Chapter 1

The N-body problem applied to exoplanets

1.1 Motivation

The N-body problem is the problem of solving the motion of $N > 1$ stars or planets that are mutually gravitationally attracting each other. The 2-body problem is a special case which can be solved analytically. The solutions to the 2-body problem are the Kepler orbits. For $N > 2$ there is, however, no analytical solution. The only way to solve the $N > 2$ problem is by numerical integration of the Newtonian equations of motion.

Of course there are limiting cases where the $N > 2$ problem reduces to a set of independent $N = 2$ problems: To first order the orbit of the planets around the Sun are Kepler orbits: $N = 2$ solutions in which the two bodies are the Sun and the planet. That is because the mutual gravitational force between e.g. Venus and Earth is *much* smaller than the gravitational force of the Sun. So for the planets of our Solar System, and for most planets around other stars, the N-body problem can, *to first order*, be simplified into $N-1$ independent 2-body problems. That means that e.g. the Earth's orbit is only very slightly modified by the gravitational attraction by Venus or Mars. For most purposes we can ignore this tiny effect, meaning that the Kepler orbital characteristic are sufficiently accurate to describe most of what we want to know about the system.

However, for exoplanetary systems where the planets are closer to each other, the situation may be different. The mutual gravitational forces between the planets will still be small compared to that of the star, but they may be nevertheless large enough to play a role. In that case, we have no other choice than to actually model the full 3-body or N-body problem.

1.2 Numerical integration of an Ordinary Differential Equation: Euler's method

Before we go to the full N-body problem, let us first try to numerically solve an ordinary differential equation (ODE) for which we know the analytic solution:

$$\frac{dy(t)}{dt} = -A\,y(t) \tag{1.1}$$

where A is a given constant number greater than 0. The analytic solution is of course

$$y(t) = B\,e^{-At} \tag{1.2}$$

where B is an *integration constant*. The value of B is usually chosen such that $y(t = t_0)$ has the value of y as it is known to be at time $t = t_0$. This is called an *initial condition*. In this case it is simple to find B for a given initial condition $y(t_0)$: we have $B = y(t_0)\,e^{At_0}$.

Now suppose we wish to integrate Eq. (1.1) numerically. How do we do that? First we have to find out *at which times do we wish to know the value of* $y(t)$. We have to know this in advance, because a numerical solution is nothing else than a table of numbers y_i belonging to a table of times t_i. So we first have to define the list of times

t_i (i.e. $t_0, t_1, \cdots, t_n$) for which we wish to know the solution values y_i (i.e. $y_0, y_1, \cdots, y_n$). By definition t_0 will be the initial condition, and $t_i > t_0$ for all $i > 0$.

Now we start at $t = t_0$, for which we know the value of y, because that is the initial condition $y(t_0)$. We now wish to find the value of y at the next time t_1, which obeys, of course, $t_1 > t_0$. How do we do that?

1.2.1 Euler method

The simplest method is called the *Euler method*. The idea is based on the fact that the differential dy/dt is defined as

$$\left.\frac{dy}{dt}\right|_{t=t_i} = \lim_{\Delta t \to 0} \frac{y(t_i + \Delta t) - y(t_i)}{\Delta t} \tag{1.3}$$

This equation is exact, because it is the definition of the derivative of an equation. But it is only exact as long as $\Delta t \to 0$, i.e. the time step goes to zero. For finite Δt the above equation is only *approximately correct*. We can write this as:

$$\left.\frac{dy}{dt}\right|_{t=t_i} = \frac{y(t_i + \Delta t) - y(t_i)}{\Delta t} + \mathcal{O}(\Delta t^2) \equiv \left.\frac{\Delta y}{\Delta t}\right|_{t=t_i} + \mathcal{O}(\Delta t^2) \tag{1.4}$$

which says that for a small, but non-zero (i.e. finite) Δt the difference between the true dy/dt and the approximate $\Delta y/\Delta t$ is a small, and this difference shrinks proportional to Δt^2 if Δt is chosen ever smaller. For small enough Δt the approximation that $dy/dt \simeq \Delta y/\Delta t$ is reasonably good.

