Planet formation (aero)dynamics project

Chris Ormel, Carsten Dominik, Jacob Arcangeli

February 2, 2016

1 Goals

This project consist of two steps:

1. Program a small N-body code and test several algorithms (Section 2)
2. Choose a project from the list (Section 3), investigate the problem, and deliver a report and a presentation.

You can show the instructors your results on part #1, to ensure you are on the right track. Part #2 is an open project: you should justify the choices/assumptions you make here and defend them (usually there are multiple strategies possible). The written report should focus on the second part.

It is advised that you discuss your progress with an instructor on a weekly basis!

Grading

is based on:

- A 10min presentation +10min discussion during the final lecture block.
- A short report. Try to be to-the-point like you are writing a scientific article: intro/motivation, methods, result, and discussion/conclusion. 3-4 pages in two-column layout is more than sufficient. Summarize your results in a couple of figures, which you discuss in some depth.
- The project level. One-star ($\star$) and double-star ($\star\star$) projects require a little more effort. If conducted satisfactory, $\frac{1}{2}$ resp. 1 pt. bonus will be added to your mark.

2 An N-body tutorial

2.1 Python syntax

We will adopt `Python` to program a simple N-body code. But it must be stressed that `Python` is not ideal for N-body simulations, particular at large N when number crunching becomes important. Here, we mainly use it for illustrative purposes as `Python` syntax is very readable. We do not optimize our code for speed (ambitious students are encouraged to proceed with this or to adopt a different language like C of Fortran).

We further use the `numpy` module for efficient array operations. We use three main arrays:

- the positions of the particles (`xarr`). This is a two-dimensional array of size $3\times N$: 3 for the number of spatial dimensions (x, y, and z) and N for the number of particles. For example, `xarr[:,i]` gives the spatial position of the i^{th} particle.
- the velocities of the particles `varr`. This is also a $3 \times N$ array.

– the masses of the particles, `marr`. This is a 1-dimensional array of length N. The masses don't change.

We will use some of `python/numpy`'s vectorial features. For example:

```
sqrt((xarr**2).sum(axis=0))
```

will result in a 1-dimensional array of size N giving the distance to the particles from the origin, while

```
(xarr*mass).sum(axis=1) /sum(mass)
```

gives the center-of-mass coordinates. If you adopt a different programming language, such statements have to be modified.

2.2 Program layout

We also use a modular approach. This means that we have 3 files:

– a main program file, *eg.* `2body.py` for the 2-body problem
– a file `pars.py` which stores all the constants and parameters of the problem;
– a file `functions.py` which contains key functions, like the computation of the forces on the particles.

In the examples below, we will work in cgs-units. Professional N-body coders sometimes normalize units and put, e.g., Newton's gravitational constant to unity, $G = 1$, as well as the mass of the Sun. But we will not adopt dimensionless units here. In addition we will treat the Sun as a normal N-body particle.

Our `pars.py` file is very simple indeed:

```
#file pars.py
#constants
gN  = 6.67408e-08 #Newton's gravitational constant
mSun= 1.9884754153381438e+33 #mass of the Sun (in grams)
mEarth = ...
au  = 1.495978707e13 #astronomical unit (in cm)
yr  = 2*pi /sqrt(gN*mSun/au**3) #1 year in seconds
...
#problem parameters
Np = 2 #2 particles
```

This is included in the main program using an `import pars` statement. For example, it is used in the main program to assign the values to the variables `tfinal` and `dt`:

Box 1: Main program

```
#main program file
from numpy import *
import functions as fn
import pars

#assign the initial positions, velocities, masses
xarr, varr, marr = fn.init_2body()

#declarations
time = 0 #start time
tfinal = 1e8 #end time (seconds)
dt = 0.01*pars.yr #timestep
```

```
#compute the total energy, used for verification
etot0 = fn.e_tot (xarr, varr)

#start main loop
while time<tfinal:
    #calculate forces, update positions and velocities
    #this involves calling the function that computes
    #the accelerations
    ...
    acc = fn.forces(xarr,varr,marr)
    ...

    #increment time
    time += dt

etot = fn.e_tot (xarr, varr, marr)
error = (etot -etot0) /etot0
```

In this program, we call three functions in the `functions.py` file: `init_2body()`, `e_tot` and `forces`. The first initializes the position and velocities, the second calculates the energy, and the third computes the forces: accelerations and (for Hermite schemes) "jerks". Let us start with the function that initializes the 2-body program:

