N-Body modelling exercise session with REBOUND

In the following, I have defined different problems that can be addressed with rebound. The complexity increases from problem to problem. You do not have to work on all of them. I suggest to start with the first one. All of the examples can be addressed with the python version of rebound.

1 The N-Body Code REBOUND

REBOUND is a software package that can integrate the motion of particles under the influence of gravity. The particles can represent stars, planets, moons, ring or dust particles. REBOUND is free software and published under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version[1]. REBOUND's maintainer is Professor Hanno Rein (`https://rein.utsc.utoronto.ca/`). The source code can be obtained via github `https://github.com/hannorein/rebound` and the complete documentation is online via `https://rebound.readthedocs.io/en/latest/`.

REBOUND's core is written in the programming language C, but it offers a very nice and user-friendly python interface. **We highly recommend to use the python interface for the exercises.**

1.1 Download and Installation

Depending on your choice of programming language, you have to install REBOUND in the following way. For the problems discussed in the following, the python version is fully sufficient, more easily to install and therefore the recommended choice for today. In addition to the following short steps, you may also read `https://www.tat.physik.uni-tuebingen.de/~schaefer/teach/f/how_to_install_rebound.pdf`. Please consult also the documentation about the installation if you face any problems.

1.1.1 Python

To install the python module for REBOUND, you can use `pip`. Make sure to use python version 3 only.

```
pip install rebound
```

To verify the installation, try to import the module in the commandline interpreter and load the help for the module

```
$ python
Python 3.10.4 | packaged by conda-forge | (main, Mar 24 2022, 17:39:37) [Clang
    12.0.1 ] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import rebound
>>> sim = rebound.Simulation()
>>> sim
<rebound.simulation.Simulation object at 0x103a07540, N=0, t=0.0>
>>>
```

[1] `http://www.gnu.org/licenses/`

Additionally, install all other packages that are required for the following exercises: `reboundx`, `numpy`, `matplotlib`.

1.1.2 C

To clone the source code from the github repository, use the following command, which also runs a first example to test if the compilation was successfull

```
git clone http://github.com/hannorein/rebound && cd rebound/examples/
    shearing_sheet && make && ./rebound
```

For your own exercises, you can simply copy one of the example files in rebound/examples and modify it and use the Makefile provided there, or you use the following Makefile template (Linux, macOS) to compile source file `foo.c` and link to rebound (make sure to set `REBOUND_SRC` accordingly)

```
OPT+= -Wall -g -Wno-unused-result
OPT+= -std=c99 -Wpointer-arith -D_GNU_SOURCE -O3
LIB+= -lm -lrt

# your preferred compiler
CC = gcc
# path to the source directory of your rebound installation
REBOUND_SRC=../../src/rebound/src/

all: librebound
        @echo ""
        @echo "Compiling problem file ..."
        $(CC) -I$(REBOUND_SRC) -Wl,-rpath,./ $(OPT) $(PREDEF) foo.c -L. -
    lrebound $(LIB) -o foo
        @echo ""
        @echo "REBOUND compiled successfully."

librebound:
        @echo "Compiling shared library librebound.so ..."
        $(MAKE) -C $(REBOUND_SRC)
        @-rm -f librebound.so
        @ln -s $(REBOUND_SRC)/librebound.so .

clean:
        @echo "Cleaning up shared library librebound.so ..."
        @-rm -f librebound.so
        $(MAKE) -C $(REBOUND_SRC) clean
        @echo "Cleaning up local directory ..."
        @-rm -vf rebound
```

1.2 Documentation and Tutorial

REBOUND is a well documented software package and the webpage offers a large number of examples both in C and python. Hence, we do not give a more detailed description in this script. Please see `https://rebound.readthedocs.io/en/latest/examples.html` for examples and `https://rebound.readthedocs.io/en/latest/quickstart.html` for the quickstart guide.

If you run into problems during the installation and fail to run an example, please ask for help.

2 Exercises

In the following subsections you find the exercises that you have to solve during this lab-work. Be sure to add your findings in your minutes and if important to the final protocol. Think about meaningful plots to state your results and add them to the protocol.

