REBOUNDx Documentation

Release 4.6.2

Dan Tamayo

May 28, 2026

CONTENTS

1 Get Started! **3**

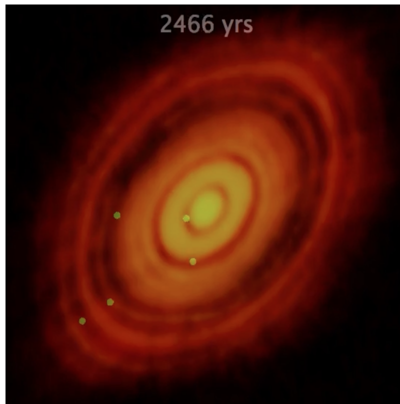
2 Attribution **5**

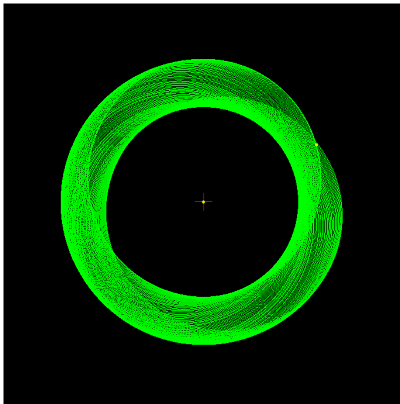
3 Contribute **7**

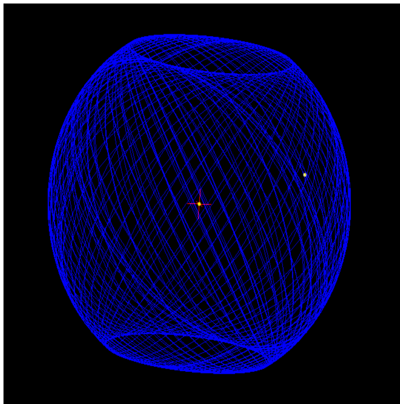
4 Issues **9**

4.1 Quickstart (Python) 9
4.2 Quickstart (C) 10
4.3 Astrophysical Effects 11
4.4 Collisions 24
4.5 Miscellaneous Utilities 24
4.6 API Documentation (C) 25
4.7 Examples (C) 43
4.8 Adding A New Effect 90

Index **99**







REBOUNDx (eXtras) allows you to easily incorporate additional physics into your REBOUND N-body integrations. The main documentation to refer back to is the *Astrophysical Effects* page. Each effect links to both a Python and C example demonstrating its use. If you are using REBOUNDx for the first time check out the quickstart guides below with installation instructions.

If you clone the repository at https://github.com/dtamayo/reboundx you can load and run all the jupyter notebook examples locally (under reboundx/ipython_examples) as well as the C examples (under reboundx/examples. In the terminal you can run the example in each folder with `make clean && make && ./rebound`).

Also see the 'saving and loading simulations' examples for how to save a REBOUNDx binary with effects and parameters. For an overview of the technical details and some practical recommendations, see Tamayo, Rein, Shi and Hernandez (2019).

CHAPTER

ONE

GET STARTED!

REBOUNDx is written in C, but we also provide a convenient Python wrapper (that's just as fast).

Quickstart (Python)

Quickstart (C)

CHAPTER

TWO

ATTRIBUTION

If you find this code useful in your research, we have tried to structure the *Astrophysical Effects* documentation in a way that makes it easy to credit the various people who have contributed, and for others to reproduce your results.

For example 'We incorporated a constant time-lag model of tides \citep{Hut81} using the \textt{tides_constant_time_lag} implementation (Baronett et al., {\em in prep.}) in {\textsc \tt REBOUNDx 3.1.0} \citep{Tamayo19}. The relevant such ADS links for each effect are provided in *Astrophysical Effects*.

CHAPTER THREE

CONTRIBUTE

REBOUNDx is a flexible framework for including new effects. You worry about incorporating the physics, and REBOUNDx takes care of how various effects should interact with the different integrators in REBOUND. *Adding A New Effect* provides a tutorial on how to do that.

Please consider contributing your effects, so we can continue to build up an efficient and well-tested repository. The documentation structure in *Astrophysical Effects* makes it easy for people to cite your particular implementation!

CHAPTER

FOUR

ISSUES

I am always happy to answer questions or otherwise help out. If you have problems, please open an issue on the GitHub page at https://github.com/dtamayo/reboundx

4.1 Quickstart (Python)

4.1.1 Installation

To quickly get up and running, simply type into a terminal (if you installed REBOUND in a virtual environment, activate it first):

```
pip install reboundx
```

At this point you are done and can skip to the Quick Start Guide below.

For a more complete installation, i.e., if you want any of the following:

- Source code
- The example files so that you can modify them locally.
- To also use the C version

First follow the installation instructions for the C version in *Quickstart (C)*. Then, to install the Python version from this repository, navigate to the *reboundx* directory and (you'd also do this to install the Python version after modifying any of the C code):

```
pip install -e ./
```

4.1.2 Quick Start Guide

You always begin by setting up your REBOUND simulation like you normally would, e.g.

```
import rebound
sim = rebound.Simulation()
sim.add(m=1.)
sim.add(a=1.)
```

To use reboundx, we first import it, and then create a `reboundx.Extras` instance, passing it the simulation we want to attach it to:

```
import reboundx
rebx = reboundx.Extras(sim)
```

We then add the effect we are interested in. There are two types of effects, forces and operators. The easiest is to go to the *Astrophysical Effects* page, which lists all effects and links to a jupyter notebook example of how to set it up. For a deeper discussion, see Tamayo et al. 2019. Loading a force/operator returns an object of the appropriate type. For example, let's add some mass loss to the star:

```
mm = rebx.load_operator("modify_mass")
rebx.add_operator(mm)
```

Each effect will have different parameters to set, listed on the *Astrophysical Effects* page and the examples. Forces, operators and particles have a params attribute that works like a dictionary. For example, let's add an exponential mass loss (i.e., negative) timescale to the star (index 0) of 1000 time units.

```
sim.particles[0].params["tau_mass"] = -1000
```

You can add as many modifications as you'd like in the same simulation. Simply add them:

```
gr = rebx.load_force("gr")
rebx.add_force(gr)
gr.params['c'] = 1.e4 # set speed of light
```

The units for the various parameters should always match the units you're using for the rest of the simulation (see the examples). When you're done setting up the modifications you want, you just run your REBOUND simulation like you normally would:

```
sim.integrate(100.)
```

Probably the quickest way to get up and running is to modify an existing example for your effect. You can find links to the appropriate examples here: *Astrophysical Effects*, as well as details of each implementation and citations.

4.2 Quickstart (C)

4.2.1 Installation

Navigate to the parent directory that holds the `rebound` folder (see below if you want to install in a custom folder). Then in a terminal:

```
git clone https://github.com/dtamayo/reboundx.git
```

(install git if you don't have it). *If you do use* a custom install location for REBOUNDx, you have to additionally set the `REB_DIR` environment variable to the path to REBOUND. You might add this to your shell's startup files, e.g. with bash:

```
export REB_DIR=/Users/dtamayo/rebound
```

4.2.2 Quick Start Guide

We assume we've already set up a `reb_simulation` called `sim`. We always begin by attaching REBOUNDx to our simulation:

```
struct rebx_extras* rebx = rebx_attach(sim);
```

We then add the effect we are interested in. There are two types of effects, forces and operators. The easiest is to check the documentation for the effect you're interested in on the *Astrophysical Effects* page, which links to a C example that shows you how to add it and set the relevant parameters. For example, to add post-Newtonian forces:

```
struct rebx_force* gr = rebx_load_force(rebx, "gr");
```

This returns a rebx_force pointer. Different effects have their own set of parameters that must be set, listed on *Astrophysical Effects* and explained in the examples. For example, general relativity effects require us to set the speed of light (always in the units we're using in the rest of the simulation).

Simple parameters of type double can be set like this:

```
rebx_set_param_double(rebx, &gr->ap, "c", 3e4);
```

Additional parameters (ap) are stored in a linked list for each force, operator and particle, so in addition to passing the rebx instance, we always pass a pointer to the head of the ap linked list (&gr->ap). We then pass the name of the parameter, and the value we want to set it to.

The same goes for a particle. For example, to set a semimajor axis damping timescale tau_a for the modify_orbits_forces effect to particles[1]:

```
rebx_set_param_double(rebx, &sim->particles[1].ap, "tau_a", -1.e4);
```

We can check the value of a particle's parameter by getting a pointer to that parameter with:

```
double* c = rebx_get_param(rebx, gr->ap, "c");
```

Note that if the parameter name (here "c") is not found, rebx_get_param will return NULL. So check for NULL before dereferencing it, or you will get undefined behavior/seg faults.

Once you've set up your force, you still have to add it to the simulation. You do that with:

```
rebx_add_force(rebx, force);
```

You can add as many effects as you'd like in the same simulation. Once you're done setting up all the effects you want, just run the REBOUND simulation as usual:

```
reb_simulation_integrate(sim, tmax);
```

Probably the quickest way to get up and running is to modify an existing example for your effect. You can find links to the appropriate examples here: *Astrophysical Effects*, along with implementation descriptions. You can find the example files in the `reboundx/examples` folder.

Even if you are using the C version, you might also take a look at the python example links at *Astrophysical Effects*, as the iPython notebooks nicely incorporate text and may therefore have a bit longer discussions about the physical details for each effect.

4.3 Astrophysical Effects

Below are descriptions for each of the effects included in REBOUNDx. Different implementations for the same effect are grouped together. All effects follow the same recipes for usage, see the Python quick-start guide (*Quick Start Guide*) or C quick-start guide (*Quick Start Guide*). Probably the quickest way to get up and running is to edit one of the linked examples for the effect you're interested in.

4.3.1 Orbit Modifications

REBOUNDx offers two ways of modifying orbital elements (semimajor axis/eccentricity/inclination damping, precession, etc.) In both cases, each particle is assigned evolution timescales for each orbital element. Positive timescales correspond to growth / progression, negative timescales correspond to damping / regression. Semimajor axes, eccentricities and inclinations grow / damp exponentially. Pericenters and nodes progress/regress linearly.

inner_disk_edge

Authors	Kajtazi, Kaltrina and D. Petit, C. Antoine
Implementation Paper	Kajtazi et al 2022.
Based on	Pichierri et al 2018.
C example	*Inner disk edge.*
Python example	InnerDiskEdge.ipynb.

This applies an inner disk edge that functions as a planet trap. Within its width the planet's migration is reversed by an opposite and roughly equal magnitude torque. Thus, stopping further migration and trapping the planet within the width of the trap. The functions here provide a way to modify the tau_a timescale in modify_orbits_forces, modify_orbit_direct, and type_I_migration. Note that the present prescription is very useful for simple simulations when an inner trap is needed during the migration but it shouldn't be considered as a realistic model of the inner edge of a disk.

Effect Parameters

Field (C type)	Required	Description
ide_position (double)	Yes	The position of the inner disk edge in code units
ide_width (double)	Yes	The disk edge width (planet will stop within ide_width of ide_position)

exponential_migration

Author	Mohamad Ali-Dib
Implementation Paper	Ali-Dib et al., 2021 AJ.
Based on	Hahn & Malhotra 2005.
C Example	*Exponential Migration*
Python Example	ExponentialMigration.ipynb.

Continuous velocity kicks leading to exponential change in the object's semimajor axis. One of the standard prescriptions often used in Neptune migration & Kuiper Belt formation models. Does not directly affect the eccentricity or inclination of the object.

Particle Parameters

Field (C type)	Required	Description
em_tau_a (double)	Yes	Semimajor axis exponential growth/damping timescale
em_aini (double)	Yes	Object's initial semimajor axis
em_afin (double)	Yes	Object's final semimajor axis

modify_orbits_forces

Authors	D. Tamayo, H. Rein
Implementation Paper	Kostov et al., 2016.
Based on	Papaloizou & Larwood 2000.
C Example	*Migration and other orbit modifications*
Python Example	Migration.ipynb EccAndIncDamping.ipynb.

This applies physical forces that orbit-average to give exponential growth/decay of the semimajor axis, eccentricity and inclination. The eccentricity damping keeps the angular momentum constant (corresponding to *p=1* in modify_orbits_direct), which means that eccentricity damping will induce some semimajor axis evolution. Additionally, eccentricity/inclination damping will induce pericenter/nodal precession. Both these effects are physical, and the method is more robust for strongly perturbed systems.

Effect Parameters

If coordinates not, defaults to using Jacobi coordinates.

Field (C type)	Required	Description
coordinates (enum)	No	Type of elements to use for modification (Jacobi, barycentric or particle). See the examples for usage.

Particle Parameters

One can pick and choose which particles have which parameters set. For each particle, any unset parameter is ignored.

Field (C type)	Required	Description
tau_a (double)	No	Semimajor axis exponential growth/damping timescale
tau_e (double)	No	Eccentricity exponential growth/damping timescale
tau_inc (double)	No	Inclination axis exponential growth/damping timescale

type_I_migration

Authors	Kajtazi, Kaltrina and D. Petit, C. Antoine
Implementation Paper	Kajtazi et al 2022.
Based on	Cresswell & Nelson 2008, and Pichierri et al 2018.
C example	*Type I migration.*
Python example	TypeIMigration.ipynb.

This applies Type I migration, damping eccentricity, angular momentum and inclination. The base of the code is the same as the modified orbital forces one written by D. Tamayo, H. Rein. It also allows for parameters describing an inner disc edge, modeled using the implementation in inner_disk_edge.c. Note that this code is not machine independent since power laws were not possible to avoid all together.

Effect Parameters

Field (C type)	Required	Description
ide_position (double)	No	The position of the inner disk edge in code units
ide_width (double)	No	The disk edge width (planet will stop within ide_width of ide_position)
tIm_surface_density_1 (double)	Yes	Disk surface density at one code unit from the star; used to find the surface density at any distance from the star
tIm_scale_height_1 (double)	Yes	The scale height at one code unit from the star; used to find the aspect ratio at any distance from the star
tIm_surface_density_expone (double)	Yes	Exponent of disk surface density, indicative of the surface density profile of the disk
tIm_flaring_index (double)	Yes	The flaring index; 1 means disk is irradiated by only the stellar flux

modify_orbits_direct

Authors	D. Tamayo
Implementation Paper	Tamayo, Rein, Shi and Hernandez, 2019.
Based on	Lee & Peale 2002.
C Example	*Migration and other orbit modifications*
Python Example	Migration.ipynb, EccAndIncDamping.ipynb.

This updates particles' positions and velocities between timesteps to achieve the desired changes to the osculating orbital elements (exponential growth/decay for a, e, inc, linear progression/regression for Omega/omega. This nicely isolates changes to particular osculating elements, making it easier to interpret the resulting dynamics. One can also adjust the coupling parameter *p* between eccentricity and semimajor axis evolution, as well as whether the damping is done on Jacobi, barycentric or heliocentric elements. Since this method changes osculating (i.e., two-body) elements, it can give unphysical results in highly perturbed systems.

Effect Parameters

If p is not set, it defaults to 0. If coordinates not set, defaults to using Jacobi coordinates.

Field (C type)	Required	Description
p (double)	No	Coupling parameter between eccentricity and semimajor axis evolution (see Deck & Batygin 2015). *p=0* corresponds to no coupling, *p=1* to eccentricity evolution at constant angular momentum.
coordinates (enum)	No	Type of elements to use for modification (Jacobi, barycentric or particle). See the examples for usage.

Particle Parameters

One can pick and choose which particles have which parameters set. For each particle, any unset parameter is ignored.

Field (C type)	Required	Description
tau_a (double)	No	Semimajor axis exponential growth/damping timescale
tau_e (double)	No	Eccentricity exponential growth/damping timescale
tau_inc (double)	No	Inclination axis exponential growth/damping timescale
tau_Omega (double)	No	Period of linear nodal precession/regression
tau_omega (double)	No	Period of linear apsidal precession/regression

4.3.2 General Relativity

gr_potential

Authors	H. Rein, D. Tamayo
Implementation Paper	Tamayo, Rein, Shi and Hernandez, 2019.
Based on	Nobili and Roxburgh 1986.
C Example	*Post-Newtonian correction from general relativity*
Python Example	GeneralRelativity.ipynb.

This is the simplest potential you can use for general relativity. It assumes that the masses are dominated by a single central body. It gets the precession right, but gets the mean motion wrong by $\mathcal{O}(GM/ac^2)$. It's the fastest option, and because it's not velocity-dependent, it automatically keeps WHFast symplectic. Nice if you have a single-star system, don't need to get GR exactly right, and want speed.

Effect Parameters

Field (C type)	Required	Description
c (double)	Yes	Speed of light, needs to be specified in the units used for the simulation.

lense_thirring

Authors	A. Akmal
Implementation Paper	None
Based on	Park et al..
C Example	*Adding Lense-Thirring effect*
Python Example	LenseThirring.ipynb.

Adds Lense-Thirring effect due to rotating central body in the simulation. Assumes the source body is particles[0]

Effect Parameters

Field (C type)	Required	Description
lt_c (double)	Yes	Speed of light in the units used for the simulation.

Particle Parameters

Field (C type)	Required	Description
I (double)	Yes	Moment of Inertia of source body
Omega (reb_vec3d)	Yes	Angular rotation frequency (Omega_x, Omega_y, Omega_z)

gr

Authors	P. Shi, D. Tamayo, H. Rein
Implementation Paper	Tamayo, Rein, Shi and Hernandez, 2019.
Based on	Anderson et al. 1975.
C Example	*Post-Newtonian correction from general relativity*
Python Example	GeneralRelativity.ipynb.

This assumes that the masses are dominated by a single central body, and should be good enough for most applications with planets orbiting single stars. It ignores terms that are smaller by of order the mass ratio with the central body. It gets both the mean motion and precession correct, and will be significantly faster than *gr_full*, particularly with several bodies. Adding this effect to several bodies is NOT equivalent to using gr_full.

Effect Parameters

Field (C type)	Required	Description
c (double)	Yes	Speed of light, needs to be specified in the units used for the simulation.

gr_full

Authors	P. Shi, H. Rein, D. Tamayo
Implementation Paper	Tamayo, Rein, Shi and Hernandez, 2019.
Based on	Newhall et al. 1983.
C Example	*Post-Newtonian correction from general relativity*
Python Example	GeneralRelativity.ipynb.

This algorithm incorporates the first-order post-newtonian effects from all bodies in the system, and is necessary for multiple massive bodies like stellar binaries.

Effect Parameters

Field (C type)	Required	Description
c (double)	Yes	Speed of light, needs to be specified in the units used for the simulation.

Particle Parameters

None

4.3.3 Radiation Forces

radiation_forces

Authors	H. Rein, D. Tamayo
Implementation Paper	Tamayo, Rein, Shi and Hernandez, 2019.
Based on	Burns et al. 1979.
C Example	*Radiation forces on a debris disk*, *Radiation forces on circumplanetary dust*.
Python Example	Radiation_Forces_Debris_Disk.ipynb, Radiation_Forces_Circumplanetary_Dust.ipynb.

This applies radiation forces to particles in the simulation. It incorporates both radiation pressure and Poynting-Robertson drag. Only particles whose *beta* parameter is set will feel the radiation.

Effect Parameters

Field (C type)	Required	Description
c (double)	Yes	Speed of light in the units used for the simulation.

Particle Parameters

If no particles have radiation_source set, effect will assume the particle at index 0 in the particles array is the source.

Field (C type)	Required	Description
radiation_source (int)	No	Flag identifying the particle as the source of radiation.
beta (float)	Yes	Ratio of radiation pressure force to gravitational force. Particles without beta set feel no radiation forces.

yarkovsky_effect

Authors	Noah Ferich, D. Tamayo
Implementation Paper	Ferich et al., in prep.
Based on	Veras et al., 2015, Veras et al., 2019.
C Example	*Yarkovsky effect on a small body*.
Python Example	YarkovskyEffect.ipynb.

Adds the accelerations and orbital perturbations created by the Yarkovsky effect onto one or more bodies in the simulation. There are two distinct versions of this effect that can be used: the 'full version' and the 'simple version'. The full version uses the full equations found in Veras et al. (2015) to accurately calculate the Yarkovsky effect on a particle. However, this version slows down simulations and requies a large amount of parameters. For these reasons, the simple version of the effect (based on Veras et al. (2019)) is available. While the magnitude of the acceleration created by the effect will be the same, this version places constant values in a crucial rotation matrix to simplify the push from the Yarkovsky effect on a body. This version is faster and requires less parameters and can be used to get an upper bound on how much the Yarkovsky effect can push an object's orbit inwards or outwards. The lists below describes which parameters are needed for one or both versions of this effect. For more information, please visit the papers and examples linked above.

Effect Parameters

Field (C type)	Required	Description
ye_lstar (float)	Yes	Luminosity of sim's star (Required for both versions).
ye_c (float)	Yes	Speed of light (Required for both versions).
ye_stef_boltz (float)	No	Stefan-Boltzmann constant (Required for full version).

Particle Parameters

Field (C type)	Required	Description
particles[i].r (float)	Yes	Physical radius of a body (Required for both versions).
ye_flag (int)	Yes	0 sets full version of effect. 1 uses simple version with outward migration. -1 uses the simple version with inward migration (see examples and paper).
ye_body_densit (float)	Yes	Density of an object (Required for both versions)
ye_rotation_peri (float)	No	Rotation period of a spinning object (Required for full version)
ye_albedo (float)	Yes	Albedo of an object (Reuired for both versions)
ye_emissivity (float)	No	Emissivity of an object (Required for full version)
ye_thermal_iner (float)	No	Thermal inertia of an object (Required for full version)
ye_k (float)	No	A constant that gets a value between 0 and 1/4 based on the object's rotation - see Veras et al. (2015) for more information on it (Required for full version)
ye_spin_axis_x (float)	No	The x value for the spin axis vector of an object (Required for full version)
ye_spin_axis_y (float)	No	The y value for the spin axis vector of an object (Required for full version)
ye_spin_axis_z (float)	No	The z value for the spin axis vector of an object (Required for full version)

4.3.4 Stochastic Forces

stochastic_forces

Authors	H. Rein
Based on	Rein and Papaloizou 2009.
Implementation Paper	Rein and Choksi 2022.
C Example	*Stochastic forces on a single planet*
Python Example	StochasticForces.ipynb, StochasticForcesCartesian.ipynb,

This applies stochastic forces to particles in the simulation.

Effect Parameters

None

Particle Parameters

All particles which have the field kappa set, will experience stochastic forces. The particle with index 0 cannot experience stochastic forces.

Field (C type)	Required	Description
kappa (double)	Yes	Strength of stochastic forces relative to gravity from central object
tau_kappa (double)	No	Auto-correlation time of stochastic forces. Defaults to orbital period if not set. The units are relative to the current orbital period.

4.3.5 Mass Modifications

modify_mass

Authors	D. Tamayo
Implementation Paper	Kostov et al., 2016.
Based on	None
C Example	*Exponential mass loss/gain*
Python Example	ModifyMass.ipynb.

This adds exponential mass growth/loss to individual particles every timestep. Set particles' `tau_mass` parameter to a negative value for mass loss, positive for mass growth.

Effect Parameters

None

Particle Parameters

Only particles with their `tau_mass` parameter set will have their masses affected.

Name (C type)	Required	Description
tau_mass (double)	Yes	e-folding mass loss (<0) or growth (>0) timescale

4.3.6 Tides

tides_spin

Authors	Tiger Lu, Hanno Rein, D. Tamayo, Sam Hadden, Rosemary Mardling, Sarah Millholland, Gregory Laughlin
Implementation Paper	Lu et al., 2023.
Based on	Eggleton et al. 1998.
C Example	*Pseudo-Synchronization (Hut 1981)*, *Obliquity Sculpting of Kepler Multis (Millholland & Laughlin 2019)*, *Kozai cycles*.
Python Example	SpinsIntro.ipynb, TidesSpinPseudoSynchronization.ipynb, TidesSpinEarthMoon.ipynb.

This effect consistently tracks both the spin and orbital evolution of bodies under constant-time lag tides raised on both the primary and on the orbiting bodies. In all cases, we need to set masses for all the particles that will feel these tidal forces. Particles with only mass are point particles.

Particles are assumed to have structure (i.e - physical extent & distortion from spin) if the following parameters are set: physical radius particles[i].r, potential Love number of degree 2 k2 (Q/(1-Q) in Eggleton 1998), and the spin angular rotation frequency vector Omega. If we wish to evolve a body's spin components, the fully dimensional moment of inertia I must be set as well. If this parameter is not set, the spin components will be stationary. Note that if the body is a test particle, this is assumed to be the specific moment of inertia. Finally, if we wish to consider the effects of tides raised on a specific body, we must set the constant time lag tau as well.