The idea behind the Euler integration method is based on this approximation of the *derivative* dy/dt as a discrete *difference* $\Delta y/\Delta t$. The ODE Eq. (1.1) then becomes:

$$\frac{y_{i+1} - y_i}{t_{i+1} - t_i} = -A\, y_i \tag{1.5}$$

We *know* the values of $t_{i+1} - t_i$ and y_i, but we want to *compute* the value of y_{i+1} (i.e. the future value of y). You can find this value of y_{i+1} simply by solving for y_{i+1} in Eq. (1.5). You obtain:

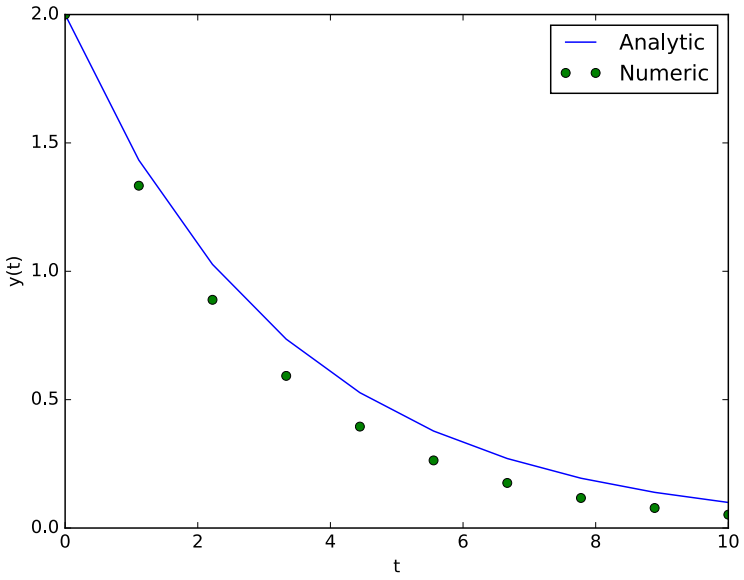
$$y_{i+1} = y_i - A\, y_i (t_{i+1} - t_i) \tag{1.6}$$

This is the *Euler integration* formula for the ODE

Here is a snippet of a Python code that does this:

```
import numpy as np
import matplotlib.pyplot as plt
A    = 0.3   # The A constant in the ODE
t0   = 0.    # Starting time
tend = 10.   # End time
nt   = 10    # Nr of time steps
t    = np.linspace(t0,tend,nt)
y    = np.zeros(nt)
y[0] = 2.    # Intial condition
for it in range(0,nt-1):
   y[it+1] = y[it] - A * y[it] * ( t[it+1] - t[it] )
yana = y[0]*np.exp(-A*t)
plt.figure()
plt.plot(t,yana,label='Analytic')
plt.plot(t,y,'o',label='Numeric')
plt.xlabel('t')
plt.ylabel('y(t)')
plt.legend()
plt.savefig('fig_ode_euler_1_1.pdf')
plt.show()
```

You can find this snippet in `snippets/snippet_ode_euler_1.py`.



More generally: if we have the ODE

$$\frac{dy(t)}{dt} = -F(y(t)) \tag{1.7}$$

where $F(y)$ is some function of y, the corresponding Euler integration formula is:

$$y_{i+1} = y_i - F(y_i)\,(t_{i+1} - t_i) \tag{1.8}$$

1.2.2 Time step cannot be too large

The Euler integration method, as well as most other numerical integration methods, has one serious weakness: while it can become arbitrarily accurate if you choose $\Delta t \equiv t_{i+1} - t_i$ arbitrarily small, it can also become arbitrarily wrong if you choose $\Delta t \equiv t_{i+1} - t_i$ arbitrarily large.

The time step size $\Delta t \equiv t_{i+1} - t_i$ is simply the time between two successive values of your chosen set of times $\{t_i\}$. However, if you choose these time intervals too large, you will get inaccurate results. Moreover, if the time step gets beyond a certain critical value, the resulting values y_i are not only wrong, they are exponentially diverging to ever larger values.

