Astronomy & Astrophysics

REBOUND: an open-source multi-purpose *N*-body code for collisional dynamics

H. Rein[1] and S.-F. Liu[2,3]

[1] Institute for Advanced Study, 1 Einstein Drive, Princeton, NJ 08540, USA
   e-mail: rein@ias.edu
[2] Kavli Institute for Astronomy and Astrophysics, Peking University, 100871 Beijing, PR China
[3] Department of Astronomy, Peking University, 100871 Beijing, PR China
   e-mail: liushangfei@pku.edu.cn

ABSTRACT

REBOUND is a new multi-purpose *N*-body code which is freely available under an open-source license. It was designed for collisional dynamics such as planetary rings but can also solve the classical *N*-body problem. It is highly modular and can be customized easily to work on a wide variety of different problems in astrophysics and beyond.

REBOUND comes with three symplectic integrators: leap-frog, the symplectic epicycle integrator (SEI) and a Wisdom-Holman mapping (WH). It supports open, periodic and shearing-sheet boundary conditions. REBOUND can use a Barnes-Hut tree to calculate both self-gravity and collisions. These modules are fully parallelized with MPI as well as OpenMP. The former makes use of a static domain decomposition and a distributed essential tree. Two new collision detection modules based on a plane-sweep algorithm are also implemented. The performance of the plane-sweep algorithm is superior to a tree code for simulations in which one dimension is much longer than the other two and in simulations which are quasi-two dimensional with less than one million particles.

In this work, we discuss the different algorithms implemented in REBOUND, the philosophy behind the code's structure as well as implementation specific details of the different modules. We present results of accuracy and scaling tests which show that the code can run efficiently on both desktop machines and large computing clusters.

Key words. methods: numerical – planets and satellites: rings – protoplanetary disks

1. Introduction

REBOUND is a new open-source collisional *N*-body code. This code, and precursors of it, have already been used in wide variety of publications (Rein & Papaloizou 2010; Crida et al. 2010; Rein et al. 2010; Rein & Liu, in prep.; Rein & Latter, in prep.). We believe that REBOUND can be of great use for many different problems and have a wide reach in astrophysics and other disciplines. To our knowledge, there is currently no publicly available code for collisional dynamics capable of solving the problems described in this paper. This is why we decided to make it freely available under the open-source license GPLv3[1].

Collisional *N*-body simulations are extensively used in astrophysics. A classical application are planetary rings (see e.g. Wisdom & Tremaine 1988; Salo 1991; Richardson 1994; Lewis & Stewart 2009; Rein & Papaloizou 2010; Michikoshi & Kokubo 2011, and references therein) which have often a collision time-scale that is much shorter than or at least comparable to an orbital time-scale. Self-gravity plays an important role, especially in the dense parts of Saturn's rings (Schmidt et al. 2009). These simulations are usually done in the shearing sheet approximation (Hill 1878).

Collisions are also important during planetesimal formation (Johansen et al. 2007; Rein et al. 2010; Johansen et al., in prep.). Collisions provide the dissipative mechanism to form a planetesimal out of a gravitationally bound swarm of boulders.

REBOUND can also be used with little modification in situations where only a statistical measure of the collision frequency is required such as in transitional and debris discs. In such systems, individual collisions between particles are not modeled exactly, but approximated by the use of super-particles (Stark & Kuchner 2009; Lithwick & Chiang 2007).

Furthermore, REBOUND can be used to simulate classical *N*-body problems involving entirely collision-less systems. A symplectic and mixed variable integrator can be used to follow the trajectories of both test-particles and massive particles.

We describe the general structure of the code, how to obtain, compile and run it in Sect. 2. The time-stepping scheme and our implementation of symplectic integrators are described in Sect. 3. The modules for gravity are described in Sect. 4. The algorithms for collision detection are discussed in Sect. 5. In Sect. 6, we present results of accuracy tests for different modules. We discuss the efficiency of the parallelization with the help of scaling tests in Sect. 7. We finally summarize in Sect. 8.

2. Overview of the code structure

REBOUND is written entirely in C and conforms to the ISO C99 standard. It compiles and runs on any modern computer platform which supports the POSIX standard such as Linux, Unix and

---

[1] The full license is distributed together with REBOUND. It can also be downloaded from http://www.gnu.org/licenses/gpl.html.

Mac OSX. In its simplest form, REBOUND requires no external libraries to compile.

Users are encouraged to install the OpenGL and GLUT libraries which enable real-time and interactive 3D visualizations. LIBPNG is required to automatically save screen-shots. The code uses OpenMP for parallelization on shared memory systems. Support for OpenMP is built-in to modern compilers and requires no libraries (for example gcc ≥4.2). An MPI library must be installed for parallelization on distributed memory systems. REBOUND also supports hybrid parallelization using both OpenMP and MPI simultaneously.

2.1. Downloading and compiling the code

The source code is hosted on the github platform and can be downloaded at http://github.com/hannorein/rebound/. A snapshot of the current repository is provided as tar and zip-files. However, users are encouraged to clone the entire repository with the revision control system git. The latter allows one to keep up-to-date with future updates. Contributions from users to the public repository are welcome. Once downloaded and extracted, one finds five main directories.

The entire source code can be found in the src/ directory. In the vast majority of cases, nothing in this directory needs to be modified.

Many examples are provided in the examples/ directory. Each example comes with a problem file, named problem.c, and a makefile named Makefile. To compile one of the examples, one has to run the make command in the corresponding directory. The code compilation is then performed in the following steps:

1. The makefile sets up environment variables which control various options such as the choice of compiler, code optimization, real time visualization and parallelization.
2. It sets symbolic links, specifying the modules chosen for this problem (see below).
3. It calls the makefile in the src/ directory which compiles and links all source files.
4. The binary file is copied to the problem directory, from where it can be executed.

Documentation of the source code can be generated in the doc/ directory with doxygen. There is no static documentation available because the code structure depends crucially on the modules currently selected. To update the documentation with the current module selection, one can simply run make doc in any directory with a makefile.

In the directory tests/ one finds tests for accuracy and scaling as well as simple unit tests. The source code of the tests presented in Sects. 6 and 7 is included as well.

The problem/ directory is the place to create new problems. It contains a template for that. Any of the examples can also be used as a starting point for new problems.

2.2. Modules

REBOUND is extremely modular. The user has the choice between different gravity, collision, boundary and integration modules. It is also possible to implement completely new modules with minimal effort.

Modules are chosen by setting symbolic links. Thus, there is no need to execute a configuration script prior to compiling the code. For example, there is one link gravity.c which points to

one of the gravity modules gravity_*.c. The symbolic links are set in each problem makefile. Only this makefile has to be changed when a different module is used. Pre-compiler macros are set automatically for situations in which different modules need to know about each other.