For spins that are synchronized with a circular orbit, the constant time lag can be related to the tidal quality factor Q as tau = 1/(2*n*tau), with n the orbital mean motion. See Lu et. al (in review) and Eggleton et. al (1998) above for discussion.

Effect Parameters

None

Particle Parameters

Field (C type)	Required	Description
particles[i].r (float)	Yes	Physical radius (required for contribution from tides raised on the body).
k2 (float)	Yes	Potential Love number of degree 2.
Omega (reb_vec3d)	Yes	Angular rotation frequency (Omega_x, Omega_y, Omega_z)
I (float)	No	Moment of inertia (for test particles, assumed to be the specific MoI I/m)
tau (float)	No	Constant time lag. If not set, defaults to 0

tides_constant_time_lag

Authors	Stanley A. Baronett, D. Tamayo, Noah Ferich
Implementation Paper	Baronett et al., 2022.
Based on	Hut 1981, Bolmont et al., 2015.
C Example	*Constant time lag model for tides (Hut 1981)*.
Python Example	TidesConstantTimeLag.ipynb.

This adds constant time lag tidal interactions between orbiting bodies in the simulation and the primary, both from tides raised on the primary and on the other bodies. In all cases, we need to set masses for all the particles that will feel these tidal forces. After that, we can choose to include tides raised on the primary, on the "planets", or both, by setting the respective bodies' physical radius particles[i].r, k2 (potential Love number of degree 2), constant time lag tau, and rotation rate Omega. See Baronett et al. (2021), Hut (1981), and Bolmont et al. 2015 above.

If tau is not set, it will default to zero and yield the conservative piece of the tidal potential.

Effect Parameters

None

Particle Parameters

Field (C type)	Required	Description
particles[i].r (float)	Yes	Physical radius (required for contribution from tides raised on the body).
tctl_k2 (float)	Yes	Potential Love number of degree 2.
tctl_tau (float)	No	Constant time lag. If not set will default to 0 and give conservative tidal potential.
OmegaMag (float)	No	Angular rotation frequency. If not set will default to 0.

tides_dynamical

Authors	D. Liveoak, S. Millholland, M. Vick, D. Tamayo
Implementation Paper	Liveoak et al., 2025.
Based on	Vick et al. 2019.
C Example	*Chaotic dynamical tides model*
Python Example	TidesDynamical.ipynb.

This updates body's orbital and modal evolution due to the presence of dynamical tides. Particles are modeled by a gamma=2 polytrope, and the f-mode is evolved at each pericentre passage. The dissipation of orbital energy due to

dynamical tides is modeled as an angular momentum-conserving kick at periapse. When mode energy grows to exceed *td_E_max*, it is non-linearly dissipated in one orbital period to *td_E_resid*. To isolate the effects of chaotic model evolution, one can set *dP_hat_crit* to disable dynamical tides whenever chaos is unlikely (see Vick et al. (2019)). Implementation is only applied to particles[1] in the simulation.

Effect Parameters

Field (C type)	Required	Description
td_disruption_flag (int)	No	Raise error if a planet becomes tidally disrupted (default:0)

Particle Parameters

Field (C type)	Required	Description
particles[1].m (float)	Yes	Mass
particles[1].r (float)	Yes	Physical radius
td_E_max (float)	No	Threshold mode energy for non-linear dissipation (default: 0.1 * E_bind)
td_E_resid (float)	No	Residual mode energy after non-linear dissipation (default: 0.001 * E_bind)
td_c_real (float)	No	Real component of mode (default: 0)
td_c_imag (float)	No	Imaginary component of mode (default: 0)
td_dP_crit (float)	No	Critical change in mode phase to enable dynamical tides (default: 0)

4.3.7 Central Force

central_force

Authors	D. Tamayo
Implementation Paper	Tamayo, Rein, Shi and Hernandez, 2019.
Based on	None
C Example	*General central force.*
Python Example	CentralForce.ipynb.

Adds a general central acceleration of the form a=Acentral*r^gammacentral, outward along the direction from a central particle to the body. Effect is turned on by adding Acentral and gammacentral parameters to a particle, which will act as the central body for the effect, and will act on all other particles.

Effect Parameters

None

Particle Parameters

Field (C type)	Required	Description
Acentral (double)	Yes	Normalization for central acceleration.
gammacentral (double)	Yes	Power index for central acceleration.

4.3.8 Gravity Fields

gravitational_harmonics

Authors	M. Broz
Implementation Paper	Tamayo, Rein, Shi and Hernandez, 2019.
Based on	None
C Example	*Adding gravitational harmonics (J2, J4) to particles*
Python Example	J2.ipynb.

Allows the user to add azimuthally symmetric gravitational harmonics (J2, J4) to bodies in the simulation. These interact with all other bodies in the simulation (treated as point masses). The implementation allows the user to specify an arbitrary spin axis orientation for each oblate body, which defines the axis of symmetry. This is specified through the angular rotation rate vector Omega. The rotation rate Omega is not currently used other than to specify the spin axis orientation. In particular, the current implementation applies the appropriate torque from the body's oblateness to the orbits of all the other planets, but does not account for the equal and opposite torque on the body's spin angular momentum. The bodies spins therefore remain constant in the current implementation. This is a good approximation in the limit where the bodies' spin angular momenta are much greater than the orbital angular momenta involved.

Effect Parameters

None

Particle Parameters

Field (C type)	Required	Description
J2 (double)	No	J2 coefficient
J4 (double)	No	J4 coefficient
R_eq (double)	No	Equatorial radius of nonspherical body used for calculating Jn harmonics
Omega (reb_vec3d)	No	Angular rotation frequency (Omega_x, Omega_y, Omega_z)

4.3.9 Gas Effects

gas_dynamical_friction

Authors	A. Generozov, H. Perets
Implementation Paper	Generozov and Perets 2022
Based on	Ostriker 1999 (with simplifications), Just et al 2012.
C Example	*Gas dynamical friction*
Python Example	GasDynamicalFriction.ipynb

Effect Parameters

Field (C type)	Required	Description
rhog (double)	Yes	Normalization of density. Density in the disk midplane is rhog*r^alpha_rhog
alpha_rhog (double)	Yes	Power-law slope of the power-law density profile.
cs (double)	Yes	Normalization of the sound speed. Sound speed has profile cs*r^alpha_cs
alpha_cs (double)	Yes	Power-law slope of the sound speed
xmin (double)	Yes	Dimensionless parameter that determines the Coulomb logarithm (ln(L) =log (1/xmin))
hr (double)	Yes	Aspect ratio of the disk
Qd (double)	Yes	Prefactor for geometric drag

Particle Parameters

None.

gas_damping_timescale

Authors	Phoebe Sandhaus
Implementation Paper	*Sandhaus et al. in prep*
Based on	Dawson et al. 2016 <https://ui.adsabs.harvard.edu/abs/2016ApJ...822...54D/abstract>; Kominami & Ida 2002
C Example	*Gas Damping Timescale Example*
Python Example	GasDampingTimescale.ipynb

This updates particles' positions and velocities between timesteps by first calculating a damping timescale for each individual particle, and then applying the timescale to damp both the eccentricity and inclination of the particle. Note: The timescale of damping should be much greater than a particle's orbital period. The damping force should also be small as compared to the gravitational forces on the particle.

Effect Parameters

Field (C type)	Required	Description
None	•	•

Particle Parameters

Field (C type)	Require	Description
d_factor (double)	Yes	Depletion factor d in Equation 16 from Dawson et al. 2016; d=1 corresponds roughly to the full minimum mass solar nebula with Sigma_gas (surface gas density) = 1700 g cm^-2 at 1 AU [for d=1]; d>1 corresponds to a more depleted nebula
cs_coeff (double)	Yes	Sound speed coefficient; Changing the value will change assumed units; Example: If you are using the units: AU, M_sun, yr –> cs_coeff = 0.272 # AU^(3/4) yr^-1
tau_coeff (double)	Yes	Timescale coefficient; Changing the value will change assumed units; Example: If you are using the units: AU, M_sun, yr –> tau_coeff = 0.003 # yr AU^-2

4.4 Collisions

Below are each of the functions for resolving collisions implemented in REBOUNDx.

4.5 Miscellaneous Utilities

Below are various miscellaneous utilities implemented in REBOUNDx.

4.5.1 track_min_distance

Authors	D. Tamayo
Implementation Paper	Tamayo, Rein, Shi and Hernandez, 2019.
Based on	None
C Example	*Tracking a particle's minimum distance from the central star.*
Python Example	TrackMinDistance.ipynb.

For a given particle, this keeps track of that particle's minimum distance from another body in the simulation. User should add parameters to the particular particle whose distance should be tracked.

Effect Parameters

None

Particle Parameters

Only particles with their `min_distance` parameter set initially will track their minimum distance. The effect will update this parameter when the particle gets closer than the value of `min_distance`, so the user has to set it initially. By default, distance is measured from sim->particles[0], but you can specify a different particle by setting the `min_distance_from` parameter to the hash of the target particle.

Name (C type)	Required	Description
min_distance (double)	Yes	Particle's mininimum distance.
min_distance_from (uint32)	No	Hash for particle from which to measure distance
min_distance_orbit (reb_orbit)	No	Parameter to store orbital elements at moment corresponding to min_distance (heliocentric)

4.5.2 interpolation

Authors	S.A. Baronett, D. Tamayo, N. Ferich
Implementation Paper	Baronett et al., 2022.
Based on	Press et al., 1992.
C Example	*Stellar evolution with interpolated mass data*
Python Example	ParameterInterpolation.ipynb.

This isn't an effect that's loaded like the others, but an object that facilitates machine-independent interpolation of parameters that can be shared by both the C and Python versions. See the examples for how to use them.

Effect Parameters

Not applicable. See examples.

Particle Parameters

Not applicable. See examples.

4.5.3 steppers

Authors	D. Tamayo, H. Rein
Implementation Paper	Tamayo, Rein, Shi and Hernandez, 2019.
Based on	Rein and Liu, 2012.
C Example	None
Python Example	CustomSplittingIntegrationSchemes.ipynb.

These are wrapper functions to taking steps with several of REBOUND's integrators in order to build custom splitting schemes.

Effect Parameters

None

Particle Parameters

None

4.6 API Documentation (C)

This documents the C API for REBOUNDx. **The main reference point in the documentation is** *Astrophysical Effects* **,which has descriptions for each effect and its parameters, citations, and links to examples.**

REBOUNDx API definition.

Author
Dan Tamayo tamayo.daniel@gmail.com, Hanno Rein

4.6.1 LICENSE

Copyright (c) 2019 Dan Tamayo, Hanno Rein

This file is part of reboundx.

reboundx is free software: you can redistribute it and/or modify it 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.

reboundx is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with rebound. If not, see http://www.gnu.org/licenses/.

Main REBOUNDx Functions

struct *rebx_extras* ***rebx_attach**(struct reb_simulation *sim)

Adds REBOUNDx functionality to a passed REBOUND simulation.

Parameters
sim – Pointer to the reb_simulation on which to add REBOUNDx functionality.

Returns
Pointer to a *rebx_extras* structure.

void **rebx_detach**(struct reb_simulation *sim, struct *rebx_extras* *rebx)

Detaches REBOUNDx from simulation, resetting all the simulation's function pointers that REBOUNDx has set.

This does not free the memory allocated by REBOUNDx (call rebx_free).

Parameters
sim – Pointer to the simulation from which to remove REBOUNDx

void **rebx_extras_cleanup**(struct reb_simulation *sim)

void **rebx_free**(struct *rebx_extras* *rebx)

Frees all memory allocated by REBOUNDx instance.

Should be called after simulation is done if memory is a concern.

Parameters
rebx – The *rebx_extras* pointer returned from the initial call to rebx_attach.

int **rebx_remove_force**(struct *rebx_extras* *rebx, struct *rebx_force* *force)

int **rebx_remove_operator**(struct *rebx_extras* *rebx, struct *rebx_operator* *operator)

void **rebx_output_binary**(struct *rebx_extras* *rebx, char *filename)

Save a binary file with all the effects in the simulation, as well as all particle and effect parameters.

Parameters

- **rebx** – Pointer to the *rebx_extras* instance
- **filename** – Filename to which to save the binary file.

struct *rebx_extras* ***rebx_create_extras_from_binary**(struct reb_simulation *sim, const char *const filename)

Reads a REBOUNDx binary file, loads all effects and parameters.

Parameters

- **sim** – Pointer to the simulation to which the effects and parameters should be added.
- **filename** – Filename of the saved binary file.

void **rebx_init_extras_from_binary**(struct *rebx_extras* *rebx, const char *const filename, enum *rebx_input_binary_messages* *warnings)

Similar to *rebx_create_extras_from_binary()*, but takes an extras instance (must be attached to a simulation) and allows for manual message handling.

Parameters

- **rebx** – Pointer to a *rebx_extras* instance to be updated.
- **filename** – Filename of the saved binary file.
- **warnings** – Pointer to an array of warnings to be populated during loading.

Functions for manipulating effects in REBOUNDx

int **rebx_add_operator**(struct *rebx_extras* *rebx, struct *rebx_operator* *operator)

Main function for adding effects in REBOUNDx.

Parameters

- **rebx** – Pointer to the *rebx_extras* instance
- **name** – Name of the effect we want to add

Returns
Returns a pointer to a rebx_effect structure for the effect.

int **rebx_add_operator_step**(struct *rebx_extras* *rebx, struct *rebx_operator* *operator, const double dt_fraction, enum *rebx_timing* timing)

int **rebx_add_force**(struct *rebx_extras* *rebx, struct *rebx_force* *force)

struct *rebx_operator* ***rebx_load_operator**(struct *rebx_extras* *const rebx, const char *name)

struct *rebx_force* ***rebx_load_force**(struct *rebx_extras* *const rebx, const char *name)

struct *rebx_operator* ***rebx_create_operator**(struct *rebx_extras* *const rebx, const char *name)

struct *rebx_force* ***rebx_create_force**(struct *rebx_extras* *const rebx, const char *name)

struct rebx_effect ***rebx_add_custom_force**(struct *rebx_extras* *rebx, const char *name, void (*custom_force)(struct reb_simulation *const sim, struct rebx_effect *const effect, struct reb_particle *const particles, const int N), const int force_is_velocity_dependent)

Function for adding a custom force in REBOUNDx.

Parameters

- **rebx** – Pointer to the *rebx_extras* instance
- **name** – String with the name of the custom effect
- **custom_force** – User-implemented function that updates the accelerations of particles.
- **force_is_velocity_dependent** – Should be set to 1 if the custom force uses particle velocities, 0 otherwise

Returns
Returns a pointer to a rebx_effect structure for the effect

struct rebx_effect ***rebx_add_custom_operator**(struct *rebx_extras* *rebx, const char *name, void (*custom_operator)(struct reb_simulation *const sim, struct rebx_effect *const effect, const double dt, enum *rebx_timing* timing))

Function for adding a custom post_timestep_modification in REBOUNDx.

Parameters

- **rebx** – Pointer to the *rebx_extras* instance
- **name** – String with the name of the custom effect
- **custom_ptm** – User-implemented function that updates particles.

Returns
Returns a pointer to a rebx_effect structure for the effect.

struct *rebx_force* ***rebx_get_force**(struct *rebx_extras* *const rebx, const char *const name)

Get a pointer to a force by name.

Parameters

- **rebx** – Pointer to the *rebx_extras* instance

- **effect_name** – Name of the force (string)

Returns
Pointer to the corresponding *rebx_force* structure, or NULL if not found.

struct *rebx_operator* ***rebx_get_operator**(struct *rebx_extras* *const rebx, const char *const name)

Get a pointer to an operator by name.

Parameters

- **rebx** – Pointer to the *rebx_extras* instance
- **effect_name** – Name of the operator (string)

Returns
Pointer to the corresponding *rebx_operator* structure, or NULL if not found.

Functions for accessing and modifying particle and effect parameters.

int **rebx_remove_param**(struct *rebx_node* **apptr, const char *const param_name)

Removes a parameter from a particle or effect.

Parameters

- **object** – Pointer to the particle or effect we want to remove a parameter from.
- **param_name** – Name of the parameter we want to remove.

Returns
1 if parameter found and successfully removed, 0 otherwise.

void ***rebx_get_param**(struct *rebx_extras* *const rebx, struct *rebx_node* *ap, const char *const param_name)

Gets a parameter from a particle or effect.

Parameters

- **ap** – Pointer from which to get the param
- **param_name** – Name of the parameter we want to get (see Effects page at http://reboundx.readthedocs.org)

Returns
A void pointer to the parameter. NULL if not found.

struct *rebx_param* ***rebx_get_param_struct**(struct *rebx_extras* *const rebx, struct *rebx_node* *ap, const char *const param_name)

void **rebx_set_param_pointer**(struct *rebx_extras* *const rebx, struct *rebx_node* **apptr, const char *const param_name, void *val)

void **rebx_set_param_double**(struct *rebx_extras* *const rebx, struct *rebx_node* **apptr, const char *const param_name, double val)

void **rebx_set_param_int**(struct *rebx_extras* *const rebx, struct *rebx_node* **apptr, const char *const param_name, int val)

void **rebx_set_param_uint32**(struct *rebx_extras* *const rebx, struct *rebx_node* **apptr, const char *const param_name, uint32_t val)

void **rebx_set_param_vec3d**(struct *rebx_extras* *const rebx, struct *rebx_node* **apptr, const char *const param_name, struct reb_vec3d val)

void **rebx_register_param**(struct *rebx_extras* *const rebx, const char *name, enum *rebx_param_type* type)

General utility functions

struct reb_vec3d **rebx_tools_spin_angular_momentum**(struct *rebx_extras* *const rebx)

Calculate spin angular momentum in the simulation of any bodies with spin parameters set (moment of inertia I and angular rotation frequency vector Omega).

Parameters

rebx – Pointer to the *rebx_extras* instance

double **rebx_tools_spin_energy**(struct *rebx_extras* *const rebx)

Calculate spin energy in the simulation of any bodies with spin parameters set (moment of inertia I and angular rotation frequency vector Omega).

Parameters

rebx – Pointer to the *rebx_extras* instance

void **rebx_simulation_irotate**(struct *rebx_extras* *const rebx, const struct reb_rotation q)

Convenience Functions for Various Effects

double **rebx_rad_calc_beta**(const double G, const double c, const double source_mass, const double source_luminosity, const double radius, const double density, const double Q_pr)

Calculates beta, the ratio between the radiation pressure force and the gravitational force from the star.

Parameters

- **G** – Gravitational constant.
- **c** – Speed of light.
- **source_mass** – Mass of the source body.
- **source_luminosity** – Luminosity of radiation source.
- **radius** – Particle physical radius.
- **density** – density of particle.
- **Q_pr** – Radiation pressure coefficient (Burns et al. 1979).

Returns

Beta parameter (double).

double **rebx_rad_calc_particle_radius**(const double G, const double c, const double source_mass, const double source_luminosity, const double beta, const double density, const double Q_pr)

Calculates the particle radius from physical parameters and beta, the ratio of radiation to gravitational forces from the star.

Parameters

- **G** – Gravitational constant.
- **c** – Speed of light.
- **source_mass** – Mass of the source body.
- **source_luminosity** – Luminosity of radiation source.
- **beta** – ratio of radiation force to gravitational force from the radiation source body.
- **density** – density of particle.
- **Q_pr** – Radiation pressure coefficient (Burns et al. 1979).

Returns
Particle radius (double).

void **rebx_spin_initialize_ode**(struct *rebx_extras* *const rebx, struct *rebx_force* *const effect)

Count how many particles have their moment of inertia and spin set, and initialize corresponding spin ODEs.

Must be called after setting the moment of inertia and spin of all particles you want to evolve. Attaches spin_ode param to passed effect struct.

Parameters

- **rebx** – Pointer to the *rebx_extras* instance.
- **effect** – (*rebx_force*) Force structure to which to attach spin_ode object.

double **rebx_central_force_Acentral**(const struct reb_particle p, const struct reb_particle primary, const double pomegadot, const double gamma)

Calculates the Aradial parameter for central_force effect required for a particle to have a particular pericenter precession rate.

Parameters

- **p** – Particle whose pericenter precession rate we want to match.
- **primary** – Central particle for the central force (to which we add the Acentral and gammacentral parameters).
- **pomegadot** – Pericenter precession rate we want to obtain.
- **gamma** – Index of the central force law.

Returns
Acentral Normalization to add to the central particle.

double **rebx_gr_hamiltonian**(struct *rebx_extras* *const rebx, const struct *rebx_force* *const gr)

Calculates the hamiltonian for gr, including the classical Hamiltonian.

Assumes there is only one source particle (with gr_source set to 1)

Parameters

- **rebx** – pointer to the REBOUNDx extras instance.
- **gr** – Force structure returned by rebx_load_force

Returns
Total Hamiltonian, including classical Hamiltonian (double).

double **rebx_gr_full_hamiltonian**(struct *rebx_extras* *const rebx, const struct *rebx_force* *const gr_full)

Calculates the hamiltonian for gr_full, including the classical Hamiltonian.

Parameters

- **rebx** – pointer to the REBOUNDx extras instance.
- **gr_full** – Force structure returned by rebx_load_force

Returns
Total Hamiltonian, including classical Hamiltonian (double).

double **rebx_gr_potential_potential**(struct *rebx_extras* *const rebx, const struct *rebx_force* *const gr_potential)

Calculates the potential for gr_potential, including the classical Hamiltonian.

Parameters

- **rebx** – pointer to the REBOUNDx extras instance.
- **gr_potential** – Force structure returned by rebx_load_force

Returns
Total Hamiltonian, including classical Hamiltonian (double).

double **rebx_tides_constant_time_lag_potential**(struct *rebx_extras* *const rebx)

Calculates the potential for the conservative piece of the tides_constant_time_lag effect.

Will be conserved if tctl_tau = 0

Parameters
rebx – pointer to the REBOUNDx extras instance.

Returns
Potential corresponding to tides_constant_time_lag effect.

double **rebx_tides_spin_energy**(struct *rebx_extras* *const rebx)

Calculates the total energy associated with bodies' spin and their tidal interactions in tides_spin effect. This includes the spin kinetic energy plus gravitational potential between the tidally and rotationally induced quadrupoles and other bodies (treated as point masses). You should add sim.energy() to include bodies' overall kinetic energy and point-mass gravitational potential energy.

Parameters
rebx – pointer to the REBOUNDx extras instance.

Returns
Energy associated with tides_spin effect.

double **rebx_central_force_potential**(struct *rebx_extras* *const rebx)

Calculates the potential for central_force effect.

Parameters
rebx – pointer to the REBOUNDx extras instance.

Returns
Potential corresponding to central_force effect.

double **rebx_gravitational_harmonics_potential**(struct *rebx_extras* *const rebx)

Calculates the potential for all particles with additional gravity field harmonics beyond the monopole (i.e., J2, J4).

Parameters
rebx – pointer to the REBOUNDx extras instance.

Returns
Potential corresponding to the effect from all particles of their additional gravity field harmonics

struct *rebx_tides_dynamical_mode* **rebx_calculate_tides_dynamical_mode_evolution**(double old_real, double old_imag, double dc_tilde, double P, double sigma)

Specialized functions for accessing and modifying particle and effect parameters.

void ***rebx_get_param_check**(struct reb_simulation *sim, struct *rebx_node* *ap, const char *const param_name, enum *rebx_param_type* param_type)

Gets the full *rebx_param* structure for a particular parameter, rather than just the pointer to the contents.

This can be useful to check properties of the parameter, like the param_type or shape.

Returns a void pointer to the parameter just like rebx_get_param, but additionally checks that the param_type matches what is expected.

Effects should use this function rather than rebx_get_param to ensure that the user appropriately set parameters if working from Python.

Parameters

- **object** – Pointer to the particle or effect that holds the parameter.
- **param_name** – Name of the parameter we want to get (see Effects page at http://reboundx.readthedocs.org)
- **object** – Pointer to the particle or effect that holds the parameter.
- **param_name** – Name of the parameter we want to get (see Effects page at http://reboundx.readthedocs.org)

Returns

Pointer to the *rebx_param* structure that holds the parameter. NULL if not found.

Returns

Void pointer to the parameter. NULL if not found or type does not match (will write error to stderr).

struct *rebx_param* ***rebx_get_or_add_param**(struct *rebx_extras* *const rebx, struct *rebx_node* **apptr, const char *const param_name)

REBOUND Stepper Functions

void **rebx_ias15_step**(struct reb_simulation *const sim, struct *rebx_operator* *const operator, const double dt)

Executes a step for passed time dt using the IAS15 integrator in REBOUND.

Parameters

- **sim** – Pointer to the simulation to step.
- **operator** – Unused pointer (kept for consistency with other operators). Can pass NULL.
- **dt** – timestep for which to step in simulation time units.

void **rebx_kepler_step**(struct reb_simulation *const sim, struct *rebx_operator* *const operator, const double dt)

Executes a Kepler step for passed time dt using the WHFast integrator in REBOUND.