1.2.3 Substepping

If your intervals between the times t_i is too large, then how to solve this? Of course, one can make Δt smaller, but that means that you need, for the same time range, more time steps. In other words, you will need more t_i. This is not always desirable. For instance, you may simply wish to have just time steps between t_0 and t_{final}.

The solution is *substepping*. Between any two time steps t_i and t_{i+1} you can do as many substeps as necessary. And these intermediate step results do not have to be stored. You can simply replace the current with the next step. This smaller time step δt would then be $\delta t = \Delta t/(n_{\text{sub}} + 1)$ where n_{sub} is the number of substeps you wish to make.

Exercise 1: Adapt the above python script to divide each time step up in 10 substeps. Plot the results over the previous results so that you can compare.

1.2.4 How small do you need to make δt?

With the substepping you can keep the t_i you want, while at the same time obtain the desired accuracy by making the substep δt small enough. But what is "small enough"? We have to find an objective criterion what "small enough" is. Let us return to the actual equation (Eq. 1.1). The constant A has the dimension of 1/second. Therefore $1/A$ defines a characteristic time scale of the problem: $t_{\rm char} = 1/A$. It is therefore reasonable to express the desired time step δt in units of $t_{\rm char} = 1/A$:

$$\delta t = C t_{\rm char} = \frac{C}{A} \tag{1.9}$$

where C is a number we choose. If we choose $C \gg 1$, then the solution will be inaccurate or even numerically unstable. If we choose $C \ll 1$ then we waste too much computational time. Presumably we wish to choose C somewhere between 0.1 and 0.5 or so.

Note that, for a given value of δt, it is not guaranteed that Δt is an integer-multiple of δt. So instead of taking the exact δt of Eq. (1.9), we take the next smaller integer fraction of Δt. Here is a way to do that:

```
import math
dt    = t[it+1] - t[it]
C     = 0.5
dtsub = C / A
nsub  = math.floor( dt / dtsub )    # Nr of substeps (round down to integer)
dtsub = dt / (nsub+1)               # New dtsub is now always an integer fraction of dt
```

where Δt=`dt` and δt=`dtsub`.

Exercise 2: How does the computational cost (i.e. the number of computer operations) scale with δt?

Exercise 3: Experiment with choices of C and find, by trial and error, the value for which the relative error is about 1%. Overplot this result over the other ones.

1.2.5 Non-linear equations

Now let us return to the more general case of Eq. (1.7), which we repeat here for convenience:

$$\frac{dy(t)}{dt} = F(y(t)) \tag{1.10}$$

The numerical Euler method follows Eq. (1.8). How do we know which time sub-step δt we should take in this case? There is no constant A here. The trick is to compute the *Jacobian*:

$$J = \frac{\partial F}{\partial y} \tag{1.11}$$

The time step formula (1.9) now becomes:

$$\delta t = C t_{\rm char} = \frac{C}{|J|} \tag{1.12}$$

For the linear case (Eq. 1.1) we know that $F(y) = -Ay$, and so $J = -A$. But this time our formalism is more general.

1.3 Higher-order integration methods: Runge-Kutta methods

The Euler integration is the simplest and most intuitive method of numerical integration. But its accuracy scales only linearly with $1/\delta t$. Or in order words: the error scales only linearly with δt. The algorithm is said to be *a first order algorithm*.

If you want to have $10\times$ higher accuracy, you need $10\times$ more computer power. Computers are nowadays so fast, that it is often affordable to indeed simply invest more computer power. But for problems that require very high accuracy, the computational cost might become nevertheless too high.

The solution lies in better algorithms: second-order algorithms have an accuracy proportional to $1/\delta t^2$. That means that for a $10\times$ higher accuracy you need only $\sqrt{10}\times$ more computational effort. Or in other words: with a $10\times$ smaller time step δt you get a $100\times$ more accurate result.