This setup allows the user to work on multiple projects at the same time using different modules. When switching to another problem, nothing has to be set-up and the problem can by compiled by simply typing make in the corresponding directory.

To implement a new module, one can just copy an existing module to the problem directory, modify it and change the link in the makefile accordingly. Because no file in the src/ directory needs to be changed, one can easily keep REBOUND in sync with new versions[2].

2.3. Computational domain and boundary conditions

In REBOUND, the computational domain consists of a collection of cubic boxes. Any integer number of boxes can be used in each direction. This allows elongated boxes to be constructed out of cubic boxes. The cubic root boxes are also used for static domain decomposition when MPI is enabled. In that case, the number of root boxes has to be an integer multiple of the number of MPI nodes. When a tree is used for either gravity or collision detection, there is one tree structure per root box (see Sect. 4.2).

REBOUND comes with three different boundary conditions. Open boundaries (boundaries_open.c) remove every particle from the simulation that leaves the computational domain. Periodic boundary conditions (boundaries_periodic.c) are implemented with ghost boxes. Any number of ghost boxes can be used in each direction. Shear-periodic boundary conditions (boundaries_shear.c) can be used to simulate a shearing sheet.

3. Integrators

Several different integrators have been implemented in REBOUND. Although all of these integrators are second order accurate and symplectic, their symplectic nature is formally lost as soon as self-gravity or collisions are approximated or when velocity dependent forces are included.

All integrators follow the commonly used Drift-Kick-Drift (DKD) scheme[3] but implement the three sub-steps differently. We describe the particles' evolution in terms of a Hamiltonian H which can often be written as the sum of two Hamiltonians $H = H_1 + H_2$. How the Hamiltonian is split into two parts depends on the integrator. Usually, one identifies $H_1(p)$ as the kinetic part and $H_2(q)$ as the potential part, where p and q are the canonical momenta and coordinates. During the first drift sub-step, the particles evolve under the Hamiltonian H_1 for half a time-step $dt/2$. Then, during the kick sub-step, the particles evolve under the Hamiltonian H_2 for a full time-step dt. Finally, the particles evolve again for half a time-step under H_1. Note that the positions and velocities are synchronized in time only at the end of the DKD time-steps. We refer the reader to Saha & Tremaine (1992) and references therein for a detailed discussion on symplectic integrators.

REBOUND uses the same time-step for all particles. By default, the time-step does not change during the simulation

---

[2] On how to do that, see for example http://gitref.org/ for an introduction to git.

[3] We could have also chosen a KDK scheme but found that a DKD scheme performs slightly better.

because in all the examples that come with `REBOUND`, the time-step can be naturally defined as a small fraction of the dynamical time of the system. However, it is straight forward to implement a variable time-step. This implementation depends strongly on the problem studied. Note that in general variable time-steps also break the symplectic nature of an integrator.

REBOUND does not choose the time-step automatically. It is up to the user to ensure that the time-step is small enough to not affect the results. This is especially important for highly collisional systems in which multiple collisions per time-step might occur and in situations where the curvature of particle trajectories is large. The easiest way to ensure numerical convergence is to run the same simulation with different time-steps. We encourage users to do that whenever a new parameter regime is studied.

3.1. Leap-frog

Leap-frog is a second-order accurate and symplectic integrator for non-rotating frames. Here, the Hamiltonian is split into the kinetic part $H_1 = \frac{1}{2}p^2$ and the potential part $H_2 = \Phi(x)$. Both the drift and kick sub-steps are simple Euler steps. First the positions of all particles are advanced for half a time-step while keeping the velocities fixed. Then the velocities are advanced for one time-step while keeping the positions fixed. In the last sub-step the velocities are again advanced for half a time-step. Leap-frog is implemented in the module `integrator_leapfrog.c`.

3.2. Wisdom-Holman mapping

A symplectic Wisdom-Holman mapping (WH, Wisdom & Holman 1991) is implemented as a module in `integrator_wh.c`. The implementation follows closely that by the SWIFT code[4]. The WH mapping is a mixed variable integrator that calculates the Keplerian motion of two bodies orbiting each other exactly up to machine precision during the drift sub-step. Thus, it is very accurate for problems in which the particle motion is dominated by a central $1/r$ potential and perturbations added in the kick sub-step are small. However, the WH integrator is substantially slower than the leap-frog integrator because Kepler's equation is solved iteratively every time-step for every particle.

The integrator assumes that the central object has the index 0 in the particle array, that it is located at the origin and that it does not move. The coordinates of all particles are assumed to be the heliocentric frame. During the sub-time-steps the coordinates are converted to Jacobi coordinates (and back) according to their index. The particle with index 1 has the first Jacobi index, and so on. This works best if the particles are sorted according to their semi-major axis. Note that this is not done automatically.

3.3. Symplectic epicycle integrator

The symplectic epicycle integrator (SEI, Rein & Tremaine 2011) for Hill's approximation (Hill 1878) is implemented in `integrator_sei.c`. When shear-periodic boundary conditions (`boundaries_shear.c`) are used, the Hill approximation is know as a shearing sheet.

SEI has similar properties to the Wisdom-Holman mapping in the case of the Kepler potential but works in a rotating frame and is as fast as a standard non-symplectic integrator. The error after one time-step scales as the third power of the time-step

---

[4] http://www.boulder.swri.edu/~hal/swift.html

---

times the ratio of the gravitational force over the Coriolis force (see Rein & Tremaine 2011, for more details on the performance of SEI).

The epicyclic frequency Ω and the vertical epicyclic frequency Ω_z can be specified individually. This can be used to enhance the particle density in the mid-plane of a planetary ring and thus simulate the effect of self-gravity (see e.g. Schmidt et al. 2001).

4. Gravity

REBOUND is currently equipped with two (self-)gravity modules. A gravity module calculates exactly or approximately the acceleration onto each particle. For a particle with index i this is given by

$$\boldsymbol{a}_i = \sum_{j=0}^{N_{\text{active}}-1} \frac{Gm_j}{\left(r_{ji}^2 + b^2\right)^{3/2}} \, \hat{\boldsymbol{r}}_{ji}, \tag{1}$$

where G is the gravitational constant, m_j the mass of particle j and $\boldsymbol{r}_{ji}$ the relative distance between particles j and i. The gravitational softening parameter b defaults to zero but can be set to a finite value in simulations where physical collisions between particles are not resolved and close encounters might lead to large unphysical accelerations. The variable N_{active} specifies the number of massive particles in the simulation. Particles with an index equal or larger than N_{active} are treated as test-particles. By default, all particles are assumed to have mass and contribute to the sum in Eq. (1).

