)
time_list3 = np.array(time_list3, dtype=np.float64)

systeme_binaire.reset_system()
systeme_binaire2.reset_system()

e = np.zeros(len(positions_list3))
a = np.zeros(len(positions_list3))
r = np.zeros(len(positions_list3))

for i in range(len(positions_list3)) :
    r = norm_vec(positions_list3[i,0,:]-positions_list3[i,1,:])
    dv = (velocities_list3[i,0,:]-velocities_list3[i,1,:])
    dr = (positions_list3[i,0,:]-positions_list3[i,1,:])
    h = np.cross(dr,dv)
    mu = (G*systeme_binaire.mass)
    e[i] = norm_vec((np.cross(dv,h)/mu) - (dr/norm_vec(dr)))
    a[i] = norm_vec(h)**2/(mu*(1-e[i]**2))
```

[10]:
```
### Plotting The results ####

fig = plt.figure(figsize=plt.figaspect(1))
ax = fig.add_subplot(1, 1, 1, projection='3d')
ax.plot(positions_list3[:,0,0]/au, positions_list3[:,0,1]/au,
 ↪positions_list3[:,0,2]/au, color='green')
ax.plot(positions_list3[:,1,0]/au, positions_list3[:,1,1]/au,
 ↪positions_list3[:,1,2]/au, color='red')
ax.plot(positions_list3[:,2,0]/au, positions_list3[:,2,1]/au,
 ↪positions_list3[:,2,2]/au, color='black')
ax.set_xlim(-1.1*(1+e_out)*a_out/au, 1.1*(1+e_out)*a_out/au)
ax.set_ylim(-1.1*(1+e_out)*a_out/au, 1.1*(1+e_out)*a_out/au)
ax.set_zlim(-1.1*(1+e_out)*a_out/au, 1.1*(1+e_out)*a_out/au)
plt.show()


fig = plt.figure(figsize=plt.figaspect(0.5))
ax = fig.add_subplot(1, 3, 1, adjustable='box', aspect=1)
ax.plot(positions_list3[:,0,0]/au, positions_list3[:,0,1]/au,
 ↪color='green')
ax.plot(positions_list3[:,1,0]/au, positions_list3[:,1,1]/au, color='red')
ax.plot(positions_list3[:,2,0]/au, positions_list3[:,2,1]/au,
 ↪color='black')
ax.set_xlim(-1.1*(1+e_out)*a_out/au, 1.1*(1+e_out)*a_out/au)
ax.set_ylim(-1.1*(1+e_out)*a_out/au, 1.1*(1+e_out)*a_out/au)
ax.set_title('Top View (x and y)')
ax = fig.add_subplot(1, 3, 2, adjustable='box', aspect=1)
ax.plot(positions_list3[:,0,0]/au, positions_list3[:,0,2]/au,
 ↪color='green')
ax.plot(positions_list3[:,1,0]/au, positions_list3[:,1,2]/au, color='red')
ax.plot(positions_list3[:,2,0]/au, positions_list3[:,2,2]/au,
 ↪color='black')
ax.set_xlim(-1.1*(1+e_out)*a_out/au, 1.1*(1+e_out)*a_out/au)
ax.set_ylim(-1.1*(1+e_out)*a_out/au, 1.1*(1+e_out)*a_out/au)
ax.set_title('Side view (x and z)')
ax = fig.add_subplot(1, 3, 3, adjustable='box', aspect=1)
ax.plot(positions_list3[:,0,1]/au, positions_list3[:,0,2]/au,
 ↪color='green')
ax.plot(positions_list3[:,1,1]/au, positions_list3[:,1,2]/au, color='red')
ax.plot(positions_list3[:,2,1]/au, positions_list3[:,2,2]/au,
 ↪color='black')
ax.set_xlim(-1.1*(1+e_out)*a_out/au, 1.1*(1+e_out)*a_out/au)
ax.set_ylim(-1.1*(1+e_out)*a_out/au, 1.1*(1+e_out)*a_out/au)
ax.set_title('Side view (y and z)')
plt.show()


plt.figure()
plt.plot(time_list3/year, a/au, label='a')
plt.plot(time_list3/year, e, label='e')
plt.ylim(0.,1.2)
plt.xlabel('Time in years')
plt.title('Time evolution of inner binary semi-major axis and
 ↪excentricity')
plt.legend()
plt.show()
```

```
[ ]: ##Create the binary system giving them an initial position in the R0
      ↪frame :
     # binary1 = body("B1", msol)
     # binary2 = body("B2", msol)

      body_list = [binary1, binary2]