The accuracy of third- and even higher-order methods scale even better with δt. But always keep in mind: *all* methods fail if the δt becomes of the order of $1/J$ (the Jacobian, see Section 1.2.5) or larger. The order of the algorithm only says how quickly the results get better as δt is taken ever smaller than $1/J$.

1.3.1 Second order method: the midpoint method (RK2)

The simplest second order method is the *midpoint method*, also often called the *second-order Runge-Kutta method*. The method first does a half-timestep to find a reasonable estimate of the value of y at half the time step:

$$y_{\text{mid}} = y_i + \frac{1}{2} F(y_i)\,(t_{i+1} - t_i) \tag{1.13}$$

Then it uses this value to compute the right-hand-side of the equation for the real integration step:

$$y_{i+1} = y_i + F(y_{\text{mid}})\,(t_{i+1} - t_i) \tag{1.14}$$

This is often called RK2 as in Runge-Kutte-2nd-order.

NOTE: Here, and in the following sections, we do not include sub-stepping in the equations (for simplicity), but the principle can be easily extended to substepping.

Exercise 4: Use the RK2 method for our simple ODE, and compare to the Euler method. Take, for the comparison, no substeps, so that the differences become more apparent.

1.3.2 Fourth order method: Runge-Kutta 4 (RK4)

The most popular higher-order integration method is RK4, a fourth-order Runge-Kutta method:

$$y_{i+1} = y_i + \frac{\delta t}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right) \tag{1.15}$$

$$t_{i+1} = t_i + \delta t \tag{1.16}$$

with

$$k_1 = F(y_i) \tag{1.17}$$

$$k_2 = F(y_i + k_1\,\delta t/2) \tag{1.18}$$

$$k_3 = F(y_i + k_2\,\delta t/2) \tag{1.19}$$

$$k_4 = F(y_i + k_3\,\delta t) \tag{1.20}$$

Exercise 5: Use the RK4 method for our simple ODE.

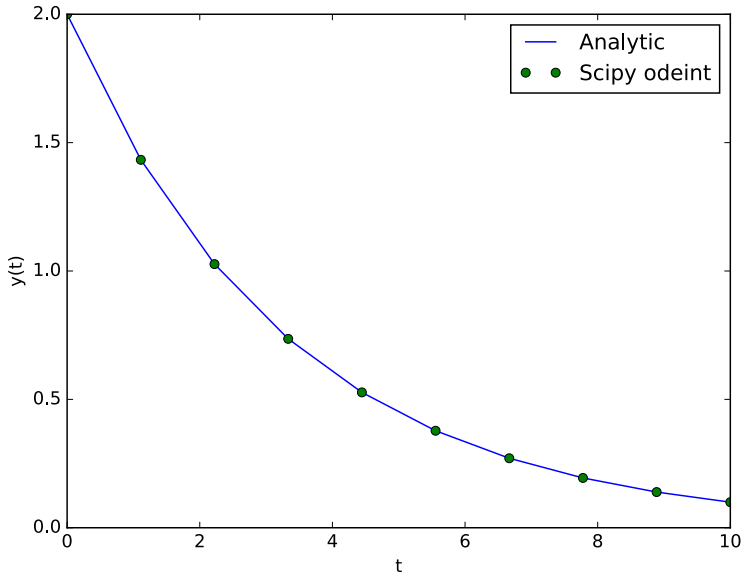
Exercise 6: Plot the error for the value at $t = 10$ (compared to the known analytic solution) as a function of n_{substep}. Overplot the same for the RK2 and Euler method.