4.1. Direct summation

The direct summation module is implemented in `gravity_direct.c` and computes Eq. (1) directly. If there are N_{active} massive particles and N particles in total, the performance scales as $O(N \cdot N_{\text{active}})$. Direct summation is only efficient with few active particles; typically $N_{\text{active}} \lesssim 10^2$.

4.2. Octree

Barnes & Hut (1986, BH hereafter) proposed an algorithm to approximate Eq. (1), which can reduce the computation time drastically from $O(N^2)$ to $O(N \log N)$. The idea is straightforward: distant particles contribute to the gravitational force less than those nearby. By grouping particles hierarchically, one can separate particles in being far or near from any one particle. The total mass and the center of mass of a group of particles which are far away can then be used as an approximation when calculating the long-range gravitational force. Contributions from individual particles are only considered when they are nearby.

We implement the BH algorithm in the module `gravity_tree.c`. The hierarchical structure is realized using a three-dimensional tree, called an octree. Each node represents a cubic cell which might have up to eight sub-cells with half the size. The root node of the tree contains all the particles, while the leaf nodes contain exactly one particle. The BH tree is initially constructed by adding particles one at a time to the root box, going down the tree recursively to smaller boxes until one reaches an empty leaf node to which the particle is then added. If the leaf node already contains a particle it is divided into eight sub-cells.

Every time the particles move, the tree needs to be updated using a tree reconstruction algorithm. We therefore keep track
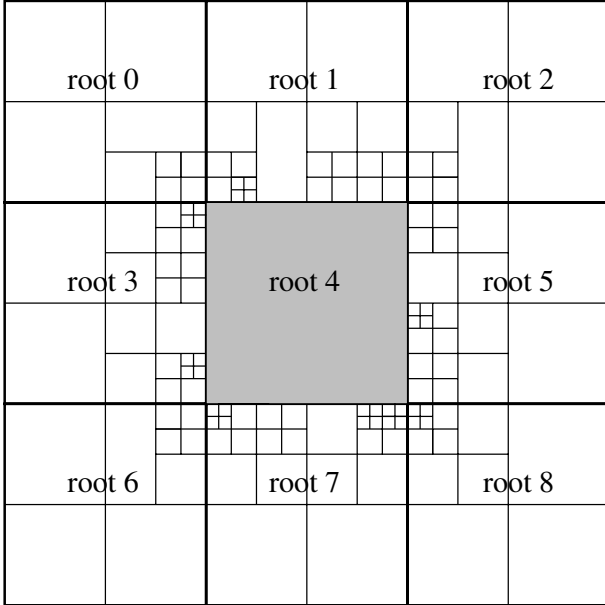
Fig. 1. Illustration of the essential trees needed by root box 4. The different levels of the tree structure which need to be copied depend on the distance to the nearest boundary of root box 4 and the opening angle θ. See text for details.

Fig. 2. Different collision detection algorithms. *Left*: curved particle trajectories are approximated by straight lines. *Right*: trajectories are not approximated, particles only collide when they are overlapping. See text for details.

of any particle crossing the boundaries of the cell it is currently in. If it has moved outside, then the corresponding leaf node is destroyed and the particle is re-added to the tree as described above. After initialization and reconstruction, we walk through the tree to update the total mass and the center of mass for each cell from the bottom-up.

To calculate the gravitational forces on a given particle, one starts at the root node and descends into sub-cells as long as the cells are considered to be close to the particle. Let us define the opening angle as $\theta = w/R$, where w is the width of the cell and R is the distance from the cell's center of mass to the particle. If the opening angle is smaller than a critical angle $\theta_{crit} > \theta$, the total mass and center of mass of the cell are used to calculate the contribution to the gravitational force. Otherwise, the sub-cells are opened until the criterion is met. One has to choose θ_{crit} appropriately to achieve a balance between accuracy and speed.

REBOUND can also include the quadrupole tensor of each cell in the gravity calculation by setting the pre-compiler flag QUADRUPOLE. The quadrupole expansion (Hernquist 1987) is more accurate but also more time consuming for a fixed θ_{crit}. We discuss how the critical opening angle and the quadrupole expansion affect the accuracy in Sect. 6.1.

With REBOUND, a static domain decomposition is used for parallelizing the tree algorithm on distributed memory systems. Each MPI node contains one or more root boxes (see also Sect. 2.3) and all particles within these boxes belong to that node. The number of root boxes N_{RB} has to be a multiple of the number of MPI nodes N_{MPI}. For example, the setup illustrated in Fig. 1 uses 9 root boxes allowing 1, 3 or 9 MPI nodes. By default, the domain decomposition is done first along the z direction, then along the y and x direction. If one uses 3 MPI nodes in the above example, the boxes 0−2 are on on node 0, the boxes 3−5 on node 1 and the remaining boxes on node 2. When a particle moves across a root box boundary during the simulation, it is send to the corresponding node and removed form the local tree.
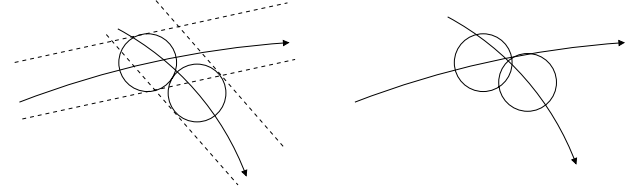
Because of the long-range nature of gravity, every node needs information from any other node during the force calculation. We distribute this information before the force calculation using an essential tree (Salmon et al. 1990) and an all-to-all communication pattern. The essential tree contains only those cells of the local tree that might be accessed by the remote node during the force calculation. Each node prepares a total of $N_{RB} - N_{RB}/N_{MPI}$ different essential trees. The cells that constitute the essential tree are copied into a buffer array and the daughter cell references therein are updated accordingly. The center of mass and quadrupole tensors (if enabled) are stored in the cell structure and automatically copied when a cell is copied. For that reason only the tree structure needs to be distributed, not individual particles. The buffer array is then sent to the other nodes using non-blocking MPI calls.

For example, suppose 9 MPI nodes are used, each node using exactly one tree in its domain. For that scenario the essential trees prepared for root box 4 are illustrated in Fig. 1. The essential trees include all cells which are close enough to the boundary of root box 4 so that they might be opened during the force calculation of a particle in root box 4 according to the opening angle criteria.

In Sect. 7 we show that this parallelization is very efficient when the particle distribution is homogeneous and there are more than a few thousand particles on every node. When the number of particles per node is small, communication between nodes dominates the total runtime.