Box 2: Initialize 2-body

```
from numpy import *
import pars

def init_2body (ecc=0.5):
    """
    construct the 2body problem; initialize at aphelion
    """

    #declare the parameters as zeros
    xarr = zeros((3,pars.Np))#positions
    varr = zeros((3,pars.Np))#velocities
    marr = zeros(pars.Np) #masses

    #consider the Sun (particle 0) and Earth (particle 1)
    marr[0] = pars.mSun
    marr[1] = pars.mEarth

    #Keplerian velocity corresponding to 1 AU
    vKep = sqrt(pars.gN*(pars.mSun+pars.mEarth) /pars.au)

    #initialize at aphelion
    xarr[:,1] = [pars.au *(1+ecc), 0., 0.]
    varr[:,1] = [0., vKep *sqrt((1-ecc) /(1+ecc)), 0.]

    return xarr, varr, marr
```

Note that the Sun is initialized at $\mathbf{x} = \mathbf{v} = 0$ and the second particle has been given the mass of the Earth (but that is arbitrary). Also note that it has been initialized with a zero radial velocity and a azimuthal velocity less than its local Keplerian value. Convince yourself that the particle is then at apoastron.

The most important function is the calculation of the gravitational forces:

$$\mathbf{a}_i = \sum_{j \neq i} \frac{Gm_j}{|x_{ji}|^3} \mathbf{x}_{ji}; \qquad \mathbf{x}_{ji} = \mathbf{x}_j - \mathbf{x}_i \tag{1}$$

Note the signs. The summation contains $N-1$ terms and the calculation should be conducted for each of the N particles. However, this can be halved by virtue of Newton's 3rd law. Hence, we "only" need $\frac{1}{2}N(N-1)$ calculations. This may be the most direct way to code this up:

Box 3: force computation (accelerations)

```python
def forces (xarr, marr):
    """
    xarr(3,Np) :positions
    marr(Np)      :masses

    Calculates the gravitational force (accelerations)
    on each particle

    returns the accelerations
    """
    acc = zeros((3,pars.Np))
    for i in range(pars.Np):
        for j in range(i+1, pars.Np):
            rji = xarr[:,j] -xarr[:,i] #relative position (vectorial)
            r2 = (rji**2).sum(axis=0) #squared distance (scalar)
            r1 = sqrt(r2)  #distance
            r3 = r1*r2    #cubed distance

            force = pars.gN*rji/r3
            acc[:,i] += force*marr[j] #add to i
            acc[:,j] -= force*marr[i] #reverse sign

    return acc
```

In Python, such an implementation with two for-loops is far from optimal. We can, quite easily, get rid of the inner for loop. But the goal is here to get the code working and consider optimizations later.

The remaining function in our program above is `e_tot`, which calculates the total energy of the system. This is super useful as it will allow us to check the accuracy of the various algorithms that integrate the equation of motions.

Exercise 1:

(a) In Box 2 we initialized the star at $\mathbf{x} = \mathbf{v} = 0$. But it will not stay there. Why?

(b) Write the function `e_tot`, which returns the total energy of the system, *ie.* the sum of the kinetic and potential energies:

$$E = \frac{1}{2} \sum_i m_i v_i^2 - \sum_i \sum_{j>i} \frac{Gm_i m_j}{|\mathbf{x}_j - \mathbf{x}_i|} \tag{2}$$

Of course, you can – and should – extend the program, for example to print intermediate results.

2.3 The forward Euler, midpoint, Runge-Kutta and Leapfrog schemes

In the forward Euler scheme, the velocities and positions are updated simply as:

Box 4: Forward Euler

```
acc = fn.forces(xarr, marr)
xarr += varr*dt
varr += acc*dt
```

Exercise 2: Run the 2-body code and verify that the error is order unity. Plot the orbit and convince yourself that something is very wrong. Try the integration with a smaller value for the timestep parameter `dt`. How small should you set `dt` in order to get a ”decent” result?

Clearly, the forward Euler scheme will be very inefficient for 2-body – let alone N-body – calculations. The result already improves for the midpoint algorithm:

Box 5: Midpoint method

```
acc = fn.forces (xarr, marr)
xmid = xarr +varr*dt/2
vmid = varr +acc*dt/2
amid = fn.forces (xmid, marr)
xarr += vmid*dt
varr += amid*dt
```

In the midpoint method, we first take a half-step, re-calculate the forces at the midpoint, and then use the velocities and forces of the midpoint to advance over the full timestep. You can verify that this already performs much better than the forward Euler, albeit at the expense of an additional force calculation.