Will use the coordinates and other options set in sim.ri_whfast

Parameters

- **sim** – Pointer to the simulation to step.
- **operator** – Unused pointer (kept for consistency with other operators). Can pass NULL.
- **dt** – timestep for which to step in simulation time units.

void **rebx_jump_step**(struct reb_simulation *const sim, struct *rebx_operator* *const operator, const double dt)

Executes a jump step for passed time dt using the WHFast integrator in REBOUND.

Will use the coordinates and other options set in sim.ri_whfast

Parameters

- **sim** – Pointer to the simulation to step.

- **operator** – Unused pointer (kept for consistency with other operators). Can pass NULL.
- **dt** – timestep for which to step in simulation time units.

void **rebx_interaction_step**(struct reb_simulation *const sim, struct *rebx_operator* *const operator, const double dt)

Executes an interaction step for passed time dt using the WHFast integrator in REBOUND.

Will use the coordinates and other options set in sim.ri_whfast

Parameters

- **sim** – Pointer to the simulation to step.
- **operator** – Unused pointer (kept for consistency with other operators). Can pass NULL.
- **dt** – timestep for which to step in simulation time units.

void **rebx_drift_step**(struct reb_simulation *const sim, struct *rebx_operator* *const operator, const double dt)

Executes a drift step for passed time dt using the leapfrog integrator in REBOUND.

Parameters

- **sim** – Pointer to the simulation to step.
- **operator** – Unused pointer (kept for consistency with other operators). Can pass NULL.
- **dt** – timestep for which to step in simulation time units.

void **rebx_kick_step**(struct reb_simulation *const sim, struct *rebx_operator* *const operator, const double dt)

Executes a kick step for passed time dt using the leapfrog integrator in REBOUND.

Parameters

- **sim** – Pointer to the simulation to step.
- **operator** – Unused pointer (kept for consistency with other operators). Can pass NULL.
- **dt** – timestep for which to step in simulation time units.

Interpolation Routines

struct *rebx_interpolator* ***rebx_create_interpolator**(struct *rebx_extras* *const rebx, const int Nvalues, const double *times, const double *values, enum *rebx_interpolation_type* interpolation)

Takes an array of times and corresponding array of values and returns a structure that allows interpolation of values at arbitrary times.

See the parameter interpolation examples in C and Python.

Parameters

- **rebx** – pointer to the REBOUNDx extras instance.
- **Nvalues** – Length of times and values arrays (must be equal for both).
- **times** – Array of times at which the corresponding values are supplied.
- **values** – Array of values at each corresponding time.
- **interpolation** – Enum specifying the interpolation method. Defaults to spline.

Returns

Pointer to a *rebx_interpolator* structure. Call rebx_interpolate to get values.

void **rebx_free_interpolator**(struct *rebx_interpolator* *const interpolator)

Frees the memory for a *rebx_interpolator* structure.

double **rebx_interpolate**(struct *rebx_extras* *const rebx, struct *rebx_interpolator* *const interpolator, const double time)

Interpolate value at arbitrary times.

Need to first rebx_create_interpolator with an array of times and corresponding values to interpolate between. See parameter interpolation examples.

Parameters

- **rebx** – Pointer to the REBOUNDx extras instance.
- **interpolator** – Pointer to the *rebx_interpolator* structure to interpolate from.
- **time** – Time at which to interpolate value.

Returns

Interpolated value at passed time.

Testing Functions

FILE ***rebx_input_inspect_binary**(const char *const filename, enum *rebx_input_binary_messages* *warnings)

Loads a binary file, reads the header, and gives back the file pointer for manual reading.

Parameters

- **filename** – File to open
- **warnings** – Pointer to warnings enum to store warnings that come up

Returns

Returns a pointer to the binary file at the position following the header

struct *rebx_binary_field* **rebx_input_read_binary_field**(FILE *inf)

Read the next field in a binary file.

Parameters

inf – Pointer to the input file

Returns

Returns *rebx_binary_field* struct, initialized to 0 if read fails

void **rebx_input_skip_binary_field**(FILE *inf, long field_size)

Skip forward in binary file.

Parameters

- **inf** – Pointer to the input file
- **field_size** – Length by which to skip from current file position

Defines

M_PI

REBXGITHASH

Enums

enum **rebx_param_type**

Data types available in REBOUNDx for parameters.

Values:

enumerator **REBX_TYPE_NONE**

enumerator **REBX_TYPE_DOUBLE**

enumerator **REBX_TYPE_INT**

enumerator **REBX_TYPE_POINTER**

enumerator **REBX_TYPE_FORCE**

enumerator **REBX_TYPE_UINT32**

enumerator **REBX_TYPE_ORBIT**

enumerator **REBX_TYPE_ODE**

enumerator **REBX_TYPE_VEC3D**

enum **REBX_COORDINATES**

Different coordinate systems.

Values:

enumerator **REBX_COORDINATES_JACOBI**

Jacobi coordinates (default)

enumerator **REBX_COORDINATES_BARYCENTRIC**

Coordinates referenced to pos/vel of system's center of mass.

enumerator **REBX_COORDINATES_PARTICLE**

Coordinates relative to pos/vel of a particular particle.

enum **rebx_timing**

Flag for whether steps should happen before or after the timestep.

Values:

enumerator **REBX_TIMING_PRE**

Pre timestep.

enumerator **REBX_TIMING_POST**

Post timestep.

enum **rebx_force_type**

Force types.

Values:

enumerator **REBX_FORCE_NONE**

Uninitialized default.

enumerator **REBX_FORCE_POS**

Force derivable from a position-dependent potential.

enumerator **REBX_FORCE_VEL**

velocity (or pos and vel) dependent force

enum **rebx_operator_type**

Operator types.

Values:

enumerator **REBX_OPERATOR_NONE**

Uninitialized default.

enumerator **REBX_OPERATOR_UPDATER**

operator that modifies x,v or m,

enumerator **REBX_OPERATOR_RECORDER**

operator that leaves state unchanged. Just records

enum **rebx_binary_field_type**

Different fields for binary files.

Values:

enumerator **REBX_BINARY_FIELD_TYPE_NONE**

enumerator **REBX_BINARY_FIELD_TYPE_OPERATOR**

enumerator **REBX_BINARY_FIELD_TYPE_PARTICLE**

enumerator **REBX_BINARY_FIELD_TYPE_REBX_STRUCTURE**

enumerator **REBX_BINARY_FIELD_TYPE_PARAM**

enumerator `REBX_BINARY_FIELD_TYPE_NAME`

enumerator `REBX_BINARY_FIELD_TYPE_PARAM_TYPE`

enumerator `REBX_BINARY_FIELD_TYPE_PARAM_VALUE`

enumerator `REBX_BINARY_FIELD_TYPE_END`

enumerator `REBX_BINARY_FIELD_TYPE_PARTICLE_INDEX`

enumerator `REBX_BINARY_FIELD_TYPE_REBX_INTEGRATOR`

enumerator `REBX_BINARY_FIELD_TYPE_FORCE_TYPE`

enumerator `REBX_BINARY_FIELD_TYPE_OPERATOR_TYPE`

enumerator `REBX_BINARY_FIELD_TYPE_STEP`

enumerator `REBX_BINARY_FIELD_TYPE_STEP_DT_FRACTION`

enumerator `REBX_BINARY_FIELD_TYPE_REGISTERED_PARAM`

enumerator `REBX_BINARY_FIELD_TYPE_ADDITIONAL_FORCE`

enumerator `REBX_BINARY_FIELD_TYPE_PARAM_LIST`

enumerator `REBX_BINARY_FIELD_TYPE_REGISTERED_PARAMETERS`

enumerator `REBX_BINARY_FIELD_TYPE_ALLOCATED_FORCES`

enumerator `REBX_BINARY_FIELD_TYPE_ALLOCATED_OPERATORS`

enumerator `REBX_BINARY_FIELD_TYPE_ADDITIONAL_FORCES`

enumerator `REBX_BINARY_FIELD_TYPE_PRE_TIMESTEP_MODIFICATIONS`

enumerator `REBX_BINARY_FIELD_TYPE_POST_TIMESTEP_MODIFICATIONS`

enumerator `REBX_BINARY_FIELD_TYPE_PARTICLES`

enumerator `REBX_BINARY_FIELD_TYPE_FORCE`

enumerator **REBX_BINARY_FIELD_TYPE_SNAPSHOT**

enum **rebx_input_binary_messages**

Possible errors that might occur during binary file reading.

Values:

enumerator **REBX_INPUT_BINARY_WARNING_NONE**

enumerator **REBX_INPUT_BINARY_ERROR_NOFILE**

enumerator **REBX_INPUT_BINARY_ERROR_CORRUPT**

enumerator **REBX_INPUT_BINARY_ERROR_NO_MEMORY**

enumerator **REBX_INPUT_BINARY_ERROR_REBX_NOT_LOADED**

enumerator **REBX_INPUT_BINARY_ERROR_REGISTERED_PARAM_NOT_LOADED**

enumerator **REBX_INPUT_BINARY_WARNING_PARAM_NOT_LOADED**

enumerator **REBX_INPUT_BINARY_WARNING_PARTICLE_PARAMS_NOT_LOADED**

enumerator **REBX_INPUT_BINARY_WARNING_FORCE_NOT_LOADED**

enumerator **REBX_INPUT_BINARY_WARNING_OPERATOR_NOT_LOADED**

enumerator **REBX_INPUT_BINARY_WARNING_STEP_NOT_LOADED**

enumerator **REBX_INPUT_BINARY_WARNING_ADDITIONAL_FORCE_NOT_LOADED**

enumerator **REBX_INPUT_BINARY_WARNING_FIELD_UNKNOWN**

enumerator **REBX_INPUT_BINARY_WARNING_LIST_UNKNOWN**

enumerator **REBX_INPUT_BINARY_WARNING_PARAM_VALUE_NULL**

enumerator **REBX_INPUT_BINARY_WARNING_VERSION**

enumerator **REBX_INPUT_BINARY_WARNING_FORCE_PARAM_NOT_LOADED**

enum **rebx_integrator**

Different schemes for integrating across the interaction step.

Values:

enumerator **REBX_INTEGRATOR_NONE**

enumerator **REBX_INTEGRATOR_IMPLICIT_MIDPOINT**

enumerator **REBX_INTEGRATOR_RK4**

enumerator **REBX_INTEGRATOR_EULER**

enumerator **REBX_INTEGRATOR_RK2**

enum **rebx_interpolation_type**

Different interpolation options.

Values:

enumerator **REBX_INTERPOLATION_NONE**

enumerator **REBX_INTERPOLATION_SPLINE**

Functions

void **rebx_error**(struct *rebx_extras* *rebx, const char *const msg)

Variables

const char ***rebx_build_str**

Date and time build string.

const char ***rebx_version_str**

Version string.

const char ***rebx_githash_str**

Current git hash.

struct **rebx_node**

#include <reboundx.h> Node structure for all REBOUNDx linked lists.

Public Members

void ***object**

Pointer to object (param, force, step, etc)

struct *rebx_node* ***next***

Pointer to next node in list.

struct **rebx_param**

#include <reboundx.h> Main structure used for all parameters added to objects.

Public Members

char ***name***

For searching linked lists and informative errors.

enum *rebx_param_type* **type**

Needed to cast value.

void ***value***

Pointer to parameter value.

struct **rebx_force**

#include <reboundx.h> Structure for REBOUNDx forces.

Public Members

char ***name***

For searching linked lists and informative errors.

struct *rebx_node* ***ap***

Additional parameters linked list.

struct reb_simulation ***sim***

Pointer to attached sim. Needed for error checks.

enum *rebx_force_type* **force_type**

Force type for internal logic.

void (***update_accelerations**)(struct reb_simulation *const sim, struct *rebx_force* *const force, struct reb_particle *const particles, const int N)

Function pointer to add additional accelerations.

struct **rebx_operator**

#include <reboundx.h> Structure for REBOUNDx operators.

Public Members

char ***name***

For searching linked lists and informative errors.

struct *rebx_node* ***ap**

Additional parameters linked list.

struct reb_simulation ***sim**

Pointer to attached sim. Needed for error checks.

enum *rebx_operator_type* **operator_type**

Operator type for internal logic.

void (***step_function**)(struct reb_simulation *sim, struct *rebx_operator* *operator, const double dt)

Function pointer to execute step.

struct **rebx_step**

#include <reboundx.h> Structure for a REBOUNDx step.

A step is just a combination of an operator with a fraction of a timestep (see Sec. 6 of REBOUNdx paper). Can use same operator for different steps of different lengths to build higher order splitting schemes.

Public Members

struct rebx_operator * operator

Pointer to operator to use.

double **dt_fraction**

Fraction of sim.dt to use each time it's called.

struct **rebx_binary_field**

#include <reboundx.h> Structure used as building block to save and load binary files.

Public Members

enum *rebx_binary_field_type* **type**

Type of object.

long **size**

Size in bytes of the object data (not including this structure). So you can skip ahead.

struct **rebx_interpolator**

Public Members

enum *rebx_interpolation_type* **interpolation**

double ***times**

double ***values**

int **Nvalues**

double ***y2**

int **klo**

struct **rebx_extras**

#include <reboundx.h> Main REBOUNDx structure.

These fields are used internally by REBOUNDx and generally should not be changed manually by the user. Use the API instead.

Public Members

struct reb_simulation ***sim**

Pointer to the simulation REBOUNDx is linked to.

struct *rebx_node* ***additional_forces**

Linked list of extra forces.

struct *rebx_node* ***pre_timestep_modifications**

Linked list of rebx_steps to apply before each timestep.

struct *rebx_node* ***post_timestep_modifications**

Linked list of rebx_steps to apply after each timestep.

struct *rebx_node* ***registered_params**

Linked list of rebx_params with all the parameter names registered with their type (for type safety)

struct *rebx_node* ***allocated_forces**

For memory management.

struct *rebx_node* ***allocated_operators**

For memory management.

struct **rebx_tides_dynamical_params**

Public Members

double **dP**

double **dE_alpha**

double **sigma**

struct **rebx_tides_dynamical_mode**

Public Members

double **real**

double **imag**

char **mode**

4.7 Examples (C)

4.7.1 Adding Post-Newtonian correction from general relativity as an operator.

This example shows how to add post-newtonian corrections to REBOUND simulations as an operator (see REBOUNDx paper and corresponding IntegrateForce.ipynb jupyter notebook for more details). If you have GLUT installed for the visualization, press 'w' and/or 'c' for a clearer view of the whole orbit.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <math.h>
#include "rebound.h"
#include "reboundx.h"

double E0;
void heartbeat(struct reb_simulation* sim);

int main(int argc, char* argv[]){
    // We first set up a system similar to the EPIC system discussed in the REBOUNDx␣
↪paper
    struct reb_simulation* sim = reb_simulation_create();
    sim->G = 4*M_PI*M_PI;

    struct reb_particle star = {0};
    star.m     = 0.93;
    reb_simulation_add(sim, star);

    double m = 1.35e-6;
    double a = 0.013; // put planet close to enhance precession so it's visible in␣
↪visualization (this would put planet inside the Sun!)
    double e = 0.01;
    double inc = 0.;
    double Omega = 0.;
    double omega = 0.;
    double f = 0.;

    struct reb_particle planet1 = reb_particle_from_orbit(sim->G, star, m, a, e, inc,␣
↪Omega, omega, f);

    m = 1.2e-5;
    a = 0.107; // put planet close to enhance precession so it's visible in visualization␣
↪(this would put planet inside the Sun!)
```

(continues on next page)

(continued from previous page)

```
    e = 0.01;

    struct reb_particle planet2= reb_particle_from_orbit(sim->G, star, m, a, e, inc,␣
↪Omega, omega, f);
    reb_simulation_add(sim, planet1);
    reb_simulation_add(sim, planet2);
    reb_simulation_move_to_com(sim);

    sim->dt = 1.e-4;
    sim->heartbeat = heartbeat;
    sim->integrator = REB_INTEGRATOR_WHFAST;

    // Now we add GR. This is a velocity dependent force. With WHFast this would cause␣
↪errors on long timescales, so we integrate the force in a separate step
    // See the REBOUNDx paper and the corresponding example in ipython_examples for more␣
↪details
    // By adding a separate force step for GR and integrating across it we keep an␣
↪oscillatory energy error

    struct rebx_extras* rebx = rebx_attach(sim);
    struct rebx_force* gr = rebx_load_force(rebx, "gr");
    rebx_set_param_double(rebx, &gr->ap, "c", 63197.8); // AU/yr

    struct rebx_operator* integrateforce = rebx_load_operator(rebx, "integrate_force");
    rebx_set_param_pointer(rebx, &integrateforce->ap, "force", gr);
    rebx_set_param_int(rebx, &integrateforce->ap, "integrator", REBX_INTEGRATOR_RK2);
    rebx_add_operator(rebx, integrateforce);

    double tmax = 10.;
    E0 = rebx_gr_hamiltonian(rebx, gr);
    reb_simulation_integrate(sim, tmax);
    rebx_free(rebx);    // this explicitly frees all the memory allocated by REBOUNDx
    reb_simulation_free(sim);
}

void heartbeat(struct reb_simulation* sim){
    if(reb_simulation_output_check(sim, 0.1)){
        struct rebx_force* gr = rebx_get_force(sim->extras, "gr");
        double E = rebx_gr_hamiltonian(sim->extras, gr);
        printf("t=%f\tEnergy Error=%e\n", sim->t, fabs((E-E0)/E0));
    }
}
```

This example is located in the directory *examples/integrate_force*

4.7.2 Type I migration.

This example shows how to add Type I migration. See detailed annotations and explanations for the parameters in the TypeIMigration ipython example and Implemented Effects documentation for REBOUNDx.

```
#include <stdio.h>
#include <string.h>
```

(continues on next page)

(continued from previous page)

```
#include "rebound.h"
#include "reboundx.h"
#include "core.h"

int main(int argc, char* argv[]){
    struct reb_simulation* sim = reb_simulation_create();
    /* sim units are ('yr', 'AU', 'Msun') */
    sim->G = 4*M_PI*M_PI;

    struct reb_particle star = {0};
    star.m     = 1.;
    reb_simulation_add(sim, star);

    double m = 0.00001;
    double a1 = 0.5;
    double a2 = 0.85;
    double e = 0;
    double inc = 0.;
    double Omega = 0.;
    double omega = 0.;
    double f = 0.;

    struct reb_particle p1 = reb_particle_from_orbit(sim->G, star, m, a1, e, inc, Omega,␣
↪omega, f);
    struct reb_particle p2 = reb_particle_from_orbit(sim->G, star, m, a2, e, inc, Omega,␣
↪omega, f);
    reb_simulation_add(sim, p1);
    reb_simulation_add(sim, p2);
    reb_simulation_move_to_com(sim);

    sim->dt = 0.002;  //The period at inner disk edge divided by 20, for a disk edge␣
↪location at 0.1 AU
    sim->integrator = REB_INTEGRATOR_WHFAST;

    struct rebx_extras* rebx = rebx_attach(sim);

    struct rebx_force* type_I_mig = rebx_load_force(rebx, "type_I_migration");
    rebx_set_param_double(rebx, &type_I_mig->ap, "ide_position", 0.3);
    rebx_set_param_double(rebx, &type_I_mig->ap, "ide_width", 0.02); //Calculated using␣
↪the scale height value given below
    rebx_set_param_double(rebx, &type_I_mig->ap, "tIm_flaring_index", 0.25);
    rebx_set_param_double(rebx, &type_I_mig->ap, "tIm_surface_density_exponent", 1);
    rebx_set_param_double(rebx, &type_I_mig->ap, "tIm_surface_density_1", 0.00011255); //
↪In units of solarmass/(AU^2) which is roughly 1000g/cm^2
    rebx_set_param_double(rebx, &type_I_mig->ap, "tIm_scale_height_1", 0.03);
    rebx_add_force(rebx, type_I_mig);

    double tmax = 1.e4;  // Note that one can calculate the timescale of the semi-major␣
↪axis of the outer planet then set the integration time to twice this value

    reb_simulation_integrate(sim, tmax);
    rebx_free(rebx);
```

(continues on next page)

(continued from previous page)

```
    reb_simulation_free(sim);
}
```

This example is located in the directory *examples/type_I_migration*

4.7.3 General central force.

This example shows how to add a general central force. If you have GLUT installed for the visualization, press 'w' and/or 'c' for a clearer view of the whole orbit.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <math.h>
#include "rebound.h"
#include "reboundx.h"

int main(int argc, char* argv[]){
    struct reb_simulation* sim = reb_simulation_create();

    struct reb_particle star = {0};
    star.m     = 1.;
    reb_simulation_add(sim, star);

    double m = 0.;
    double a = 1.; // put planet close to enhance precession so it's visible in␣
↪visualization (this would put planet inside the Sun!)
    double e = 0.2;
    double inc = 0.;
    double Omega = 0.;
    double omega = 0.;
    double f = 0.;

    struct reb_particle planet = reb_particle_from_orbit(sim->G, star, m, a, e, inc,␣
↪Omega, omega, f);
    reb_simulation_add(sim, planet);
    reb_simulation_move_to_com(sim);

    struct rebx_extras* rebx = rebx_attach(sim);
    struct rebx_force* force = rebx_load_force(rebx, "central_force");
    rebx_add_force(rebx, force);

    /* We first choose a power for our central force (here F goes as r^-1).
     * We then need to add it to the particle(s) that will act as central sources for␣
↪this force.*/

    struct reb_particle* ps = sim->particles;
    double gammacentral = -1.;
    rebx_set_param_double(rebx, &ps[0].ap, "gammacentral", gammacentral);

    // The other parameter to set is the normalization Acentral (F=Acentral*r^
↪gammacentral). E.g.,
```

(continues on next page)

(continued from previous page)

```
    rebx_set_param_double(rebx, &ps[0].ap, "Acentral", 1.e-4);

    /* We can also use the function rebx_central_force_Acentral to calculate the
↪Acentral required
     * for particles[1] (around primary particles[0]) to have a pericenter precession
↪rate of
     * pomegadot, given a gammacentral value: */

    double pomegadot = 1.e-3;
    double Acentral = rebx_central_force_Acentral(ps[1], ps[0], pomegadot, gammacentral);
    rebx_set_param_double(rebx, &ps[0].ap, "Acentral", Acentral);

    double tmax = 3.e4;
    reb_simulation_integrate(sim, tmax);
    rebx_free(rebx);    // this explicitly frees all the memory allocated by REBOUNDx
    reb_simulation_free(sim);
}
```

This example is located in the directory *examples/central_force*

4.7.4 Tracking a particle's minimum distance from the central star.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <math.h>
#include "rebound.h"
#include "reboundx.h"

int main(int argc, char* argv[]){
    struct reb_simulation* sim = reb_simulation_create();
    struct rebx_extras* rebx = rebx_attach(sim);

    struct reb_particle star = {0};
    star.m     = 1.;
    star.hash  = reb_hash("star");
    reb_simulation_add(sim, star);

    double m = 0.;
    double a = 1.;
    double e = 0.9;
    double inc = 0.;
    double Omega = 0.;
    double omega = 0.;
    double f = M_PI;

    struct reb_particle planet = reb_particle_from_orbit(sim->G, star, m, a, e, inc,
↪Omega, omega, f);
    reb_simulation_add(sim, planet);
    reb_simulation_move_to_com(sim);
```

(continues on next page)

(continued from previous page)

```
    struct rebx_operator* track_min_distance = rebx_load_operator(rebx, "track_min_
↪distance");
    rebx_add_operator(rebx, track_min_distance);

    // Wee add a min_distance parameter to the particle whose distance we want to track,␣
↪and set it
    // to a particular value. In any timestep that the distance drops below this value,␣
↪the value is updated.
    rebx_set_param_double(rebx, &sim->particles[1].ap, "min_distance", 5.);

    // By default distance is measured from sim->particles[0].  We can specify a␣
↪different particle by a hash (unnecessary here):

    rebx_set_param_uint32(rebx, &sim->particles[1].ap, "min_distance_from", sim->
↪particles[0].hash);


    struct reb_orbit orbit = {0};
    rebx_set_param_pointer(rebx, &sim->particles[1].ap, "min_distance_orbit", &orbit);

    double tmax = 10.;
    reb_simulation_integrate(sim, tmax);

    // At any point in the integration, we can check the `min_distance` parameter and␣
↪output it as needed.
    double* min_distance = rebx_get_param(rebx, sim->particles[1].ap, "min_distance");
    struct reb_orbit* orbitptr = rebx_get_param(rebx, sim->particles[1].ap, "min_
↪distance_orbit");

    printf("Particle's minimum distance from the star over the integration = %e\n", *min_
↪distance);
    printf("Semimajor axis and eccentricity at closest approach: a=%.2f, e=%.2f\n",␣
↪orbitptr->a, orbitptr->e);

    rebx_free(rebx);    // this explicitly frees all the memory allocated by REBOUNDx
    reb_simulation_free(sim);
}
```

This example is located in the directory *examples/track_min_distance*

4.7.5 Adding Lense-Thirring effect

This example shows how to add the Lense-Thirring effect to a simulation. If you have GLUT installed for the visualization, press 'w' and/or 'c' for a clearer view of the whole orbit.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <math.h>
#include "rebound.h"
#include "reboundx.h"
```

(continues on next page)

(continued from previous page)

```
int main(int argc, char* argv[]){
    struct reb_simulation* sim = reb_simulation_create();
    sim->G = 4*M_PI*M_PI; // units of AU, yr, Msun
    struct reb_particle star = {0};
    star.m     = 1.;
    reb_simulation_add(sim, star);
    double omega = 90.361036076; //solar rotation rate in rad/year
    double C_I = 0.06884; //solar moment of inertia prefactor
    double R_eq = 0.00465247264; //solar equatorial radius in AU

    struct reb_particle planet = {0};  // add a planet on a circular orbit (with default␣
↪units where G=1)
    const double mp = 1.7e-7;   // approximate values for mercury in units of Msun and AU
    const double a = 0.39;
    const double e = 0.21;
    reb_simulation_add_fmt(sim, "m a e", mp, a, e);

    struct rebx_extras* rebx = rebx_attach(sim);  // first initialize rebx
    struct rebx_force* lense  = rebx_load_force(rebx, "lense_thirring"); // add our new␣
↪force
    rebx_add_force(rebx, lense);
    rebx_set_param_vec3d(rebx, &sim->particles[0].ap, "Omega", (struct reb_vec3d){.x=0, .
↪y=0, .z=omega});
    rebx_set_param_double(rebx, &sim->particles[0].ap, "I", C_I*star.m*R_eq*R_eq);

    // Have to set speed of light in right units (set by G & initial conditions).  Here␣
↪we use default units of AU/(yr/2pi)
    rebx_set_param_double(rebx, &lense->ap, "lt_c", 63241.077); // speed of light in AU/
↪yr

    double tmax = 1000.;
    reb_simulation_integrate(sim, tmax);
    struct reb_orbit orb = reb_orbit_from_particle(sim->G, sim->particles[1], sim->
↪particles[0]);
    printf("Pericenter precession rate = %.3f arcsec/century\n", orb.pomega*206265/10.);
}
```

This example is located in the directory *examples/lense_thirring*

4.7.6 Kozai cycles

This example uses the IAS15 integrator to simulate a Lidov Kozai cycle of a planet perturbed by a distant star. The integrator automatically adjusts the timestep so that even very high eccentricity encounters are resolved with high accuracy.

