

Package ‘orbitr’

October 2, 2026

Type Package

Title A Tidy Physics Engine for Building and Visualizing Orbital Simulations

Version 1.0.0

Description A lightweight, fully vectorized N-body physics engine built for the R ecosystem. Simulate and visualize complex orbital mechanics, celestial trajectories, and gravitational interactions using tidy data principles. Features multiple numerical integration methods, including the energy-conserving velocity Verlet algorithm (Verlet (1967) <doi:10.1103/PhysRev.159.98>), to ensure highly stable orbital propagation. Gravitational N-body methods follow Aarseth (2003, ISBN:0-521-43272-3).

URL <https://orbit-r.com/>, <https://github.com/DRosenman/orbitr>

BugReports <https://github.com/DRosenman/orbitr/issues>

License MIT + file LICENSE

Encoding UTF-8

Imports dplyr, ggplot2, Rcpp, tibble

Suggests plotly, gganimate, gifski, magick, knitr, rmarkdown, pkgdown, testthat (>= 3.0.0)

Config/testthat/edition 3

VignetteBuilder knitr

LinkingTo Rcpp

Config/roxygen2/version 8.1.0

NeedsCompilation yes

Author Dave Rosenman [aut, cre]

Maintainer Dave Rosenman <dave.rosenman.data@gmail.com>

Depends R (>= 4.1.0)

Repository CRAN

Date/Publication 2026-10-02 00:10:02 UTC

Contents

add_body	2
add_body_keplerian	3
add_planet	5
add_sun	7
animate_system	8
animate_system_3d	9
conserved_quantities	10
continue_simulation	11
create_system	13
export_bodies	13
get_bodies	14
get_energy	15
get_momentum	16
get_orbital_elements	17
load_solar_system	18
load_system	19
physical_constants	20
plot_orbits	23
plot_orbits_3d	24
plot_system	25
plot_system_3d	26
print.orbit_system	26
remove_body	27
save_system	28
shift_reference_frame	29
simulate_system	30
system_from_simulation	31
Index	33

add_body	<i>Add a physical body to the system</i>
----------	--

Description

This function introduces a new celestial or physical body into your ‘orbit_system’. You must provide a unique identifier and its mass. By default, the body will be placed at the origin (0, 0, 0) with zero initial velocity unless specified.

Usage

```
add_body(system, id, mass, x = 0, y = 0, z = 0, vx = 0, vy = 0, vz = 0)
```

Arguments

system	An 'orbit_system' object created by 'create_system()'.
id	A unique character string to identify the body (e.g., "Earth", "Apollo").
mass	The mass of the object in kilograms.
x	Initial X-axis position in meters (default 0).
y	Initial Y-axis position in meters (default 0).
z	Initial Z-axis position in meters (default 0).
vx	Initial velocity along the X-axis in meters per second (default 0).
vy	Initial velocity along the Y-axis in meters per second (default 0).
vz	Initial velocity along the Z-axis in meters per second (default 0).

Value

The updated 'orbit_system' object containing the newly added body.

Examples

```
my_universe <- create_system() |>
  add_body(id = "Earth", mass = 5.97e24) |>
  add_body(id = "Moon", mass = 7.34e22, x = 3.84e8, vy = 1022)
```

add_body_keplerian	<i>Add a body using Keplerian orbital elements</i>
--------------------	--

Description

A convenience wrapper around [add_body()] that lets you specify an orbit using classical Keplerian elements instead of raw Cartesian state vectors. The elements are converted to position and velocity in the reference frame of the parent body, which must already exist in the system.

Usage

```
add_body_keplerian(
  system,
  id,
  mass,
  a,
  e = 0,
  i = 0,
  lan = 0,
  arg_pe = 0,
  nu = 0,
  parent
)
```

Arguments

system	An 'orbit_system' object created by [create_system()].
id	A unique character string to identify the body.
mass	The mass of the body in kilograms.
a	Semi-major axis in meters. Positive for a bound orbit ('e < 1'); negative for a hyperbolic orbit ('e > 1'), in which case the periapsis distance is 'a * (1 - e)', which is positive.
e	Eccentricity (0 = circle, 0 < e < 1 = ellipse, e > 1 = hyperbola). Default 0. Exactly parabolic orbits ('e = 1') are not supported; use a value slightly above or below 1.
i	Inclination in degrees. Default 0.
lan	Longitude of ascending node in degrees. Default 0.
arg_pe	Argument of periapsis in degrees. Default 0.
nu	True anomaly in degrees. Default 0 (body starts at periapsis). For a hyperbolic orbit, 'nu' must lie between the asymptotes, 'abs(nu) < acos(-1/e) * 180 / pi'.
parent	Character id of the parent body (must already exist in 'system'). The orbital elements are defined relative to this body.