Continuing the approach, more intermediate evaluations can be conducted to produce more accurate results. Together, this schemes are known as Runge-Kutta methods. In particular, the fourth order method, RK4, is frequently-used and requires four evaluations of the force function. A sample algorithm (there are several variations) could read:

$$\mathbf{x}_{i+1} = \mathbf{x}_i + (\mathbf{v}_{k1} + 2\mathbf{v}_{k2} + 2\mathbf{v}_{k3} + \mathbf{v}_{k4})\Delta t/6 \tag{3}$$
$$\mathbf{v}_{i+1} = \mathbf{v}_i + (\mathbf{a}_{k1} + 2\mathbf{a}_{k2} + 2\mathbf{a}_{k3} + \mathbf{a}_{k4})\Delta t/6 \tag{4}$$

where the $\{\mathbf{v}_k\}$ and $\{\mathbf{a}_k\}$ are intermediate results:

$$\mathbf{v}_{k1} = \mathbf{v}_i \qquad \mathbf{a}_{k1} = \mathbf{a}_i \tag{5}$$
$$\mathbf{v}_{k2} = \mathbf{v}_i + \mathbf{a}_{k1}\Delta t/2 \qquad \mathbf{a}_{k2} = \mathbf{a}(\mathbf{x}_i + \mathbf{v}_{k1}\Delta t/2) \tag{6}$$
$$\mathbf{v}_{k3} = \mathbf{v}_i + \mathbf{a}_{k2}\Delta t/2 \qquad \mathbf{a}_{k3} = \mathbf{a}(\mathbf{x}_i + \mathbf{v}_{k2}\Delta t/2) \tag{7}$$
$$\mathbf{v}_{k4} = \mathbf{v}_i + \mathbf{a}_{k3}\Delta t \qquad \mathbf{a}_{k4} = \mathbf{a}(\mathbf{x}_i + \mathbf{v}_{k3}\Delta t) \tag{8}$$

2.4 The Leapfrog scheme

For gravitational N-body simulations (where energy is conserved) there is a simple and accurate alternative scheme: the Leapfrog. In its simplest form it considers half steps:

$$\mathbf{v}_{i+\frac{1}{2}} = \mathbf{v}_{i-\frac{1}{2}} + \mathbf{a}_i\Delta t \tag{9}$$
$$\mathbf{x}_{i+1} = \mathbf{x}_i + \mathbf{v}_{i+\frac{1}{2}}\Delta t \tag{10}$$

from which the name ”Leapfrog” is obvious. The key property of the Leapfrog is its time-reversibility: reversing the sign of Δt guarantees that you will follow the same orbit, but backwards. Integrators that harbour time-reversal are called symplectic.

Box 6: Leapfrog scheme

```
varr += acc*dt/2
xarr += varr*dt
acc = fn.forces (xarr, marr) #update accelerations
varr += acc*dt/2
```

Box 7: Alternative Leapfrog scheme (predictor-corrector)

```
old_x = copy(xarr) #use numpy's copy function to produce a clone of arrays
old_v = copy(varr)
old_a = copy(acc)
xarr += varr*dt +acc*dt**2/2 #predicted position
acc = fn.forces (xarr, marr)
varr += (acc+old_a)*dt/2
xarr = old_x +(old_v+varr)*dt/2 +(old_a-acc)*dt**2/4 #corrected position
```

Another way to write the Leapfrog without the need for introducing half-steps is as follows:

$$\mathbf{x}_{i+1} = \mathbf{x}_i + \mathbf{v}_i \Delta t + \frac{1}{2}\mathbf{a}_i (\Delta t)^2 \tag{11a}$$

$$\mathbf{v}_{i+1} = \mathbf{v}_i + \frac{1}{2}(\mathbf{a}_i + \mathbf{a}_{i+1})\Delta t \tag{11b}$$

