EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

MATHEMATISCH-
NATURWISSENSCHAFTLICHE
FAKULTÄT

Practical Course in Astronomy

N-Body Simulations with REBOUND

Christoph Schäfer

Kepler Center for Astro and Particle Physics
Institut für Astronomie und Astrophysik
Abteilung Computational Physics
Auf der Morgenstelle 10
72076 Tübingen

Version: 5 September 2024

http://www.uni-tuebingen.de/de/4203

Contents

Note

The goal of the labwork at hand for the student is to learn and apply the widely-used N-Body integrator software package REBOUND. The first section is identical to the first section of the other computational astrophysics labwork "Chaos in the planetary system".

If you have no or scarce experience in programming and computer science, we advise you to favour this labwork. You will start with the installation of a scientific python programming environment on your Laptop/PC/Workstation, e.g., anaconda[1], followed by the installation of REBOUND and with a little effort, you will be able to solve the exercises from section 4.

You need a computer running either Linux or macOS or Windows 10 with the Windows Subsystem for Linux to run REBOUND. We strongly suggest to use the python interface. Make sure to install REBOUND before the labwork class. If you face any problems during the installation, please mail to Christoph Schaefer (ch.schaefer@uni-tuebingen.de).

Introduction

The classical N-body problem in Astrophysics is important to understand the evolution and stability of planetary systems, star clusters and galactic nuclei. The dynamics of these many body systems is dominated by pairwise gravitational interactions between single bodies. Hence, this two-body interaction has to be accounted with high accuracy in the numerical treatment.
Typical numbers of the aforementioned systems are ~ 10 for planetary systems, 10^4 to 10^6 for star clusters and more than over 10^8 for galactic nuclei.
For particle numbers of this magnitude, the methods of statistical mechanics can only be applied partially, and one has to rely on the direct integration of each individual particle trajectory with regard of the mutual gravitational forces.
In this practical course, we will use the existing REBOUND software package to address several small N-body problems. In the first section, the classical N-body problem and the numerical methods to solve it are presented. In the second section, the software package REBOUND and its installation is described. The third section contains the exercises.

1 The Classical N-Body-Problem

The motion of N point masses in their mutual gravitational field is the classical N-body problem. Each particle i with mass m_i has the location $\mathbf{r}_i$ and the velocity $\mathbf{v}_i$ at time t. The Hamiltonian of this system reads

$$H = \sum_{i=1}^{N} \frac{{\mathbf{p}_i}^2}{2m_i} - \sum_{i=1}^{N} \sum_{j=i+1}^{N} \frac{G m_i m_j}{|\mathbf{q}_i - \mathbf{q}_j|} \tag{1}$$

[1] https://www.anaconda.com/distribution/

with the canonical coordinates momentum $\mathbf{p}_i = m_i \mathbf{v}_i$ and location $\mathbf{q}_i = \mathbf{r}_i$ for all $i = 1 \dots N$ point masses. The Hamiltonian Equations yield the equations of motion for particle i

$$\dot{\mathbf{q}}_i(t) = \frac{\partial H}{\partial \mathbf{p}_i} \qquad \Longrightarrow \qquad \frac{\mathrm{d}\mathbf{r}_i}{\mathrm{d}t} = \mathbf{v}_i \tag{2}$$

$$\dot{\mathbf{p}}_i(t) = -\frac{\partial H}{\partial \mathbf{q}_i} \qquad \Longrightarrow \qquad \frac{\mathrm{d}\mathbf{v}_i}{\mathrm{d}t} = \mathbf{a}_i \tag{3}$$

with the acceleration

$$\mathbf{a}_i(t) = \sum_{j \neq i}^{N} G m_j \frac{\mathbf{r}_{ij}}{r_{ij}^3} \quad . \tag{4}$$

Later, we will also need the time derivative of the acceleration, the so-called jerk

$$\dot{\mathbf{a}}_i(t) = \sum_{j \neq i}^{N} G m_j \left(\frac{\mathbf{v}_{ij}}{r_{ij}^3} - \frac{3(\mathbf{v}_{ij} \cdot \mathbf{r}_{ij})}{r_{ij}^5} \mathbf{r}_{ij} \right) \tag{5}$$

where $\mathbf{r}_{ij} := \mathbf{r}_j(t) - \mathbf{r}_i(t)$, $r_{ij} := |\mathbf{r}_{ij}|$, $\mathbf{v}_{ij} := \mathbf{v}_j(t) - \mathbf{v}_i(t)$, $v_{ij} := |\mathbf{v}_{ij}|$.
To calculate all accelerations at a certain time, one needs to evaluate $N(N-1)/2$ terms (using symmetry), that is the computing time for large numbers of N is asymptotically $\sim N^2$.