Value

The updated 'orbit_system' with the new body added.

Keplerian Elements

Six numbers fully describe a Keplerian orbit:

- 'a' (semi-major axis)** The size of the orbit — half the longest diameter of the ellipse, in meters. Negative for a hyperbolic orbit.
- 'e' (eccentricity)** The shape of the orbit. 0 is a perfect circle; values between 0 and 1 are ellipses; values above 1 are hyperbolas (unbound flybys).
- 'i' (inclination)** The tilt of the orbital plane relative to the reference plane, in degrees.
- 'lan' (longitude of ascending node)** The angle from the reference direction to where the orbit crosses the reference plane going "upward," in degrees. Sometimes written as Ω .
- 'arg_pe' (argument of periapsis)** The angle within the orbital plane from the ascending node to the closest-approach point, in degrees. Sometimes written as ω .
- 'nu' (true anomaly)** Where the body currently sits along its orbit, measured as an angle from periapsis in degrees. 0 = at periapsis (closest), 180 = at apoapsis (farthest).

Examples

```
# Earth orbiting the Sun with real orbital elements
system <- create_system() |>
  add_sun() |>
  add_body_keplerian(
    "Earth", mass = mass_earth,
```

```

    a = distance_earth_sun, e = 0.0167, i = 0.00005,
    parent = "Sun"
  )

# An interstellar visitor on a hyperbolic orbit (e > 1 needs a < 0):
# 'Oumuamua-like, perihelion 0.255 AU, approaching from 140 degrees
# before perihelion
q <- 0.2553 * distance_earth_sun
e <- 1.2
system <- system |>
  add_body_keplerian(
    "Visitor", mass = 1e10,
    a = -q / (e - 1), e = e, i = 122.7, nu = -140,
    parent = "Sun"
  )

# Mars with its notable eccentricity
system <- system |>
  add_body_keplerian(
    "Mars", mass = mass_mars,
    a = distance_mars_sun, e = 0.0934, i = 1.85,
    lan = 49.6, arg_pe = 286.5, nu = 0,
    parent = "Sun"
  )

```

add_planet

Add a known solar system body by name

Description

A convenience wrapper around [add_body_keplerian()] that looks up real orbital elements for well-known solar system bodies. Instead of typing out mass, semi-major axis, eccentricity, and inclination by hand, just give the name and parent:

Usage

```

add_planet(
  system,
  name,
  parent,
  nu = 0,
  a = NULL,
  e = NULL,
  i = NULL,
  lan = NULL,
  arg_pe = NULL,
  mass = NULL
)

```

Arguments

system	An 'orbit_system' object.
name	The name of the body. Must be one of: "Mercury", "Venus", "Earth", "Mars", "Jupiter", "Saturn", "Uranus", "Neptune", "Moon", or "Pluto". Case-sensitive.
parent	Character id of the parent body, which must already exist in the system. For planets and Pluto this is typically "Sun"; for the Moon it is "Earth".
nu	True anomaly in degrees (default 0, body starts at periapsis). This is the most commonly overridden element — use it to spread planets around their orbits instead of starting them all at periapsis.
a	Override semi-major axis (meters).
e	Override eccentricity.
i	Override inclination (degrees).
lan	Override longitude of ascending node (degrees).
arg_pe	Override argument of periapsis (degrees).
mass	Override mass (kg).

Details

```
““ create_system() |> add_sun() |> add_planet("Earth", parent = "Sun") |> add_planet("Moon", parent = "Earth") ““
```

Any Keplerian element can be overridden to explore "what if" scenarios while keeping the rest of the real values:

```
““ # What if Mars had zero eccentricity? add_planet("Mars", parent = "Sun", e = 0) ““
```

Value

The updated 'orbit_system' with the body added.