Yet another method to write the Leapfrog is using a predictor-corrector scheme. First, we express the Leapfrog in implicit form:

$$\mathbf{v}_{i+1} = \mathbf{v}_i + \frac{1}{2}(\mathbf{a}_i + \mathbf{a}_{i+1})\Delta t; \tag{12a}$$

$$\mathbf{x}_{i+1} = \mathbf{x}_i + \frac{1}{2}(\mathbf{v}_i + \mathbf{v}_{i+1})\Delta t + \frac{1}{4}(\mathbf{a}_i - \mathbf{a}_{i+1})(\Delta t)^2 \tag{12b}$$

Appreciate again the symmetry in these expression. The ”problem” now is that we don't know the acceleration at the next timestep, which occurs in Equation (12) on the RHS. To solve this, we first calculate a predicted position, $\mathbf{x}_p$, based on a Taylor series around the point $\mathbf{x}_i$. Thus:

1. Predict:
$$\mathbf{x}_p = \mathbf{x}_i + \mathbf{v}_i \Delta t + \frac{1}{2}\mathbf{a}_i (\Delta t)^2 \tag{13}$$
2. Calculate the forces at the predicted position: $\mathbf{a}_p = \mathbf{a}(\mathbf{x}_p)$
3. Correct according to the time-symmetric equations above:
$$\mathbf{v}_{i+1} = \mathbf{v}_i + \frac{1}{2}(\mathbf{a}_i + \mathbf{a}_p)\Delta t. \tag{14}$$
and Equation (12b).

Such schemes are known as predictor-corrector schemes. Steps 2 and 3 may be repeated more than once for increased precision.

Exercise 3:

(a) Convince yourself that Equation (11) is time-reversible and that Equation (12) is equivalent to Equation (11).

(b) An implementation of the Leapfrog scheme of Equation (11) is given in Box 6. Verify that the algorithm indeed corresponds to Equation (11). However, the program may crash or give erroneous results, as the accelerations in the top line are not yet known when the program first enters the while loop. How should you correct this?

(c) An implementation for the predictor-corrector variant of the leapfrog is given in Box 7. Note the use of `numpy`'s `copy()` function to produce a clone of the array.

(d) Program the midpoint, and both leapfrog methods (Eq. [11] and Eq. [12]). Then, do an algorithmic comparison among the methods by calculations of the energy error in the 2-body problem after $t = 10^8$ s. Construct a table like:

Δt [yr]	Euler	Midpoint	RK4	Leapfrog	Leapfrog-2	Hermite
10^{-2}						
10^{-3}						
10^{-4}						
10^{-5}						

(You can fill in the Hermite column in the next section.)

2.5 Hermite integration

The Hermite algorithm can be regarded as a higher-order generalization of the predictor-corrector variant of the Leapfrog. First we predict the new positions/velocities:

$$\mathbf{x}_p = \mathbf{x}_i + \mathbf{v}_i \Delta t + \frac{1}{2}\mathbf{a}_i(\Delta t)^2 + \frac{1}{6}\mathbf{j}_i(\Delta t)^3 \tag{15a}$$

$$\mathbf{v}_p = \mathbf{v}_i + \mathbf{a}_i \Delta t + \frac{1}{2}\mathbf{j}_i(\Delta t)^2 \tag{15b}$$

where $\mathbf{j}_i$ – the "jerk" term – is the time-derivative of the accelerations $\mathbf{j}_i = \dot{\mathbf{a}}_i$. We will see that for the calculation of $\mathbf{j}$ we also need the velocities. Hence, the predicted velocity $\mathbf{v}_p$. After approximating $\mathbf{a}_{i+1}$ and $\mathbf{j}_{i+1}$ from the predicted positions and velocities, we "correct" these using a similar scheme as in Equation (12):

$$\mathbf{v}_{i+1} = \mathbf{v}_i + \frac{1}{2}(\mathbf{a}_i + \mathbf{a}_{i+1})\Delta t + \frac{1}{12}(\mathbf{j}_i - \mathbf{j}_{i+1})(\Delta t)^2 \tag{16a}$$

$$\mathbf{x}_{i+1} = \mathbf{x}_i + \frac{1}{2}(\mathbf{v}_i + \mathbf{v}_{i+1})\Delta t + \frac{1}{12}(\mathbf{a}_i - \mathbf{a}_{i+1})(\Delta t)^2 \tag{16b}$$

where, in the numerical implementation, the $i+1$ terms on the RHS should be replaced by the predicted values. There is some freedom in choosing the coefficients of the higher-order terms in Equation (16b) (*eg.* Kokubo & Makino 2004). Here we have chosen a scheme such that the $O(\Delta t)^3$ term disappears (Hut et al. 1995). The expressions above guarantee that the error is $O(\Delta t)^5$ – in other words, the Hermite is an 4th order scheme. Like the Leapfrog schemes the forces (accelerations and jerk terms) have to be calculated only once per iteration in the while loop.