The Two-Body Problem

The Hamiltonian (1) of the two-body problem can be separated into the motion of the center of mass and the relative motion, that is $H = H(\mathbf{r}_1, \mathbf{r}_2, \mathbf{p}_1, \mathbf{p}_2) \rightarrow H(\mathbf{r}, \mathbf{p}, \mathbf{p}_{cm})$ with

$$H = H_{\mathrm{cm}} + H_{\mathrm{rel}} = \frac{\mathbf{p}_{\mathrm{cm}}^2}{2M} + \frac{\mathbf{p}^2}{2\mu} - \frac{G m_1 m_2}{r} \tag{6}$$

with the total mass $M := m_1 + m_2$, the reduced mass $\mu := m_1 m_2 / M$, the distance vector between the two bodies $\mathbf{r} := \mathbf{r}_1 - \mathbf{r}_2$, the distance $r := |\mathbf{r}|$, the relative momentum $\mathbf{p} := \mathbf{p}_1 - \mathbf{p}_2$ and the momentum of the center of mass $\mathbf{p}_{\mathrm{cm}}$.
The Hamiltonian of the center of mass motion H_{cm} (the first term on the right hand side of equation (6)) has trivial solutions. This follows from the face, that H_{cm} depends only on the momemtum of the center of mass $\mathbf{p}_{\mathrm{cm}}$ and not from the conjugate variable $\mathbf{r}_{\mathrm{cm}}$, the location of the center of mass. It follows a steady motion of the center of mass.
The Hamiltonian of the relative motion H_{rel} yields the equation of motion of the classical two-body problem

$$\dot{\mathbf{p}} = \mu \dot{\mathbf{v}} = -\frac{G m_1 m_2}{r^3} \mathbf{r} \tag{7}$$

with $\mathbf{v} = \mathbf{v}_1 - \mathbf{v}_2 = \mathbf{p}/\mu$, and finally in the familiar notation

$$\ddot{\mathbf{r}} = \dot{\mathbf{v}} = -\frac{GM}{r^3} \mathbf{r} \quad . \tag{8}$$

The solution of the Kepler-problem are conic sections with the orbital plane, the motion of $\mathbf{r}$ is characterised by an ellipse, a parabola or a hyperbola. The energy E, the specific angular momentum

$$\mathbf{j} = \mathbf{r} \times \mathbf{v} \tag{9}$$

and the Runge-Lenz vector

$$\mathbf{e} = \frac{\mathbf{v} \times \mathbf{j}}{GM} - \frac{\mathbf{r}}{r} \tag{10}$$

are conserved values of the two-body motion. Using eq. (8), we find for the evolution of the angular momentum

$$\frac{\mathrm{d}\mathbf{j}}{\mathrm{d}t} = \mathbf{v} \times \mathbf{v} + \mathbf{r} \times \dot{\mathbf{v}} = -\frac{GM}{r^3}(\mathbf{r} \times \mathbf{r}) = 0 \quad . \tag{11}$$

To prove the conservation of $\mathbf{e}$, we need some calculus with the use of the vector identity $(\mathbf{A} \times \mathbf{B}) \times \mathbf{C} = \mathbf{B}(\mathbf{A}\cdot\mathbf{C}) - \mathbf{A}(\mathbf{B}\cdot\mathbf{C})$. It follows

$$\frac{\mathbf{j} \times \mathbf{r}}{r^3} = \frac{(\mathbf{r} \times \mathbf{v}) \times \mathbf{r}}{r^3} = \frac{\mathbf{v}}{r} - \mathbf{r}\frac{(\mathbf{r}\cdot\mathbf{v})}{r^3} = \frac{\mathrm{d}}{\mathrm{d}t}\left(\frac{\mathbf{r}}{r}\right) \quad . \tag{12}$$

With this term and the help of eq. (8) and (11), we find

$$\frac{\mathrm{d}\mathbf{e}}{\mathrm{d}t} = -\frac{GM}{r^3}\frac{\mathbf{r} \times \mathbf{j}}{GM} - \frac{\mathrm{d}}{\mathrm{d}t}\left(\frac{\mathbf{r}}{r}\right) = \frac{\mathbf{j} \times \mathbf{r}}{r^3} - \frac{\mathrm{d}}{\mathrm{d}t}\left(\frac{\mathbf{r}}{r}\right) = 0 \quad . \tag{13}$$

Additionally, we regard the following term

$$\mathbf{r}\cdot\mathbf{e} + r = \frac{\mathbf{r}\cdot(\mathbf{v} \times \mathbf{j})}{GM} = \frac{(\mathbf{r} \times \mathbf{v})\cdot\mathbf{j}}{GM} = \frac{j^2}{GM} \quad . \tag{14}$$

The scalar product $\mathbf{r}\cdot\mathbf{e}$ is expressed with the angle $\phi - \phi_0$ between $\mathbf{e}$ and the location vector $\mathbf{r}$, and $re\cos(\phi - \phi_0) + r = j^2/(GM)$, it follows

$$r(\phi) = \frac{j^2/(GM)}{1 + e\cos(\phi - \phi_0)}. \tag{15}$$

This is the well-known equation of a conic section. For a constrained motion, the absolute value of the Runge-Lenz vector $e = |\mathbf{e}|$ specifies the eccentricity of the ellipse. Then, the maximum and minimum distance of the two point masses is given by

$$r_{\mathrm{max/min}} = \frac{j^2/(GM)}{1 \pm e}, \tag{16}$$

the semi-major axis of the ellipse is given by

$$a = \frac{1}{2}(r_{\mathrm{min}} + r_{\mathrm{max}}) = \frac{j^2/(GM)}{1 - e^2}, \tag{17}$$

and the semi-minor axis is given by the geometric mean

$$b = \sqrt{r_{\mathrm{min}} r_{\mathrm{max}}}. \tag{18}$$

Because of Kepler's third law ("The square of the orbital period of a planet is directly proportional to the cube of the semi-major axis of its orbit."), it follows for the orbital period

$$\omega = \sqrt{\frac{GM}{a^3}}. \tag{19}$$

Please see fig. 1 for an illustrative sketch of the properties of a bound Kepler orbit.