2.1 The Two-Body Problem

Consider the two-body problem with the following setup: eccentricity $e = 0.3$, semi-major axis $a = 1$, $m_1 = 1$, $m_2 = 10^{-3}$. With the gravitational constant G set to 1, the orbital period is $\approx 2\pi$. Energy and angular momentum should be conserved, they are given by

$$E = -\mu \frac{GM}{2a},$$
$$L = \mu \sqrt{(1-e^2)GMa},$$

where μ denotes the reduced mass and M the total mass

$$\mu = \frac{m_1 m_2}{M},$$
$$M = m_1 + m_2.$$

Choose the z-axis as the direction of the angular momentum $L_z = L$. Choose the x-axis as the direction of the periapsis to body 2. At time $t_0 = t(0) = 0$, both bodies are at closest distance $d_p = a(1-e)$. At time t_0, the positions and the velocities of the two bodies are given by $v_{y1}(0) = -L/(d_p m_1)$, $x_1(0) = -d_p m_2/M$ and $v_{y2}(0) = L/(d_p m_2)$, $x_2(0) = d_p m_1/M$. The motion of the two bodies is solely in the xy-plane.

We want to test the quality of the various integrators from the REBOUND software package. Integrate the system for 10^3 orbital periods at first with the Leap-Frog integrator with different fixed time-steps 1 to 10^{-6} and observe the energy, angular momentum, and the orbital elements. Try also another integrator!

There are additional diagnosis functions to compute conserved values, e.g., in C `reb_calculate_energy(struct reb_simulation *)` or in Python `sim.calculate_energy()`.

2.1.1 Solution with C

```c
// note: this is not the complete source code, just a snippet
// add the missing lines

#define TWOPI 6.283185307179586

struct reb_simulation* r = reb_create_simulation();

/* setup of the simulation
 * create two particles via
 * struct reb_particle p1 and p2
 * add the two particles to the simulation via reb_add(r, p)
 *
 */
```

```c
 // choose Leap-frog integrator
r->integrator = REB_INTEGRATOR_LEAPFROG;
// fixed time step
r->dt = 1e-4; // and 1 and 1e-1 and 1e-2 and 1e-3 and ...

// function pointer to the heartbeat function.
// the function is called after each dt
r->heartbeat = heartbeat;

// move all to center of mass frame
reb_move_to_com(r);

// now integrate for 1000 orbits, 2pi is one orbit
reb_integrate(r, 1000*TWOPI);
```

Define a heartbeat function `void heartbeat(struct reb_simulation *r)` to write the orbits, eccentricities and energy with the help of the builtin functions `reb_output_ascii` and/or `reb_output_orbits` to files and plot the values (using gnuplot or python or your preferred plotting tool).

2.1.2 Solution with Python

```python
#!/usr/bin/env python3
# integrate two body problem, awful formatting to fit on a single page
import sys
import numpy as np
import matplotlib.pyplot as plt
import rebound

# create a sim object
sim = rebound.Simulation()

# set the integrator to type REB_INTEGRATOR_LEAPFROG
sim.integrator = "leapfrog"
# and use a fixed time step
sim.dt = 1e-4
# here add your particles....
sim.add(m=1.0)
sim.add(m=1e-3, a=1.0, e=0.3)
# do not forget to move to the center of mass
sim.move_to_com()
# create time array, let's say 1 orbit, plot 250 times per orbit
Norbits = 1
Nsteps = Norbits*250
times = np.linspace(0, Norbits*2*np.pi, Nsteps)
x = np.zeros((sim.N, Nsteps)) # coordinates for both particles
y = np.zeros_like(x)
energy = np.zeros(Nsteps) # energy of the system
# now integrate
for i, t in enumerate(times):
    print(t, end="\r")
    sim.integrate(t, exact_finish_time=0)
    energy[i] = sim.calculate_energy()
    for j in range(sim.N):
        x[j,i] = sim.particles[j].x
        y[j,i] = sim.particles[j].y
```

```
print("Done, now plotting...")
# plot the orbit
fig, ax = plt.subplots()
ax.scatter(x,y, s=2)
ax.set_title("Orbit with step size %g" % sim.dt)
ax.set_aspect("equal")
ax.set_xlabel("x-coordinate")
ax.set_ylabel("y-coordinate")
plt.grid(True)
fig.savefig("orbit"+str(sim.dt)+".pdf")
# plot the energy
fig, ax = plt.subplots()
ax.scatter(times, np.abs(energy-energy[0])/np.abs(energy[0]), s=2)
print(energy)
ax.set_title("Energy with step size %g" % sim.dt)
ax.set_xlabel("time")
ax.set_yscale("log")
ax.set_ylabel("energy")
plt.grid(True)
fig.savefig("energy"+str(sim.dt)+".pdf")
print("Done.")
```