Taking the time-derivative of Equation (1), and applying the chain rule, the jerk term readily follows:

$$\mathbf{j}_i = \frac{d}{dt}\mathbf{a}_i = \sum_{k \neq i} \frac{G m_k}{|x_{ki}|^3}\mathbf{v}_{ki} - \sum_{k \neq i} \frac{G m_k}{|x_{ki}|^5}(3\mathbf{v}_{ki} \cdot \mathbf{x}_{ki})\mathbf{x}_{ki} \tag{17}$$

(Here I switched the dummy index j to k to avoid confusion with the vectorial $\mathbf{j}$ on the LHS). Equation (17) looks intimidating, but in fact the only novelty is the calculation of the relative velocity $\mathbf{v}_{ki}$ and of the $(3\mathbf{v}_{ki} \cdot \mathbf{x}_{ki})$ dot product. Box 8 gives an example of how the main part of the `forces` function may look like (again without considering optimizations):

Box 8: forces (Hermite schemes)

```
def forces_Hermite (xarr, varr, marr):
    """
    computes the accelerations and the jerk terms
    """
    acc = zeros((3,pars.Np))#accelerations
    jer = zeros((3,pars.Np))#time-derivative of accelerations
    for i in range(pars.Np):
        for j in range(i+1, pars.Np):
            rji = xarr[:,j] -xarr[:,i] #relative position
            vji = varr[:,j] -varr[:,i] #relative velocity

            r2 = sum(rji**2)
            r1 = sqrt(r2)
            r3 = r1*r2

            rv = sum(rji*vji) #calculate the dot product
            rv /= r2

            force = pars.gN*rji/r3
            acc[:,i] += force*marr[j]
            acc[:,j] -= force*marr[i]

            #add the jerk terms
            jerk = pars.gN *(vji -3*rv*rji)/r3
            jer[:,i] += jerk*marr[j]
            jer[:,j] -= jerk*marr[i]
    return acc, jer
```

2.6 Individual timesteps⋆⋆

Until now, our program featured a single timestep. While this is sufficient for many applications, a fixed Δt becomes problematic in the case of close encounters. In close encounters, for example, the timestep should certainly be less than the duration of the encounter, $t_{\text{int},1} \sim x_{ij}/v_{ij}$ where x_{ij}, v_{ij} are the magnitudes of $\mathbf{x}_{ij}, \mathbf{v}_{ij}$. In that case the timestep must be reduced. An alternative estimate for the interaction time, in case of low (initial) velocities, is $t_{\text{int},2} \simeq \sqrt{x_{ij}/a_{ij}}$.

In the force computation algorithm above we could simply calculate these timescales at every point in the double for loop. The timestep over which particle i should be advanced is then the minimum in $\{t_{\text{int},ij}\}$ for all j.

There are many alternative, more sophisticated expressions for the timestep calculation. A popular choice for Hermite scheme's is (Aarseth 2003):

$$\Delta t_{i+i} = \eta \sqrt{\frac{a_{i+i}a^{(2)}_{i+1} + \dot{a}^2_{i+1}}{\dot{a}_{i+1}a^{(3)}_{i+1} + \left(a^{(2)}_{i+1}\right)^2}} \tag{18}$$

where η is a control parameter. In this method the timestep calculation does not involve the double while loop, but is calculated from the acceleration and its higher order derivatives. Note that only the accelerations a_{i+1} and jerks $\dot{a}_{i+1}$ are calculated explicitly. Using a Taylor series, we therefore estimate the higher-order time derivatives from the old values of these quantities:

$$\mathbf{a}_{i+1}^{(2)} \approx \frac{6(\mathbf{a}_i - \mathbf{a}_{i+1}) + (2\dot{\mathbf{a}}_i + 4\dot{\mathbf{a}}_{i+1})\Delta t_i}{(\Delta t_i)^2} \tag{19}$$

$$\mathbf{a}_{i+1}^{(3)} \approx \frac{12(\mathbf{a}_i - \mathbf{a}_{i+1}) + 6(\dot{\mathbf{a}}_i + \dot{\mathbf{a}}_{i+1})\Delta t_i}{(\Delta t_i)^3} \tag{20}$$