1.4 Using built-in Python ODE integrator

Python has several standard methods built in into the `scipy` package. Here is an example of how to do the same as above, but now with a built-in method of `scipy`. We use the `odeint()` method, which is a Python interface to the famous `lsoda()` integrator from the `odepack` fortran library.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint

def f(y,t,A):
    return -A*y

A    = 0.3   # The A constant in the ODE
t0   = 0.    # Starting time
tend = 10.   # End time
nt   = 10    # Nr of time steps
t    = np.linspace(t0,tend,nt)
y0   = 2.    # Intial condition
sol  = odeint(f,y0,t,args=(A,))
y    = sol[:,0].T
yana = y[0]*np.exp(-A*t)
plt.figure()
plt.plot(t,yana,label='Analytic')
plt.plot(t,y,'o',label='Scipy odeint')
plt.xlabel('t')
plt.ylabel('y(t)')
plt.legend()
plt.savefig('fig_ode_scipy_1_1.pdf')
plt.show()
```

You can find this snippet in `snippets/snippet_ode_scipy_1.py`.



Note that this `odeint()` method automatically makes its internal substepping, so that you do not have to worry about it. In fact, as you can see, the results are extremely accurate. The reason is that `odeint()` allows you to specify the accuracy yourself by setting the *error tolerance*. The default tolerance is 10^{-8} (but see the manual of `odeint()` for more detailed information), meaning that `odeint()` will make as many substeps as necessary to achieve the desired accuracy.

Exercise 7: Inspect what the relative error of the solution at the final time step is. Look for the online manual of `scipy.integrate.odeint()` and find out how one can set the accuracy of the integration. Try out the above snippet with a different error tolerance and show that it was succesful.

1.5 Orbital integration

Now let us move to an astrophysical example: the integration of the orbit of a planet around the Sun. In this initial example we make the assumption that the planet's mass is infinitely much smaller than that of the Sun (we will have to drop this assumption later, when we include multiple planets).

The equations of motion of the planet are:

$$\frac{d\mathbf{x}}{dt} = \mathbf{v} \tag{1.21}$$

$$\frac{d\mathbf{v}}{dt} = -\frac{GM_\odot}{r^3}\mathbf{x} \tag{1.22}$$

$$\tag{1.23}$$

with $r = \sqrt{|\mathbf{x}|}$. Since both $\mathbf{x}$ and $\mathbf{v}$ are 3-D vectors, the full set of equations consists of 6 coupled ODEs.

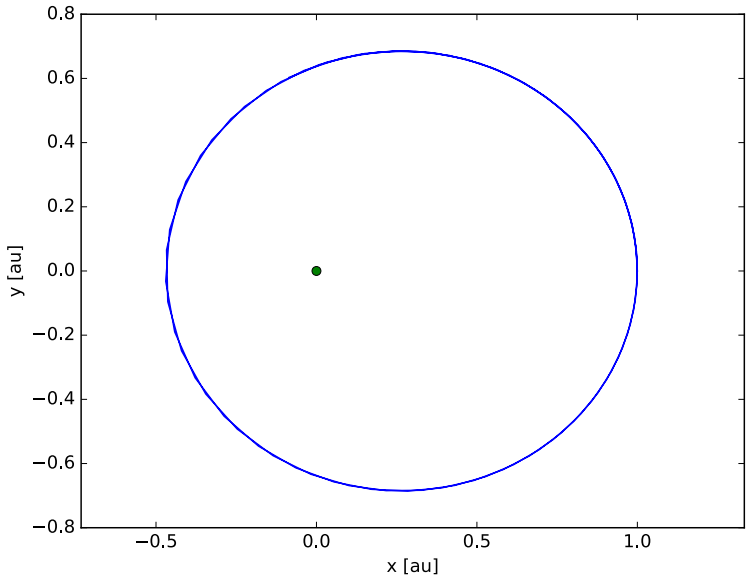
Here is an example Python snippet that models the orbit of the Earth if we would slow down the earth by 20%.

NOTE: We use CGS units here (as is usually done in astronomy).

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint

def f(y,t,mstar):
    G    = 6.67408e-08    # Gravitational constant in CGS units [cm^3/g/s^2]
    gm   = G*mstar
    x    = y[0:3].copy()
    v    = y[3:6].copy()
    r    = np.sqrt(x[0]**2+x[1]**2+x[2]**2)
    dxdt = v
    dvdt = -gm*x/r**3
    dy   = np.hstack((dxdt,dvdt))
    return dy

msun = 1.98892e33             # Solar mass              [g]
year = 31557600.e0            # Year                    [s]
au   = 1.49598e13             # Astronomical Unit       [cm]
t0   = 0.                     # Starting time
tend = 1.5*year               # End time
nt   = 100                    # Nr of time steps
t    = np.linspace(t0,tend,nt)
r0   = au                     # Initial distance of Earth to Sun
vp0  = 0.8 * 2*np.pi*au/year  # Initial velocity of Earth
x0   = np.array([r0,0.,0.])   # 3-D initial position of Earth
v0   = np.array([0.,vp0,0.])  # 3-D initial velocity of Earth
y0   = np.hstack((x0,v0))     # Make a 6-D vector of x0 and v0
sol  = odeint(f,y0,t,args=(msun,))
x    = sol[:,0:3].T
v    = sol[:,3:6].T
plt.figure()
plt.plot(x[0,:]/au,x[1,:]/au)
plt.plot([0.],[0.],'o')
plt.xlabel('x [au]')
plt.ylabel('y [au]')
plt.axis('equal')
```

```
plt.savefig('fig_kepler_1_1.pdf')
plt.show()
```