The orbital elements are also stored in `sim.particles`, e.g., the eccentricity of the second particle is given by `sim.particles[1].e`. Plot the orbital elements for different fixed time steps and test if the conservation of energy and angular momentum is given.

2.2 Stability of a planetary system — The co-planar Three-Body-Problem

This example follows the ideas of the publication by Gladman (1993): We have a co-planar planetary system with a massive central mass m_1 and two planets with masses m_2 and m_3, please see fig. 1 for a sketch. The initial setup is as follows: A central mass m_1, which is initially orbited by two smaller bodies (planets) on circular orbits, which start in opposition ($\delta\phi = \pi$). The mutual gravitational perturbations will change the shape of the orbits, the eccentricities e and semi-major axes a. Large changes are expected when the two planets are close in conjunction. A system is called Hill-stable (or simply stable), if close encounters are excluded for all times.

The stability of the system can be examined in dependence of the initial conditions and the masses. We will investigate the conditions described in Fig. 1.

For small initial separations Δ of the two given masses m_2, m_3, we will expect larger gravitational perturbations of the two masses and hence higher changes in e and a.

Gladman (1993) finds for small masses m_2 and m_3 and initially circular orbits the following criterion for stability for the initial separation of the two circular planetary orbits

$$\Delta_c \simeq 2.40\,(\mu_2 + \mu_3)^{1/3}. \tag{1}$$

For non-circular orbits, more general criteria can be derived, but this is off the scope of this experiment.

In order to study the stability, we start with a given Δ and perform the integration of the trajectory for 10^3 orbits. The time evolution of the orbital elements have to be plotted.

Instability will be detected by close-encounters. A close-encounter is given, when the distance between the two planets is lower than the Hill radius of the more massive one of

the two bodies. The Hill radius is given by

$$R_{in} \approx a\sqrt[3]{\mu/3}, \tag{2}$$

where the distance of the planet to the star is denoted by a and μ denotes the mass ratio of the smaller object to the more massive one.

In the exercise, we determine numerically the critical distance Δ_c. For values lower than Δ_c, the system is unstable. The precise value of Δ_c might depend on the applied numerical integrator.

REBOUND can automatically find collisions between the objects if they were given a radius when added to the simulation[2]. It allows to define a function which is called for each collision. We will use this feature to add a global counter which we increase if the planets get closer than their Hill's sphere.

```
# count number of collisions
def collision_print_only(sim_pointer, collision):
    global cnt
    sim = sim_pointer.contents           # get simulation object from pointer
    print(sim.t)                         # print time
    # print(sim.particles[collision.p1].x) # x position of particle 1
    # print(sim.particles[collision.p2].x) # x position of particle 2
    cnt += 1
    return 0                             # Don't remove either particle

# and in the main scope
cnt = 0
# collision detection
sim.collision = "direct"
# call this function if a collision is found
sim.collision_resolve = collision_print_only
```