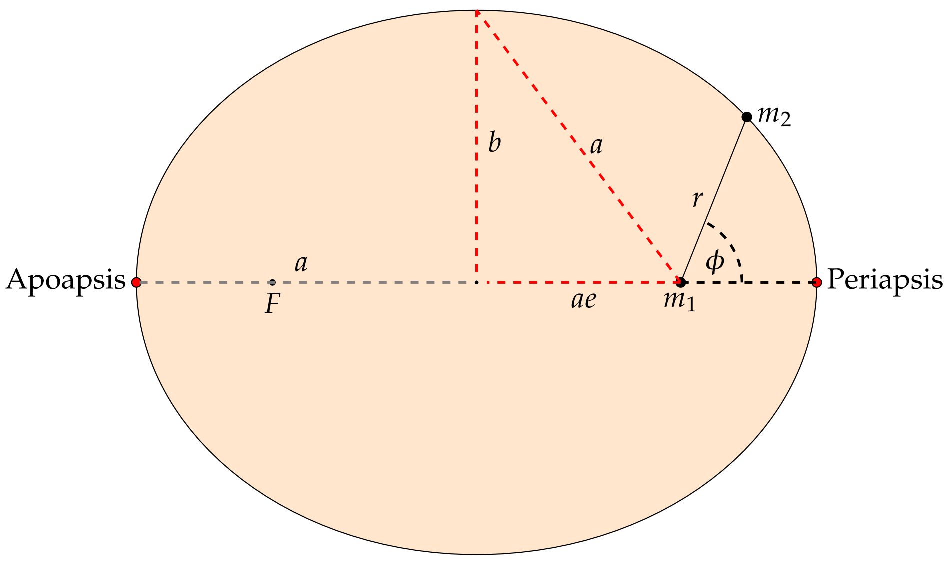


Figure 1: Bound Two-Body orbit. The mass m_1 is in the focus of the ellipse with semi-major and semin-minor axes a and b, and eccentricity e. The distance of m_1 to apoapsis is $r_{\max}$ and to periapsis $r_{\min}$. The equation 15 denotes the orbit with the choice of the coordinate system so that $\phi_0 = 0$.

2 Numerical Solution of the Equation of Motion: Time integrators

In order to calculate the trajectories of the point masses, one has to solve the corresponding equation of motion for each particle i, whic is the system of ODEs given by eqs. (2) and (3) with appropriate initial values for locations and velocites. In general, this is only possible by the help of numerical methods, which leads to the fact, that the particle distribution can only be determined at discrete time values. Starting from the current point in time $t = t_n$, the location and the velocity at a later point in time $t_{n+1} = t_n + \Delta t$ are calculated. Then, the new point in time becomes the current one and the algorithm restarts at the new time.
In the following, different numerical schemes will be presented which can be used to solve a system of ODEs of the form $\mathrm{d}y/\mathrm{d}t = f(y, t)$ with initial values. These schemes are called time integrators for obvious reasons in this context. They can, however, be applied for any initial value problem. The indices in the following refer always to the equivalent point in time, that is $r_n = r(t_n)$, and so forth. The index i for the different point masses will be left in the following.

2.1 Single-Step Methods

Basic idea: Consider the differentials $\mathrm{d}y$ and $\mathrm{d}t$ as finite intervals Δy and Δt

$$\frac{\mathrm{d}y}{\mathrm{d}t} \Rightarrow \frac{\Delta y}{\Delta t} = f(t, y)$$

and discretise with $\Delta t = h$

$$\Delta y = y_{n+1} - y_n = \Delta t f = hf.$$

Generally, single-step methods can be written in the following form

$$y_{n+1} = y_n + h\Phi(t_n, y_n, y_{n+1}, h),$$

where Φ is called the increment function. The scheme is called explicit if $\Phi = \Phi(t_n, y_n, h)$ and implicit for $\Phi = \Phi(t_n, y_n, y_{n+1}, h)$. The step is from t_n to t_{n+1} and from y_n to y_{n+1}.
Single-step methods may be constructed by Taylor expansion. At first, we will discuss the simple explicit one-step method, the explicit Euler-method with $\Phi(t_n, y_n, h) = f(t_n, y_n)$.