This is the same Kozai example implemented in Lu et. al (2023) Also, see the ipython examples prefixed TidesSpin for in-depth exploration of the parameters that can be set in this simulation.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <math.h>
#include "rebound.h"
```

(continues on next page)

(continued from previous page)

```
#include "reboundx.h"
#include "tides_spin.c"

void heartbeat(struct reb_simulation* r);
double tmax = 1e5; // kept short to run quickly.
                   // set to 3e5 for a full cycle
                   // or 7e6 * 2 * M_PI to reproduce the paper plot

int main(int argc, char* argv[]){
    struct reb_simulation* sim = reb_simulation_create();
    // Initial conditions
    // Setup constants
    sim->dt                 = M_PI*1e-1;     // initial timestep
    sim->integrator         = REB_INTEGRATOR_IAS15; // IAS15 is used for its adaptive␣
↪timestep:
                                                   // in a Kozai cycle the planet␣
↪experiences close encounters during the high-eccentricity epochs.
                                                   // A fixed-time integrator (for␣
↪example, WHFast) would need to apply the worst-case timestep to the whole simulation
    sim->heartbeat          = heartbeat;

    // Initial conditions
    struct reb_particle star = {0};
    star.m  = 1;
    star.r = 0.00465;
    reb_simulation_add(sim, star);

    // struct reb_particle planet = {0};
    double planet_m  = 0.054 * 9.55e-4; // A Jupiter-like planet
    double planet_r = 0.3 * 4.676e-4;
    double planet_a = 2.;
    double planet_e = 0.001;
    reb_simulation_add_fmt(sim, "m r a e", planet_m, planet_r, planet_a, planet_e);

    // The perturber - treated as a point particle
    double perturber_m  = 1;
    double perturber_a = 50.;
    double perturber_e = 0.7 * M_PI / 180.;
    double perturber_inc = 80. * M_PI / 180.;
    reb_simulation_add_fmt(sim, "m a e inc", perturber_m, perturber_a, perturber_e,␣
↪perturber_inc);

    // Add REBOUNDx effects
    // First tides_spin
    struct rebx_extras* rebx = rebx_attach(sim);

    struct rebx_force* effect = rebx_load_force(rebx, "tides_spin");
    rebx_add_force(rebx, effect);


    // Sun
    const double solar_k2 = 0.07;
```

(continues on next page)

(continued from previous page)

```
    rebx_set_param_double(rebx, &sim->particles[0].ap, "k2", solar_k2);
    rebx_set_param_double(rebx, &sim->particles[0].ap, "I", 0.07 * star.m * star.r *␣
↪star.r);

    const double solar_spin_period = 4.6 * 2. * M_PI / 365.;
    const double solar_spin = (2 * M_PI) / solar_spin_period;
    rebx_set_param_vec3d(rebx, &sim->particles[0].ap, "Omega", (struct reb_vec3d){.
↪z=solar_spin}); // Omega_x = Omega_y = 0 by default

    const double solar_Q = 1e6;
    struct reb_orbit orb = reb_orbit_from_particle(sim->G, sim->particles[1], sim->
↪particles[0]);
    // In the case of a spin that is synchronous with a circular orbit, tau is related␣
↪to the tidal quality factor Q through the orbital mean motion n (see Lu et al. 2023␣
↪for discussion). Clearly that's not the case here, but gives us a reasonable starting␣
↪point to start turning this knob
    double solar_tau = 1 / (2 * solar_Q * orb.n);
    rebx_set_param_double(rebx, &sim->particles[0].ap, "tau", solar_tau);

    // Planet
    const double planet_k2 = 0.4;
    rebx_set_param_double(rebx, &sim->particles[1].ap, "k2", planet_k2);
    rebx_set_param_double(rebx, &sim->particles[1].ap, "I", 0.25 * planet_m * planet_r *␣
↪planet_r);

    const double spin_period_p = 1. * 2. * M_PI / 365.; // days to reb years
    const double spin_p = (2. * M_PI) / spin_period_p;
    const double theta_p = 0. * M_PI / 180.;
    const double phi_p = 0. * M_PI / 180;
    struct reb_vec3d Omega_sv = reb_tools_spherical_to_xyz(spin_p, theta_p, phi_p);
    rebx_set_param_vec3d(rebx, &sim->particles[1].ap, "Omega", Omega_sv);

    const double planet_Q = 3e5;
    rebx_set_param_double(rebx, &sim->particles[1].ap, "tau", 1./(2.*planet_Q*orb.n));


    // add GR precession:
    struct rebx_force* gr = rebx_load_force(rebx, "gr_potential");
    rebx_add_force(rebx, gr);
    rebx_set_param_double(rebx, &gr->ap, "c", 10065.32); // in default units

    reb_simulation_move_to_com(sim);

    // Let's create a reb_rotation object that rotates to new axes with newz pointing␣
↪along the total ang. momentum, and x along the line of
    // nodes with the invariable plane (along z cross newz)
    struct reb_vec3d newz = reb_vec3d_add(reb_simulation_angular_momentum(sim), rebx_
↪tools_spin_angular_momentum(rebx));
    struct reb_vec3d newx = reb_vec3d_cross((struct reb_vec3d){.z =1}, newz);
    struct reb_rotation rot = reb_rotation_init_to_new_axes(newz, newx);
    rebx_simulation_irotate(rebx, rot); // This rotates our simulation into the␣
↪invariable plane aligned with the total ang. momentum (including spin)
```

(continues on next page)

(continued from previous page)

```
    rebx_spin_initialize_ode(rebx, effect);

    system("rm -v output.txt");        // delete previous output file
    reb_simulation_integrate(sim, tmax);

    rebx_free(rebx);
    reb_simulation_free(sim);
}

void heartbeat(struct reb_simulation* sim){
    // Output spin and orbital information to file
    if(reb_simulation_output_check(sim, 10)){        // outputs every 10 REBOUND years
      struct rebx_extras* const rebx = sim->extras;
      FILE* of = fopen("output.txt", "a");
      if (of==NULL){
          reb_simulation_error(sim, "Can not open file.");
          return;
      }

      struct reb_particle* sun = &sim->particles[0];
      struct reb_particle* p1 = &sim->particles[1];
      struct reb_particle* pert = &sim->particles[2];

      // orbits
      struct reb_orbit o1 = reb_orbit_from_particle(sim->G, *p1, *sun);
      double a1 = o1.a;
      double e1 = o1.e;
      double i1 = o1.inc;
      double Om1 = o1.Omega;
      double pom1 = o1.pomega;
      struct reb_vec3d n1 = o1.hvec;

      struct reb_particle com = reb_particle_com_of_pair(sim->particles[0],sim->
→particles[1]);
      struct reb_orbit o2 = reb_orbit_from_particle(sim->G, *pert, com);
      double a2 = o2.a;
      double e2 = o2.e;
      double i2 = o2.inc;
      double Om2 = o2.Omega;
      double pom2 = o2.pomega;

      struct reb_vec3d* Omega_sun = rebx_get_param(rebx, sun->ap, "Omega");

      // Interpret planet spin in the rotating planet frame
      struct reb_vec3d* Omega_p_inv = rebx_get_param(rebx, p1->ap, "Omega");

      // Transform spin vector into planet frame, w/ z-axis aligned with orbit normal␣
→and x-axis aligned with line of nodes
      struct reb_vec3d line_of_nodes = reb_vec3d_cross((struct reb_vec3d){.z =1}, n1);
      struct reb_rotation rot = reb_rotation_init_to_new_axes(n1, line_of_nodes); //␣
→Arguments to this function are the new z and x axes
```

(continues on next page)

(continued from previous page)

```
        struct reb_vec3d srot = reb_vec3d_rotate(*Omega_p_inv, rot); // spin vector in the␣
↪planet's frame

        // Interpret the spin axis in the more natural spherical coordinates
        double mag_p;
        double theta_p;
        double phi_p;
        reb_tools_xyz_to_spherical(srot, &mag_p, &theta_p, &phi_p);

        fprintf(of, "%e,%e,%e,%e,%e,%e,%e,%e,%e,%e,%e,%e,%e,%e,%e,%e,%e\n", sim->t, Omega_
↪sun->x, Omega_sun->y, Omega_sun->z, mag_p, theta_p, phi_p, a1, e1, i1, Om1, pom1, a2,␣
↪e2, i2, Om2, pom2); // print spins and orbits

        fclose(of);
    }

    if(reb_simulation_output_check(sim, 20.*M_PI)){        // outputs to the screen
        reb_simulation_output_timing(sim, tmax);
    }
}
```

This example is located in the directory *examples/tides_spin_kozai*

4.7.7 Constant time lag model for tides (Hut 1981)

In particular, this simulates post-main sequence tidal interactions between the Earth and Sun near its tip-RGB phase. Definitely see the corresponding ipython example, as well as the documentation, for more explanations along the way of the various parameters and assumptions.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <math.h>
#include "rebound.h"
#include "reboundx.h"

int main(int argc, char* argv[]){
    struct reb_simulation* sim = reb_simulation_create();
    sim->G = 4*M_PI*M_PI;           // Units of AU, yr and Msun

    struct reb_particle sun = {0};
    sun.m = 0.86;                   // post-MS in Msun
    reb_simulation_add(sim, sun);

    struct reb_orbit eo = {0};      // for Earth
    eo.a = 1.0;                     // in AU
    eo.e = 0.03;
    double e_mass = 2.988e-6;       // if planets don't have mass, they don't raise any␣
↪tides
    struct reb_particle ep = reb_particle_from_orbit(sim->G, sun, e_mass, eo.a, eo.e, eo.
↪inc, eo.Omega, eo.omega, eo.f);
    reb_simulation_add(sim, ep);
```

(continues on next page)

(continued from previous page)

```
    reb_simulation_move_to_com(sim);

    // Add REBOUNDx Additional Effect
    struct rebx_extras* rebx = rebx_attach(sim);
    struct rebx_force* tides = rebx_load_force(rebx, "tides_constant_time_lag");
    rebx_add_force(rebx, tides);

    // We first have to give the body being tidally distorted a physical radius, or␣
↪nothing will happen
    sim->particles[0].r = 0.85;     // AU

    // At a minimum, have to set tctl_k2 (potential Love number of degree 2) parameter,␣
↪which will add the conservative potential of the equilibrium tidal distortion
    rebx_set_param_double(rebx, &sim->particles[0].ap, "tctl_k2", 0.03);

    // We add dissipation by adding a constant time lag tctl_tau
    rebx_set_param_double(rebx, &sim->particles[0].ap, "tctl_tau", 0.04);

    // We also can set the angular rotation rate of bodies OmegaMag. Implementation␣
↪assumes Omega is along z axis
    // If you need a more general implementation, see the tides_spin effect.␣
↪Implementation will assume zero if not specified.

    rebx_set_param_double(rebx, &sim->particles[0].ap, "OmegaMag", 2*M_PI/(30./365.)); //
↪ 30 day rotation rate
    //exit(1);
    // Run simulation
    double tmax = 2.5e5; // years
    reb_simulation_integrate(sim, tmax);
    rebx_free(rebx);
    reb_simulation_free(sim);
}
```

This example is located in the directory *examples/tides_constant_time_lag*

4.7.8 Stellar evolution with interpolated mass data

This example shows how to change a particle's mass by interpolating time-series data during a REBOUND simulation. If you have GLUT installed for visualization, press 'w' to see the orbits as wires. You can zoom out by holding shift, holding down the mouse and dragging. Press 'c' to better see migration/e-damping.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <math.h>
#include "rebound.h"
#include "reboundx.h"

void heartbeat(struct reb_simulation* sim);
struct rebx_interpolator* stellarmass;
struct rebx_extras* rebx;
double tmax = 1e4;
```

(continues on next page)

(continued from previous page)

```
int main(int argc, char* argv[]) {
     struct reb_simulation* sim = reb_simulation_create();
    sim->G = 4*M_PI*M_PI; // use units of AU, yr and solar masses
     sim->heartbeat = heartbeat;
     sim->integrator = REB_INTEGRATOR_WHFAST;

     struct reb_particle sun = {0};
     sun.m    = 1.;
     reb_simulation_add(sim, sun);
     // Initialize planets on circular orbits, each 2 times farther than last.
     struct reb_particle planet = {0};
     planet.x = 1.;
     planet.vy = 2.*M_PI;
     reb_simulation_add(sim, planet);
     planet.x *= 2.;
     planet.vy /= sqrt(2.);
     reb_simulation_add(sim, planet);
     planet.x *= 2.;
     planet.vy /= sqrt(2.);
     reb_simulation_add(sim, planet);

     reb_simulation_move_to_com(sim);
     rebx = rebx_attach(sim); // initialize reboundx

     // To set how the mass will change, we pass three equally-sized arrays
     // necessary to interpolate the time-mass values. Here we have six (6)
     // values that correspond to a star losing mass with an e-damping timescale
     // of tmax (1e4) up to 12,500 yr. The effect will use a cubic spline to
     // interpolate any intermediate values needed by the simulation.
    int n = 6; // size of arrays
     double times[] = {0, 2500, 5000, 7500, 10000, 12500}; // in yr
     double values[] = {1., 0.77880078307, 0.60653065971, 0.47236655274, 0.36787944117,␣
↪0.28650479686}; // in Msun
    stellarmass = rebx_create_interpolator(rebx, n, times, values, REBX_INTERPOLATION_
↪SPLINE);

    reb_simulation_integrate(sim, tmax);
    rebx_free_interpolator(stellarmass);
     rebx_free(rebx); // explicitly free all the memory allocated by REBOUNDx
}

void heartbeat(struct reb_simulation* sim) {
     if (reb_simulation_output_check(sim, tmax/1000)) {
        sim->particles[0].m = rebx_interpolate(rebx, stellarmass, sim->t);
        reb_simulation_move_to_com(sim);
        struct reb_orbit o = reb_orbit_from_particle(sim->G, sim->particles[1], sim->
↪particles[0]);
        printf("t=%e, Sun mass = %f, planet semimajor axis = %f\n", sim->t, sim->
↪particles[0].m, o.a);
     }
}
```

This example is located in the directory *examples/parameter_interpolation*

4.7.9 Pseudo-Synchronization (Hut 1981)

This example shows how to add quadrupole and tidal distortions to bodies with structure, letting us consistently track the spin-axis and dynamical evolution of the system. In particular, this simulates the pseudo-synchronization of a fiducial hot Jupiter. The hot Jupiter is initialized with a slight eccentricity, nontrivial obliquity and fast rotation. Under the influence of tidal dissipation, we see the following rapidly occur: circularization of the orbit, obliquity damping down to 0, and rotation settling to the pseudo-synchronous value predicted by Hut (1981) Definitely see the corresponding ipython example, as well as the documentation, for more in-depth explanations regarding the various parameters that may be set in this simulation. Also see Lu et. al (in review), Eggleton et. al (1998), Hut (1981).

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <math.h>
#include "rebound.h"
#include "reboundx.h"
#include "tides_spin.c"

void heartbeat(struct reb_simulation* sim);
double tmax = 1000 * 2 * M_PI;

int main(int argc, char* argv[]){
    struct reb_simulation* sim = reb_simulation_create();

    // Star
    const double solar_mass = 1.;
    const double solar_rad = 0.00465; // Bodies with structure require radius! This is␣
↪one solar radius
    reb_simulation_add_fmt(sim, "m r", solar_mass, solar_rad);// Central object

    // Fiducial hot Jupiter
    const double p1_mass = 1. * 9.55e-4; // in Jupiter masses * 1 Jupiter Mass / 1 Solar␣
↪Mass
    const double p1_rad = 1. * 4.676e-4; // in Jupiter rad * 1 jupiter rad / 1 AU
    const double p1_e = 0.01;
    const double p1_inc = 0.01;
    reb_simulation_add_fmt(sim, "m a e inc r", p1_mass, 0.04072, p1_e, p1_inc, p1_rad); /
↪/ Planet 1

    sim->N_active = 2;
    sim->integrator = REB_INTEGRATOR_WHFAST;
    sim->dt = 1e-3;
    sim->heartbeat = heartbeat;

    // Add tides_spin as a REBOUNDx additional effect
    struct rebx_extras* rebx = rebx_attach(sim);
    struct rebx_force* effect = rebx_load_force(rebx, "tides_spin");
    rebx_add_force(rebx, effect);
    // Star
    // The following parameters are mandatory to set for the body to have structure:
    // The Love number (k2)
    // The three components of the anguarl spin frequency (Omega_x, Omega_y, Omega_z)
    const double solar_k2 = 0.07;
    rebx_set_param_double(rebx, &sim->particles[0].ap, "k2", solar_k2);
```

(continues on next page)

(continued from previous page)

```
    const double solar_spin_period = 27 * 2 * M_PI / 365; // 27 Days in REBOUND time␣
↪units
    const double solar_spin = (2 * M_PI) / solar_spin_period;
    rebx_set_param_vec3d(rebx, &sim->particles[0].ap, "Omega", (struct reb_vec3d){.
↪z=solar_spin}); // Omega.x = Omega.y = 0 by default

    // While not technically necessary, the fully dimensional moment of inertia (I)␣
↪should also be set
    // This is required to evolve the spin axis! Without setting this value, the spin␣
↪axis will remain stationary.
    rebx_set_param_double(rebx, &sim->particles[0].ap, "I", 0.07 * solar_mass * solar_
↪rad * solar_rad);

    // Finally, the last parameter is the constant time lag tau
    const double solar_Q = 1e6; // Often tidal dissipation is expressed in terms of a␣
↪tidal quality factor Q
    struct reb_orbit orb = reb_orbit_from_particle(sim->G, sim->particles[1], sim->
↪particles[0]);
    // In the case of a spin that is synchronous with a circular orbit, tau is related␣
↪to the tidal quality factor Q through the orbital mean motion n
    double solar_tau = 1 / (2 * solar_Q * orb.n);
    // See Lu et. al (2023) for discussion of the cases in which this approximation␣
↪relating Q and tau is valid

    rebx_set_param_double(rebx, &sim->particles[0].ap, "tau", solar_tau);

    // Planet - all of the above applies here too
    const double spin_period_1 = 0.5 * 2. * M_PI / 365.; // 0.5 days in reb years
    const double spin_1 = (2. * M_PI) / spin_period_1;
    const double planet_Q = 10000.;
    const double theta_1 = 30. * (M_PI / 180.);
    const double phi_1 = 0 * (M_PI / 180);
    rebx_set_param_double(rebx, &sim->particles[1].ap, "k2", 0.3);
    rebx_set_param_double(rebx, &sim->particles[1].ap, "I", 0.25 * p1_mass * p1_rad * p1_
↪rad);

    // Let's consider a tilted planetary spin axis. The spin frequency vector is␣
↪expressed in the same inertial reference frame as the simulation.
    // Given a polar angle theta (from the z axis) and azimuthal angle phi (from the x␣
↪axis), we could either manually set spin axis components:
    struct reb_vec3d Omega_1;
    Omega_1.x = spin_1 * sin(theta_1) * cos(phi_1);
    Omega_1.y = spin_1 * sin(theta_1) * sin(phi_1);
    Omega_1.z = spin_1 * cos(theta_1);

    // Or use the built-in convenience function, which returns the Cartesian coordinates␣
↪of the spin vector given:
    // magnitude, obliquity, and phase angle
    Omega_1 = reb_tools_spherical_to_xyz(spin_1, theta_1, phi_1);

    // For your own use case, you would just use whichever of the above two methods is␣
↪most convenient (here we've done it three ways)
```

(continues on next page)

(continued from previous page)

```
    // whichever of the two methods we use above, we need to remember to set the spin␣
↪axis values
    rebx_set_param_vec3d(rebx, &sim->particles[1].ap, "Omega", Omega_1);
    rebx_set_param_double(rebx, &sim->particles[1].ap, "tau", 1./(2*planet_Q*orb.n));

    reb_simulation_move_to_com(sim);

    // Let's create a reb_rotation object that rotates our simulation to new axes with␣
↪newz pointing along the total ang. momentum, and x along the line of
    // nodes with the invariable plane (along z cross newz)
    struct reb_vec3d newz = reb_vec3d_add(reb_simulation_angular_momentum(sim), rebx_
↪tools_spin_angular_momentum(rebx));
    struct reb_vec3d newx = reb_vec3d_cross((struct reb_vec3d){.z =1}, newz);
    struct reb_rotation rot = reb_rotation_init_to_new_axes(newz, newx);
    rebx_simulation_irotate(rebx, rot); // This rotates our simulation into the␣
↪invariable plane aligned with the total ang. momentum (including spin)
    rebx_spin_initialize_ode(rebx, effect); // We must call this function before␣
↪starting our integration to track the spins

    system("rm -v output.txt"); // remove previous output file
    reb_simulation_integrate(sim, tmax);
    rebx_free(rebx);
    reb_simulation_free(sim);
}

void heartbeat(struct reb_simulation* sim){
    // Output spin and orbital information to file
    if(reb_simulation_output_check(sim, 10)){        // outputs every 10 REBOUND years
      struct rebx_extras* const rebx = sim->extras;
      FILE* of = fopen("output.txt", "a");
      if (of==NULL){
          reb_simulation_error(sim, "Can not open file.");
          return;
      }

      struct reb_particle* star = &sim->particles[0];
      struct reb_particle* p = &sim->particles[1];

      struct reb_orbit orb = reb_orbit_from_particle(sim->G, *p, *star);
      double a = orb.a;
      double Om = orb.Omega;
      double inc = orb.inc;
      double pom = orb.pomega;
      double e = orb.e;

      // The spin vector in the inertial (in this case, invariant frame)
      struct reb_vec3d* Omega_inv = rebx_get_param(rebx, p->ap, "Omega");

      // Transform spin vector into planet frame, w/ z-axis aligned with orbit normal␣
↪and x-axis aligned with line of nodes
      struct reb_vec3d orbit_normal = orb.hvec;
      struct reb_vec3d line_of_nodes = reb_vec3d_cross((struct reb_vec3d){.z =1}, orbit_
```

(continues on next page)