The setup

Use the standard IAS15 integrator and (a) masses $m_1 = 1$, $m_2 = m_3 = 10^{-5}$, $a_2 = 1$, $a_3 = a_2 + \Delta$, $e_{2,3} = 0$. Add the masses to the REBOUND simulation and indicate their collision radius by using the argument r in function `sim.add`, i.e.

```
# here add your particles....
# with something like this
m1 = 1.0
m2 = m3 = 1e-5
deltacritical = 2.4 * (m2/m1 + m3/m1)**(1./3)
rhill = 1.0*(m2/(3.*m1))**(1./3)
sim.add(m=m1)
sim.add(m=m2, a=1.0, e=0.0, r=rhill)

# choose DELTA accordingly: for DELTA < deltacritical, we expect a lot of
    close encounters, for DELTA > deltacritical, the system is expected to be
    stable, place the second planet in opposition using omega=np.pi
sim.add(m=m3, a=1.0+DELTA, e=0.0, r=rhill, omega=np.pi)
# do not forget to move to the center of mass
sim.move_to_com()
```

[2] https://rebound.readthedocs.io/en/latest/collisions/

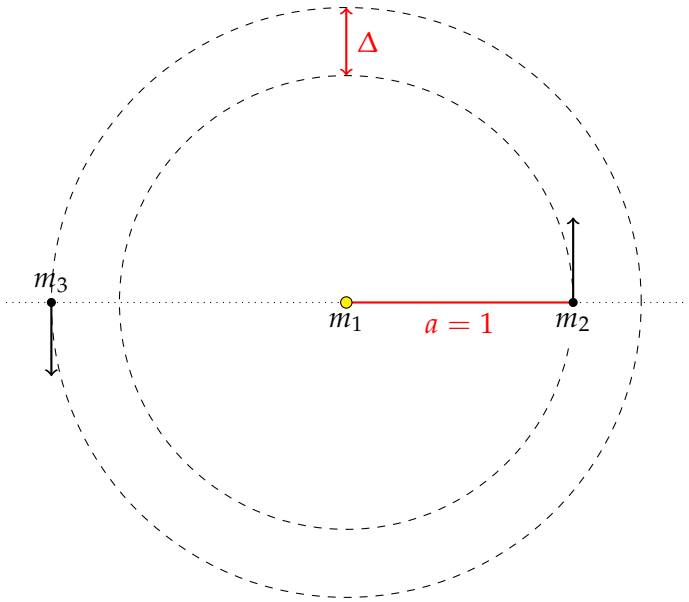

Figure 1: Schematic representation of a three body problem, where two smaller test masses m_2 and m_3 orbit a larger central mass m_1, it holds $m_2 + m_3 \ll m_1$. The initial semi-major axis of mass m_2 is normed to 1 in respect to the central mass m_1, and the semi-major axis of m_3 to $1 + \Delta$. The initial locations of the smaller bodies are opposed, $\delta\phi = \pi$, (after Gladman, 1993).

Simulate the system for approximately 10^4 orbits (one period is approximately 2π), count the numbers of close encounters and plot the evolution of the semi-major axes and eccentricities of the two planets for five different values of the initial separation Δ: 10%, 50%, 100%, 150% and 1000% of Δ_c.

(b) Repeat exercise (a) with masses $m_1 = 1$, $m_2 = 10^{-5}$, $m_3 = 10^{-7}$, $a_2 = 1$, $a_3 = a_2 + \Delta$, $e_{2,3} = 0$. Describe your findings!

2.3 Stability of Saturn's rings

The idea for this exercise is based on the paper by Vanderbei & Kolemen (2007)[3]. They give a self-contained modern linear stability analysis of a system of n equal mass bodies in circular orbit around a single more massive body. We will investigate their analytical findings with simulations.