5. Collisions

REBOUND supports several different modules for collision detection which are described in detail below. All of these methods search for collisions only approximately, might miss some of the collisions or detect a collision where there is no collision. This is because either curved particle trajectories are approximated by straight lines (collisions_sweep.c and collisions_sweepphi.c) or particles have to be overlapping to collide (collisions_direct.c and collisions_tree.c). This is also illustrated in Fig. 2.

In all modules, the order of the collisions is randomized. This ensures that there is no preferred ordering which might lead to spurious correlations when one particles collides with multiple particles during one time-step. Note that REBOUND uses a fixed time-step for all particles. Therefore one has to ensure that the time-step is chosen small enough so that one particle does collide with no more than one other particle during one time-step, at least on average. See also the discussion in Sect. 3.

A free-slip, hard-sphere collision model is used. Individual collisions are resolved using momentum and energy conservation. A constant or an arbitrary velocity dependent normal

coefficient of restitution ϵ can be specified to model inelastic collisions. The relative velocity after one collision is then given by

$$v'_n = -\epsilon \, v_n$$
$$v'_t = v_t, \tag{2}$$

where v_n and v_t are the relative normal and tangential velocities before the collision. Particle spin is currently not supported.

5.1. Direct nearest neighbor search

A direct nearest neighbor collisions search is the simplest collision module in REBOUND. It is implemented in collisions_direct.c,

In this module, a collision is detected whenever two particles are overlapping at the end of the DKD time-step, i.e. the middle of the drift sub-step, where positions and velocities are synchronized in time (see Sect. 3). This is illustrated in the right panel of Fig. 2. Then, the collision is added to a collision queue. When all collisions have been detected, the collision queue is shuffled randomly. Each individual collision is then resolved after checking that the particles are approaching each other.

Every pair of particles is checked once per time-step, making the method scale as $O(N^2)$. Similar to the direct summation method for gravity, this is only useful for a small number of particles. For most cases, the nearest neighbor search using a tree is much faster (see next section).

5.2. Octree

The octree described in Sect. 4.2 can also be used to search for nearest neighbors. The module collisions_tree.c implements such a nearest neighbor search. It is parallelized with both OpenMP and MPI. It can be used in conjunction with any gravity module, but when both tree modules gravity_tree.c and collisions_tree.c are used simultaneously, only one tree structure is needed. When collisions_tree.c is the only tree module, center of mass and octopole tensors are not calculated in tree cells.

To find overlapping particles for particle i, one starts at the root of the tree and descents into daughter cells as long as the distance of the particle to the cell center r_{ic} is smaller than a critical value:

$$r_{ic} < R_i + R_{max} + \frac{\sqrt{3}}{2} w_c, \tag{3}$$

where R_i is the size of the particle, R_{max} is the maximum size of a particle in the simulation and w_c is the width of the current cell. When two particles are found to be overlapping, a collision is added to the collision queue and resolved later in the same way as above.

If MPI is used, each node prepares the tree and particle structures that are close to the domain boundaries as these might be needed by other nodes (see Fig. 1). This essential tree is send to other nodes and temporarily added to the local tree structure. The nearest neighbor search can then be performed in the same way as in the serial version. The essential tree and particles are never modified on a remote node.

This essential tree is different from the essential tree used for the gravity calculation in two ways. First, this tree is needed at the end of the time-step, whereas the gravity tree is needed at the beginning of the kick sub time-step. Second, the criteria for cell opening, Eq. (3), is different.
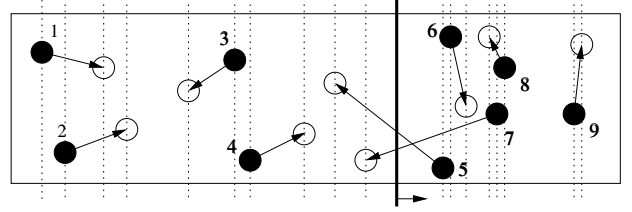

Fig. 3. Illustration of the plane-sweep algorithm. The plane is intersecting the trajectories of particles 5 and 7. See text for details.

A nearest neighbor search using the octree takes on average $O(\log(N))$ operations for one particle and therefore $O(N \log(N))$ operations for all N particles.

5.3. Plane-sweep algorithm

We further implement two collision detection modules based on a plane-sweep algorithm in collisions_sweep.c and collisions_sweepphi.c. The plane-sweep algorithm makes use of a conceptual plane that is moved along one dimension.

The original algorithm described by Bentley & Ottmann (1979) maintains a binary search tree in the orthogonal dimensions and keeps track of line crossings. In our implementation, we assume the dimension normal to the plane is much longer than the other dimensions. This allows us to simplify the Bentley-Ottmann algorithm and get rid of the binary search tree which further speeds up the calculation.

In REBOUND the sweep is either performed along the x-direction or along the azimuthal angle ϕ (measured in the xy-plane from the origin). The sweep in the x direction can also be used in the shearing sheet. The sweep in the ϕ direction is useful for (narrow) rings in global simulations. Here, we only discuss the plane-sweep algorithm in the Cartesian case (along the x-direction) in detail. The ϕ sweep implementation is almost identical except of the difference in periodicity and the need to calculate the angle and angular frequency for every particle at the beginning of the collision search.

Our plane-sweep algorithm can be described as follows (see also Fig. 3):

1. If a tree is not used to calculate self-gravity, the particles are sorted according to their x coordinate[5]. During the first time-step, quicksort is used as the particles are most likely not pre-sorted. In subsequent time-steps, the adaptive sort algorithm insertionsort is used. It can make use of the pre-sorted array from the previous time-step and has an average performance of $O(N)$ as long as particles do not completely randomize their positions in one time-step.
2. The x coordinate of every particle before and after the drift step is inserted into an array SWEEPX. The trajectory is approximated by a line (see left panel of Fig. 2). In general, the real particle trajectories will be curved. In that case the positions are only approximately the start and end points of the particle trajectory. The particle radius is subtracted/added to the minimum/maximum x coordinate. The array contains $2N$ elements when all particles have been added.
3. If a tree is not used, the array SWEEPX is sorted with the x position as a key using the insertionsort algorithm. Because