     ##Parameters
     # e_bin=0. #Excentricity of the binary
     # a_bin = 1*au
     # phi_bin=0*np.pi/180          #Initial inclination of the third body
      ↪plane
     # simu_time=(9/12)*year        #Total time of the simulation


     # systeme_binaire = binary_system(binary1, binary2, rotation_angle =
      ↪phi_bin)
     # systeme_binaire.initialize_system(e_bin, a_bin)

      # to_test = 'opt_step'

     # if to_test=='energy_from_traj' :
     #     P, V, T = hermite_integrator(systeme_binaire.body_list, 0.
      ↪713*day,(9/12)*year)
     #     P = np.array(P)
     #     V = np.array(V)
     #     T = np.array(T)/year
     #     erreur = energy_from_trajectory(systeme_binaire, P, V, T,1)

     # if to_test=='opt_step':
     #     step_opt_herm = find_integration_step_method(hermite_integrator,
      ↪body_list, 10**(-5), year, simu_time)
     #     print('')
     #     step_opt_leap = find_integration_step_method(leapfrog_integrator,
      ↪body_list, 10**(-5), year, simu_time)
     #     systeme_binaire.reset_system()
```

```
[ ]: # time_H=[365.256,182.628,91.314,45.657,28.29,14.14,5.707,2.854,1.427,0.
      ↪713]
     # error_H=[21.7,6.53,1.67,5.54,0.463,0.0243,0.00278,0.000338,0.0000425,0.
      ↪00000529]

     # time_V=[365.256,182.628,91.314,45.657,28.29,14.14,5.707,2.854,1.427,0.
      ↪713,0.357,0.178,0.089,0.045,0.022,0.011146739501953124,0.006,0.003,0.
      ↪0015,0.0007,0.00035,0.00017]
     # error_V=[1.18,0.53,3.61,5.87,31.8,125,0.466,0.152,0.0662,0.0309,0.015,0.
      ↪00738,0.00666,0.00182,0.00091,0.000455,0.000227,0.000114,0.0000568,0.
      ↪0000284,0.0000142,0.00000710]

     # proc_H=[0.0,0.0,0.0,0.0,0.0009999275207519531,0.003513336181640625,0.
      ↪005513191223144531,0.01009178161621093 8,0.020220279693603516,0.
      ↪03678083419799805]
     # proc_V=[0.0,0.0,0.0,0.0,0.001005411148071289,0.0,0.
      ↪0009989738464355469,0.002000093460083008,0.004001140594482422,0.
      ↪006031990051269531]

     # plt.figure()
     # plt.plot(time_H,error_H, label='Hermite')
     # plt.plot(time_V,error_V,label='Verlet')
```

```
# plt.axhline(y=1e-5, color='r', linestyle='--')

# plt.yscale('log')
# plt.xscale('log')
# plt.xlabel('Time step in days')
# plt.ylabel('Relative Error')
# plt.legend()
# plt.gca().invert_xaxis()

# plt.figure()
# plt.plot(time_H,proc_H, label='Hermite')
# plt.plot(time_H,proc_V,label='Verlet')
# plt.xscale('log')
# plt.xlabel('Time step in days')
# plt.ylabel('Time to execute in seconds')
# plt.legend()
# plt.gca().invert_xaxis()
```

[ ]:
```
# #EARTH-SUN SYSTEM :
# binary1 = body("B1", msol)
# binary2 = body("B2", earth*msol)

# body_list = [binary1, binary2]

# #Parameters
# e_bin=0.0167 #Excentricity of the binary
# a_bin = 1*au
# phi_bin=0*np.pi/180          #Initial inclination of the third body
 ↪plane
# simu_time=5*(12/12)*year   #Total time of the simulation


# systeme_binaire = binary_system(binary1, binary2, rotation_angle =
 ↪phi_bin)
# systeme_binaire.initialize_system(e_bin, a_bin)

# positions_list, velocities_list, time_list = systeme_binaire.
 ↪compute_motion(hermite_integrator, day, simu_time)
# systeme_binaire.reset_system()

# P = np.array(positions_list, dtype = np.float64)
# V = np.array(velocities_list, dtype = np.float64)
# T = np.array(time_list, dtype = np.float64)/year

# fig = plt.figure(figsize=plt.figaspect(1))
# ax = fig.add_subplot(1, 1, 1, projection='3d')
# ax.plot(P[:,0,0]/au, P[:,0,1]/au, P[:,0,2]/au, color='green')
# ax.plot(P[:,1,0]/au, P[:,1,1]/au, P[:,1,2]/au, color='blue')
# #ax.set_title("Trajectory in space")
# plt.show()

# print(V[0,1,1])
```