Consider following scenario: one high mass body with mass M is orbited by n lower mass bodies with equal masses m at distance $r = 1$. The initial angular velocity ω of the lower mass bodies is

$$\omega = \sqrt{\frac{GM}{r^3} + \frac{GmI_n}{r^3}}, \tag{3}$$

[3] https://arxiv.org/abs/astro-ph/0606510

n	linear stability analysis γ	numerical simulation analysis γ
8	2.412	2.4121
10	2.375	2.3753
36	2.306	2.3066
100	2.300	2.2999

Table 1: Values for the parameter γ found by Vanderbei & Kolemen.

where I_n is given by

$$I_n = \sum_{k=1}^{n-1} \frac{1}{4} \frac{1}{\sin(\pi k/n)}.$$

The result of Vanderbei & Kolemen is that the system is linear stable as long as

$$m \leq \frac{\gamma M}{n^3}, \tag{4}$$

with the values for γ given in table 1. The goal of this exercise is to reproduce the analysis by Vanderbei & Kolemen. Can you reproduce their result with your simulations? Use Saturn's mass for the central object ($M = 2.857\,166\,56 \times 10^{-4} \mathrm{M}_\odot$), set the mass unit to $\mathrm{M}_\odot$ and use $G = 1$, set the initial distance of n lower mass bodies to $r = 1$ and the velocity according to eq. (3), use the leap frog integrator with $\Delta t \approx 10^{-5} P = 2\pi/\omega \times 10^{-5}$. Vary both n and the mass m of the n smaller objects, check if you expect a stable system according to eq. (4) and integrate the system for 100 t_p. If the masses are still orbiting Saturn, we consider the systems as stable. Try different systems using values from table 1, i.e. start with $n = 8$ and vary the mass from a value which gives a stable configuration to a value that yields instability in 10-20 steps, proceed in the same manner for $n = 10, 36, 100$.

2.4 Jupiter and Kirkwood gaps

In this problem we want to address the influence of Jupiter on the asteroid belt. A Kirkwood gap is a gap in the distribution of the semi-major axes of the asteroids. They correspond to the locations of orbital resonances with Jupiter. We will use inactive particles to model the asteroids and consider only the Sun, Jupiter and Mars as gravitating objects (`r->N_active=3`). Hence, the asteroids have no feedback on Sun, Jupiter and Mars and are just tracer particles. Set up the Sun-Mars-Jupiter system with the help of the NASA Horizons web-interface to get initial conditions for Mars and Jupiter (Ephemeris Type VECTORS) and add randomly 10 000 inactive particles between 2 and 4 au in the midplane.

Let the system evolve for several one hundred thousands of years and look for the Kirkwood gaps (cf. Figure 2).

Note: This will probably require to run the code over night (depending on your hardware).

Generate meaningful plots (eccentricity over semi-major axis, histogram plot of number of asteroids over semi-major axis) at different times (e.g., every one hundred thousand years) until at least one million years to show the formation of the dips. Add the known Kirkwood gaps to your plots. Do you find agreement with your results?