---

[5] Each tree cell keeps a reference to the particle it contains. This reference has to be updated every time a particle is moved in the particle array which would lead to larger overhead.
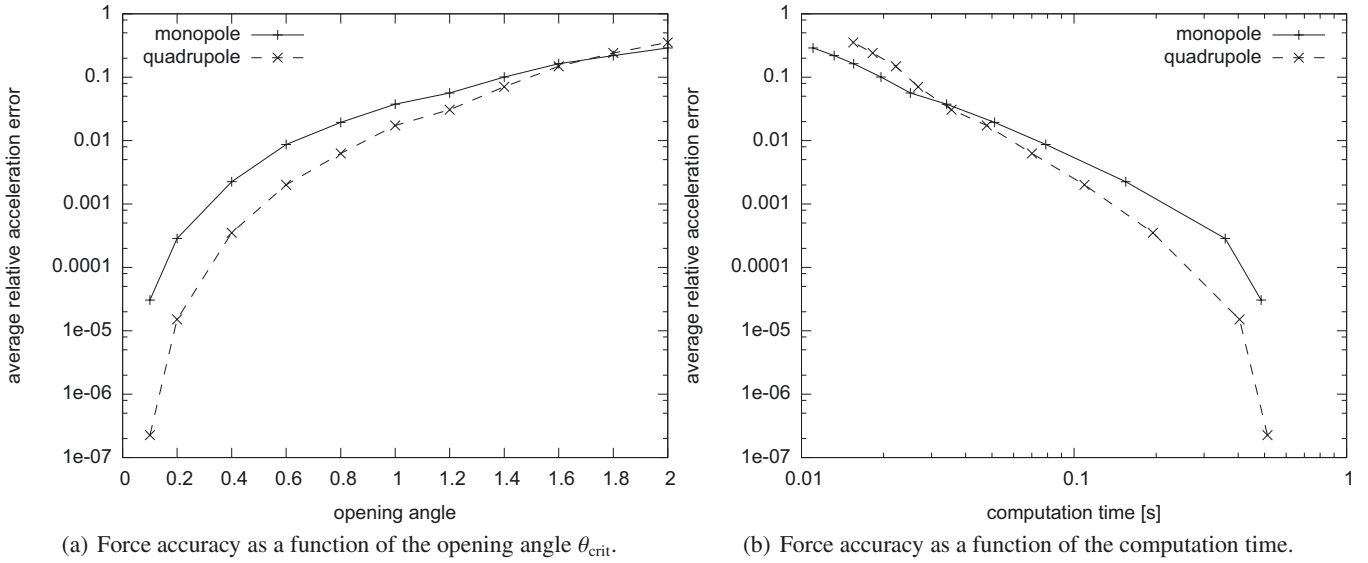
(a) Force accuracy as a function of the opening angle θ_{crit}.

(b) Force accuracy as a function of the computation time.

Fig. 4. Comparison of the monopole and quadrupole expansion.

the particle array is pre-sorted, insertionsort runs in approximately $O(N)$ operations. If a tree is used, the array is sorted with quicksort.

4. A conceptual plane with its normal vector in the x direction is inserted at the left side of the box. While going through the array SWEEPX, we move the plane towards the right one step at a time according to the x coordinate of the current element in the array. We thus move the plane to the other side of the box in a total of $2N$ stops.

5. The plane is intersecting particle trajectories. We keep track of these intersection using a separate array SWEEPL. Whenever a particle appears for the first time in the array SWEEPX the particle is added to the SWEEPL array. The particle is removed from the array SWEEPL when it appears in the array SWEEPX for the second time. In Fig. 3, the plane is between stop 10 and 11, intersecting the trajectories of particles 5 and 7.

6. When a new particle is inserted into the array SWEEPL, we check for collisions of this particle with any other particle in SWEEPL during the current time-step. The collision is recorded and resolved later. In Fig. 3 the array SWEEPL has two entries, particles 5 and 7. Those will be checked for collisions.

The time needed to search for a collision at each stop of the plane is $O(N_{\mathrm{SWEEPL}})$, where N_{SWEEPL} is the number of elements in the array SWEEPL. This could be reduced with a binary search tree to $O(\log(N_{\mathrm{SWEEPL}}))$ as in the original algorithm by Bentley & Ottmann (1979). However tests have shown that there is little to no performance gain for the problems studied with REBOUND because a more complicated data structure has to be maintained. One entire time-step with the plane-sweep algorithm is completed in $O(N \cdot N_{\mathrm{SWEEPL}})$. It is then easy to see that this method can only be efficient when $N_{\mathrm{SWEEPL}} \lesssim \log(N)$, as a tree code is more efficient otherwise.

Indeed, experiments have shown (see Sect. 7.4) that the plane-sweep algorithm is more efficient than a nearest neighbor search with an octree by many orders of magnitude for low dimensional systems in which N_{SWEEPL} is small.

6. Test problems

We present several tests in this section which verify the implementation of all modules. First, we measure the accuracy of the tree code. Then we check for energy and momentum conservation. We use a long term integration of the outer solar system as a test of the symplectic WH integrator. Finally, we measure the viscosity in a planetary ring which is a comprehensive test of both self-gravity and collisions.

6.1. Force accuracy

We measure the accuracy of the tree module gravity_tree.c by comparing the force onto each particle to the exact result obtained by direct summation (Eq. (1)). We set up 1000 randomly distributed particles with different masses in a box. We do not use any ghost boxes and particles do not evolve.

We sum up the absolute value of the acceleration error for each particle and normalize it with respect to the total acceleration (see Hernquist 1987, for more details).

This quantity is plotted as a function of the critical opening angle θ_{crit} in Fig. 4a. One can see that the force quickly converges towards the correct value for small θ_{crit}. The quadrupole expansion performs one order of magnitude better then the monopole expansion for $\theta_{\mathrm{crit}} \sim 0.5$ and two orders of magnitude better for $\theta_{\mathrm{crit}} \sim 0.1$.

In Fig. 4b we plot the errors of the same simulations as a function of the computation time. The quadrupole expansion requires more CPU time than the monopole expansion for fixed θ_{crit}. However, the quadrupole expansion is faster when $\theta_{\mathrm{crit}} \lesssim 1$ for a fixed accuracy. Note that including the quadrupole tensor also increases communication costs between MPI nodes.

6.2. Energy and momentum conservation in collisions

In a non-rotating simulation box with periodic boundaries and non-gravitating collisional particles, we test both momentum and energy conservation. Using a coefficient of restitution of unity (perfectly elastic collisions), the total momentum and
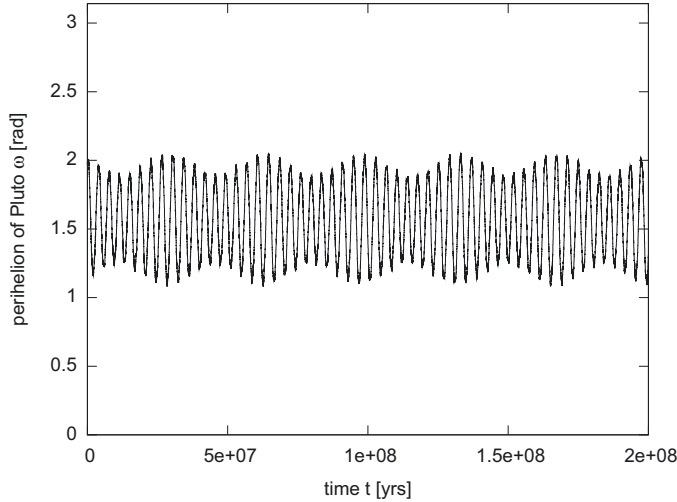
Fig. 5. Libration pattern of Pluto with two distinct frequencies of 3.8 Myr and 34 Myr.