(continued from previous page)

```
↪normal);
        struct reb_rotation rot = reb_rotation_init_to_new_axes(orbit_normal, line_of_
↪nodes); // Arguments to this function are the new z and x axes
        struct reb_vec3d srot = reb_vec3d_rotate(*Omega_inv, rot); // spin vector in the␣
↪planet's frame

        // Interpret the spin axis in the more natural spherical coordinates
        double mag;
        double theta;
        double phi;
        reb_tools_xyz_to_spherical(srot, &mag, &theta, &phi);

        fprintf(of, "%e,%e,%e,%e,%e,%e,%e,%e,%e\n", sim->t, a, inc, e, pom, Om, mag, theta,
↪ phi);
        fclose(of);
    }

    if(reb_simulation_output_check(sim, 20.*M_PI)){        // outputs to the screen
        reb_simulation_output_timing(sim, tmax);
    }
}
```

This example is located in the directory *examples/tides_spin_pseudo_synchronization*

4.7.10 Saving and loading simulations

This example demonstrates how to restart a simulation with all REBOUNDx effects and parameters.

```
#include <stdio.h>
#include <string.h>
#include "rebound.h"
#include "reboundx.h"
#include "core.h"

int main(int argc, char* argv[]){
    struct reb_simulation* sim = reb_simulation_create(); // make a simple sim with star␣
↪and 1 planet
    struct reb_particle p = {0};
    p.m     = 1.;
    reb_simulation_add(sim, p);
    double m = 0.;
    double a = 1.e-4;
    double e = 0.2;
    double inc = 0.;
    double Omega = 0.;
    double omega = 0.;
    double f = 0.;

    struct reb_particle p1 = reb_particle_from_orbit(sim->G, p, m, a, e, inc, Omega,␣
↪omega, f);
    reb_simulation_add(sim, p1);
    sim->dt = 1.e-8;
```

(continues on next page)

(continued from previous page)

```
    sim->integrator = REB_INTEGRATOR_WHFAST;

    struct rebx_extras* rebx = rebx_attach(sim);

    // Add some (arbitrary) parameters to effects and particles

    struct rebx_force* gr = rebx_load_force(rebx, "gr");
    rebx_set_param_double(rebx, &gr->ap, "c", 3e4);
    rebx_set_param_int(rebx, &gr->ap, "gr_source", 0);
    rebx_add_force(rebx, gr);

    struct rebx_operator* mm = rebx_load_operator(rebx, "modify_mass");
    rebx_set_param_double(rebx, &sim->particles[0].ap, "tau_mass", -1.e4);
    rebx_add_operator(rebx, mm);

    struct reb_particle* pptr = &sim->particles[1];
    rebx_set_param_double(rebx, &pptr->ap, "tau_e", 1.5);   // no meaning, just to test␣
↪storage
    rebx_set_param_int(rebx, &pptr->ap, "max_iterations", 42);

    double* c = rebx_get_param(rebx, gr->ap, "c");
    int* gr_source = rebx_get_param(rebx, gr->ap, "gr_source");
    double* tau_mass = rebx_get_param(rebx, sim->particles[0].ap, "tau_mass");
    double* tau_e = rebx_get_param(rebx, sim->particles[1].ap, "tau_e");
    int* max_iterations = rebx_get_param(rebx, sim->particles[1].ap, "max_iterations");

    struct rebx_operator* integforce = rebx_load_operator(rebx, "integrate_force");
    rebx_set_param_pointer(rebx, &integforce->ap, "force", gr);
    rebx_add_operator(rebx, integforce);

    printf("c: Original = %f\n", *c);
    printf("gr_source: Original = %d\n", *gr_source);
    printf("tau_mass: Original = %f\n", *tau_mass);
    printf("tau_e: Original = %f\n", *tau_e);
    printf("max_iterations: Original = %d\n", *max_iterations);

    // We now have to save both a REBOUND binary (for the simulation) and a REBOUNDx one␣
↪(for parameters and effects)
    reb_simulation_integrate(sim, 1.e-4);
    reb_simulation_save_to_file(sim, "reb.bin");
    rebx_output_binary(rebx, "rebx.bin");

    rebx_free(rebx);
    reb_simulation_free(sim);

    // We now reload the simulation and the rebx instance (which adds previously loaded␣
↪effects to the simulation)
    sim = reb_simulation_create_from_file("reb.bin", 0);
    rebx = rebx_create_extras_from_binary(sim, "rebx.bin");

    gr = rebx_get_force(rebx, "gr");
    mm = rebx_get_operator(rebx, "modify_mass");
```

(continues on next page)

(continued from previous page)

```
    c = rebx_get_param(rebx, gr->ap, "c");
    gr_source = rebx_get_param(rebx, gr->ap, "gr_source");
    tau_mass = rebx_get_param(rebx, sim->particles[0].ap, "tau_mass");
    tau_e = rebx_get_param(rebx, sim->particles[1].ap, "tau_e");
    max_iterations = rebx_get_param(rebx, sim->particles[1].ap, "max_iterations");

    printf("c: Loaded = %f\n", *c);
    printf("gr_source: Loaded = %d\n", *gr_source);
    printf("tau_mass: Loaded = %f\n", *tau_mass);
    printf("tau_e: Loaded = %f\n", *tau_e);
    printf("max_iterations: Loaded = %d\n", *max_iterations);

    // You would now integrate as usual
    double tmax = 1.e-4;
    reb_simulation_integrate(sim, tmax);
    rebx_free(rebx);
    reb_simulation_free(sim);
}
```

This example is located in the directory *examples/saving_and_loading_simulations*

4.7.11 Radiation forces on circumplanetary dust

This example shows how to integrate circumplanetary dust particles under the action of radiation forces using IAS15. We use Saturn's Phoebe ring as an example, a distant ring of debris, The output is custom, outputting the semi-major axis of every dust particle relative to the planet.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <math.h>
#include "rebound.h"
#include "reboundx.h"

void heartbeat(struct reb_simulation* r);

double tmax = 1e10;

int main(int argc, char* argv[]){
    struct reb_simulation* sim = reb_simulation_create();
    // Setup constants
    sim->integrator     = REB_INTEGRATOR_IAS15;
    sim->G              = 6.674e-11;    // Use SI units
    sim->dt             = 1e4;          // Initial timestep in sec
    sim->N_active       = 2;            // Only the sun and the planet affect other␣
↪particles gravitationally
    sim->heartbeat      = heartbeat;
    sim->usleep     = 1000;             // Slow down integration (for visualization only)

    // sun
    struct reb_particle sun = {0};
```

(continues on next page)

(continued from previous page)

```
    sun.m  = 1.99e30;                     // mass of Sun in kg
    reb_simulation_add(sim, sun);

    // Saturn (simulation set up in Saturn's orbital plane, i.e., inc=0, so only need 4␣
↪orbital elements
    double mass_sat = 5.68e26;            // Mass of Saturn
    double a_sat = 1.43e12;               // Semimajor axis of Saturn in m
    double e_sat = 0.056;                 // Eccentricity of Saturn
    double pomega_sat = 0.;               // Angle from x axis to pericenter
    double f_sat = 0.;                    // True anomaly of Saturn
    double inc = 0.;
    double Omega = 0.;
    struct reb_particle saturn = reb_particle_from_orbit(sim->G, sun, mass_sat, a_sat, e_
↪sat, inc, Omega, pomega_sat, f_sat);

    reb_simulation_add(sim, saturn);

    // Add REBOUNDx
    struct rebx_extras* rebx = rebx_attach(sim);
    struct rebx_force* rad = rebx_load_force(rebx, "radiation_forces");
    double c = 3.e8;                      // speed of light in SI units
    rebx_set_param_double(rebx, &rad->ap, "c", c);

    // Will assume particles[0] is the radiation source by default. You can also add a␣
↪flag to a particle explicitly
    rebx_set_param_int(rebx, &sim->particles[0].ap, "radiation_source", 1);

    /* Dust particles
     Here we imagine particles launched from Saturn's irregular Satellite Phoebe.
     Such grains will inherit the moon's orbital elements (e.g. Tamayo et al. 2011)

     In order for a particle to feel radiation forces, we have to set their beta␣
↪parameter,
     the ratio of the radiation pressure force to the gravitional force from the star␣
↪(Burns et al. 1979).
     We do this in two ways below.*/

    double a_dust = 1.30e10;              // semimajor axis of satellite Phoebe, in m
    double e_dust = 0.16;                 // eccentricity of Phoebe
    double inc_dust = 175.*M_PI/180.;     // inclination of Phoebe to Saturn's orbital plane
    double Omega_dust = 0.;               // longitude of ascending node
    double omega_dust = 0.;               // argument of pericenter
    double f_dust = 0.;                   // true anomaly

    // We first set up the orbit and add the particles
    double m_dust = 0.;                   // treat dust particles as massless
    struct reb_particle p = reb_particle_from_orbit(sim->G, sim->particles[1], m_dust, a_
↪dust, e_dust, inc_dust, Omega_dust, omega_dust, f_dust);
    reb_simulation_add(sim, p);

    // For the first particle we simply specify beta directly.
    rebx_set_param_double(rebx, &sim->particles[2].ap, "beta", 0.1);
```

(continues on next page)

(continued from previous page)

```
    // We now add a 2nd particle on the same orbit, but set its beta using physical␣
↪parameters.
    struct reb_particle p2 = reb_particle_from_orbit(sim->G, sim->particles[1], 0., a_
↪dust, e_dust, inc_dust, Omega_dust, omega_dust, f_dust);
    reb_simulation_add(sim, p2);

    /* REBOUNDx has a convenience function to calculate beta given the gravitational␣
↪constant G, the star's luminosity and mass, and the grain's physical radius, density␣
↪and radiation pressure coefficient Q_pr (Burns et al. 1979). */

    // Particle parameters
    double radius = 1.e-5;              // in meters
    double density = 1.e3;              // kg/m3 = 1g/cc
    double Q_pr = 1.;                   // Equals 1 in limit where particle radius >>␣
↪wavelength of radiation
    double L = 3.85e26;                 // Luminosity of the sun in Watts

    double beta = rebx_rad_calc_beta(sim->G, c, sim->particles[0].m, L, radius, density,␣
↪Q_pr);
    rebx_set_param_double(rebx, &sim->particles[3].ap, "beta", beta);

    printf("Particle 2 has beta = %f\n", beta);

    reb_simulation_move_to_com(sim);

    printf("Time\t\tEcc (p)\t\tEcc (p2)\n");
    reb_simulation_integrate(sim, tmax);
    rebx_free(rebx);                /* free memory allocated by REBOUNDx */
}

void heartbeat(struct reb_simulation* sim){
    if(reb_simulation_output_check(sim, 1.e8)){
        const struct reb_particle* particles = sim->particles;
        const struct reb_particle saturn = particles[1];
        const double t = sim->t;
        struct reb_orbit orbit = reb_orbit_from_particle(sim->G, sim->particles[2],␣
↪saturn); /* calculate orbit of particles[2] around Saturn */
        double e2 = orbit.e;
        orbit = reb_orbit_from_particle(sim->G, sim->particles[3], saturn);
        double e3 = orbit.e;
        printf("%e\t%f\t%f\n", t, e2, e3);
    }
}
```

This example is located in the directory *examples/rad_forces_circumplanetary*

4.7.12 Migration and other orbit modifications

This example shows how to add migration, eccentricity damping and pericenter precession to a REBOUND simulation. If you have GLUT installed for visualization, press ‘w’ to see the orbits as wires. You can zoom out by holding shift, holding down the mouse and dragging. Press ‘c’ to better see migration/e-damping.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <math.h>
#include <string.h>
#include "rebound.h"
#include "reboundx.h"

void heartbeat(struct reb_simulation* sim);

int main(int argc, char* argv[]){
    struct reb_simulation* sim = reb_simulation_create();
    // Setup constants
    sim->integrator = REB_INTEGRATOR_WHFAST;
    sim->dt             = 0.012;        // initial timestep.
    sim->heartbeat = heartbeat;

    struct reb_particle p = {0};
    p.m     = 1.;
    reb_simulation_add(sim, p);

    double m = 0.;
    double a1 = 1.;
    double a2 = 2.;
    double e = 0.4;
    double inc = 0.;
    double Omega = 0.;
    double omega = 0.;
    double f = 0.;

    struct reb_particle p1 = reb_particle_from_orbit(sim->G, p, m, a1, e, inc, Omega,␣
↪omega, f);
    struct reb_particle p2 = reb_particle_from_orbit(sim->G, p, m, a2, e, 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);

    // There are two options for how to modify orbits.  You would only choose one␣
↪(comment the other out).
    // You can't set precession separately with modify_orbits_forces (eccentricity and␣
↪inclination damping induce pericenter and nodal precession).

    struct rebx_operator* mo = rebx_load_operator(rebx, "modify_orbits_direct");       ␣
↪                           // directly update particles' orbital elements each timestep
    rebx_add_operator(rebx, mo);
    //struct rebx_force* mo = rebx_load_force(rebx, "modify_orbits_forces");           ␣
↪                           // add forces that orbit-average to give exponential a and␣
↪e damping
    //rebx_add_force(rebx, mo);
```

(continues on next page)

(continued from previous page)

```
    // Set the timescales for each particle.
    double tmax = 5.e4;

     rebx_set_param_double(rebx, &sim->particles[1].ap, "tau_a", -tmax);         // add␣
↪semimajor axis damping on inner planet (e-folding timescale)
    rebx_set_param_double(rebx, &sim->particles[1].ap, "tau_omega", -tmax/10.); // add␣
↪linear precession (set precession period). Won't do anything for modify_orbits_forces
    rebx_set_param_double(rebx, &sim->particles[2].ap, "tau_e", -tmax/10.);     // add␣
↪eccentricity damping on particles[2] (e-folding timescale)

    /* One can also adjust a coupling parameter between eccentricity and semimajor axis␣
↪damping.  We use the parameter p
     * as defined by Deck & Batygin (2015).  The default p=0 corresponds to no coupling,␣
↪while p=1 corresponds to e-damping
     * at constant angular momentum.  This is only implemented for modify_orbits_direct.
      * modify_orbits_forces damps eccentricity at constant angular momentum.
     *
     * Additionally, the damping by default is done in Jacobi coordinates.  If you'd␣
↪prefer to use barycentric
     * coordinates, or coordinates referenced to a particular particle, set a␣
↪coordinates parameter in the effect
     * parameters returned by rebx_add to REBX_COORDINATES_BARYCENTRIC or REBX_
↪COORDINATES_PARTICLE.
     * If the latter, add a 'primary' flag to the reference particle (not neccesary for␣
↪barycentric):
     */

     rebx_set_param_int(rebx, &mo->ap, "coordinates", REBX_COORDINATES_PARTICLE);
     rebx_set_param_double(rebx, &mo->ap, "p", 1.); // doesn't do anything for modify_
↪orbits_forces
     rebx_set_param_int(rebx, &sim->particles[0].ap, "primary", 1);

    reb_simulation_integrate(sim, tmax);
    rebx_free(rebx);    // Free all the memory allocated by rebx
    reb_simulation_free(sim);
}

void heartbeat(struct reb_simulation* sim){
    // output a e and pomega (Omega + omega) of inner body
    if(reb_simulation_output_check(sim, 5.e2)){
        const struct reb_particle sun = sim->particles[0];
        const struct reb_orbit orbit = reb_orbit_from_particle(sim->G, sim->particles[1],
↪ sun); // calculate orbit of particles[1]
        printf("%f\t%f\t%f\t%f\n",sim->t,orbit.a, orbit.e, orbit.pomega);
    }
}
```

This example is located in the directory *examples/modify_orbits*

4.7.13 Inner disk edge.

This example shows how to add an inner disk edge when using modify_orbits_forces or modify_orbits_direct. See detailed annotations and explanations for the parameters in the InnerDiskEdge ipython example and Implemented Effects documentation for REBOUNDx.

```
#include <stdio.h>
#include <string.h>
#include "rebound.h"
#include "reboundx.h"
#include "core.h"

int main(int argc, char* argv[]){
    struct reb_simulation* sim = reb_simulation_create();
    /*sim units are ('yr', 'AU', 'Msun')*/
    sim->G = 4*M_PI*M_PI;

    struct reb_particle star = {0};
    star.m     = 1.;
    reb_simulation_add(sim, star);

    double m = 0.00001;
    double a = 1;
    double e = 0;
    double inc = 0.;
    double Omega = 0.;
    double omega = 0.;
    double f = 0.;

    struct reb_particle p1 = reb_particle_from_orbit(sim->G, star, m, a, e, inc, Omega,␣
↪omega, f);
    reb_simulation_add(sim, p1);
    reb_simulation_move_to_com(sim);

    sim->dt = 0.002;  //The period at the inner disk edge divided by 20, for a disk edge␣
↪location at 0.1 AU
    sim->integrator = REB_INTEGRATOR_WHFAST;

    struct rebx_extras* rebx = rebx_attach(sim);

    struct rebx_force* mof = rebx_load_force(rebx, "modify_orbits_forces");
    rebx_set_param_double(rebx, &mof->ap, "ide_position", 0.1);
    rebx_set_param_double(rebx, &mof->ap, "ide_width", 0.02); // Planet will stop within␣
↪0.02 AU of the inner disk edge
    rebx_add_force(rebx, mof);

    double tmax = 1.e4;
    rebx_set_param_double(rebx, &sim->particles[1].ap, "tau_a", -tmax);
    rebx_set_param_double(rebx, &sim->particles[1].ap, "tau_e", -tmax/100.);

    reb_simulation_integrate(sim, tmax);
    rebx_free(rebx);
    reb_simulation_free(sim);
}
```

This example is located in the directory *examples/inner_disk_edge*

4.7.14 Stochastic forces on a single planet

This example shows how to add a stochastic force to a single planet. As a result, the planet will undergo a random walk in its orbital parameters. See also [Rein (2010)](https://ui.adsabs.harvard.edu/abs/2010PhDT.......196R/abstract) and [Rein and Papaloizou (2009)](https://ui.adsabs.harvard.edu/abs/2009A%26A...497..595R/abstract)

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <math.h>
#include "rebound.h"
#include "reboundx.h"

void heartbeat(struct reb_simulation* r);

double tmax = 1e4*M_PI*2.0; // 1e4 orbits

int main(int argc, char* argv[]){
    // Let's first setup the simulation.
    // Note that we are using the WHFast integrator with a fixed timestep.
    // It's important to point out that the IAS15 integrator is not well
    // suited for stochastic forces because it automatically reduces the
    // timestep if it doesn't achieve an accuracy near machine precision.
    // Because the stochastic forces are random, it might never converge.

    struct reb_simulation* sim = reb_simulation_create();
    sim->integrator     = REB_INTEGRATOR_WHFAST;
    sim->dt             = 1e-2;         // At ~100 AU, orbital periods are ~1000 yrs,␣
↪so here we use ~1% of that, in sec
    sim->heartbeat      = heartbeat;

    reb_simulation_add_fmt(sim, "m", 1.); // Sun
    reb_simulation_add_fmt(sim, "m a", 1e-3, 1.0); // Jupiter mass planet at 1AU
    reb_simulation_move_to_com(sim);

    // Next, we add the `stochastic_forces` module in REBOUNDx
    struct rebx_extras* rebx = rebx_attach(sim);
    struct rebx_force* sto = rebx_load_force(rebx, "stochastic_forces");
    rebx_add_force(rebx, sto);

    // We can now turn on stochastic forces for a particle by setting
    // the field `kappa` to a finite value. This parameter determines
    // the strength of the stochastic forces and is relative to the
    // gravitational force that the particle experiences from the
    // center of mass interior to its orbit.
    rebx_set_param_double(rebx, &sim->particles[1].ap, "kappa", 1e-5);

    // The auto-correlation time of the stochastic forces is by default
    // the orbital period. We can set it to a fraction or multiple of
    // the orbital period by chanhging the `tau_kappa` parameter.

    // rebx_set_param_double(rebx, &sim->particles[1].ap, "tau_kappa", 1.); // Not␣
```

(continues on next page)

(continued from previous page)

```
↪needed. Already the default.


    // Let's integrate the system.
    reb_simulation_integrate(sim, tmax);

    rebx_free(rebx);
    reb_simulation_free(sim);
}

void heartbeat(struct reb_simulation* sim){
    // Periodically output the semi-major axis of the planet.
    if(reb_simulation_output_check(sim, 1.e2*M_PI*2.0)){
        reb_simulation_output_timing(sim, tmax);
        FILE* f = fopen("orbit.txt", "a+");
        struct reb_orbit o =  reb_orbit_from_particle(sim->G, sim->particles[1], sim->
↪particles[0]);
        fprintf(f, "%.8e\t%.8e\n", sim->t, o.a);
        fclose(f);
    }
}
```

This example is located in the directory *examples/stochastic_forces*

4.7.15 Adding gravitational harmonics (J2, J4) to particles

This example shows how to add a J2 and J4 harmonic to particles. See the J2.ipynb ipython_example for more details on setting tilted spins, and on manipulating the outputs for analysis. If you have GLUT installed for the visualization, press 'w' and/or 'c' for a clearer view of the whole orbit.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <math.h>
#include "rebound.h"
#include "reboundx.h"

int main(int argc, char* argv[]){
    struct reb_simulation* sim = reb_simulation_create();

    struct reb_particle star = {0};
    star.m     = 1.;
    star.hash  = reb_hash("star");
    reb_simulation_add(sim, star);

    double m = 0.;
    double a = 1.;
    double e = 0.2;
    double inc = 0.;
    double Omega = 0.;
    double omega = 0.;
    double f = 0.;
```

(continues on next page)

(continued from previous page)

```
    struct reb_particle planet = reb_particle_from_orbit(sim->G, star, m, a, e, inc,␣
↪Omega, omega, f);
    planet.hash = reb_hash("planet");
    reb_simulation_add(sim, planet);
    reb_simulation_move_to_com(sim);

    struct rebx_extras* rebx = rebx_attach(sim);
    struct rebx_force* gh = rebx_load_force(rebx, "gravitational_harmonics");
    rebx_add_force(rebx, gh);

    rebx_set_param_double(rebx, &sim->particles[0].ap, "J2", 0.1);
    rebx_set_param_double(rebx, &sim->particles[0].ap, "J4", 0.01);
    rebx_set_param_double(rebx, &sim->particles[0].ap, "R_eq", 0.01);

    double tmax = 1.e5;
    reb_simulation_integrate(sim, tmax);
    rebx_free(rebx);    // this explicitly frees all the memory allocated by REBOUNDx
    reb_simulation_free(sim);
}
```

This example is located in the directory *examples/J2*

4.7.16 Exponential mass loss/gain

This example shows how to add exponential mass loss to bodies in the integration. The primary's mass loss causes planets' orbits to expand. For more flexible variations in mass or other parameters see the parameter interpolation example.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <math.h>
#include "rebound.h"
#include "reboundx.h"

void heartbeat(struct reb_simulation* sim);
double tmax = 1.e4;

int main(int argc, char* argv[]){
     struct reb_simulation* sim = reb_simulation_create();
    sim->G = 4*M_PI*M_PI;                 // use units of AU, yr and solar masses
     sim->heartbeat = heartbeat;

     struct reb_particle sun = {0};
     sun.m   = 1.;
     reb_simulation_add(sim, sun);

     struct reb_particle planet = {0};   // Initialize planets on circular orbits, each␣
↪2 times farther than last.
     planet.m = 1.e-3;
     planet.x = 1.;
```

(continues on next page)

(continued from previous page)

```
    planet.vy = 2.*M_PI;
    reb_simulation_add(sim,planet);
    planet.x *= 2.;
    planet.vy /= sqrt(2.);
    reb_simulation_add(sim, planet);
    planet.x *= 2.;
    planet.vy /= sqrt(2.);
    reb_simulation_add(sim, planet);

    reb_simulation_move_to_com(sim);

    struct rebx_extras* rebx = rebx_attach(sim); // initialize reboundx
   struct rebx_operator* modify_mass = rebx_load_operator(rebx, "modify_mass");

   /* The function rebx_add_operator will choose how to add the operator to the␣
↪integration
    * scheme based on the integrator being used and the properties of the operator.
    * This is typically a half operator timestep before the main REBOUND timestep, and␣
↪half afterward.
    */

    rebx_add_operator(rebx, modify_mass);

   /* If you wanted to make your own choices, you can add individual operator steps.
    * In this case you would pass additional parameters. Say we wanted to add a full␣
↪operaator timestep after the main REBOUND timestep;
    *
    * dt_fraction = 1. // Fraction of a REBOUND timestep (sim->dt) operator should act
    * timing = REBX_TIMING_POST; // Should happen POST timestep
    * name = "modify_mass_post"; // Name identifier
    * rebx_add_operator_step(rebx, modify_mass, dt_fraction, timing, name);
    */

    // To set an exponential mass loss rate, we set the e-folding timescale (positive␣
↪for growth, negative for loss)
   // Here have the star lose mass with e-damping timescale = tmax
   rebx_set_param_double(rebx, &sim->particles[0].ap, "tau_mass", -tmax);