Examples

```
# Build the inner solar system
create_system() |>
  add_sun() |>
  add_planet("Mercury", parent = "Sun") |>
  add_planet("Venus", parent = "Sun") |>
  add_planet("Earth", parent = "Sun") |>
  add_planet("Mars", parent = "Sun") |>
  simulate_system(time_step = seconds_per_day, duration = seconds_per_year) |>
  plot_orbits()

# Earth-Moon system
create_system() |>
  add_body("Earth", mass = mass_earth) |>
  add_planet("Moon", parent = "Earth") |>
  simulate_system(time_step = seconds_per_hour, duration = seconds_per_day * 28) |>
```

```

    plot_orbits()

# What if Jupiter were twice as massive?
create_system() |>
  add_sun() |>
    add_planet("Jupiter", parent = "Sun", mass = mass_jupiter * 2)

```

add_sun

Add the Sun to the system

Description

A convenience function that adds the Sun as the central body of a simulation. By default it is placed at the origin with zero velocity, which is the natural choice for a heliocentric reference frame. Position and velocity can be overridden for advanced use cases such as barycentric coordinates.

Usage

```
add_sun(system, mass = mass_sun, x = 0, y = 0, z = 0, vx = 0, vy = 0, vz = 0)
```

Arguments

system	An ‘orbit_system’ object created by [create_system()].
mass	Mass of the Sun in kilograms. Defaults to [mass_sun] (1.989 x 10 ³⁰ kg).
x	Initial X-axis position in meters (default 0).
y	Initial Y-axis position in meters (default 0).
z	Initial Z-axis position in meters (default 0).
vx	Initial velocity along the X-axis in m/s (default 0).
vy	Initial velocity along the Y-axis in m/s (default 0).
vz	Initial velocity along the Z-axis in m/s (default 0).

Details

This pairs naturally with [add_planet()]:

```
““ create_system() |> add_sun() |> add_planet("Earth", parent = "Sun") |> add_planet("Mars", parent = "Sun") ““
```

Value

The updated ‘orbit_system’ with the Sun added.

Examples

```
# Typical usage - Sun at the origin
create_system() |>
  add_sun()

# Full solar system in three lines
create_system() |>
  add_sun() |>
  add_planet("Earth", parent = "Sun") |>
  add_planet("Mars", parent = "Sun") |>
  simulate_system(time_step = seconds_per_day, duration = seconds_per_year) |>
  plot_orbits()
```

animate_system	<i>Animate the System Over Time (Smart 2D/3D Dispatch)</i>
----------------	--

Description

Plays the simulation forward as an animation. Bodies move through their orbits frame by frame, optionally leaving a fading wake behind them. This is the animated counterpart to `[plot_system()]` — a moving snapshot rather than a single frozen one.

Usage

```
animate_system(
  sim_data,
  fps = 20,
  duration = 10,
  trails = FALSE,
  three_d = NULL
)
```

Arguments

<code>sim_data</code>	A tibble output from <code>[simulate_system()]</code> .
<code>fps</code>	Frames per second of the rendered animation. Default ‘20’.
<code>duration</code>	Length of the animation in seconds. Default ‘10’. Together with ‘fps’, this determines how many simulation time steps are sampled into frames (‘fps * duration’). If your simulation has fewer steps than that, every step becomes a frame.
<code>trails</code>	Logical. If ‘TRUE’ (the default), each body leaves a fading wake of its recent positions behind it. Set ‘FALSE’ for naked moving dots.
<code>three_d</code>	Logical. If ‘TRUE’, forces a 3D animation even for planar data.

Details

If any body has non-zero motion in the Z dimension (or `'three_d = TRUE'`), `[animate_system_3d()]` is used; otherwise a 2D `'ganimate'` animation is returned.

The 2D path requires the `'ganimate'` package, which is in `'Suggests'`. Install it with `'install.packages("ganimate")'`. Rendering a 2D animation is much slower than a static plot — expect tens of seconds for typical simulations, since every frame is drawn and encoded as a GIF (or MP4).

The 3D path uses `'plotly'`'s built-in `'frame'` aesthetic, which produces an interactive HTML widget with a play button and time slider. No GIF encoding is involved, so 3D animations render essentially instantly.

Value

A rendered `'ganimate'` animation (2D) or a `'plotly'` HTML widget with built-in play/pause controls (3D). The 2D return value can be saved to disk with `[ganimate::anim_save()]`.