Fig. 6. Individual components of the viscosity as a function of the non-dimensional particle radius.

energy is conserved up to machine precision for all collision detection algorithms.

6.3. Long term integration of Solar System

To test the long-term behavior of our implementation of the Wisdom-Holman Mapping, we integrate the outer Solar System for 200 million years. We use the initial conditions given by Applegate et al. (1986) with 4 massive planets and Pluto as a test particle. The direct summation module has been used to calculate self-gravity. With a 40 day time-step and an integration time of 200 Myr, the total runtime on one CPU was less then 2 h.

In Fig. 5, we plot the perihelion of Pluto as a function of time. One can clearly see two distinct libration frequencies with 3.8 Myr and 34 Myr time-scales respectively. This is in perfect agreement with Applegate et al. (1986).

6.4. Viscosity in planetary rings

Daisaka et al. (2001) calculate the viscosity in a planetary ring using numerical simulations. We repeat their analysis as this is an excellent code test as the results depend on both self-gravity and collisions. The quasi-equilibrium state is dominated by either self-gravity or collisions, depending on the ratio of the Hill radius over the physical particle radius, $r_h^\star$.

In this simulation we use the octree implementation for gravity and the plane-sweep algorithm for collisions. The geometric optical depth is $\tau = 0.5$ and we use a constant coefficient of restitution of $\epsilon = 0.5$. The separate parts of the viscosity are calculated directly as defined by Daisaka et al. (2001) for various $r_h^\star$ and plotted in dimensionless units in Fig. 6.

The results are in good agreement with previous results. At large $r_h^\star$, the collisional part of the viscosity is slightly higher in our simulations when permanent particle clumps form. This is most likely due to the the different treatment of collisions and some ambiguity in defining the collisional viscosity when particles are constantly touching each other (Daisaka, private comm.).
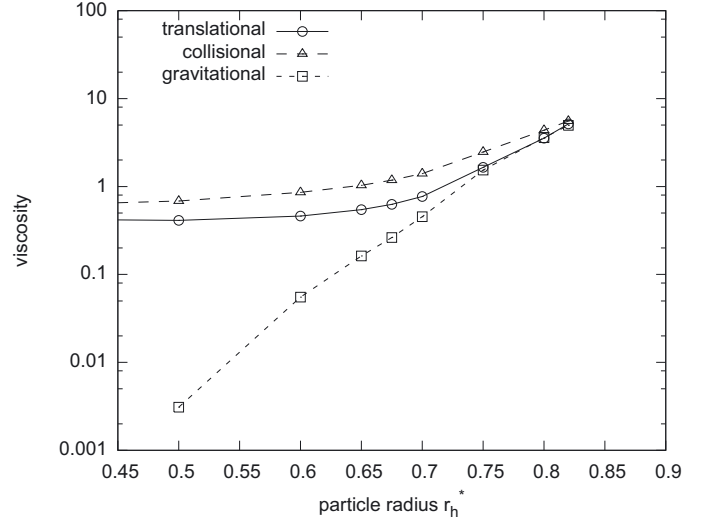
7. Scaling

Using the shearing sheet configuration with the tree modules `gravity_tree.c` and `collisions_tree.c`, we measure the scaling of REBOUND and the efficiency of the parallelization. The simulation parameters have been chosen to resemble those of a typical simulation of Saturn's A-ring with an optical depth of order unity and a collision time-scale being much less than one orbit. The opening angle is $\theta_{crit} = 0.7$. The `problem.c` files for this and all other tests can be found in the `test/` directory.

All scaling tests have been performed on the IAS aurora cluster. Each node has dual quad-core 64-bit AMD Opteron Barcelona processors and 16 GB RAM. The nodes are interconnected with 4x DDR Infiniband.

7.1. Strong scaling

In the strong scaling test the average time to compute one time-step is measured as a function of the number of processors for a fixed total problem size (e.g. fixed total number of particles). We use only the MPI parallelization option.

The results for simulations using $N = 12.5k, 50k, 200k$ and $800k$ particles are plotted in Fig. 7a. One can see that for a small number of processors the scaling is linear for all problems. When the number of particles per processor is below a critical value, $N_{pp} \sim 2000$, the performance drops. Below the critical value, a large fraction of the tree constitutes the essential tree which needs to be copied to neighboring nodes every time-step. This leads to an increase in communication.

The results show that we can completely utilize 64 processors cores with one million particles.

7.2. Weak scaling

In the weak scaling test we measure the average time to compute one time-step as a function of the number of processors for a fixed number of particles per processor. Again, we only use the MPI parallelization option.

The results for simulations using $N_{pp} = 25k, 50k$ and $100k$ particles per processor are plotted in Fig. 7b. One can easily confirm that the runtime for a simulation using k processors is

(a) Strong scaling test: constant problem size, varying number of nodes.

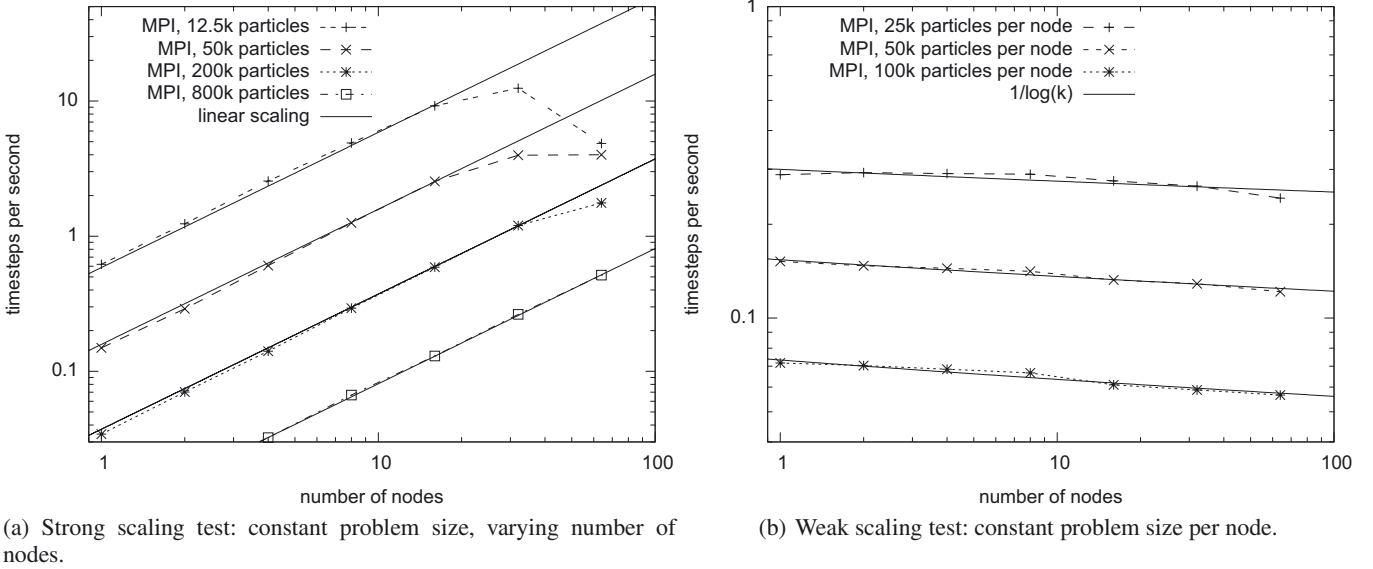(b) Weak scaling test: constant problem size per node.