    // We can approximate a linear mass loss/growth rate if the rate is small by taking␣
↪tau_mass = M_initial / mass_loss_rate (or growth)
    double M_dot = 1.e-12; // mass growth rate for the planet (in simulation units--
↪here Msun/yr)
   double tau_mass = sim->particles[1].m / M_dot; // first planet gains mass at linear␣
↪rate M_dot
   rebx_set_param_double(rebx, &sim->particles[1].ap, "tau_mass", tau_mass);

    reb_simulation_integrate(sim, tmax);
    rebx_free(rebx);        // this explicitly frees all the memory allocated by␣
↪REBOUNDx
}

void heartbeat(struct reb_simulation* const sim){
```

(continues on next page)

(continued from previous page)

```
    // Output masses and semimajor of the inner planet 100 times over the time of the␣
↪simulation
   if(reb_simulation_output_check(sim, tmax/100.)){
            struct reb_orbit o = reb_orbit_from_particle(sim->G, sim->particles[1], sim-
↪>particles[0]);
            printf("t=%e, Sun mass = %f, planet mass = %e, planet semimajor axis = %f\n
↪", sim->t, sim->particles[0].m, sim->particles[1].m, o.a);
    }
}
```

This example is located in the directory *examples/modify_mass*

4.7.17 Yarkovsky effect on a small body

This example simulates a single asteroid at .5 AU orbiting around the Sun for 100,000 years to demonstartate how the Yarkovsky effect can change the semi-major axis of an orbiting body. Changing the value of the 'yark_flag' parameter between -1, 0, and 1 switches which version of the effect is being used. For more information on the different versions of the effect and what they're good for, please visit the ipython example and the documentation for this effect.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <math.h>
#include "rebound.h"
#include "reboundx.h"

int main(int argc, char* argv[]) {

    struct reb_simulation* sim = reb_simulation_create(); //creates simulation

    sim->G = 4*M_PI*M_PI;  // use units of AU, yr and solar masses
    sim->dt = .05;         //timestep for simulation in yrs
    sim->integrator = REB_INTEGRATOR_WHFAST; //integrator for sim

    //following adds star with mass of Sun to sim
    struct reb_particle star = {0};
    star.m = 1.;
    reb_simulation_add(sim, star);

    //following variables are the orbital elements of only asteroid in sim
    double m = 0;
    double a = .5;
    double e = 0;
    double inc = 0;
    double Omega = 0;
    double omega = 0;
    double f = 0;

    //creates asteroid
    struct reb_particle asteroid_1 = reb_particle_from_orbit(sim->G, star, m, a, e, inc,␣
↪Omega, omega, f);
```

(continues on next page)

(continued from previous page)

```
    reb_simulation_add(sim,asteroid_1);

    struct reb_particle* const particles = sim->particles; //pointer for the particles␣
↪in the sim

    struct rebx_extras* rebx = rebx_attach(sim);
    struct rebx_force* yark = rebx_load_force(rebx, "yarkovsky_effect");

    double au_conv = 1.495978707e11;
    double msun_conv = 1.9885e30;
    double yr_conv = 31557600.0;

    double density = (3000.0*au_conv*au_conv*au_conv)/msun_conv;
    double c = (2.998e8*yr_conv)/au_conv;
    double lstar = (3.828e26*yr_conv*yr_conv*yr_conv)/(msun_conv*au_conv*au_conv);
    double albedo = .017;
    double stef_boltz = ((5.670e-8)*yr_conv*yr_conv*yr_conv)/(msun_conv);
    double emissivity = .9;
    double k = .25;
    double Gamma = (310*sqrt(yr_conv)*yr_conv*yr_conv)/msun_conv;
    double rotation_period = 15470.9/yr_conv;
    double sx = 0.0;
    double sy = 0.0;
    double sz = 1.0;

    rebx_set_param_double(rebx, &sim->particles[1].ap, "ye_body_density", density);
    rebx_set_param_int(rebx, &sim->particles[1].ap, "ye_flag", 0);
    rebx_set_param_double(rebx, &yark->ap, "ye_lstar", lstar);
    rebx_set_param_double(rebx, &yark->ap, "ye_c", c);
    particles[1].r = 1000/au_conv;

    rebx_set_param_double(rebx, &sim->particles[1].ap, "ye_albedo", albedo);
    rebx_set_param_double(rebx, &yark->ap, "ye_stef_boltz", stef_boltz);
    rebx_set_param_double(rebx, &sim->particles[1].ap, "ye_emissivity", emissivity);
    rebx_set_param_double(rebx, &sim->particles[1].ap, "ye_k", k);
    rebx_set_param_double(rebx, &sim->particles[1].ap, "ye_thermal_inertia", Gamma);
    rebx_set_param_double(rebx, &sim->particles[1].ap, "ye_rotation_period", rotation_
↪period);
    rebx_set_param_double(rebx, &sim->particles[1].ap, "ye_spin_axis_x", sx);
    rebx_set_param_double(rebx, &sim->particles[1].ap, "ye_spin_axis_y", sy);
    rebx_set_param_double(rebx, &sim->particles[1].ap, "ye_spin_axis_z", sz);

    rebx_add_force(rebx, yark);

    double tmax = 100000;

    reb_simulation_integrate(sim, tmax); //integrates system for tmax years

    struct reb_orbit o= reb_orbit_from_particle(sim->G, sim->particles[1], sim->
↪particles[0]); //gives orbital parameters for asteroid after sim

    double final_a = o.a; //final semi-major axis of asteroid after sim
```

(continues on next page)

(continued from previous page)

```
    double final_e = o.e;

    printf("CHANGE IN SEMI-MAJOR AXIS: %1.30f\n", (final_a-a)); //prints difference␣
↪between the intitial and final semi-major axes of asteroid

    printf("CHANGE IN ECCENTRICITY: %1.30f\n", (final_e-e));

}
```

This example is located in the directory *examples/yarkovsky_effect*

4.7.18 Adding parameters to objects in REBOUNDx

This example walks through adding parameters to particles, forces and operators in REBOUNDx

```
#include <stdio.h>
#include <math.h>
#include "rebound.h"
#include "reboundx.h"

int main(int argc, char* argv[]){
    // We start by making a simulation and adding GR with REBOUNDx
    struct reb_simulation* sim = reb_simulation_create();

    struct reb_particle star = {0};
    star.m     = 1.;
    reb_simulation_add(sim, star);

    struct reb_particle planet = {0};
    planet.x = 1.;
    planet.vy = 1.;

    reb_simulation_add(sim, planet);

    struct rebx_extras* rebx = rebx_attach(sim);
    struct rebx_force* gr = rebx_load_force(rebx, "gr");
    rebx_add_force(rebx, gr);

    /* The documentation page https://reboundx.readthedocs.io/en/latest/effects.html␣
↪lists the various required and optional parameters that need to be set for each effect␣
↪in REBOUNDx.
     *
     * Parameters are stored in particles, forces and operators through the linked list
↪'ap', which is common to all three objects.
     *
     * For simple types (int and double) we add them through their corresponding setters.
     * In both cases, we pass the rebx struct, a pointer to the head of the linked list␣
↪that we want to modify, and the value we want to set.
     */

    double c = 10064.915; // speed of light in default units of AU, Msun and yr/2pi
    rebx_set_param_double(rebx, &gr->ap, "c", c);
```

(continues on next page)

(continued from previous page)

```
    // After setting the parameters we want to set, we would integrate as usual.

    double tmax = 10.;
    reb_simulation_integrate(sim, tmax);

    /* At any point, we can access the parameters we set (e.g., some effects could␣
↪update these values as the simulation progresses). We get all parameter types back␣
↪with rebx_get_param, which returns a void pointer that we are responsible for casting␣
↪to the correct type. Since we are not modifying the linked list, we don't pass a␣
↪reference to ap like above*/

    double* new_c = rebx_get_param(rebx, gr->ap, "c");

    /* It is important to check returned parameters!
     * If the parameter is not found, it will return a NULL pointer.
     * If this happens and you dereference it, you will get a segmentation fault.
     * If you get a seg fault, the first thing you should check are returned pointers.
     */

    if (new_c != NULL){
        printf("c=%f\n", *new_c);
    }

    /* The above functionality is probably enough for most users.
     * If you find you need to do more complicated things, read below!
     *
     * More complicated types can be added with set_param_pointer.
     * Instead of passing a value, we now pass a pointer to the variable.
     * As a simple example adding the gr pointer to particles[1] (with no meaning␣
↪whatsoever):
     */

    rebx_set_param_pointer(rebx, &sim->particles[1].ap, "force", gr);

    /* In order to be able to write binary files, and for interoperability with Python,␣
↪REBOUNDx keeps a registered list of parameter names and their corresponding types.
     * Above we have used parameter names that have been registered by various effects.
     * If you try to use a name that's not on the list, REBOUNDx will print an error and␣
↪continue execution without adding the parameter.
     */

    rebx_set_param_int(rebx, &gr->ap, "q", 7);

    int* q = rebx_get_param(rebx, gr->ap, "q");

    if (q != NULL){
        printf("q=%d\n", *q);
    }
    else{
        printf("q is NULL because we didn't register the name first.\n");
    }
```

(continues on next page)

(continued from previous page)

```
    /* You could register the name permanently in rebx_register_params in core.c, or you␣
↪can do it manually in problem.c, passing a name and rebx_param_type enum (defined in␣
↪reboundx.h):
     */

    rebx_register_param(rebx, "q", REBX_TYPE_INT);

    rebx_set_param_int(rebx, &gr->ap, "q", 7);
    q = rebx_get_param(rebx, gr->ap, "q");

    if (q != NULL){
        printf("q=%d\n", *q);
    }
    else{
        printf("q is NULL because we didn't register the name first.\n");
    }

    /* Finally, you might want to add your own custom structs (e.g. from another␣
↪library).
     * This can be done straightforwardly by adding it as a pointer.
     */

    struct SPH_sim{
        double dt;
        int Nparticles;
    };

    struct SPH_sim my_sph_sim = {.dt = 0.1, .Nparticles=10000};

    rebx_register_param(rebx, "sph", REBX_TYPE_POINTER);
    rebx_set_param_pointer(rebx, &gr->ap, "sph", &my_sph_sim);

    // If you need to get it back:

    struct SPH_sim* sph = rebx_get_param(rebx, gr->ap, "sph");

    if (sph != NULL){
        printf("dt=%f\tNparticles=%d\n", sph->dt, sph->Nparticles);
    }

    /* One caveat is that since REBOUNDx does not know the structure definition, custom␣
↪parameters will not be written or read from REBOUNDx binary files*/

    rebx_free(rebx);    // this explicitly frees all the memory allocated by REBOUNDx
    reb_simulation_free(sim);
}
```

This example is located in the directory *examples/parameters*

4.7.19 Adding custom post-timestep modifications and forces.

This allows the user to use the built-in functions of REBOUNDx but also include their own specialised functions.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <math.h>
#include "rebound.h"
#include "reboundx.h"

/* There are two qualitatively different options for adding in effects. One can add a␣
↪force, which evaluates accelerations to add
 * on top of gravitational accelerations every timestep and will be integrated␣
↪numerically.
 * Alternatively one can apply operators before and/or after each REBOUND timestep that␣
↪update particle properties
 * (masses, positions/velocities, or other parameters).
 *
 * Here's an example of a force, for whichi we have to update the particle accelerations␣
↪in the passed particles array (not sim->particles!)
 * Function must have the same prototype as below.
 * We can have the user assign parameters to the force structure in main(), and our␣
↪force function read and/or update those parameters each timestep when called.
```

```
void stark_force(struct reb_simulation* const sim, struct rebx_force* const starkforce,␣
↪struct reb_particle* const particles, const int N){
    double* starkconst = rebx_get_param(sim->extras, starkforce->ap, "starkconst");    //
↪ get parameters we want user to set

    if(starkconst != NULL){
        particles[1].ax += (*starkconst);           // make sure you += not =, which␣
↪would overwrite other accelerations
    }
}

/* Here's a very simple example of an operator to change the planet's orbit
 *
 * For a post_timestep_modification we update the particle states (positions, velocities,
↪ masses etc.)
 * Function must have the same prototype as below, passing simulation and operator␣
↪pointers (which can be used to hold parameters),
 * and the timestep over which the operator should act.
```

```
void simple_drag(struct reb_simulation* const sim, struct rebx_operator* const␣
↪dragoperator, const double dt){
    double* dragconst = rebx_get_param(sim->extras, dragoperator->ap, "dragconst");    //
↪ get parameters we want user to set

    if(dragconst != NULL){
        sim->particles[1].vx *= 1. - (*dragconst)*dt;
        sim->particles[1].vy *= 1. - (*dragconst)*dt;
        sim->particles[1].vz *= 1. - (*dragconst)*dt;
```

(continues on next page)

(continued from previous page)

```
    }
}


int main(int argc, char* argv[]){
    struct reb_simulation* sim = reb_simulation_create();
    struct reb_particle p = {0};
    p.m     = 1.;
    reb_simulation_add(sim, p);

    double m = 0.;
    double a1 = 1.;
    double e = 0.4;
    double inc = 0.;
    double Omega = 0.;
    double omega = 0.;
    double f = 0.;

    struct reb_particle p1 = reb_particle_from_orbit(sim->G, p, m, a1, e, inc, Omega,␣
↪omega, f);
    reb_simulation_add(sim,p1);
    reb_simulation_move_to_com(sim);

    struct rebx_extras* rebx = rebx_attach(sim);  // first initialize rebx

    /* We now add our custom functions. We do this in the same way as we add REBOUNDx␣
↪forces and operators.
     * We'll still get back a force and operator struct, respectively, with a warning␣
↪that the name wasn't found
     * in REBOUNDx and that we're reponsible for setting their type flags and function␣
↪pointers.
     */

    struct rebx_force* stark = rebx_create_force(rebx, "stark_force");

    /* We first set a flag for whether our force only depends on particle positions␣
↪(REBX_FORCE_POS)
     * or whether it depends on velocities (or velocities and positions, REBX_FORCE_VEL)
     */
    stark->force_type = REBX_FORCE_VEL;
    stark->update_accelerations = stark_force;  // set the function pointer to what we␣
↪wrote above
    rebx_add_force(rebx, stark);                // Now it's initialized, add to REBOUNDx

    struct rebx_operator* drag = rebx_create_operator(rebx, "simple_drag"); // Now we␣
↪create our custom operator

    /* We first set the operator_type enum for whether our operator modifies dynamical␣
↪variables (positions, velocities
     * or masses, REBX_OPERATOR_UPDATER), or whether it's just passively recording the␣
↪state of the simulation, or
     * updating parameters that don't feed back on the dynamics (REBX_OPERATOR_RECORDER)
```

(continues on next page)

(continued from previous page)

```
     */

    drag->operator_type = REBX_OPERATOR_UPDATER;
    drag->step_function = simple_drag;  // set function pointer to what we wrote above
    rebx_add_operator(rebx, drag);      // Now it's initialized, add to REBOUNDx

    /* Now we set the parameters that our custom functions above need
     * Before setting them, we need to register them with their corresponding types
     */

    rebx_register_param(rebx, "starkconst", REBX_TYPE_DOUBLE);
    rebx_register_param(rebx, "dragconst", REBX_TYPE_DOUBLE);

    rebx_set_param_double(rebx, &stark->ap, "starkconst", 1.e-5);
    rebx_set_param_double(rebx, &drag->ap, "dragconst", 1.e-5);

    /* To simplify things for other users, we can always add the new effects and␣
↪register parameters in the REBOUNDx
     * repo itself. Feel free to send a pull request or contact me (tamayo.daniel@gmail.
↪com) about adding it
     */

    double tmax = 5.e4;
    reb_simulation_integrate(sim, tmax);
    rebx_free(rebx);                          // Free all the memory allocated by rebx
}
```

This example is located in the directory *examples/custom_effects*

4.7.20 Post-Newtonian correction from general relativity

This example shows how to add post-newtonian corrections to REBOUND simulations with reboundx. If you have GLUT installed for the visualization, press ‘w’ and/or ‘c’ for a clearer view of the whole orbit.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <math.h>
#include "rebound.h"
#include "reboundx.h"

int main(int argc, char* argv[]){
    struct reb_simulation* sim = reb_simulation_create();
    sim->dt = 1.e-8;

    struct reb_particle star = {0};
    star.m     = 1.;
    star.hash  = reb_hash("star");
    reb_simulation_add(sim, star);

    double m = 1.e-5;
    double a = 1.e-4; // put planet close to enhance precession so it's visible in␣
```

(continues on next page)

(continued from previous page)

```
↪visualization (this would put planet inside the Sun!)
    double e = 0.2;
    double inc = 0.;
    double Omega = 0.;
    double omega = 0.;
    double f = 0.;

    struct reb_particle planet = reb_particle_from_orbit(sim->G, star, m, a, e, inc,␣
↪Omega, omega, f);
    planet.hash = reb_hash("planet");
    reb_simulation_add(sim, planet);
    reb_simulation_move_to_com(sim);

    struct rebx_extras* rebx = rebx_attach(sim);
    // Could also add "gr" or "gr_full" here.  See documentation for details.
    struct rebx_force* gr = rebx_load_force(rebx, "gr");
    rebx_add_force(rebx, gr);
    // Have to set speed of light in right units (set by G & initial conditions).  Here␣
↪we use default units of AU/(yr/2pi)
    rebx_set_param_double(rebx, &gr->ap, "c", 10065.32);

    double tmax = 5.e-1;
    reb_simulation_integrate(sim, tmax);
    rebx_free(rebx);    // this explicitly frees all the memory allocated by REBOUNDx
    reb_simulation_free(sim);
}
```

This example is located in the directory *examples/gr*

4.7.21 Obliquity Sculpting of Kepler Multis (Millholland & Laughlin 2019)

In particular, this simulates the obliquity sculpting of the Kepler multis due to convergent migration over 4 Myr. Two planets are initialized just wide of the 3:2 MMR (0.173 and 0.233 AU) are migrated inward. After 2 Myr, migration is turned off. After a period of chaotic obliquity evolution, the inner planet is excited to high obliquity and maintained indefinitely Result is based on Figure 3 in Millholland & Laughlin (2019), and reproduces Figure 3 in Lu et. al (2023) For a more in-depth description of the various parameters that can be set in this simulation, please see the ipython examples for consistent tides & spin (any notebook with the prefix TidesSpin) and migration (Migration.ipynb)

integration time is artificially shortened to run quickly. Full run in paper with tmax = 4e6 orbits takes ~ 2 days

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <math.h>
#include "rebound.h"
#include "reboundx.h"
#include "tides_spin.c"

void heartbeat(struct reb_simulation* sim);
double tmax = 100 * 2 * M_PI; // set short to run quickly. Set to 4e6 * 2 * M_PI in paper

int main(int argc, char* argv[]){
    struct reb_simulation* sim = reb_simulation_create();
```

(continues on next page)

(continued from previous page)

```
    // Exact parameters from Millholland & Laughlin (2019)
    const double solar_mass = 1.;
    const double solar_rad = 0.00465;
    reb_simulation_add_fmt(sim, "m r", solar_mass, solar_rad);// Central object

    const double p1_mass = 5. * 3.0e-6; // in Earth masses * 1 Earth Mass / 1 Solar Mass
    const double p1_rad = 2.5 * 4.26e-5; // in Earth rad * 1 Earth rad / 1 AU
    reb_simulation_add_fmt(sim, "m a e r inc Omega pomega M", p1_mass, 0.17308688, 0.01,␣
↪p1_rad, 0.5 * (M_PI / 180.), 0.0 * (M_PI / 180.), 0.0 * (M_PI / 180.), 0.0 * (M_PI /␣
↪180.)); // Planet 1

    const double p2_mass = 5. * 3.0e-6;
    const double p2_rad = 2.5 * 4.26e-5;
    reb_simulation_add_fmt(sim, "m a e r inc Omega pomega M", p2_mass, 0.23290608, 0.01,␣
↪p2_rad, -0.431 * (M_PI / 180.), 0.0 * (M_PI / 180.), 0.0 * (M_PI / 180.), 0.0 * (M_PI /
↪ 180.)); // Planet 2
    sim->N_active = 3;
    sim->integrator = REB_INTEGRATOR_WHFAST;
    sim->dt = 1e-3;
    sim->heartbeat = heartbeat;

    // Add REBOUNDx Additional effects
    // First Spin
    struct rebx_extras* rebx = rebx_attach(sim);

    struct rebx_force* effect = rebx_load_force(rebx, "tides_spin");
    rebx_add_force(rebx, effect);
    // Exact parameters from Millholland & Laughlin (2019)
    // Sun
    const double solar_spin_period = 20 * 2 * M_PI / 365;
    const double solar_spin = (2 * M_PI) / solar_spin_period;
    const double solar_Q = 1000000.;
    rebx_set_param_double(rebx, &sim->particles[0].ap, "k2", 0.07);
    rebx_set_param_double(rebx, &sim->particles[0].ap, "I", 0.07 * solar_mass * solar_
↪rad * solar_rad);
    rebx_set_param_vec3d(rebx, &sim->particles[0].ap, "Omega", (struct reb_vec3d){.z =␣
↪solar_spin}); // Omega_x = Omega_y = 0 by default

    // We assume tau = 1/(2*n*Q) with n the mean motion, even though the spin is not␣
↪synchronized with the orbit (see Lu et al. (2023))
    struct reb_orbit orb = reb_orbit_from_particle(sim->G, sim->particles[1], sim->
↪particles[0]);
    rebx_set_param_double(rebx, &sim->particles[0].ap, "tau", 1./(2.*orb.n*solar_Q));

    // P1
    const double spin_period_1 = 5. * 2. * M_PI / 365.; // 5 days in REBOUND time units
    const double spin_1 = (2. * M_PI) / spin_period_1;
    const double planet_Q = 10000.;
    rebx_set_param_double(rebx, &sim->particles[1].ap, "k2", 0.4);
    rebx_set_param_double(rebx, &sim->particles[1].ap, "I", 0.25 * p1_mass * p1_rad * p1_
↪rad);
    rebx_set_param_vec3d(rebx, &sim->particles[1].ap, "Omega", (struct reb_vec3d){.
```

(continues on next page)

(continued from previous page)

```
↪y=spin_1 * -0.0261769, .z=spin_1 * 0.99965732});

    rebx_set_param_double(rebx, &sim->particles[1].ap, "tau", 1./(2.*orb.n*planet_Q));

    // P2
    double spin_period_2 = 3. * 2. * M_PI / 365.; // 3 days in REBOUND time units
    double spin_2 = (2. * M_PI) / spin_period_2;
    rebx_set_param_double(rebx, &sim->particles[2].ap, "k2", 0.4);
    rebx_set_param_double(rebx, &sim->particles[2].ap, "I", 0.25 * p2_mass * p2_rad * p2_
↪rad);
    rebx_set_param_vec3d(rebx, &sim->particles[2].ap, "Omega", (struct reb_vec3d){.
↪y=spin_2 * 0.0249736, .z=spin_2 * 0.99968811});

    struct reb_orbit orb2 = reb_orbit_from_particle(sim->G, sim->particles[2], sim->
↪particles[0]);
    rebx_set_param_double(rebx, &sim->particles[2].ap, "tau", 1./(2.*orb2.n*planet_Q));

    // And migration
    struct rebx_force* mo = rebx_load_force(rebx, "modify_orbits_forces");
    rebx_add_force(rebx, mo);

    // Set migration parameters
    rebx_set_param_double(rebx, &sim->particles[1].ap, "tau_a", -5e6 * 2 * M_PI);
    rebx_set_param_double(rebx, &sim->particles[2].ap, "tau_a", (-5e6 * 2 * M_PI) / 1.1);

    reb_simulation_move_to_com(sim);

    // Let's create a reb_rotation object that rotates to new axes with newz pointing␣
↪along the total ang. momentum, and x along the line of
    // nodes with the invariable plane (along z cross newz)
    struct reb_vec3d newz = reb_vec3d_add(reb_simulation_angular_momentum(sim), rebx_
↪tools_spin_angular_momentum(rebx));
    struct reb_vec3d newx = reb_vec3d_cross((struct reb_vec3d){.z =1}, newz);
    struct reb_rotation rot = reb_rotation_init_to_new_axes(newz, newx);
    rebx_simulation_irotate(rebx, rot); // This rotates our simulation into the␣
↪invariable plane aligned with the total ang. momentum (including spin)
    rebx_spin_initialize_ode(rebx, effect);

    // Run simulation
    system("rm -v output_orbits.txt"); // remove previous output files
    system("rm -v output_spins.txt");

    reb_simulation_integrate(sim, tmax/2);

    printf("Migration Switching Off\n");
    rebx_set_param_double(rebx, &sim->particles[1].ap, "tau_a", INFINITY);
    rebx_set_param_double(rebx, &sim->particles[2].ap, "tau_a", INFINITY);

    reb_simulation_integrate(sim, tmax);

    rebx_free(rebx);
    reb_simulation_free(sim);
```

(continues on next page)

(continued from previous page)