Examples

```
sim <- create_system() |>
  add_sun() |>
  add_body("Earth", mass = mass_earth, x = distance_earth_sun, vy = speed_earth) |>
  simulate_system(time_step = seconds_per_day, duration = seconds_per_year)

# 2D fading-wake animation (requires ganimate)
anim <- animate_system(sim, fps = 20, duration = 8)
anim

# Save to disk
ganimate::anim_save(file.path(tempdir(), "earth_orbit.gif"), anim)
```

`animate_system_3d`*Animate the System Over Time in Interactive 3D*

Description

The 3D counterpart to `[animate_system()]`. Builds a `'plotly'` 3D scene with the bodies as moving markers and an interactive Play / Pause control plus a time slider. Optionally shows the full orbit paths drawn faintly behind.

Usage

```
animate_system_3d(sim_data, fps = 20, duration = 10, trails = FALSE)
```

Arguments

sim_data	A tibble output from [simulate_system()].
fps	Frames per second target for playback. Default '20'. Combined with 'duration', controls how many time steps are sampled into frames.
duration	Total playback length in seconds. Default '10'.
trails	Logical. If 'TRUE' (the default), the full orbit paths are drawn faintly behind the animated markers.

Value

A 'plotly' HTML widget with a built-in play button and time slider.

Examples

```
create_system() |>
  add_body("Earth", mass = mass_earth) |>
  add_body("Moon", mass = mass_moon,
    x = distance_earth_moon, vy = speed_moon, vz = 100) |>
  simulate_system(time_step = seconds_per_hour, duration = seconds_per_day * 30) |>
  animate_system_3d()
```

conserved_quantities *Conserved quantities and how well a simulation conserved them*

Description

Joins [get_energy()], [get_momentum()], and [get_angular_momentum()] into one tibble and adds the relative error of each quantity against its value at the first time step. In exact Newtonian gravity all three are constant, so the errors measure the integrator, and their shape tells you what is wrong when something is.

Usage

```
conserved_quantities(sim_data, G = NULL, softening = NULL)
```

Arguments

sim_data	A tibble output from [simulate_system()].
G	The gravitational constant the simulation was run with. Defaults to the value recorded by [simulate_system()] in the "'G'" attribute of 'sim_data', or to [gravitational_constant] if that attribute is absent.
softening	The softening length (in meters) the simulation was run with. The potential energy is computed with the same softened distance, ' $\sqrt{r^2 + \text{softening}^2}$ ', that the force used; if the two do not match, the energy will appear to drift when it has not. Defaults to the value recorded by [simulate_system()], or 0.

Details

‘energy_error’ is $(E - E_0)/|E_0|$. ‘momentum_error’ is $|\mathbf{P} - \mathbf{P}_0|$ divided by $\sum_j m_j |\mathbf{v}_j|$ at the first step (total momentum is often exactly zero, so it cannot be its own scale). ‘angular_momentum_error’ is $|\mathbf{L} - \mathbf{L}_0|/|\mathbf{L}_0|$. An error is ‘NA’ when its scale is zero.

Value

A tibble with one row per time step and columns ‘time’, ‘kinetic’, ‘potential’, ‘energy’, ‘px’, ‘py’, ‘pz’, ‘Lx’, ‘Ly’, ‘Lz’, ‘energy_error’, ‘momentum_error’, and ‘angular_momentum_error’.

What to expect

The Velocity Verlet and Euler-Cromer integrators are built from "kicks" and "drifts" that conserve linear and angular momentum exactly, so for ‘method = "verlet"’ or ‘euler_cromer’ those two errors should stay at the level of floating-point rounding (around 1e-15) for any time step. Energy is conserved only approximately: with Verlet its error oscillates once per orbit within a band whose width scales as the square of the time step, and does not grow. A steady drift in energy means the time step is too large for the fastest or most eccentric orbit in the system, or that ‘method = "euler"’ was used. A sudden jump marks a close encounter the step could not resolve. A drift in momentum or angular momentum under Verlet points to a problem in the setup rather than the integrator.

Examples

```
sim <- create_system() |>
  add_body("Star", mass = 1e30) |>
  add_body("Planet", mass = 1e24, x = 1e11, vy = 30000) |>
  simulate_system(time_step = seconds_per_hour * 6,
    duration = seconds_per_year * 2)

cq <- conserved_quantities(sim)

# Verlet: bounded energy error, momenta at rounding level
range(cq$energy_error)
max(cq$angular_momentum_error)

plot(cq$time / seconds_per_day, cq$energy_error, type = "l",
  xlab = "Day", ylab = "Relative energy error")
```

continue_simulation *Continue a simulation from its last state*

Description

Rebuilds the system from the final time step of a simulation, runs it forward, and appends the new rows with ‘time’ continuing from where the previous run ended. Use it to extend a run without starting over, to change the time step partway through (small steps through a close approach, large ones elsewhere), or to apply a change between segments, such as a velocity kick or an added body, by editing the tibble before continuing.