Now that each particle has a different timestep Δt_i how should we modify the algorithm? There are several options:

- Keep a global timestep (same Δt for every particle), but take it the minimum of all the ”interaction times” $\{t_{\mathrm{int}}\}$. We update all particles. This conservative approach will work, but for large N it comes at a significant efficiency overhead, as most of the particles do not need to be advanced over such small timesteps.
- Quantize time. Every particle is advanced over a time $\Delta t_i = A2^n$ with A constant and n the largest integer that satisfies $\Delta t_i \leq t_{\mathrm{int}}$. This means that particles will be updated at (future) times $\{t_n\}$ and that at time t only a subset of the particles (those where $t = t_n$) will be advanced. This is the so-called block timestep method.

3 Projects

For each of the project you should decide which of the algorithms above is the best-suited for your projects.

The meaning of the stars is the following:

(no stars):	a 3-body problem. Should be relatively straightforward to program, so you can focus on the physics.
$\star$:	a little more challenging
$\star\star$:	quite challenging. You will probably want to apply some optimizations to your code as it would otherwise be slow.

A starless project does not imply you cannot get a high grade.

3.1 3-body problem: interactions near the Hill sphere

Consider three particles:

1. The Sun;
2. The Earth, approximated to move on a circular orbit ($e = 0$), placed at a distance of a_E =1 AU;
3. A test particle ("test" means very low mass), which has s semi-major axis axis *similar* (but not entirely the same) as the Earth, $a_T = a_E + \Delta a$ but is initialized at the opposite side of the Sun, also at zero eccentricity.

Because the semi-major axis differ (slightly), the distance between the test particle and the Earth will decrease; after half the synodical time, they will hence experience an interaction. Your task is to explore this 3-body interaction as function of Δa. When do you see horseshoe orbits, Hill-penetrating orbits, or encounters that show little interaction (distant encounters). Plot, for example, the change in eccentricity the test particle has experienced as function of Δa.

3.2 Orbital decay

In this 2-body problem (the Sun and a test particle), you investigate the rate of orbital decay by adding a gas drag force of the form:

$$\mathbf{F} = -\frac{\mathbf{v} - \mathbf{v}_{\text{gas}}}{t_{\text{stop}}}; \qquad \mathbf{v}_{\text{gas}} = (v_K(a) - v_{\text{hw}})\mathbf{e}_\phi \tag{21}$$

to the equations of motion. For simplicity, let the headwind velocity be a constant. The goal is to test the radial and azimuthal drift formulas that you derived in class. You can first start with a fixed t_{stop} or a fixed $\tau_p \equiv t_{\text{stop}}\Omega$ but at some point you should switch to a fixed particle size and allow t_{stop} to depend on position (since the gas density in the disk also depends on radius).

3.3 Pressure bump*

This is a variation of the above, but modify it at two points:

- instead of taking a power law for the disk (midplane) pressure profile, $P \propto r^p$, (with $p < 0$ for a positive headwind) modify it in such a way that P has a local maximum at a certain radius r_0. This will make $v_{\text{hw}}(r)$ a function of radius r.
- also add a small radial component to $\mathbf{v}_{\text{gas}}$ in such a way that $\dot{M} = v_r\Sigma(2\pi r)$ is constant and negative (gas is accreting onto the star).

Use reasonable parameters. We are interested in the behavior around $r = r_0$. What happens to the drifting particles?

3.4 Settling

Consider a particle at 3 scaleheights, $z = 3h_{\rm gas}$ in the protoplanetary disk. Because of the vertical component of the solar gravity, the particle settles to the midplane. Your task is to describe this behavior as function of particle size. You can again take Equation (21) for the gas drag law.

3.5 Pebble accretion or escape*

Consider three bodies:

1. The Sun;
2. A protoplanet of mass $10^{-3}M_\oplus \le M_p \le M_\oplus$ located on a circular orbit
3. A pebble, characterized by a fixed dimensionless friction time $\tau_p = t_{\rm stop}\Omega$. In addition to the 2-body forces, the pebble also experiences a gas drag force of Equation (21).

The pebble starts at a radius exterior to the protoplanet, but it will drift inwards due to radial drift. Therefore, it will either be accreted by the protoplanet or pass by it. Start with $\tau_p = 1$ particles (when radial drift is fastest) and determine the fraction of particles that will be accreted.