2.2 Simple time integrators: The Euler-method and the Euler-Cromer-method

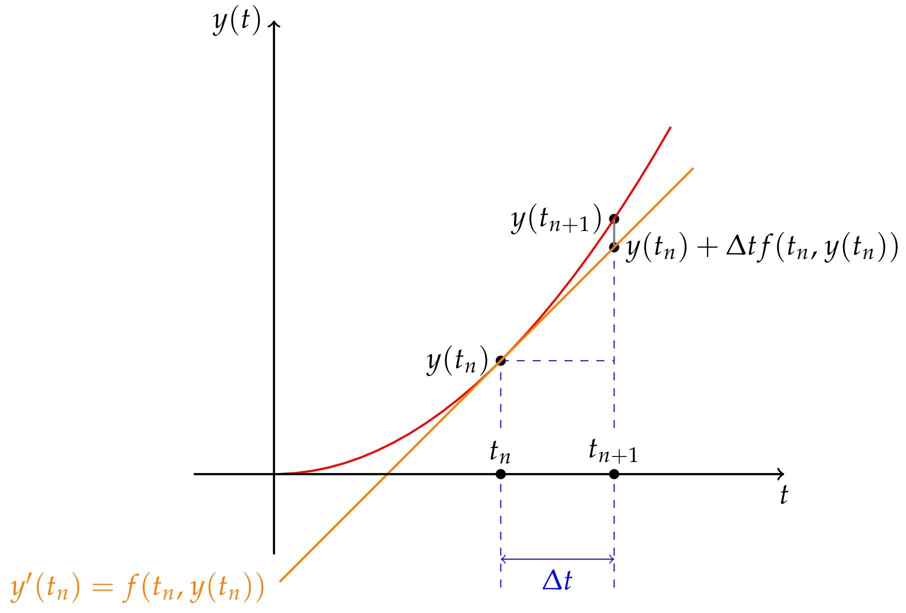
The simplest way for a discrete time integration scheme is achieved with the help of the Taylor expansion of location and velocity

$$\begin{aligned} v(t_n + \Delta t) &= v(t_n) + \frac{\mathrm{d}v}{\mathrm{d}t}(t_n)\Delta t + \mathcal{O}(\Delta t^2), && (20)\\ &= v(t_n) + a(t_n)\Delta t + \mathcal{O}(\Delta t^2), && (21)\\ r(t_n + \Delta t) &= r(t_n) + \frac{\mathrm{d}r}{\mathrm{d}t}(t_n)\Delta t + \mathcal{O}(\Delta t^2), && (22)\\ &= r(t_n) + v(t_n)\Delta t + \mathcal{O}(\Delta t^2), && (23) \end{aligned}$$

where the acceleration a is given by eq. (4).
These equations directly yield an algorithm, the so-called Euler-method, where one calculates

$$\begin{aligned} v_{n+1} &= v_n + a_n\Delta t, && (24)\\ r_{n+1} &= r_n + v_n\Delta t && (25) \end{aligned}$$

for every time step.
The Euler-method uses the simple slope triangle to perform one forward step


$y(t)$
t
$y(t_{n+1})$
$y(t_n) + \Delta t f(t_n, y(t_n))$
$y(t_n)$
t_n
t_{n+1}
Δt
$y'(t_n) = f(t_n, y(t_n))$

For each point in time t_n, the evolution of the function $y(t)$ from time t_n to $t_{n+1} = t_n + \Delta t$ is approximated by $\Delta t y'(t_n) = \Delta t f(t_n, y(t_n))$. The discretization error is given by the difference $y(t_{n+1}) - [y(t_n) + \Delta t f(t_n, y(t_n))]$.
As an alternative to the Euler-method where the velocity is calculated first, there is the so-called Euler-Cromer-method. Here, the new locations are calculated with the velocities at the new time step t_{n+1} instead of t_n like in the standard Euler-method.

$$v_{n+1} = v_n + a_n \Delta t, \tag{26}$$
$$r_{n+1} = r_n + v_{n+1} \Delta t. \tag{27}$$

Averaging both methods yields the scheme

$$v_{n+1} = v_n + a_n \Delta t, \tag{28}$$
$$r_{n+1} = r_n + \frac{1}{2}\left(v_n + v_{n+1}\right) \Delta t. \tag{29}$$

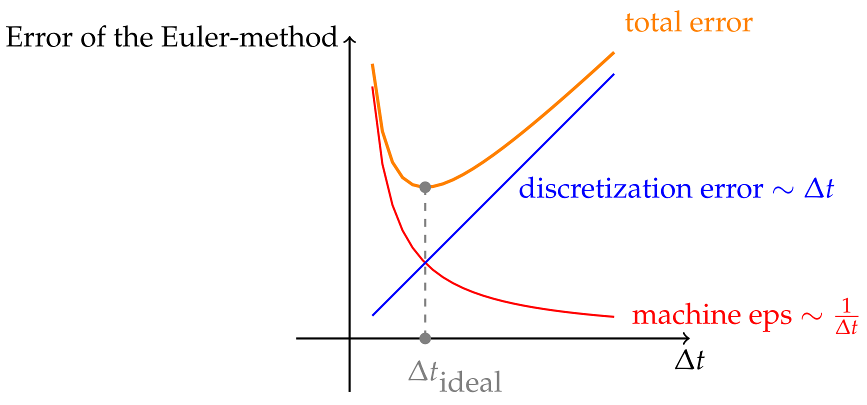
2.2.1 Accuracy of the scheme

The main disadvantage of these methods is the low accuracy. If the trajectories of particles have to be determined until the space in time $t_{\max}$ and each time step is of length Δt, we need $N_t := t_{\max}/\Delta t$ time steps. The error per time step is of the order $O(\Delta t^2)$ (see Taylor expansion), leading to a total error of the simulation of the order $O(N_t \Delta t^2) = O(\Delta t)$. And the computational accuracy was not even accounted, yet.