Fig. 7. Strong and weak scaling tests using a shearing sheet configuration with the `gravity_tree.c` and `collisions_tree.c` modules.
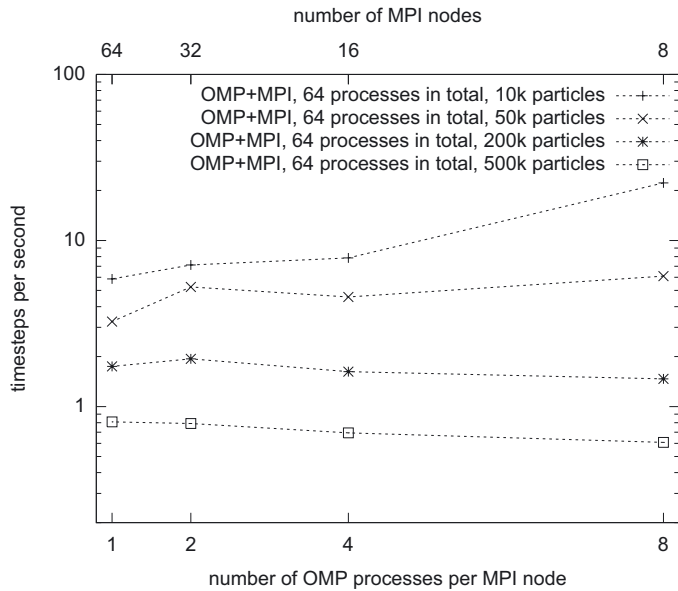

Fig. 8. Comparison between OpenMP and MPI. Each run uses 64 CPU cores. A shearing sheet configuration the with `gravity_tree.c` and `collisions_tree.c` modules is used.

$O(N_{pp} \log(N_{pp} k))$, as expected. By increasing the problem size, the communication per processor does not increase for the collision detection algorithm as only direct neighbors need to be evaluated on each node. The runtime and communication for the gravity calculation is increasing logarithmically with the total number of particles (which is proportional to the number of processors in this case).

These results shows that REBOUND's scaling is as good as it can possibly get with a tree code. The problem size is only limited by the total number of available processors.

7.3. OpenMP/MPI trade-off

The previous results use only MPI for parallelization. REBOUND also supports parallelization with OpenMP for shared memory systems.

OpenMP has the advantage over MPI that no communication is needed. On one node, different processes share the same memory and work on the same tree and particle structures. However, the tree building and reconstruction routines are not parallelized. These routines can only be parallelized efficiently when a domain decomposition is used (as used for MPI, see above).

Results of hybrid simulations using both OpenMP and MPI at the same time are shown in Fig. 8. We plot the average time to compute one time-step as a function of the number of OpenMP processes per MPI node. The total number of particles and processors (64) is kept fixed.

One can see that OpenMP does indeed perform better than MPI when the particle number per node is small and the runtime is dominated by communication (see also Sect. 7.1). For large particle numbers, the difference between OpenMP and MPI is smaller, as the sequential tree reconstruction outweighs the gains. Eventually, for very large simulations ($N_{pp} \gtrsim 5000$) the parallelization with MPI is faster.

Thus, in practice OpenMP can be used to accelerate MPI runs which are bound by communication. It is also an easy way to accelerate simulations on desktop computer which have multiple CPU cores.

7.4. Comparison of collision detection algorithms

The collision modules described in Sect. 5 have very different scaling behaviors and are optimized for different situations. Here, we illustrate their scalings using two shearing sheet configurations with no self-gravity. We plot the average number of time-steps per second as a function of the problem size in Fig. 9 for the plane-sweep algorithm and both the octree and direct nearest neighbor collision search.

In simulations used in Fig. 9a, we vary both the azimuthal size, L_y, and radial size, L_x, of the computational domain. The aspect ratio of the simulation box is kept constant. For the plane-sweep algorithm, the number of particle trajectories intersecting the plane[6] scales as $N_{SWEEPL} \sim L_y \sim \sqrt{N}$. Thus, the

---

[6] Note that a disk is effectively a two dimensional system. In three dimensions $N_{SWEEPL} \sim L_y L_z \sim N^{2/3}$.

(a) Varying the size of the simulation box and keeping a constant aspect ratio.