Usage

```
continue_simulation(sim_data, time_step, duration, G = NULL, ...)
```

Arguments

<code>sim_data</code>	A tibble output from <code>[simulate_system()]</code> or from a previous call to <code>'continue_simulation()'</code> .
<code>time_step</code>	The time increment per step in seconds for the new segment. Need not match the previous segment's.
<code>duration</code>	Total time in seconds to simulate in the new segment.
<code>G</code>	The gravitational constant. Defaults to the value recorded by <code>[simulate_system()]</code> , or to <code>[gravitational_constant]</code> .
<code>...</code>	Further arguments passed to <code>[simulate_system()]</code> : <code>'method'</code> , <code>'softening'</code> , and <code>'use_cpp'</code> . When not supplied, <code>'method'</code> and <code>'softening'</code> default to the values recorded from the previous segment.

Details

The last row of each body in `'sim_data'` is a complete state, so the restart is exact: the new segment begins from precisely where the old one stopped. The new segment's first step duplicates the old segment's last and is dropped, so `'time'` is strictly increasing in the result.

Each segment is integrated independently with a fixed step. Changing the step between segments disturbs the integration by about one step's worth of error at the switch, which is usually far smaller than the error saved by using a small step only where it is needed. A comet's perihelion passage is the typical use: run with a step of days out to a few AU, a step of hours through perihelion, and days again on the way out.

Value

A tibble with the same columns as `'sim_data'` and the new time steps appended.

Examples

```
# A comet on a highly eccentric orbit, started at aphelion
comet <- create_system() |>
  add_sun() |>
  add_body_keplerian("Comet", mass = 1e14, parent = "Sun",
    a = 5 * distance_earth_sun, e = 0.9, nu = 180)

# Coarse steps on the way in, fine steps through perihelion, coarse again
sim <- comet |>
  simulate_system(time_step = seconds_per_day * 5,
    duration = seconds_per_year * 5) |>
  continue_simulation(time_step = seconds_per_hour * 2,
    duration = seconds_per_year * 1.2) |>
  continue_simulation(time_step = seconds_per_day * 5,
    duration = seconds_per_year * 5)

range(sim$time) / seconds_per_year
```

create_system	<i>Initialize an orbitr simulation system</i>
---------------	---

Description

Sets up the foundational data structure for an orbital simulation. Universal N-body gravity is automatically integrated into the system using the specified gravitational constant.

Usage

```
create_system(G = gravitational_constant)
```

Arguments

G	The gravitational constant. Defaults to the real-world value ('gravitational_constant', 6.6743e-11 m ³ kg ⁻¹ s ⁻²). To simulate a zero-gravity environment (inertia only), set 'G = 0'.
---	---

Value

An empty 'orbit_system' object ready for bodies to be added.

Examples

```
# Creates a system with standard gravity
my_universe <- create_system()

# Creates a universe with 10x stronger gravity
heavy_universe <- create_system(G = gravitational_constant * 10)

# Creates a zero-gravity sandbox
floating_universe <- create_system(G = 0)
```

export_bodies	<i>Export body states to CSV</i>
---------------	----------------------------------

Description

Writes the body table (id, mass, position, and velocity) from an 'orbit_system' to a CSV file. This is useful for sharing initial conditions with collaborators or loading them into other tools like Python or Excel.

Usage

```
export_bodies(system, path)
```

Arguments

system An 'orbit_system' object.
 path File path to save to. Should end in '.csv'.

Value

'system', invisibly.

Examples

```
sys <- create_system() |>
  add_sun() |>
  add_planet("Earth", parent = "Sun") |>
  add_planet("Mars", parent = "Sun")

export_bodies(sys, file.path(tempdir(), "bodies.csv"))
```

get_bodies	<i>Extract the body table from a system</i>
------------	---

Description

Returns the bodies in an 'orbit_system' as a standalone tibble. Useful when you want to inspect, filter, or save the body states without dealing with the full system object.

Usage

```
get_bodies(system)
```

Arguments

system An 'orbit_system' object.

Value

A tibble with columns 'id', 'mass', 'x', 'y', 'z', 'vx', 'vy', 'vz'.