You can find this snippet in `snippets/snippet_kepler_1.py`.



Exercise 8: Explain the results. What is it that you are (physically) seeing? Why does the curve have slight wobbles on the left, but not (at least not discernable) on the right? Are these wobbles numerical errors of the integration scheme?

Exercise 9: By default both `rtol=1e-8` and `atol=1e-8`. Which of the two determine time substepping size for the `odeint()` method? Tip: What are the dimensions of `rtol` and `atol`?

1.6 N-body integration

Now let's extend what we learned to a multi-planet system, in particular to a system where the planets gravitationally influence each other. We use again the `odeint()` subroutine, but now we include not only the additional planets, *but also the star*. This is important, because for the N-body problem the reaction of the star to the pull of the planets can also affect the interactions between the planets.

1.6.1 A Python implementation

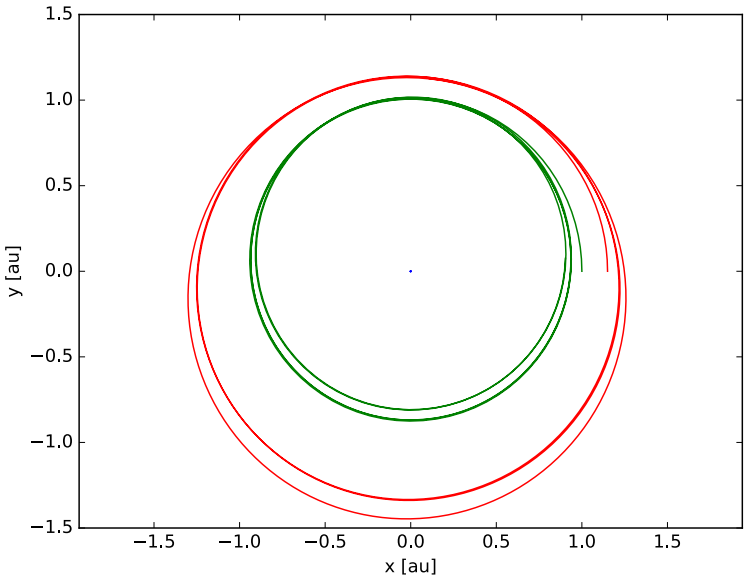
Here is a possible implementation in Python. For fun this example has two Jupiter mass planets on orbits not far from each other, so that they affect each other's orbits.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint

def f(y,t,m):
    G    = 6.67408e-08    # Gravitational constant in CGS units [cm^3/g/s^2]
    n    = len(m)
    plan = y.reshape((n,6))
    frc  = np.zeros((n,3))
    dy   = np.zeros((n,6))
```