(b) Varying the radial size of the simulation box and keeping a constant azimuthal width.
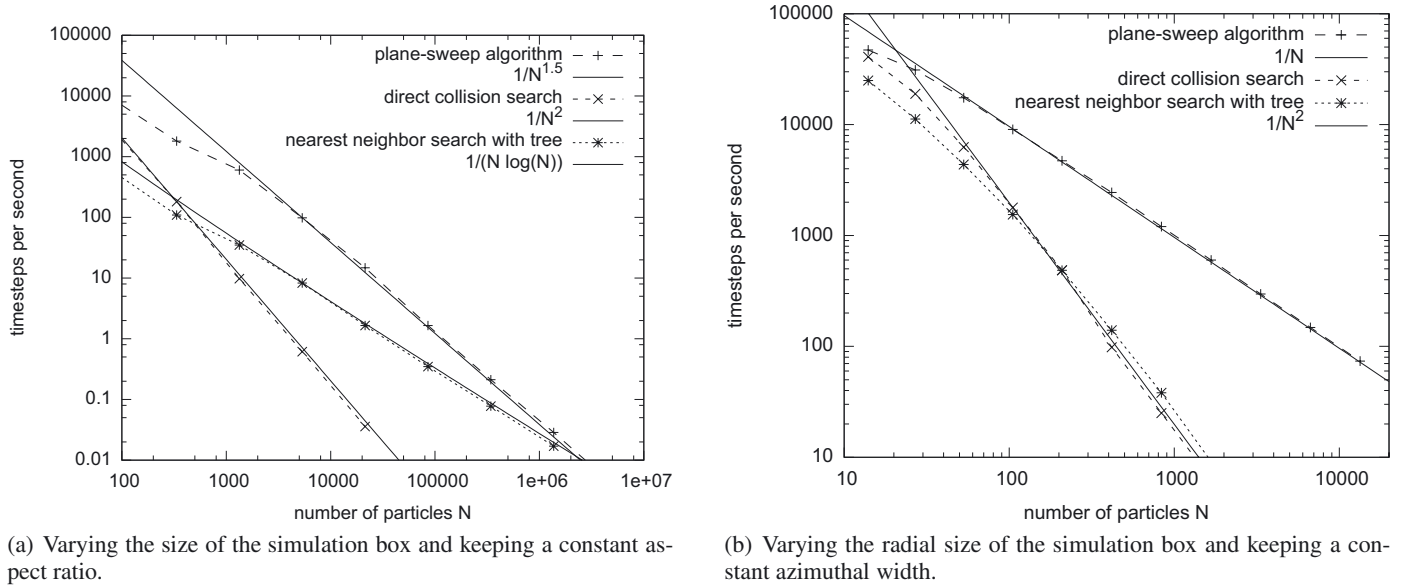
Fig. 9. Scalings of the plane-sweep algorithm, the octree and direct nearest neighbor search as a function of particle number. A shearing sheet configuration without self-gravity is used.

overall scaling of the plane-sweep method is $O(N^{1.5})$, which can be verified in Fig. 9a. Both the tree and direct detection methods scale unsurprisingly as $O(N\log(N))$ and $O(N^2)$, respectively.

For simulations used in Fig. 9b, we vary the radial size of the computational domain and keep the azimuthal size fixed at 20 particle radii. Thus, the aspect ratio changes and the box becomes very elongated for large particle numbers. If a tree is used in REBOUND, an elongated box is implemented as many independent trees, each being a cubic root box (see Sect. 2.3). Because each tree needs to be accessed at least one during the collision search, this makes the tree code scale as $O(N^2)$ for large N, effectively becoming a direct nearest neighbor search. The plane-sweep algorithm on the other hand scales as $O(N)$, as the number of particle trajectories intersecting the plane is constant, $N_{\text{sweep}} \sim L_y = \text{const}$. Again, the direct nearest neighbor search scales unsurprisingly as $O(N^2)$.

From these test cases, it is obvious that the choice of collision detection algorithm strongly depends on the problem. Also note that if the gravity module is using a tree, the collision search using the same tree comes at only a small additional cost.

The plane-sweep module can be faster for non-self-gravitating simulations by many orders of magnitude, especially if the problem size is varied only in one dimension.

8. Summary

In this paper, we presented REBOUND, a new open-source multi-purpose *N*-body code for collisional dynamics. REBOUND is available for download at http://github.com/hannorein/rebound and can be redistributed freely under the GPLv3 license.

The code is written in a modular way, allowing users to choose between different numerical integrators, boundary conditions, self-gravity solvers and collision detection algorithms. With minimal effort, one can also implement completely new modules.

The octree self-gravity and collision detection modules are fully parallelized with MPI and OpenMP. We showed that both

run efficiently on multi-core desktop machines as well as on large clusters. Results from a weak scaling test show that there is no practical limit on the maximum number of particles that REBOUND can handle efficiently except by the number of available CPUs. We will use this in future work to conduct extremely elongated simulations that can span the entire circumference of Saturn's rings.

Two new collision detection methods based on a plane-sweep algorithm are implemented in REBOUND. We showed that the plane-sweep algorithm scales linearly with the number of particles for effectively low dimensional systems and is therefor superior to a nearest neighbor search with a tree. Examples of effectively low dimensional systems include very elongated simulation domains and narrow rings. Furthermore, the simpler data-structure of the plane-sweep algorithm makes it also superior for quasi-two dimensional simulations with less than about one million particles.

Three different integrators have been implemented, for rotating and non-rotating frames. All of these integrators are symplectic. Exact long-term orbit integrations can be performed with a Wisdom-Holman mapping.

Given the already implemented features as well as the open and modular nature of REBOUND, we expect that this code will find many applications both in the astrophysics community and beyond. For example, molecular dynamics and granular flows are subject areas where the methods implemented in REBOUND can be readily applied. We strongly encourage users to contribute new algorithms and modules to REBOUND.

References

Applegate, J. H., Douglas, M. R., Gursel, Y., Sussman, G. J., & Wisdom, J. 1986, AJ, 92, 176
Barnes, J., & Hut, P. 1986, Nature, 324, 446
Bentley, J., & Ottmann, T. 1979, Computers, IEEE Transactions on, C-28, 643
Crida, A., Papaloizou, J., Rein, H., Charnoz, S., & Salmon, J. 2010, AJ, submitted
Daisaka, H., Tanaka, H., & Ida, S. 2001, Icarus, 154, 296
Hernquist, L. 1987, ApJS, 64, 715
Hill, G. W. 1878, Astron. Nachr., 91, 251
Johansen, A., Oishi, J. S., Low, M.-M. M., et al. 2007, Nature, 448, 1022
Lewis, M. C., & Stewart, G. R. 2009, Icarus, 199, 387
Lithwick, Y., & Chiang, E. 2007, ApJ, 656, 524
Michikoshi, S., & Kokubo, E. 2011, ApJ, 732, L23

Rein, H., & Papaloizou, J. C. B. 2010, A&A, 524, A22
Rein, H., & Tremaine, S. 2011, MNRAS, 415, 3168
Rein, H., Lesur, G., & Leinhardt, Z. M. 2010, A&A, 511, A69
Richardson, D. C. 1994, MNRAS, 269, 493
Saha, P., & Tremaine, S. 1992, AJ, 104, 1633
Salmon, J., Quinn, P. J., & Warren, M. 1990, Using parallel computers for very large N-body simulations (Dynamics and Interactions of Galaxies), 216
Salo, H. 1991, Icarus, 90, 254
Schmidt, J., Salo, H., Spahn, F., & Petzschmann, O. 2001, Icarus, 153, 316
Schmidt, J., Ohtsuki, K., Rappaport, N., Salo, H., & Spahn, F. 2009, Dynamics of Saturn's Dense Rings (Springer), 413
Stark, C. C., & Kuchner, M. J. 2009, ApJ, 707, 543
Wisdom, J., & Holman, M. 1991, AJ, 102, 1528
Wisdom, J., & Tremaine, S. 1988, AJ, 95, 925