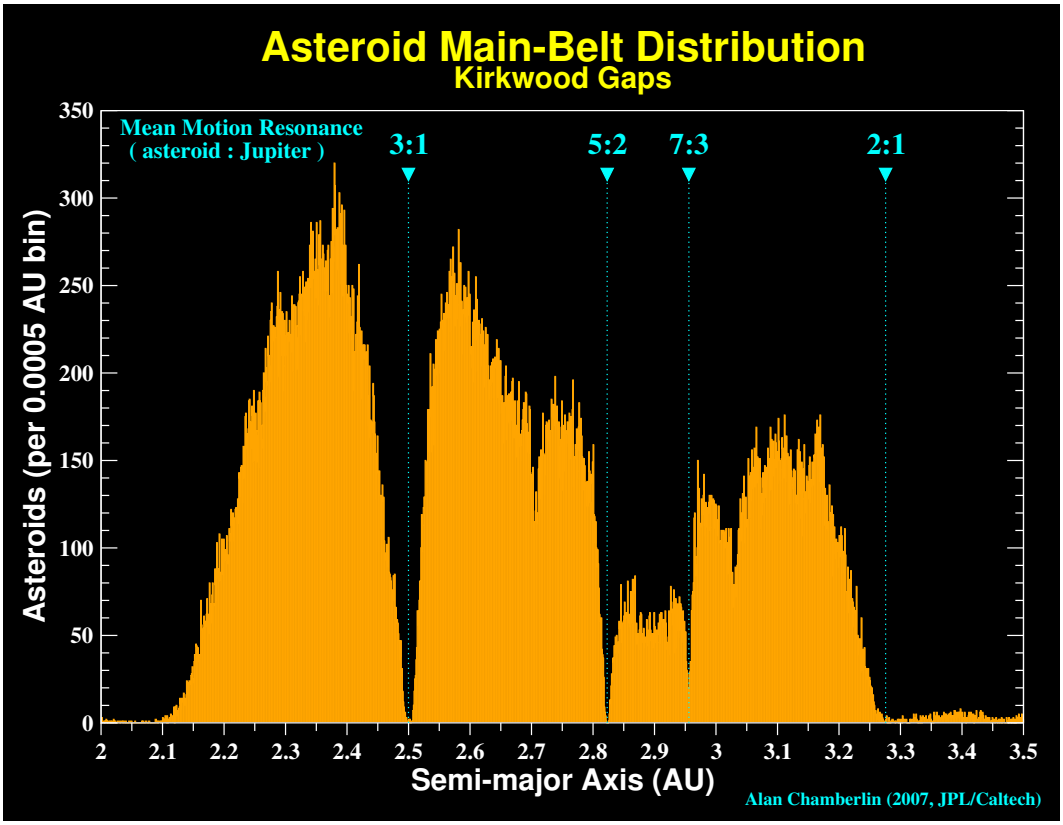

Figure 2: Kirkwood gaps in the asteroid belt (Chamberlin 2007).

2.4.1 Solution with C

The following code snippet can be used to generate testparticles between 2 and 4 au with eccentricities between 0 and 0.5.

```
    double au = 1.496e11;
    while (r->N < 10000) {
        // here we need to randomly generated particles between 2 and 4 au
        double a = 2*((double)rand()/RAND_MAX)+2;
        // and eccentricity between 0 and 0.5
        double e = 0.5*((double)rand()/RAND_MAX);
        // we use SI units, we do not know why, it'd better to use more
    convenient units
        a *= au;
        // requires rebound version >=3.17
        reb_add_fmt(r, "a e", a, e);
    }
```

2.4.2 Solution with Python

The following code snippet can be used to generate testparticles between 2 and 4 au with eccentricities between 0 and 0.5.

```
solarmass = 1.989e30
au = 1.496e11
```

```
gravitationalconstant = 6.67408e-11

# create a sim object
sim = rebound.Simulation()
# use SI units
sim.G = gravitationalconstant

# add 10000 inactive tracerparticles between 2 and 4 au
N_testparticle = 10000
a_ini = np.linspace(2*au, 4*au, N_testparticle)
for a in a_ini:
    sim.add(a=a, f=np.random.rand()*2.*np.pi, e=0.5*np.random.rand()) # mass
    is set to 0 by default, random true anomaly, random eccentricity

# set a reasonable time step
# we use 1e-2 of an orbit at 2 au
orbit = 2*np.pi*np.sqrt(8*au*au*au/(gravitationalconstant*solarmass));
sim.dt = orbit*1e-2
```

2.5 Resonant capture of planet

In this problem, we investigate the phenomenon of planet migration and resonant trapping using REBOUNDx. REBOUNDx (x for eXtras) is a library for additional effects for REBOUND N-body simulations. The package was created and is maintained by Dan Tamayo, see the official github repo for more details and examples: `https://github.com/dtamayo/reboundx`. REBOUNDx offers some physical effects as out-of-the-box modules, which can be easily added to an existing REBOUND simulation. These effects include for example radiation forces, relativistic corrections, gravitational corrections for oblate objects (J2, J4 harmonics), and various options to model migration.

We will use REBOUNDx' module `exponential_migration` to simulate a migrating, Neptune-sized planet which can capture an existing inner planet in a resonance.

An orbital resonance occurs when the orbital periods of two orbiting bodies are related by a ratio of integers[4]. A resonance can either stabilize or de-stabilize the system. In the example from the last section, the Kirkwood gaps are located at the mean-motion resonances with Jupiter.