```
}

void heartbeat(struct reb_simulation* sim){
  if(reb_simulation_output_check(sim, tmax/100000)){        // outputs every 100 REBOUND␣
↪years
    struct rebx_extras* const rebx = sim->extras;
    FILE* of_orb = fopen("output_orbits.txt", "a");
    FILE* of_spins = fopen("output_spins.txt", "a");
    if (of_orb == NULL || of_spins == NULL){
        reb_simulation_error(sim, "Can not open file.");
        return;
    }

    struct reb_particle* sun = &sim->particles[0];
    struct reb_particle* p1 = &sim->particles[1];
    struct reb_particle* p2 = &sim->particles[2];

    // Orbit information
    struct reb_orbit o1 = reb_orbit_from_particle(sim->G, *p1, *sun);
    double a1 = o1.a;
    double e1 = o1.e;
    double i1 = o1.inc;
    double Om1 = o1.Omega;
    double pom1 = o1.pomega;
    struct reb_vec3d norm1 = o1.hvec;

    struct reb_orbit o2 = reb_orbit_from_particle(sim->G, *p2, *sun);
    double a2 = o2.a;
    double e2 = o2.e;
    double i2 = o2.inc;
    double Om2 = o2.Omega;
    double pom2 = o2.pomega;
    struct reb_vec3d norm2 = o2.hvec;

    // Spin vectors - all initially in invariant plane
    struct reb_vec3d* Omega_sun = rebx_get_param(rebx, sun->ap, "Omega");

    // Interpret both planet spin vectors in the rotating planet frame in spherical␣
↪coordinates
    struct reb_vec3d* Omega_p1 = rebx_get_param(rebx, p1->ap, "Omega");

    struct reb_vec3d lon1 = reb_vec3d_cross((struct reb_vec3d){.z =1}, norm1);  // Line␣
↪of nodes is the new x-axis
    struct reb_rotation rot1 = reb_rotation_init_to_new_axes(norm1, lon1);      //␣
↪Arguments to this function are the new z and x axes
    struct reb_vec3d sv1 = reb_vec3d_rotate(*Omega_p1, rot1);

    double mag1;
    double theta1;
    double phi1;
    reb_tools_xyz_to_spherical(sv1, &mag1, &theta1, &phi1);
```

(continues on next page)

(continued from previous page)

```
    struct reb_vec3d* Omega_p2 = rebx_get_param(rebx, p2->ap, "Omega");

    struct reb_vec3d lon2 = reb_vec3d_cross((struct reb_vec3d){.z =1}, norm2); // Line␣
↪of nodes is the new x-axis
    struct reb_rotation rot2 = reb_rotation_init_to_new_axes(norm2, lon2); // Arguments␣
↪to this function are the new z and x axes
    struct reb_vec3d sv2 = reb_vec3d_rotate(*Omega_p2, rot2);

    double mag2;
    double theta2;
    double phi2;
    reb_tools_xyz_to_spherical(sv2, &mag2, &theta2, &phi2);

    fprintf(of_orb, "%e,%e,%e,%e,%e,%e,%e,%e,%e,%e,%e,%e,%e,%e,%e,%e,%e\n", sim->t, a1,␣
↪e1, i1, pom1, Om1, norm1.x, norm1.y, norm1.z, a2, e2, i2, pom2, Om2, norm2.x, norm2.y,␣
↪norm2.z);  // prints the spins and orbits of all bodies
    fclose(of_orb);
    fprintf(of_spins, "%e,%e,%e,%e,%e,%e,%e,%e,%e,%e,%e,%e,%e,%e,%e,%e\n", sim->t, Omega_
↪sun->x, Omega_sun->y, Omega_sun->z, mag1, theta1, phi1, mag2, theta2, phi2, Omega_p1->
↪x, Omega_p1->y, Omega_p1->z, Omega_p2->x, Omega_p2->y, Omega_p2->z);
    fclose(of_spins);
  }

  if(reb_simulation_output_check(sim, 100.*M_PI)){        // outputs to the screen
      reb_simulation_output_timing(sim, tmax);
  }
}
```

This example is located in the directory *examples/tides_spin_migration_driven_obliquity_tides*

4.7.22 Exponential Migration

This example shows how to add exponential migration to a REBOUND simulation. Both the Reboundx force (exponential_migration.c) and this example are based on modify_orbits_forces

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <math.h>
#include <string.h>
#include "rebound.h"
#include "reboundx.h"

void heartbeat(struct reb_simulation* sim);

int main(int argc, char* argv[]){
    struct reb_simulation* sim = reb_simulation_create();
    // Setup constants
    sim->dt             = 0.012;        // initial timestep.
    sim->heartbeat = heartbeat;

    struct reb_particle p = {0};
```

(continues on next page)

(continued from previous page)

```
    p.m     = 1.;
    reb_simulation_add(sim, p);

    double m = 0.0001;
    double a1 = 24.0;
    double e = 0.01;
    double inc = 0.;
    double Omega = 0.;
    double omega = 0.;
    double f = 0.;

    struct reb_particle p1 = reb_particle_from_orbit(sim->G, p, m, a1, e, inc, Omega,␣
↪omega, f);
    reb_simulation_add(sim,p1);
    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);

    // Set the timescales for each particle.
    double tmax = 4.e4;

    rebx_set_param_double(rebx, &sim->particles[1].ap, "em_tau_a", 2.e3);         // 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", 30.0);         // add␣
↪final semimajor axis

     rebx_set_param_int(rebx, &em->ap, "coordinates", REBX_COORDINATES_PARTICLE);
     rebx_set_param_int(rebx, &sim->particles[0].ap, "primary", 1);

    reb_simulation_integrate(sim, tmax);
    rebx_free(rebx);    // Free all the memory allocated by rebx
    reb_simulation_free(sim);
}

void heartbeat(struct reb_simulation* sim){
    // output a e body
    if(reb_simulation_output_check(sim, 1.e2)){
        const struct reb_particle sun = sim->particles[0];
        const struct reb_orbit orbit = reb_orbit_from_particle(sim->G, sim->particles[1],
↪ sun); // calculate orbit of particles[1]
        printf("%f\t%f\t%f\t%f\n",sim->t,orbit.a, orbit.e);
    }
}
```

This example is located in the directory *examples/exponential_migration*

4.7.23 Radiation forces on a debris disk

This example shows how to integrate a debris disk around a Sun-like star, with dust particles under the action of radiation forces using WHFast.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <math.h>
#include "rebound.h"
#include "reboundx.h"

void heartbeat(struct reb_simulation* r);

double tmax = 3e12;                     // in sec, ~ 10^5 yrs

int main(int argc, char* argv[]){
    struct reb_simulation* sim = reb_simulation_create();
    // Setup constants
    double AU = 1.5e11;                 // in meters
    sim->integrator     = REB_INTEGRATOR_WHFAST;
    sim->G              = 6.674e-11;    // Use SI units
    sim->dt             = 1e8;          // At ~100 AU, orbital periods are ~1000 yrs, so␣
↪here we use ~1% of that, in sec
    sim->N_active       = 1;            // The dust particles do not interact with one␣
↪another gravitationally
    sim->heartbeat      = heartbeat;
    sim->usleep         = 1000;             // Slow down integration (for visualization␣
↪only). Remove in your integrations!

    // sun
    struct reb_particle sun = {0};
    sun.m  = 1.99e30;                   // mass of Sun in kg
    reb_simulation_add(sim, sun);

    struct rebx_extras* rebx = rebx_attach(sim);
    struct rebx_force* rad = rebx_load_force(rebx, "radiation_forces");
    rebx_add_force(rebx, rad);

    rebx_set_param_double(rebx, &rad->ap, "c", 3.e8);

    /* The effect assumes the radiation source is particles[0].  You can set it to␣
↪another one by adding a radiation_source flag to it:
     * rebx_set_param_int(rebx, &sim->particles[3].ap, "radiation_source", 1);

     * Initialize dust particles
     * We idealize a perfectly coplanar debris disk with particles that have semimajor␣
↪axes between 100 and 120 AU.
     * We initialize particles with 0.01 eccentricity, random pericenters and azimuths,␣
↪and uniformly distributed
     * semimajor axes in the above range.  We take all particles to have a beta␣
↪parameter (ratio of radiation
     * force to gravitational force from the star) of 0.1.
     *
```

(continues on next page)

(continued from previous page)

```
     ***** Only particles that have their beta parameter set will feel radiation forces.␣
↪*****/

    double amin = 100.*AU;
    double awidth = 20.*AU;
    double e = 0.1;
    double inc = 0.;
    double Omega = 0.;
    double Ndust = 1000;                // Number of dust particles

    int seed = 3;                       // random number generator seed
    srand(seed);
    double a, pomega, f;
    struct reb_particle p;
    double beta = 0.1;
    for(int i=1; i<=Ndust; i++){
                                        // first we set up the orbit from sets of␣
↪uniformly drawn orbital elements.
        a = amin + awidth*(double)rand() / (double)RAND_MAX;
        pomega = 2*M_PI*(double)rand() / (double)RAND_MAX;
        f = 2*M_PI*(double)rand() / (double)RAND_MAX;

        double m=0;                     // We treat the dust grains as massless.
        p = reb_particle_from_orbit(sim->G, sim->particles[0], m, a, e, inc, Omega,␣
↪pomega, f);
        reb_simulation_add(sim, p);
                                        // Only particles with beta set will feel␣
↪radiation forces
        rebx_set_param_double(rebx, &sim->particles[i].ap, "beta", beta);
    }

    reb_simulation_move_to_com(sim);
    reb_simulation_integrate(sim, tmax);

    /* Note that the debris disk will seem to undergo oscillations at first.
     * This is actually the correct behavior.  We set up the
     * particles with orbital elements (that assume only gravity is acting).  When we␣
↪turn on radiation,
     * all of a sudden particles are moving too fast for the reduced gravity they feel␣
↪(due to radiation
     * pressure), so they're all at pericenter.  All the particles therefore move outward␣
↪in phase.  It
     * takes a few cycles before the differential motion randomizes the phases. */

    rebx_free(rebx);
    reb_simulation_free(sim);
}

void heartbeat(struct reb_simulation* sim){
    if(reb_simulation_output_check(sim, 1.e8)){
        //reb_simulation_output_timing(sim, tmax);
    }
```

(continues on next page)

(continued from previous page)

```
}
```

This example is located in the directory *examples/rad_forces_debris_disk*

4.7.24 Chaotic dynamical tides model

Close pericenter passages excite fundamental (f) modes in the planet. This effect models the chaotic growth in energy in these modes, and their non-linear dissipation at large enough amplitudes. See the corresponding ipython example, as well as the paper, for more explanations along the way of the various parameters and assumptions.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <math.h>
#include "rebound.h"
#include "reboundx.h"

void heartbeat(struct reb_simulation* r);
double tmax = 2.e3*2*M_PI; // years

int main(int argc, char* argv[]){
    struct reb_simulation* sim = reb_simulation_create();
    sim->heartbeat = heartbeat;

    struct reb_particle sun = {0};
    sun.m = 1.;
    reb_simulation_add(sim, sun);

    double a0 = 1.5;                // in AU
    double e0 = 0.987;
    double Rp = 1.6*4.67e-4;        // 1.6 Jup radii in AU
    double Mp = 1.e-3;              // 1 Jup mass in Msun
    struct reb_particle p = reb_particle_from_orbit(sim->G, sun, Mp, a0, e0, 0, 0, 0, 0);
    reb_simulation_add(sim, p);
    sim->particles[1].r = Rp;       // We must set both the mass and radius of the planet
    reb_simulation_move_to_com(sim);

    // Add dynamical tides
    struct rebx_extras* rebx = rebx_attach(sim);
    struct rebx_force* tides = rebx_load_force(rebx, "tides_dynamical");
    rebx_add_force(rebx, tides);

    // See the ipython example for various options that can be set

    // Run simulation
    reb_simulation_integrate(sim, tmax);
    rebx_free(rebx);
    reb_simulation_free(sim);
}

void heartbeat(struct reb_simulation* sim){
    // Periodically output the semi-major axis and eccentricity of the planet.
```

(continues on next page)

(continued from previous page)

```
    if(reb_simulation_output_check(sim, 1.e2*M_PI*2.0)){
        struct reb_orbit o =  reb_orbit_from_particle(sim->G, sim->particles[1], sim->
↪particles[0]);
        printf("%.3f\t%.3f\t%.3f\n", sim->t, o.a, o.e);
    }
}
```

This example is located in the directory *examples/tides_dynamical*

4.7.25 Gas dynamical friction

This example shows how to add the gas_df force.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <math.h>
#include "rebound.h"
#include "reboundx.h"

int main(int argc, char* argv[]){
    struct reb_simulation* sim = reb_simulation_create();
    sim->integrator = REB_INTEGRATOR_BS;

    struct reb_particle bh = {0};
    bh.m     = 4e6;
    reb_simulation_add(sim, bh);

    double m = 1;
    double a = 206000;
    double e = 0.01;
    double inc = 0.17;
    double Omega = 0.;
    double omega = 0.;
    double f = 0.;

    struct reb_particle star = reb_particle_from_orbit(sim->G, bh, m, a, e, inc, Omega,␣
↪omega, f);
    reb_simulation_add(sim, star);
    reb_simulation_move_to_com(sim);

    struct rebx_extras* rebx = rebx_attach(sim);
    struct rebx_force* gdf = rebx_load_force(rebx, "gas_dynamical_friction");
    rebx_add_force(rebx, gdf);

    rebx_set_param_double(rebx, &gdf->ap, "gas_df_rhog", 0.20);
    rebx_set_param_double(rebx, &gdf->ap, "gas_df_alpha_rhog", -1.5);
    rebx_set_param_double(rebx, &gdf->ap, "gas_df_cs", 20);
    rebx_set_param_double(rebx, &gdf->ap, "gas_df_alpha_cs", -0.5);
    rebx_set_param_double(rebx, &gdf->ap, "gas_df_xmin", 0.045);
    rebx_set_param_double(rebx, &gdf->ap, "gas_df_hr", 0.01);
    rebx_set_param_double(rebx, &gdf->ap, "gas_df_Qd", 5.0);
```

(continues on next page)

(continued from previous page)

```
    double delta_t = 6.28e5;
    for (int i = 0; i < 100; i++){
        reb_simulation_integrate(sim, sim->t + delta_t);
        struct reb_orbit o = reb_orbit_from_particle(sim->G, sim->particles[1], sim->
→particles[0]);
        printf("%f %f %f %e\n", sim->t, o.a, o.e, o.inc);
    }

    rebx_free(rebx);    // this explicitly frees all the memory allocated by REBOUNDx
    reb_simulation_free(sim);
}
```

This example is located in the directory *examples/gas_dynamical_friction*

4.7.26 Gas Damping Timescale Example

This example shows how to add gas damping to a REBOUND simulation.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <math.h>
#include <string.h>
#include "rebound.h"
#include "reboundx.h"

void heartbeat(struct reb_simulation* sim);

int main(int argc, char* argv[]){

    /* start the rebound simulation here */
    struct reb_simulation* sim = reb_simulation_create();

    // set units to use throughout the simulation
    sim->G = 4*M_PI*M_PI;  //  Gravitational constant in AU^3/M_sun/yr^2

    sim->heartbeat = heartbeat;

    // add the host star
    struct reb_particle star = {0};
    star.m      = 1.;
    reb_simulation_add(sim, star);

    double m = 3.e-6;                // roughly 1 Earth Mass in Solar Masses
    double a = 0.1;
    double e = 0.05;
    double inc = 5.*M_PI/180.;        // 5 degrees in radians
    double Omega = 0.;
    double omega = 0.;
    double f = 0.;
```

(continues on next page)

(continued from previous page)

```
    struct reb_particle planet = reb_particle_from_orbit(sim->G, star, m, a, e, inc,␣
↪Omega, omega, f);
    reb_simulation_add(sim,planet);

    // move simulation to center-of-mass frame
    reb_simulation_move_to_com(sim);

    // add in reboundx and load/add new force
    struct rebx_extras* rebx = rebx_attach(sim);
    struct rebx_force* gd = rebx_load_force(rebx, "gas_damping_timescale");
    rebx_add_force(rebx, gd);

    // Set the total simulation time
    double tmax = 5.e2;      // 500 years

    // Set parameters for simulation
    rebx_set_param_double(rebx, &gd->ap, "cs_coeff", 0.272);         // set speed sound␣
↪coefficient to 0.272 AU^(3/4) yr^-1
    rebx_set_param_double(rebx, &gd->ap, "tau_coeff", 0.003);        // set timescale␣
↪coefficient to 0.003 yr AU^-2

    // Set parameter for particle
    rebx_set_param_double(rebx, &sim->particles[1].ap, "d_factor", 5.);         // set␣
↪depletion factor to 5

    // integrate simulation
    reb_simulation_integrate(sim, tmax);
    rebx_free(rebx);            // free all the memory allocated by reboundx
    reb_simulation_free(sim);   // free all the memory allocated by rebound
}

void heartbeat(struct reb_simulation* sim){
    // output a e i of the planet
    if(reb_simulation_output_check(sim, 50.)){
        const struct reb_particle star = sim->particles[0];
        const struct reb_orbit orbit = reb_orbit_from_particle(sim->G, sim->particles[1],
↪ star); // calculate orbit of planet
        printf("%f\t%f\t%f\t%f\n",sim->t, orbit.a, orbit.e, orbit.inc);
    }
}
```

This example is located in the directory *examples/gas_damping_timescale*

4.8 Adding A New Effect

Whether you're playing around with new physics in your integrations or want to contribute a new effect to REBOUNDx, the process is easy.

The *Basic Force* section shows you how to get your new effect working in C and python within 10 minutes. The *Adding Parameters* section shows you how to allow the user to set effect and particle parameters at runtime, and the *Contributing your effect to REBOUNDx* section goes over how to add your new effect to the REBOUNDx repository, so others can use it and find it in the documentation.

Our hope is that as people use the package and work on new problems, they will contribute their new effects, so others can use (and cite!) their implementations. Contributing the effects you implement (and may want to use in the future) to REBOUNDx has the added benefit of ensuring that they stay up to date as REBOUND and REBOUNDx expand.

Do I Have To Write It In C?

Forces and operators are called every timestep, and the overhead of REBOUND calling a Python function each timestep makes it a factor of a few slower than if the effect was written in C. Therefore all effects in REBOUNDx are written in C. Often you might want to quickly try something out in Python, and the Custom_Effects.ipynb example shows you how to do that. Here we walk through an example of porting the stark force in Custom_Effect.ipynb into C.

Writing Forces

There are two kinds of effects, forces and operators. We first consider a force, which boils down to writing a C function that will evaluate and add the relevant accelerations to all the particles. Behind the scenes, REBOUNDx then takes care of when your acceleration function gets called depending on the REBOUND integrator the user has chosen. Adding operators is very analogous, and we discuss the differences in *Operators*.

Note: Different implementations for the same effect should be added as separate effects. For example, there are separate files for `gr`, `gr_potential`, and `gr_full`.

4.8.1 Basic Force

effect.c

First copy an existing `.c` file in `reboundx/src`, so that you have the license and right code structure to work from. Typically, you'll be able to reuse many parts of the code.

The function prototype should always stay the same, so we just change the name

```
void rebx_stark_force(struct reb_simulation* const sim, struct rebx_force* const force,␣
↪struct reb_particle* const particles, const int N){
```

All forces are always passed as a pointer to the simulation `sim`, a force structure `force`, a `particles` array, and the number `N` of particles in the array (real particles, not counting variational particles–you don't have to worry if you don't know what those are). The only thing our function needs to do is evaluate the accelerations for each particle, and add those to each particles' acceleration vector, as we'll do below. Note that we should specifically update the passed `particles` array (NOT `sim->particles`).

As a simple first example, let's hardcode a constant acceleration along the x direction for the first particle, following the ipynb example:

```
void rebx_stark_force(struct reb_simulation* const sim, struct rebx_force* const force,␣
↪struct reb_particle* const particles, const int N){
    particles[1].ax += 0.01;
}
```

Note that we ADD (+=) to the particles' accelerations, rather than overwrite them (i.e., particles[1].ax = 0.01). This way the various accelerations acting on particles can be accumulated.

core.c and core.h

You need to add your new force as a new `else if` in the `rebx_load_force` function in reboundx/src/core.c, referencing the function you've written, and the type of force. If evaluation of your accelerations involves the particle velocities, set `REBX_FORCE_VEL`, otherwise `REBX_FORCE_POS`:

```
...
else if (strcmp(name, "stark_force") == 0){
    force->update_accelerations = rebx_stark_force;
```

(continues on next page)

(continued from previous page)

```
    force->force_type = REBX_FORCE_POS;
}
else{
...
```

You also need to add your function prototype at the bottom of reboundx/src/core.h under Force prototypes:

```
...
void rebx_stark_force(struct reb_simulation* const sim, struct rebx_force* const force,␣
↪struct reb_particle* const particles, const int N);
```

Now

```
cd your_path_to/reboundx/scripts
python add_new_effect.py
```

This script updates all the makefiles and the pip installation file to include your new effect.

That's it! Your new force now works from both C and Python. Let's try it out.

C Example

Let's test this first in C. This could then turn into a C example for others if you contributed it to REBOUNDx (all REBOUNDx effects have corresponding C examples). Navigate to the `reboundx/examples` folder, and copy any folder to a new one named `stark_force`. Now we just modify the `problem.c` file in our new `stark_force` folder, e.g.:

```
#include "rebound.h"
#include "reboundx.h"

int main(int argc, char* argv[]){
    struct reb_simulation* sim = reb_simulation_create();
    struct reb_particle star = {0};
    star.m     = 1.;
    reb_simulation_add(sim, star);

    struct reb_particle planet = {0};  // add a planet on a circular orbit (with default␣
↪units where G=1)
    planet.x = 1.;
    planet.vy = 1.;
    reb_simulation_add(sim, planet);

    struct rebx_extras* rebx = rebx_attach(sim);  // first initialize rebx
    struct rebx_force* stark = rebx_load_force(rebx, "stark_force"); // add our new force
    rebx_add_force(rebx, stark);

    double tmax = 100000.;
    reb_simulation_integrate(sim, tmax);
}
```

In the terminal in the `stark_force` folder then just `make clean`, `make` and then run it with `./rebound`. In the visualization press 'w' to see the orbits. You should see a mess with the orbit getting more and less eccentric. If you get an error about OpenGL or GLUT, just google *install openGL glut libraries <your OS here>* for instructions, or open your `Makefile` and set OPENGL=0 to turn it off.

Python Example

Our new effect will now work out of the box without any extra python code. We just need to make sure that whenever we change C code (like we did above), we reinstall REBOUNDx, i.e. `pip install -e .` in the root `reboundx` directory. Then, e.g. in a jupyter notebook:

```
import rebound
import reboundx

sim = rebound.Simulation()
sim.add(m=1.)
sim.add(a=1.)

rebx = reboundx.Extras(sim)
stark = rebx.load_force("stark_force")
rebx.add_force(stark)

sim.integrate(1e5)
```

will run with our new effect. We could plot the eccentricity vs time, just like in the Custom_Effects.ipynb ipython_example where we code the effect in python (and is a factor of a few slower than our new C code).

That's all there is to it. If you want to make your effect more flexible, so that users can change parameters at runtime, check out *Adding Parameters*, and *Contributing your effect to REBOUNDx* if you want to add your effect to REBOUNDx so others can also use it.

Operators

While the above example shows how to add a new force, adding operators is very analogous. As opposed to updating accelerations, operators should update the particle states (typically their velocities). Operators make up splitting schemes, and you should read our REBOUNDx paper if you're not familiar with them.

The only difference in implementation from the above is that you would update `rebx_load_operator` instead of `rebx_load_force`, and your function prototype should look like

```
void rebx_my_operator(struct reb_simulation* const sim, struct rebx_operator* const␣
↪operator, const double dt){
```

where `sim` is again a pointer to the simulation, `operator` is an operator struct analogous to the `force` struct, and `dt` is the length of time over which the operator should act. See `modify_mass.c` for an example.

4.8.2 Adding Parameters

In *Basic Force* we went over how to add a simple new force, where we simply hardcoded which particles were effected, and by how much. REBOUNDx also makes it easy to add parameters to forces and particles so that the user can have the flexibility to choose these values at runtime, can write a script that sets parameters individually on particles, can inspect them to write output to files, change values halfway through, etc. This will show you how to do that with your effect.

Particularly if you are used to the Python side of REBOUND/REBOUNDx, you should read *Quickstart (C)* to see how to access parameters, and it can be very useful to look at forces that are already implemented.