2.2.2 Computational accuracy

Let ϵ be the machine accuracy, the relative error due to rounding in floating point arithmetic. The error caused by ϵ grows with $O(N_t \epsilon)$. In order to improve the accuracy of the numerical scheme, one can choose smaller time steps Δt, which consequently yields a higher error

by ϵ, since we need a higher number of time steps N_t. Hence, global values of a N-body simulation like the total energy E have an error of the order $O(N_t N \epsilon)$.
Some numbers for clarification: Depending on the numerical scheme, approximately 100 to 1000 steps per orbit are required for the two-body problem to achieve an error for the energy in the order of the machine accuracy, that is $\Delta E/E \sim 10^{-13}$. The error in the energy adds up to $\Delta E/E \sim 10^{-9}$ for a complete orbit. To simulate the evolution of a star cluster with about 10^6 stars and for the time of about 10^4 typical orbital periods, one has to evaluate 10^{10} orbits with 1000 time steps each, in total 10^{13} time steps, eventually leading to an error of the total energy of $\Delta E \sim 100$ %.



2.3 The Leap-Frog-method

One improvement of the Euler-method is to use the velocity for the calculation of the new locations at the point in time $t_{n+1/2} = t_n + \Delta t/2$ and not on the times t_n or t_{n+1}. The scheme of this method is

$$\begin{aligned}
&\text{First step:} \\
&\qquad \mathbf{r}_{1/2} = \mathbf{r}_0 + \mathbf{v}_0 \frac{\Delta t}{2} \quad , \\
&\qquad \text{Calculate } \mathbf{a}_{1/2} = \mathbf{a}(t_{1/2}, \mathbf{r}_{1/2}) \quad , \\
&\text{Regular steps:} \\
&\qquad \mathbf{v}_{n+1} = \mathbf{v}_n + \mathbf{a}_{n+1/2}\, \Delta t \quad , \\
&\qquad \mathbf{r}_{n+3/2} = \mathbf{r}_{n+1/2} + \mathbf{v}_{n+1}\, \Delta t \quad , \\
&\text{Last step:} \\
&\qquad \mathbf{r}_{n+1} = \mathbf{r}_{n+1/2} + \mathbf{v}_{n+1} \frac{\Delta t}{2} \quad .
\end{aligned}$$

The name of the this scheme “Leap Frog” originates from the fact that the positions and velocities are updated at interleaved time points separated by $\Delta t/2$, staggered in such a way, that they leapfrog over each other. Hence, locations and velocities are not known at the same point in time In the limit of $\Delta t \to 0$, one gets the usual canonical equations. Moreover, for finite Δt, the scheme is symplectic, which means, there is a Hamiltonian $\hat{H} = H + \Delta t^2 H_2 + \Delta t^4 H_4 + \ldots$, which is solved exactly by the scheme. This has a positive impact on the consistency of conserved quantities.

2.4 The Verlet-method

The Leap-Frog-method can be written as the so-called kick-drift-kick algorithm

$$\tilde{\mathbf{v}}(\Delta t) = \mathbf{v}(t) + \frac{1}{2}\Delta t\, \mathbf{a}(t) \tag{30}$$

$$\mathbf{r}(t+\Delta t) = \mathbf{r}(t) + \Delta t\, \tilde{\mathbf{v}}(t) \tag{31}$$

$$\mathbf{v}(t+\Delta t) = \tilde{\mathbf{v}}(t) + \frac{1}{2}\Delta t\, \mathbf{a}(t+\Delta t) \tag{32}$$

The intermediate value $\tilde{\mathbf{v}}(\Delta t)$ denotes the velocity at time $t + 1/2\Delta t$.
The advantage of this velocity Verlet-method is the higher accuracy in the calculation of the locations. The Verlet-method is identical to the Leap-frog-method. Note, that also in this method, only one evaluation of the force term is required per time step.

2.5 Runge-Kutta-Integrators

The error of one single step in the Euler method can be analyzed by looking at the Taylor expansion. The Taylor expansion of $y(t+h)$ at t yields with $\frac{\mathrm{d}y}{\mathrm{d}t} = f(y,t)$

$$y(t+h) = y(t) + h\frac{\mathrm{d}y}{\mathrm{d}t} + \mathcal{O}(h^2) \tag{33}$$

$$= y(t) + hf(y,t) + \mathcal{O}(h^2) \tag{34}$$

Hence, we could get better accuracy if we also include higher terms.

2.5.1 A 2nd order RK scheme

Now, let's expand up to third order

$$y(t+h) = y(t) + h\frac{\mathrm{d}y}{\mathrm{d}t} + \frac{h^2}{2}\frac{\mathrm{d}^2y}{\mathrm{d}t^2} + \mathcal{O}(h^3) \tag{35}$$

$$= y(t) + hf(y,t) + \frac{h^2}{2}\frac{\mathrm{d}f}{\mathrm{d}t} + \mathcal{O}(h^3) \tag{36}$$

The derivative of f with respect to t is given by

$$\frac{\mathrm{d}f(y(t),t)}{\mathrm{d}t} = \frac{f(y(t+h),t+h) - f(y(t),t)}{h} + \mathcal{O}(h). \tag{37}$$

Unfortunately, we do not know $y(t+h)$. By using the Euler scheme to estimate $y(t+h)$, we get

$$\frac{\mathrm{d}f(y(t),t)}{\mathrm{d}t} = \frac{f(y(t)+hf(y(t),t),t+h) - f(y(t),t)}{h} + \mathcal{O}(h). \tag{38}$$