```
    for i in range(n):
        x    = plan[i,0:3]
        for j in range(i+1,n):
            x1    = plan[j,0:3]
            dx    = x1-x
            r     = np.sqrt(dx[0]**2+dx[1]**2+dx[2]**2)
            df    = G*m[i]*m[j]*dx/r**3
            frc[i,0:3] += df[0:3]
            frc[j,0:3] -= df[0:3]
    for i in range(n):
        x    = plan[i,0:3].copy()
        v    = plan[i,3:6].copy()
        dxdt = v
        dvdt = frc[i,0:3]/m[i]
        dy[i,0:6] = np.hstack((dxdt,dvdt))
    return dy.reshape((6*n))

G    = 6.67408e-08            # Gravitational constant in CGS units [cm^3/g/s^2]
Msun = 1.98892e33             # Solar mass              [g]
Mju  = 1.899e30               # Mass of Jupiter         [g]
year = 31557600.e0            # Year                    [s]
au   = 1.49598e13             # Astronomical Unit       [cm]
t0   = 0.                     # Starting time
tend = 4.5*year               # End time
nt   = 400                    # Nr of time steps
t    = np.linspace(t0,tend,nt)
m    = np.array([Msun,Mju,Mju])   # Masses
a0   = np.array([0.,au,1.15*au,]) # Semi-major axes
phi0 = np.array([0.,0.,0.])       # Orbital locations in degrees
nb   = len(m)
x0   = np.zeros((nb,3))
v0   = np.zeros((nb,3))
pos  = np.zeros(3)
mom  = np.zeros(3)
for i in range(1,nb):         # Loop over all planets (i.e. excluding star)
    x0[i,0] = a0[i]*np.cos(phi0[i])
    x0[i,1] = a0[i]*np.sin(phi0[i])
    vphi    = np.sqrt(G*m[0]/a0[i])
    v0[i,0] = -vphi*np.sin(phi0[i])
    v0[i,1] =  vphi*np.cos(phi0[i])
    pos[:] += m[i]*x0[i,:]
    mom[:] += m[i]*v0[i,:]
x0[0,:] = -pos[:]/m[0]        # Total center of mass must be at (0,0,0)
v0[0,:] = -mom[:]/m[0]        # Total momentum must be (0,0,0)
xv0 = np.zeros((nb,6))
for i in range(nb):
    xv0[i,0:3] = x0[i,0:3]
    xv0[i,3:6] = v0[i,0:3]
y0   = xv0.reshape(6*nb)          # The full nb*6 element vector
sol  = odeint(f,y0,t,args=(m,))  # Solve the N-body problem
xv   = sol.reshape((nt,nb,6))    # Now extract again the x and v
x    = np.zeros((nb,3,nt))
v    = np.zeros((nb,3,nt))
for i in range(nb):
    for idir in range(3):
        x[i,idir,:] = xv[:,i,idir]
        v[i,idir,:] = xv[:,i,3+idir]
plt.figure()
for ibody in range(nb):
    plt.plot(x[ibody,0,:]/au,x[ibody,1,:]/au)
plt.xlabel('x [au]')
```