3.6 Viscous stirring of planetesimals**

In this N-body project, consider N equal mass planetesimals orbiting the Sun. Place the planetesimals initially on near-circular orbits with a very low eccentrity e and inclination i, for example with rms-values of $\approx 10^{-8}$. Space them randomly (however, they should not be initialized too close to each other!). The goal is to plot the rms-values of the eccentricity and inclination as function of time.

3.7 Scattering of planetesimals**

Like the previous project, but make one of the planetesimals 1000x more massive and place it at the center of the belt. Describe what happens.

3.8 Resonance trapping*

Consider three bodies:

1. A solar-mass star;
2. A Jupiter-mass planet at 1 AU;
3. An Earth-mass planet at 2 AU.

Initialize the planets on circular orbits. However, assume that the outer planet experiences a gravitational force from the gas disk, which damps the eccentricity of the planet (confusingly, this force is sometimes referred to as the tidal force). For simplicity, let the form of the gravitational force be given by Equation (21), except that we substitute $t_{\rm grav}$ for $t_{\rm stop}$. Take $t_{\rm grav}$ at least 10 yr. The second parameter is the value of $v_{\rm hw}$.

Because of the gravitational gas drag force, the Earth-mass planet will migrate inwards. However, at some point it could end up (”captured”) in resonance with the Jupiter-mass planet. Your task is to investigate when resonance trapping takes place and when not.

3.9 Orbit crossing**

Consider 10 (or more) Mars-sized protoplanets ($M = 0.1\ M_\oplus$) separated by $\tilde{b}$ mutual Hill radii around 1 AU. Initially, the N-bodies are on circular orbits. Integrate the system in time until the point of the first orbit crossing t_cross, that is, until the point that the apoapsis and periapsis of neighboring protoplanets start to become equal. Stop the simulation and record the crossing time $t_\mathrm{cross}(\tilde{b})$. Start with $\tilde{b} = 4$ and subsequently choose larger $\tilde{b}$. Consider at least 3 simulations for every $\tilde{b}$ to get some estimate in the spread of t_cross.

References

Aarseth, S. J. 2003, Gravitational N-Body Simulations, ed. Aarseth, S. J. (Cambridge University Press)

Hut, P., Makino, J., & McMillan, S. 1995, ApJ, 443, L93

Kokubo, E. & Makino, J. 2004, Publications of the Astronomical Society of Japan, 56, 861

A Orbital elements

Here is a short algorithm to obtain the orbital elements (a, e, i, etc.) from the positions $\mathbf{x}$ and velocities $\mathbf{v}$. In this calculations $\mathbf{x}$ and $\mathbf{v}$ are always relative to the central body.

1. Calculate the angular momentum vector $\mathbf{l} = \mathbf{r} \times \mathbf{v}$. The vertical component is l_z. The inclination of the orbit is given by $i = \arccos(l_z/|\mathbf{l}|)$.
2. Calculate the eccentricity vector:
$$\mathbf{e} = \frac{\mathbf{v} \times \mathbf{l}}{G(m_\star + m_p)} - \frac{\mathbf{r}}{r} \tag{22}$$
It can be shown that $\mathbf{e}$ is a constant of motion and that it points in the direction of periapsis (you can verify this). The eccentricity of the orbit is the magnitude of the eccentricity vector $e = |\mathbf{e}|$.
3. The semi-major axis is given by:
$$a = \frac{h^2}{G(m_\star + m_p)(1 - e^2)} \tag{23}$$
4. Calculate the node vector: $\mathbf{n} = \hat{\mathbf{z}} \times \mathbf{l}$ This vector points in the direction of the ascending node, providing the orientation of the orbit in space. In case the motion lies in the plane (such that the cross product is identically zero) you can assign it arbitrarily, *eg.* $\mathbf{n} = (1, 0, 0)$.
5. The longitude of ascending node Ω_node, argument of periapsis ω, and true anomaly ν are:
$$\Omega_\mathrm{node} = \arccos\left(\frac{n_x}{|\mathbf{n}|}\right) \tag{24}$$
$$\omega = \arccos\left(\frac{\mathbf{n}\cdot\mathbf{e}}{|\mathbf{n}|e}\right) \tag{25}$$
$$\nu = \arccos\left(\frac{\mathbf{e}\cdot\mathbf{x}}{|\mathbf{x}|e}\right) \tag{26}$$