Inserting eq. (38) into (36) yields

$$y(t+h) = y(t) + hf(y(t),t) + \frac{h}{2}\left\{f(y(t)+hf(y(t),t),t+h) - f(y(t),t)\right\} + \mathcal{O}(h^3) \tag{39}$$

$$= y(t) + \frac{h}{2}\left\{f(y(t),t) + f(y(t)+hf(y(t),t),t+h)\right\} + \mathcal{O}(h^3). \tag{40}$$

We have improved the accuracy for the cost of an additional calculation of f at substep $y(t) + hf(y,t)$. This is the basic idea of any Runge-Kutta method. In fact, we have just derived a RK 2nd order scheme, the Heun method.
However, often RK integrators are confused with Taylor integration schemes. The basic form of Taylor integrators are based on the Taylor expansion of the solution function $y(t)$. One obtains the value of the solution $y(t)$ at the time $t+h$ by

$$y(t+h) = y(t) + hf(t,y(t)) + \frac{h^2}{2!}f'(t,y(t)) + \ldots + \frac{h^2}{n!}f^{(n-1)}(t,y(t)). \tag{41}$$

For this, the higher order derivatives of $y(t)$ have to be known. These are not required for the RK integration schemes, where only the first derivative of $y(t)$, $f(t,y)$, is needed. The basic idea of a RK integrator is based on the transformation of the ODE into an integral equation

$$y' = f(t,y) \Leftrightarrow \int_t^{t+h} \frac{\mathrm{d}y}{\mathrm{d}t}\mathrm{d}t = \int_t^{t+h} f(t,y)\mathrm{d}t, \tag{42}$$

$$y(t+h) = y(t) + \int_t^{t+h} f(t,y)\mathrm{d}t, \tag{43}$$

and then find an approximation to the integral on the right hand side, e.g. the trapezoidal rule (cf. above)

$$\int_t^{t+h} f(t,y)\mathrm{d}t = \frac{h}{2}\left[f(t,y(t)) + f(t+h,y(t+h))\right] \tag{44}$$

with the problem that we do not know $y(t+h)$ and we have to approximate it with $y(t+h) = y(t) + hf(t,y(t))$.
Finally, the general form of an explicit RK method is

$$y_{n+1} = y_n + h\sum_i b_i k_i, \tag{45}$$

where the k_i denote the substeps and b_i are some weights. Please see a textbook for further details. Here, we provide only the terms for the RK4 method.

2.5.2 The classical Runge-Kutta scheme

A system of first order ODEs can be written in vector notation

$$\mathbf{y}' = \mathbf{f}(t,\mathbf{y}(t)), \quad \mathbf{y}(t_0) = \mathbf{y}_0 \tag{46}$$

with the vector for the initial values $\mathbf{y}_0$. In our N-body problem, $\mathbf{y}$ will be $(\mathbf{r},\mathbf{v})$ with initial values $(\mathbf{r}(0),\mathbf{v}(0))$. A very common integration scheme is the so-called 4th order Runge-Kutta-method (RK4). In this scheme, estimated values for the derivative at single substeps in the interval t_n, t_n+h are calculated. In vector notation, it reads

$$\mathbf{K}_1 = \mathbf{f}(t_n,\mathbf{y}_n) \tag{47}$$

$$\mathbf{K}_2 = \mathbf{f}\left(t_n + \frac{1}{2}h, \mathbf{y}_n + \frac{h}{2}\mathbf{K}_1\right) \tag{48}$$

$$\mathbf{K}_3 = \mathbf{f}\left(t_n + \frac{1}{2}h, \mathbf{y}_n + \frac{h}{2}\mathbf{K}_2\right) \tag{49}$$

$$\mathbf{K}_4 = \mathbf{f}\left(t_{n+1}, \mathbf{y}_n + h\mathbf{K}_3\right) \tag{50}$$

where $\mathbf{y}_n, \mathbf{y}_{n+1}, \mathbf{K}_1, \mathbf{K}_2, \mathbf{K}_3, \mathbf{K}_4$ and $\mathbf{f}$ are vectors in the vector space R^m. The value of the function $\mathbf{y}$ at the new time is calculated by

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \frac{h}{6}\left(\mathbf{K}_1 + 2\mathbf{K}_2 + 2\mathbf{K}_3 + \mathbf{K}_4\right). \tag{51}$$

The error of the scheme is of the order $\mathcal{O}(h^5)$. The price for this high accuracy is the required calculation of the function which is four times per time step. However, the stability is also improved compared to the Euler-method. For a more detailed description of the RK4-method, we refer to the book "Numerical Recipes"[2] and wikipedia. We will apply the RK4-method in this course on the system of equations (2) and (3).