```
plt.ylabel('y [au]')
plt.axis('equal')
plt.savefig('fig_nbody_1_1.pdf')
plt.show()
```

You can find this snippet in `snippets/snippet_nbody_1.py`.



Exercise 10: Make a new version of this snippet in which you create a separate subroutine `nbody(m,x0,v0,t)`, which does the N-body integration for the initial conditions given by `x0[nb,0:3]` and `v0[nb,0:3]`, and for the time array `t`. In other words: put all the re-packaging of the data into (and from) a big array `y` into the `nbody()` subroutine so that you do not have to worry about this data-management anymore.

1.6.2 Long-term evolution of the orbits

So far we have integrated only a few orbits. If we wish to analyze the behavior of the system over thousands of orbits, then directly plotting the orbits themselves becomes problematic, because one will not recognize anything.

One solution is to plot two of the most important orbital parameters that would remain constant for single-planetary systems, and change only due to the interaction among the planets: the semi-major axis a and the eccentricity e of the planets.

The semi-major axis a of the planet can be computed from the center-of-mass velocity $v = |\mathbf{v}|$ and radius $r = |\mathbf{x}|$ through the specific orbital energy formula:

$$E = \frac{1}{2}v^2 - \frac{\mu}{r} = -\frac{\mu}{2a} \tag{1.24}$$

where $\bar{\mu} = GM_*^3/(M_* + M_p)^2$. Note that this formula is derived with all variables in the center-of-mass frame.

The eccentricity e of a planet with respect to the star can be obtained from the *eccentricity vector* $\mathbf{e}$ defined by[1]

$$\mathbf{e} = \left(\frac{|\Delta\mathbf{v}|^2}{\mu} - \frac{1}{|\Delta\mathbf{x}|}\right)\Delta\mathbf{x} - \frac{\Delta\mathbf{x}\cdot\Delta\mathbf{v}}{\mu}\Delta\mathbf{v} \tag{1.25}$$

where $\mu = G(M_* + M_p)$ where M_p is the mass of the planet and

$$\Delta\mathbf{x} = \mathbf{x_p} - \mathbf{x}_*, \qquad \Delta\mathbf{v} = \mathbf{v_p} - \mathbf{v}_* \tag{1.26}$$

[1] `https://en.wikipedia.org/wiki/Eccentricity_vector`

The eccentricity is the length of this vector:

$$e = |\mathbf{e}| \tag{1.27}$$

Note that these formulae are all done with the relative position and velocity between planet and star.

Exercise 11: Program a function that computes a and e. Plot how e change over time for 15 years. Explain why the eccentricities appear to change in sudden jumps.

1.7 The TRAPPIST-1 exoplanetary system

One of the most spectacular exoplanetary systems known today is the TRAPPIST-1 system[2]. It is a multi-planet system around a relatively nearby M-dwarf star. Its 7 known planets have masses not very different from the Earth, and several of them lie in the so-called *habitable zone*, meaning that they are at the right distance from the star that they have temperatures such that liquid water *could perhaps* exist on them. The distance to their host star is much less than the Earth-Sun distance, but still their temperatures are not very different from the Earht, because the star is so much less bright than the Sun.

The discovery paper is[3]: Gillon, M., Triaud, A. H. M. J., Demory, B.-O., Jehin, E., Agol, E., Deck, K. M., et al. (2017). Seven temperate terrestrial planets around the nearby ultracool dwarf star TRAPPIST-1. Nature, 542(7), 456

The planets were discovered through the *transit method*. In this method the existence of a planet is inferred by the slight dimming of the stellar light by the passage of the planet in front of the star. In the case of TRAPPIST-1, many planets passed in front of the star, causing a multitude of periodic dimming events.

The orbits of this mini-planetary system lie all very close to each other, so that it is to be expected that the planets influence each other's orbits through their mutual gravity.

The most direct consequence of this mutual interaction between the planets is that the transits of the planets will vary slightly in their timing. If a planet would be a single planet, then the transit (if it occurs at all) will have a perfect orbital period: each transit happens at exactly the same time interval as the previous one. However, if there are more than 1 planet in the system, this *transit timing* may vary a bit, depending on the locaiton of the other planets. The transit may thus occur slightly earlier or later than expected based on a single planetary orbit. In fact, through *transit timing variations (TTV)* it is possible to infer the existence of planets that may not be in the same plane, and thus may not transit at all. But, equally importantly, these TTV allow the calculation of the planet masses.

To do these things, one must perform *N-body calculations*, to compute the effect of the mutual gravitational pull. Finding the proper orbital parameters from the observations requires a trial-and-error procedure, which can be quite time-consuming. But from the already-inferred orbital parameters we can redo our own N-body calculation of the system.

Exercise 12 (voluntary): Check out the discovery paper and find the parameters of the TRAPPIST-1 system. Set up the appropriate initial conditions and simulate a few orbits. Try to find from the simulations the typical expected magnitude of TTV (in minutes) of the planets.

[2] `http://www.trappist.one/`
[3] `http://doi.org/10.1038/nature21360`