First, you should decide whether force parameters belong on your force or on particles. For example, the `radiation_forces` effect needs to know the speed of light (which will vary if the user changes units), so `c` is a parameter that is the same for all particles, and is added to the force. In our case, if our constant stark acceleration was the same for all particles, we might add it to the force. If each particle could feel a different acceleration, we would add

them to the particles. That will depend on the physics you're trying to put in–let's add the parameter to the particles as an example.

The first thing to do is register the parameter name in `rebx_register_default_params` in `src/core.c`. You cannot use particle parameter names that are in use by other effects, so search first for the name you are planning to add. In order to avoid clashes, we have implemented a convention for new effects that any parameters must start with the acronym for the effect. So for example the tau parameter for `tides_constant_time_lag` is `tctl_tau`.

You also have to specify the type. The vast majority of parameters will be integers (REBX_TYPE_INT) or doubles (REBX_TYPE_DOUBLE). We go through what to do with new custom types below.

Here let's call our parameter `stark_acc`, and it should be a double:

```
...
rebx_register_param(rebx, "stark_acc", REBX_TYPE_DOUBLE);
```

The user will now be able to set and check the value of this parameter on all particles. Now we have to do something with it in our `stark_force` implementation, following the basic example in *Adding A New Effect*:

```
void rebx_stark_force(struct reb_simulation* const sim, struct rebx_force* const force,␣
→struct reb_particle* const particles, const int N){
    struct rebx_extras* const rebx = sim->extras;
    for (int i=0; i<N; i++){
        const double* stark_acc = rebx_get_param(rebx, particles[i].ap, "stark_acc");
        if (stark_acc != NULL){
            particles[i].ax += *stark_acc;
        }
    }
}
```

We now iterate through the particle list, check whether each one has its `stark_acc` param, and if so, update its x acceleration. It's good practice to always check the parameter pointers you get back from `rebx_get_param` for NULL, since otherwise you will get a seg fault when you dereference them if they have not been set by the user.

C Example

Now in C we can set our particle parameter in our `problem.c` file:

```
rebx_set_param_double(rebx, &sim->particles[1].ap, "stark_acc", 0.01);
```

and everything will work as before.

Python Example

Since we've edited the C code, to use it from Python we have to go back to the root `reboundx` directory and `pip install -e .`. After that, we can access and change our new particle parameters out of the box:

```
sim.particles[1].params['stark_acc'] = 0.01
```

That's it!

Adding Custom Types

Most of the time, you'll be using integer and double types for the parameters. But there may be times where you want to, e.g., use a custom struct. There is a catchall void pointer type (REBX_TYPE_POINTER) for such cases. This is convenient in C (see the bottom of the `reboundx/examples/parameters/problem.c` example), but in python

involves casting things manually (see the bottom of `reboundx/ipython_examples/GettingStartedParameters.ipynb`).

Here we will take that ipython_example with a made up SPH_sim struct and show how to make your new struct easily accessible in python.

First in `src/reboundx.h`, we need to add a new enum to `rebx_param_type`:

```
enum rebx_param_type{
    REBX_TYPE_NONE,
    ...
    REBX_TYPE_SPHSIM
};
```

Then we need to define this struct below under 'Basic types in REBOUNDx':

```
struct rebx_SPH_sim {
    double dt;
    int Nparticles;
};
```

Then in `src/core.c`, under `rebx_register_default_params`, we need to register it with its new type:

```
void rebx_register_default_params(struct rebx_extras* rebx){
    ...
    rebx_register_param(rebx, "sph_sim", REBX_TYPE_SPHSIM);
```

On the Python side, at the bottom of `reboundx/reboundx/extras.py` we then have to define the ctypes Structure that matches our C structure (google ctypes documentation or follow the existing examples):

```
class SPH_sim(Structure):
    _fields_ = [("dt",  c_double),
                ("Nparticles", c_int)]
```

and on the line below we have to update the mapping `REBX_C_TO_CTYPES`, which goes from the `rebx_param_type` enum (the first thing we edited in this section) to the Python ctypes structure that we just created (SPH_sim). The order in this list must match exactly with what's in the `rebx_param_type` enum.

```
REBX_C_TO_CTYPES = [["REBX_TYPE_NONE", None], ["REBX_TYPE_DOUBLE", c_double], ["REBX_
→TYPE_INT",c_int], ["REBX_TYPE_POINTER", c_void_p], ["REBX_TYPE_FORCE", Force], ["REBX_
→TYPE_UNIT32", c_uint32], ["REBX_TYPE_ORBIT", rebound.Orbit], ["REBX_TYPE_SPHSIM", SPH_
→sim]]
```

Finally, in `reboundx/reboundx/params.py`, we have to import our new structure and add a matching if clause in `__setitem__`:

```
from .extras import SPH_sim
...

if ctype == SPH_sim:
    if not isinstance(value, SPH_sim):
        raise AttributeError("REBOUNDx Error: Parameter '{0}' must be assigned a SPHsim␣
→object.".format(key))
    clibreboundx.rebx_set_param_pointer(self.rebx, byref(self.ap), c_char_p(key.encode(
→'ascii')), byref(value))
```

Now, if we follow the ipython_examples/GettingStartedParameters.ipynb, rather than manually casting things under Custom Parameters, we can simply do

```
from reboundx.extras import SPH_sim
my_sph_sim = SPH_sim()
my_sph_sim.dt = 0.1
my_sph_sim.Nparticles = 10000
gr.params['sph_sim'] = my_sph_sim
gr.params['sph_sim'].Nparticles
```

which will output 10000.

Note that these custom structs will still not be written to REBOUNDx binaries. If this is important to you, feel free to get in touch.

4.8.3 Contributing your effect to REBOUNDx

You've followed the previous steps in *Adding A New Effect* and *Adding Parameters*, and now want to add it to REBOUNDx so others can use (and cite!) your implementation. Great!

A checklist:

- Have you added a C example in reboundx/examples/?
- Have you added a python example in reboundx/ipython_examples/?
- Have you added documentation, so people can find your effect? (see below)
- Is your code machine independent? (see below)

I'm happy to help with any of these. Once you're ready, send me a pull request (see bottom of this page)

Add Your Effect to the Main Documentation Page!

You add the documentation for your effect directly within your effect.c file. It will then automatically get built into the *Astrophysical Effects* page. Easiest is if you copy-paste from another effect source file.

At the top of the comment block, you should edit the file, brief and author lines. The rest of the documentation goes Below the dollar signs. For example, for the stark_force implementation we did:

```
* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
*
* $Dcoumentation Examples$       // Effect category (must be the first non-blank line␣
↪after dollar signs and between dollar signs to be detected by script).
*
* ======================= ===============================================
* Authors                 Jane Done
* Implementation Paper    `Doe and Smith, 2019 <http://labs.adsabs.harvard.edu/adsabs/
↪abs/2019Nature...30...12/>`_,
* Based on                `Newton and Halley 1692 <http://labs.adsabs.harvard.edu/adsabs/
↪abs/1692/>`_.
* C Example               :ref:`c_example_stark_force`
* Python Example          `Stark_Force.ipynb <https://github.com/dtamayo/reboundx/blob/
↪master/ipython_examples/Stark_Force.ipynb>`_,
* ======================= ===============================================
*
* This applies a constant acceleration along the x direction (Stark force)
*
```

(continues on next page)

(continued from previous page)

```
* **Effect Parameters**
*
* ============================ ===========␣
↪=================================================================
* Field (C type)               Required    Description
* ============================ ===========␣
↪=================================================================
* None                         -           -
* ============================ ===========␣
↪=================================================================
*
* **Particle Parameters**
*
* Any particles with their stark_acc parameter set will feel the corresponding␣
↪acceleration along x
*
* ============================ ===========␣
↪=================================================================
* Field (C type)               Required    Description
* ============================ ===========␣
↪=================================================================
* stark_acc (double)           No          Size of the acceleration along the x direction
* ============================ ===========␣
↪=================================================================
*
*/
```

We first add the group that our effect belongs to, between dollar signs. This keeps different implementations of, e.g., general relativity corrections in the same place. Here I made up a new one called $Documentation Examples$. If you want to make a new category like here, you have to add it to the /reboundx/doc/effect_headers.rst file.

When you create a new category in that file, you can optionally add a description general to all implementations in the category following the format in the file, which will show up in *Astrophysical Effects*.

```
$$$$$$$$$$$$$$$$$$$$$$
Documentation Examples
^^^^^^^^^^^^^^^^^^^^^^
These are effects that have been added as documentation examples
```

You can also compare with the Orbit Modifications category in that file and how it shows up in the list of effects in the documentation at *Astrophysical Effects*.

Then fill in the table: `Authors` says who wrote the code. `Implementation paper` is the paper that you'd like to be cited by people using your implementation. `Based on` is the paper that the equations you used come from.

`C Example` is a link to the C Example you wrote. All C examples in the `reboundx/examples` directory are automatically built into the documentation, and have cross-reference targets of the form `c_example_foldername`, where foldername is the name of your example folder in `reboundx/examples`. Here it's `c_example_star_force`.

For the `Python Example` line, edit the link from another documentation entry with the name of your ipython notebook filename (in both the title and bracketed URL).

Underneath your table, provide a description that will inform users when it's appropriate to apply your effect (and when it's not!).

Finally, if your effect requires the user to set (possibly optionally) particular effect or particle parameters, we create

tables for them too.

To check how everything looks the way it should, you need to

```
pip install breathe sphinx
```

and you need to install doxygen. Then

```
cd reboundx/doc/doxygen
doxygen Doxyfile
cd reboundx/doc
make clean
make html
```

Then navigate to `reboundx/doc/_build/html` and open `index.html` in your browser. The main effects page (with the tables) is on the left: REBx Effects & Parameters. The automatically included documentation will be under API Documentation (Python) and API Documentation (C).

Is your code machine independent?

This is not a requirement, but worth thinking about, given that the rest of REBOUND is machine independent, allowing anyone to replicate one another's integrations. In short, the C99 standard guarantees that arithmetic operations (+,-,*,/) and the sqrt function are machine independent. All other math library functions (e.g., sin, cos, exp etc.) are heavily optimized for hardware and can give different results (in the last bit) between architectures. If you can find a way to write your function to only use basic operations, you can be confident that your code is machine independent.

Putting together a Pull Request

If you'd rather e-mail me your code, I'm happy to incorporate it, but if you'd like for GitHub to show your account as a contributor to the project, send me a pull request!

If you've never done this before, follow the instructions at Time to Submit Your First PR up until "Tadaa!" to fork the REBOUNDx repository and make your own local branch.

Now you can modify the code as described below, and can incrementally commit changes. As a starting point, you can check out this guide.

After working through this document and making all the changes, you can then send me a pull request by following the rest of the instructions in the pull request tutorial above. We're always happy to help. Let us know if you have any questions or suggestions for how to improve this tutorial by opening an issue on the REBOUNDx GitHub page!.

INDEX

M

M_PI (*C macro*), 34

R

rebx_add_custom_force (*C++ function*), 27
rebx_add_custom_operator (*C++ function*), 27
rebx_add_force (*C++ function*), 27
rebx_add_operator (*C++ function*), 26
rebx_add_operator_step (*C++ function*), 27
rebx_attach (*C++ function*), 25
rebx_binary_field (*C++ struct*), 41
rebx_binary_field::size (*C++ member*), 41
rebx_binary_field::type (*C++ member*), 41
rebx_binary_field_type (*C++ enum*), 36
rebx_binary_field_type::REBX_BINARY_FIELD_TYPE_ADDITIONAL_FORCE (*C++ enumerator*), 37
rebx_binary_field_type::REBX_BINARY_FIELD_TYPE_ADDITIONAL_FORCES (*C++ enumerator*), 37
rebx_binary_field_type::REBX_BINARY_FIELD_TYPE_ALLOCATED_FORCES (*C++ enumerator*), 37
rebx_binary_field_type::REBX_BINARY_FIELD_TYPE_ALLOCATED_OPERATORS (*C++ enumerator*), 37
rebx_binary_field_type::REBX_BINARY_FIELD_TYPE_END (*C++ enumerator*), 37
rebx_binary_field_type::REBX_BINARY_FIELD_TYPE_FORCE (*C++ enumerator*), 37
rebx_binary_field_type::REBX_BINARY_FIELD_TYPE_FORCE_TYPE (*C++ enumerator*), 37
rebx_binary_field_type::REBX_BINARY_FIELD_TYPE_NAME (*C++ enumerator*), 36
rebx_binary_field_type::REBX_BINARY_FIELD_TYPE_NONE (*C++ enumerator*), 36
rebx_binary_field_type::REBX_BINARY_FIELD_TYPE_OPERATOR (*C++ enumerator*), 36
rebx_binary_field_type::REBX_BINARY_FIELD_TYPE_OPERATOR_TYPE (*C++ enumerator*), 37
rebx_binary_field_type::REBX_BINARY_FIELD_TYPE_PARAM (*C++ enumerator*), 36
rebx_binary_field_type::REBX_BINARY_FIELD_TYPE_PARAM_LIST (*C++ enumerator*), 37
rebx_binary_field_type::REBX_BINARY_FIELD_TYPE_PARAM_TYPE (*C++ enumerator*), 37
rebx_binary_field_type::REBX_BINARY_FIELD_TYPE_PARAM_VALUE (*C++ enumerator*), 37
rebx_binary_field_type::REBX_BINARY_FIELD_TYPE_PARTICLE (*C++ enumerator*), 36
rebx_binary_field_type::REBX_BINARY_FIELD_TYPE_PARTICLE_IN (*C++ enumerator*), 37
rebx_binary_field_type::REBX_BINARY_FIELD_TYPE_PARTICLES (*C++ enumerator*), 37
rebx_binary_field_type::REBX_BINARY_FIELD_TYPE_POST_TIMEST (*C++ enumerator*), 37
rebx_binary_field_type::REBX_BINARY_FIELD_TYPE_PRE_TIMESTE (*C++ enumerator*), 37
rebx_binary_field_type::REBX_BINARY_FIELD_TYPE_REBX_INTEGR (*C++ enumerator*), 37
rebx_binary_field_type::REBX_BINARY_FIELD_TYPE_REBX_STRUCT (*C++ enumerator*), 36
rebx_binary_field_type::REBX_BINARY_FIELD_TYPE_REGISTERED_ (*C++ enumerator*), 37
rebx_binary_field_type::REBX_BINARY_FIELD_TYPE_REGISTERED_ (*C++ enumerator*), 37
rebx_binary_field_type::REBX_BINARY_FIELD_TYPE_SNAPSHOT (*C++ enumerator*), 37
rebx_binary_field_type::REBX_BINARY_FIELD_TYPE_STEP (*C++ enumerator*), 37
rebx_binary_field_type::REBX_BINARY_FIELD_TYPE_STEP_DT_FRA (*C++ enumerator*), 37
rebx_build_str (*C++ member*), 39
rebx_calculate_tides_dynamical_mode_evolution (*C++ function*), 31
rebx_central_force_Acentral (*C++ function*), 30
rebx_central_force_potential (*C++ function*), 31
REBX_COORDINATES (*C++ enum*), 35
REBX_COORDINATES::REBX_COORDINATES_BARYCENTRIC (*C++ enumerator*), 35
REBX_COORDINATES::REBX_COORDINATES_JACOBI (*C++ enumerator*), 35
REBX_COORDINATES::REBX_COORDINATES_PARTICLE (*C++ enumerator*), 35
rebx_create_extras_from_binary (*C++ function*), 26
rebx_create_force (*C++ function*), 27
rebx_create_interpolator (*C++ function*), 33

rebx_create_operator (*C++ function*), 27
rebx_detach (*C++ function*), 26
rebx_drift_step (*C++ function*), 33
rebx_error (*C++ function*), 39
rebx_extras (*C++ struct*), 42
rebx_extras::additional_forces (*C++ member*), 42
rebx_extras::allocated_forces (*C++ member*), 42
rebx_extras::allocated_operators (*C++ member*), 42
rebx_extras::post_timestep_modifications (*C++ member*), 42
rebx_extras::pre_timestep_modifications (*C++ member*), 42
rebx_extras::registered_params (*C++ member*), 42
rebx_extras::sim (*C++ member*), 42
rebx_extras_cleanup (*C++ function*), 26
rebx_force (*C++ struct*), 40
rebx_force::ap (*C++ member*), 40
rebx_force::force_type (*C++ member*), 40
rebx_force::name (*C++ member*), 40
rebx_force::sim (*C++ member*), 40
rebx_force::update_accelerations (*C++ member*), 40
rebx_force_type (*C++ enum*), 36
rebx_force_type::REBX_FORCE_NONE (*C++ enumerator*), 36
rebx_force_type::REBX_FORCE_POS (*C++ enumerator*), 36
rebx_force_type::REBX_FORCE_VEL (*C++ enumerator*), 36
rebx_free (*C++ function*), 26
rebx_free_interpolator (*C++ function*), 33
rebx_get_force (*C++ function*), 27
rebx_get_operator (*C++ function*), 28
rebx_get_or_add_param (*C++ function*), 32
rebx_get_param (*C++ function*), 28
rebx_get_param_check (*C++ function*), 31
rebx_get_param_struct (*C++ function*), 28
rebx_githash_str (*C++ member*), 39
rebx_gr_full_hamiltonian (*C++ function*), 30
rebx_gr_hamiltonian (*C++ function*), 30
rebx_gr_potential_potential (*C++ function*), 30
rebx_gravitational_harmonics_potential (*C++ function*), 31
rebx_ias15_step (*C++ function*), 32
rebx_init_extras_from_binary (*C++ function*), 26
rebx_input_binary_messages (*C++ enum*), 38
rebx_input_binary_messages::REBX_INPUT_BINARY_ERROR_CORRUPT (*C++ enumerator*), 38
rebx_input_binary_messages::REBX_INPUT_BINARY_ERROR_NOMEMORY (*C++ enumerator*), 38
rebx_input_binary_messages::REBX_INPUT_BINARY_ERROR_NOFILE (*C++ enumerator*), 38
rebx_input_binary_messages::REBX_INPUT_BINARY_ERROR_REBX_N (*C++ enumerator*), 38
rebx_input_binary_messages::REBX_INPUT_BINARY_ERROR_REGIST (*C++ enumerator*), 38
rebx_input_binary_messages::REBX_INPUT_BINARY_WARNING_ADDI (*C++ enumerator*), 38
rebx_input_binary_messages::REBX_INPUT_BINARY_WARNING_FIEL (*C++ enumerator*), 38
rebx_input_binary_messages::REBX_INPUT_BINARY_WARNING_FORC (*C++ enumerator*), 38
rebx_input_binary_messages::REBX_INPUT_BINARY_WARNING_FORC (*C++ enumerator*), 38
rebx_input_binary_messages::REBX_INPUT_BINARY_WARNING_LIST (*C++ enumerator*), 38
rebx_input_binary_messages::REBX_INPUT_BINARY_WARNING_NONE (*C++ enumerator*), 38
rebx_input_binary_messages::REBX_INPUT_BINARY_WARNING_OPER (*C++ enumerator*), 38
rebx_input_binary_messages::REBX_INPUT_BINARY_WARNING_PARA (*C++ enumerator*), 38
rebx_input_binary_messages::REBX_INPUT_BINARY_WARNING_PARA (*C++ enumerator*), 38
rebx_input_binary_messages::REBX_INPUT_BINARY_WARNING_PART (*C++ enumerator*), 38
rebx_input_binary_messages::REBX_INPUT_BINARY_WARNING_STEP (*C++ enumerator*), 38
rebx_input_binary_messages::REBX_INPUT_BINARY_WARNING_VERS (*C++ enumerator*), 38
rebx_input_inspect_binary (*C++ function*), 34
rebx_input_read_binary_field (*C++ function*), 34
rebx_input_skip_binary_field (*C++ function*), 34
rebx_integrator (*C++ enum*), 38
rebx_integrator::REBX_INTEGRATOR_EULER (*C++ enumerator*), 39
rebx_integrator::REBX_INTEGRATOR_IMPLICIT_MIDPOINT (*C++ enumerator*), 39
rebx_integrator::REBX_INTEGRATOR_NONE (*C++ enumerator*), 39
rebx_integrator::REBX_INTEGRATOR_RK2 (*C++ enumerator*), 39
rebx_integrator::REBX_INTEGRATOR_RK4 (*C++ enumerator*), 39
rebx_interaction_step (*C++ function*), 33
rebx_interpolate (*C++ function*), 34
rebx_interpolation_type (*C++ enum*), 39
rebx_interpolation_type::REBX_INTERPOLATION_NONE (*C++ enumerator*), 39
rebx_interpolation_type::REBX_INTERPOLATION_SPLINE (*C++ enumerator*), 39
rebx_interpolator (*C++ struct*), 41
rebx_interpolator::interpolation (*C++ member*), 41

rebx_interpolator::klo (*C++ member*), 42
rebx_interpolator::Nvalues (*C++ member*), 41
rebx_interpolator::times (*C++ member*), 41
rebx_interpolator::values (*C++ member*), 41
rebx_interpolator::y2 (*C++ member*), 42
rebx_jump_step (*C++ function*), 32
rebx_kepler_step (*C++ function*), 32
rebx_kick_step (*C++ function*), 33
rebx_load_force (*C++ function*), 27
rebx_load_operator (*C++ function*), 27
rebx_node (*C++ struct*), 39
rebx_node::next (*C++ member*), 39
rebx_node::object (*C++ member*), 39
rebx_operator (*C++ struct*), 40
rebx_operator::ap (*C++ member*), 40
rebx_operator::name (*C++ member*), 40
rebx_operator::operator_type (*C++ member*), 41
rebx_operator::sim (*C++ member*), 41
rebx_operator::step_function (*C++ member*), 41
rebx_operator_type (*C++ enum*), 36
rebx_operator_type::REBX_OPERATOR_NONE (*C++ enumerator*), 36
rebx_operator_type::REBX_OPERATOR_RECORDER (*C++ enumerator*), 36
rebx_operator_type::REBX_OPERATOR_UPDATER (*C++ enumerator*), 36
rebx_output_binary (*C++ function*), 26
rebx_param (*C++ struct*), 40
rebx_param::name (*C++ member*), 40
rebx_param::type (*C++ member*), 40
rebx_param::value (*C++ member*), 40
rebx_param_type (*C++ enum*), 35
rebx_param_type::REBX_TYPE_DOUBLE (*C++ enumerator*), 35
rebx_param_type::REBX_TYPE_FORCE (*C++ enumerator*), 35
rebx_param_type::REBX_TYPE_INT (*C++ enumerator*), 35
rebx_param_type::REBX_TYPE_NONE (*C++ enumerator*), 35
rebx_param_type::REBX_TYPE_ODE (*C++ enumerator*), 35
rebx_param_type::REBX_TYPE_ORBIT (*C++ enumerator*), 35
rebx_param_type::REBX_TYPE_POINTER (*C++ enumerator*), 35
rebx_param_type::REBX_TYPE_UINT32 (*C++ enumerator*), 35
rebx_param_type::REBX_TYPE_VEC3D (*C++ enumerator*), 35
rebx_rad_calc_beta (*C++ function*), 29
rebx_rad_calc_particle_radius (*C++ function*), 29
rebx_register_param (*C++ function*), 28
rebx_remove_force (*C++ function*), 26
rebx_remove_operator (*C++ function*), 26
rebx_remove_param (*C++ function*), 28
rebx_set_param_double (*C++ function*), 28
rebx_set_param_int (*C++ function*), 28
rebx_set_param_pointer (*C++ function*), 28
rebx_set_param_uint32 (*C++ function*), 28
rebx_set_param_vec3d (*C++ function*), 28
rebx_simulation_irotate (*C++ function*), 29
rebx_spin_initialize_ode (*C++ function*), 30
rebx_step (*C++ struct*), 41
rebx_step::dt_fraction (*C++ member*), 41
rebx_tides_constant_time_lag_potential (*C++ function*), 31
rebx_tides_dynamical_mode (*C++ struct*), 42
rebx_tides_dynamical_mode::imag (*C++ member*), 43
rebx_tides_dynamical_mode::mode (*C++ member*), 43
rebx_tides_dynamical_mode::real (*C++ member*), 43
rebx_tides_dynamical_params (*C++ struct*), 42
rebx_tides_dynamical_params::dE_alpha (*C++ member*), 42
rebx_tides_dynamical_params::dP (*C++ member*), 42
rebx_tides_dynamical_params::sigma (*C++ member*), 42
rebx_tides_spin_energy (*C++ function*), 31
rebx_timing (*C++ enum*), 35
rebx_timing::REBX_TIMING_POST (*C++ enumerator*), 35
rebx_timing::REBX_TIMING_PRE (*C++ enumerator*), 35
rebx_tools_spin_angular_momentum (*C++ function*), 29
rebx_tools_spin_energy (*C++ function*), 29
rebx_version_str (*C++ member*), 39
REBXGITHASH (*C macro*), 34