2.5.3 Special integrators for N-Body problems

Although the presented integrators from the previous sections may yield profound and solid results for ODEs, special integrators for the N-Body problem have been developed in the recent years. The N-Body code that you will use in this labwork features several symplectic integrators (WHFast, WHFastHelio, SEI, LEAPFROG) and a high accuracy non-symplectic integrator with adaptive timestepping (IAS15). These integrators are described nicely in the following publications by Hanno Rein

1. Rein & Liu 2012 (Astronomy and Astrophysics, Volume 537, A128) describe the code structure and the main feature including the gravity and collision routines for many particle systems. http://adsabs.harvard.edu/abs/2012A%26A...537A.128R

2. Rein & Spiegel 2015 (Monthly Notices of the Royal Astronomical Society, Volume 446, Issue 2, p.1424-1437) describe the versatile high order integrator IAS15 which is now part of REBOUND. http://adsabs.harvard.edu/abs/2015MNRAS.446.1424R

3. Rein & Tamayo 2015 (Monthly Notices of the Royal Astronomical Society, Volume 452, Issue 1, p.376-388) describe WHFast, the fast and unbiased implementation of a symplectic Wisdom-Holman integrator for long term gravitational simulations. http://adsabs.harvard.edu/abs/2015MNRAS.452..376R

4. Rein & Tamayo 2016 (Monthly Notices of the Royal Astronomical Society, Volume 459, Issue 3, p.2275-2285) develop the framework for second order variational equations. http://arxiv.org/abs/1603.03424

5. Rein & Tamayo 2017 (Monthly Notices of the Royal Astronomical Society, Volume 467, Issue 2, p.2377-2383) describes the Simulation Archive for exact reproducibility of N-body simulations. https://arxiv.org/abs/1701.07423

6. Rein & Tamayo 2018 (Monthly Notices of the Royal Astronomical Society, Volume 473, Issue 3, p.3351–3357) describes the integer based JANUS integrator. https://arxiv.org/abs/1704.07715

7. Rein, Hernandez, Tamayo, Brown, Eckels, Holmes, Lau, Leblanc & Silburt 2019 (Monthly Notices of the Royal Astronomical Society, Volume 485, Issue 4, p.5490-5497) describes the hybrid symplectic intergator MERCURIUS. https://arxiv.org/abs/1903.04972

[2]http://apps.nrbook.com/c/index.html, p.710

8. Rein, Tamayo & Brown 2019 (submitted) describes the implementation of the high order symplectic intergators SABA, SABAC, SABACL, WHCKL, WHCKM, and WHCKC. https://arxiv.org/abs/1907.11335

3 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[3]. 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.** Note that the API of REBOUND changed slightly with major release 4.0 (e.g., `sim.save` became `sim.save_to_file`, etcpp.). You may find some errors related to this change in one of the examples below.

3.1 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, which might be outdated. Please consult the official documentation about the installation if you face any problems, installation on windows is now also officially supported, see https://rebound.readthedocs.io/en/latest/quickstart_installation/.

3.1.1 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
```

[3] http://www.gnu.org/licenses/

```
<rebound.simulation.Simulation object at 0x103a07540, N=0, t=0.0>
>>>
```

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

3.1.2 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 successful

```
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
```

3.2 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 quick-start guide.

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

4 Exercises

In the following subsections you find the exercises that you have to solve during this labwork. 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.
There are also a lot of code listings below. Please note that copy and paste from the pdf does not always work correctly and will give you encoding errors for special characters. Always look at the very first error that the compiler or interpreter lists.

4.1 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

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

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_simlation_energy(struct reb_simulation *)` or in Python `sim.energy()`.

4.1.1 Solution with 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_simulation_create();

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

 // 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_simulation_move_to_com(r);

// now integrate for 1000 orbits, 2pi is one orbit
reb_simulation_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).

4.1.2 Solution with 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.energy()
    for j in range(sim.N):
        x[j,i] = sim.particles[j].x
        y[j,i] = sim.particles[j].y
print("Done, now saving and plotting...")
# it's best to store the information to a file if we
# want to continue it later
sim.save_to_file("my_sim.bin")
# 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. See the documentation on how to use fixed timestepping for the different integrators: For `ias15`, you will need something like

```
sim.ri_ias15.min_dt = 1e-3
sim.ri_ias15.epsilon = 0.0
```

in your Python code to turn off the adaptive timestep and set the time step to 10^{-3}.

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

This problem is equivalent to one of the problems in the CHAOS experiment and 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. 2 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. 2.
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

$$\Delta_c \simeq 2.40\,(\mu_2 + \mu_3)^{1/3}. \tag{52}$$

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 10^4 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

$$R_{in} \approx a\sqrt[3]{\mu/3}, \tag{53}$$

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[4]. 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
```

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

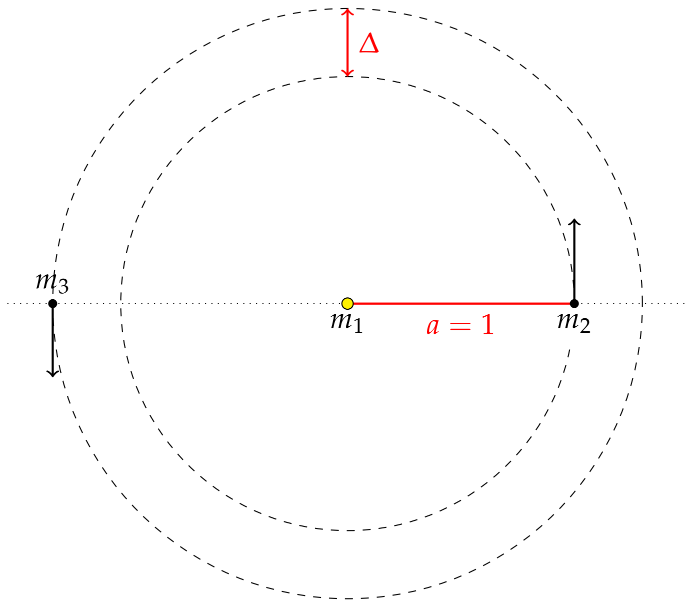


Figure 2: Schematic representation of a three body problem, where two smaller test masses m_2 and m_3 orbit a larger central mass m_1, it holds $m_2 + m_3 \ll m_1$. The initial semi-major axis of mass m_2 is normed to 1 in respect to the central mass m_1, and the semi-major axis of m_3 to $1 + \Delta$. The initial locations of the smaller bodies are opposed, $\delta\phi = \pi$, (after Gladman, 1993).

```python
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()
```

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!