The following code snippet adds an exponential migration on the object called Neptune, which migrates from 24 au to 10 au with an e-folding time of 10^5 years. Depending on the migration speed, Neptune will be able to capture the inner planet in a resonance. Setup a simulation with the three objects using the following snippets depending on your choice of programming language and integrate the system for at least one million years. Plot the evolution of the semi-major axes and eccentricities of the planets for different migration rates of Neptune. Additionally track the ratio of the orbital periods of the two planets. What do you find?

2.5.1 Solution with C

For this example in C, you must have installed REBOUNDx from `https://github.com/dtamayo/reboundx`.

[4] A very nice example for resonances is given by Jupiter's Galilean moons

```c
int main(int argc, char* argv[]) {
    struct reb_simulation* sim = reb_create_simulation();
    // Setup constants, that's G for sim.units = ('yr', 'AU', 'Msun')
    sim->G = 39.476926421373679764;
    sim->heartbeat = heartbeat;
    struct reb_particle p = {0};
    p.m     = 1.;
    reb_add(sim, p);
    double m = 5.1e-5;
    double a1 = 24.0;
    double e = 0.01;
    double inc = 0.;
    double Omega = 0.;
    double omega = 0.;
    double f = 0.;
    double m2 = 6e-6;
    double a2 = 10.0;
    double e2 = 0.0;
    struct reb_particle p1 = reb_tools_orbit_to_particle(sim->G, p, m, a1, e,
   inc, Omega, omega, f);
    struct reb_particle p2 = reb_tools_orbit_to_particle(sim->G, p, m2, a2, e2
   , inc, Omega, omega, f);
    reb_add(sim,p1);
    reb_add(sim,p2);
    reb_move_to_com(sim);

    struct rebx_extras* rebx = rebx_attach(sim);

    struct rebx_force* em = rebx_load_force(rebx, "exponential_migration");
   // add force that adds velocity kicks to give exponential migration
    rebx_add_force(rebx, em);
    double tmax = TMAX; // TMAX is a global define
    rebx_set_param_double(rebx, &sim->particles[1].ap, "em_tau_a", 1e5);
      // add migration e-folding timescale
    rebx_set_param_double(rebx, &sim->particles[1].ap, "em_aini", 24.0);
      // add initial semimajor axis
    rebx_set_param_double(rebx, &sim->particles[1].ap, "em_afin", 10.0);
      // add final semimajor axis
    reb_integrate(sim, tmax);
    rebx_free(rebx);    // Free all the memory allocated by rebx
    reb_free_simulation(sim);
}
```

2.5.2 Solution with Python

For this example, you must have installed the reboundx module for python.

```python
sim = rebound.Simulation()  # Initiate rebound simulation
sim.units = ('yr', 'AU', 'Msun')
sim.add(m=1)
sim.add(m=5.1e-5, a=24., e=0.01, hash="neptune") # Add Neptune (pre-migration)
    at 24 AU
sim.add(m=6e-6, a=10., e=0.0, hash="planet") # Add 2nd earth mass planet

sim.move_to_com()

rebx = reboundx.Extras(sim) # Initiate reboundx
```

```
mod_effect = rebx.load_force("exponential_migration")  # Add the migration
    force
rebx.add_force(mod_effect)  # Add the migration force

# based on example from https://github.com/dtamayo/reboundx/blob/master/
    ipython_examples/ExponentialMigration.ipynb
sim.particles['neptune'].params["em_aini"] = 24.  # parameter 1: Neptune's
    initial semimajor axis
sim.particles['neptune'].params["em_afin"] = 10.0  # parameter 2: Neptune's
    final semimajor axis
# this parameter is to play around with
# using 1e5 the resonance will be 3:2, using 1e6 it's 2:1
sim.particles[1].params["em_tau_a"] = 1e5  # parameter 3: the migration e-
    folding time
```

References

Gladman, B. 1993. Dynamics of systems of two close planets. *Icarus* 106, 247–+.

Happy Coding! ☺☕

Anna Penzlin, Christoph Schäfer, last changed 1 July 2022