4.3 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

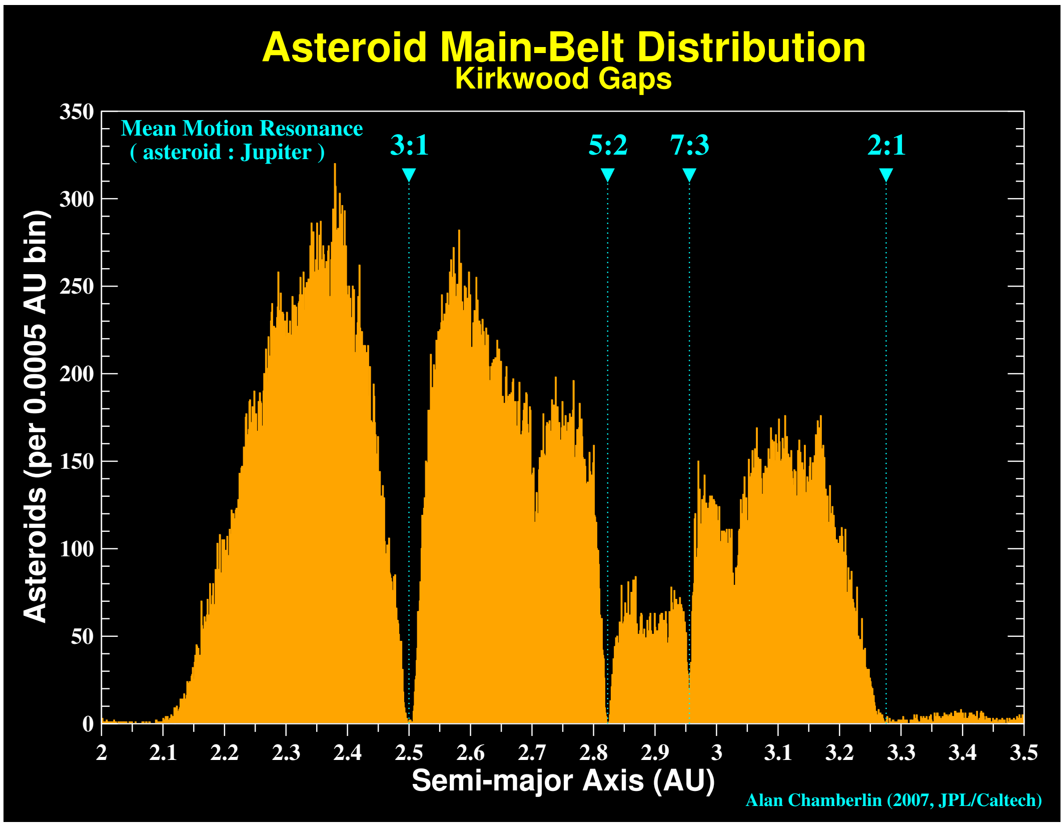


Figure 3: Kirkwood gaps in the asteroid belt (Chamberlin 2007).

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 3).
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?

4.3.1 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_simulation_add_fmt(r, "a e", a, e);
    }
```

4.3.2 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
```

4.4 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[5]. A resonance can either stabilize or de-stabilize the system. In the example

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

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?

4.4.1 Solution with C

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

```c
int main(int argc, char* argv[]) {
    struct reb_simulation* sim = reb_simulation_create();
    // 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_simulation_add(sim,p1);
    reb_simulation_add(sim,p2);
    reb_simulation_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_simulation_integrate(sim, tmax);
    rebx_free(rebx);    // Free all the memory allocated by rebx
    reb_simulation_free(sim);
}
```

4.4.2 Solution with Python

For this example, you must have installed the reboundx module for 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=3e-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
```

Hints for the experiment protocol

During the course of this experiment and especially for the protocol, you will produce a lot of data that you want to visualize and present. Before you add an enormous number of figures or data plots to your protocol, first think about what you really have to plot (see the exercises) and how you can combine several plots into one figure or one figure with several subfigures. For example, when you plot the time evolution of the energy for different time integrators and different time step sizes, you can essentially create one figure with different panels. **I will not accept protocols with more than 30 pages.**
Although a nice diagram and a figure speaks more than one million words, please add your results and findings in text form also to your protocol. Additionally, keep in mind that the protocol should be readable and understandable without the experiment instructions! Write down your initial conditions, explain what you do and why you do it. Do not copy and paste from the instructions, write your own words in own sentences. Do not add code listings to the protocol, send your source files separately, including instructions on how to run your code.
Moreover, grammar does not really matter as long as you get the formulae right in Physics, but you mihgt agrea that it impruves the readbillitty. No one wants to read a protocol with

a lot of typos.
See the guidelines of the labwork for more information.

References

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

Happy Coding!