Examples

```
sys <- create_system() |>
  add_sun() |>
  add_planet("Earth", parent = "Sun") |>
  add_planet("Mars", parent = "Sun")

# Get the tibble
get_bodies(sys)
```

```
# Use with dplyr

get_bodies(sys) |>
  dplyr::filter(mass > 1e24)
```

get_energy	<i>Total energy of a simulated system</i>
------------	---

Description

Computes the total kinetic energy, total gravitational potential energy, and their sum at every time step of a simulation. In exact Newtonian gravity the total is constant; with a numerical integrator it is not, and how far it wanders is a direct measure of integration error (see [conserved_quantities()] for the error itself).

Usage

```
get_energy(sim_data, G = NULL, softening = NULL)
```

Arguments

sim_data	A tibble output from [simulate_system()].
G	The gravitational constant the simulation was run with. Defaults to the value recorded by [simulate_system()] in the "G" attribute of 'sim_data', or to [gravitational_constant] if that attribute is absent.
softening	The softening length (in meters) the simulation was run with. The potential energy is computed with the same softened distance, 'sqrt(r^2 + softening^2)', that the force used; if the two do not match, the energy will appear to drift when it has not. Defaults to the value recorded by [simulate_system()], or 0.

Details

At each time step, 'kinetic' is $\sum_j \frac{1}{2} m_j v_j^2$, 'potential' is $-\sum_{j < k} G m_j m_k / \sqrt{r_{jk}^2 + \varepsilon^2}$ summed over every unordered pair of bodies, and 'energy' is their sum. Potential energy is a property of pairs, not of individual bodies, which is why the function reports system totals only.

Value

A tibble with one row per time step and columns 'time', 'kinetic', 'potential', and 'energy', all in joules.

Examples

```
sim <- create_system() |>
  add_body("Star", mass = 1e30) |>
  add_body("Planet", mass = 1e24, x = 1e11, vy = 30000) |>
  simulate_system(time_step = seconds_per_hour * 6,
                  duration = seconds_per_year)

energy <- get_energy(sim)
energy

# Kinetic and potential trade off along the orbit; the total barely moves
plot(energy$time / seconds_per_day, energy$kinetic, type = "l",
     xlab = "Day", ylab = "Kinetic energy (J)")
```

get_momentum	<i>Total linear and angular momentum of a simulated system</i>
--------------	--

Description

‘get_momentum()’ computes the total linear momentum $\mathbf{P} = \sum_j m_j \mathbf{v}_j$ and ‘get_angular_momentum()’ the total angular momentum about the origin, $\mathbf{L} = \sum_j m_j \mathbf{r}_j \times \mathbf{v}_j$, at every time step of a simulation.

Usage

```
get_momentum(sim_data)

get_angular_momentum(sim_data)
```

Arguments

sim_data A tibble output from [simulate_system()].

Details

Both are conserved exactly by Newtonian gravity, because every force comes in an equal and opposite pair directed along the line between the two bodies. The Velocity Verlet and Euler-Cromer integrators also conserve both to floating-point rounding, at any time step, so a drift in either points to a problem in the setup rather than the step size. A system whose total momentum is not zero has a center of mass that drifts at a constant velocity; see ‘shift_reference_frame(sim_data, "barycenter")’.

Value

A tibble with one row per time step: ‘time, px, py, pz’ (kg m/s) for ‘get_momentum()’, and ‘time, Lx, Ly, Lz’ (kg m²/s) for ‘get_angular_momentum()’.

Examples

```
# A binary built with zero total momentum
sim <- create_system() |>
  add_body("A", mass = 2e30, x = 5e10, vy = 15000) |>
  add_body("B", mass = 1e30, x = -1e11, vy = -30000) |>
  simulate_system(time_step = seconds_per_hour, duration = seconds_per_year)

get_momentum(sim)          # px, py, pz all zero to rounding
get_angular_momentum(sim)  # Lz constant to rounding
```

get_orbital_elements *Osculating orbital elements of a body relative to a parent*

Description

The inverse of [add_body_keplerian()]: computes the classical Keplerian elements of ‘body’'s orbit about ‘parent’ at every time step of a simulation, from their relative position and velocity. In a two-body system the elements are constant (up to integration error). With more bodies present they drift as the orbit is perturbed, and the value at each instant is the *osculating* orbit: the ellipse the body would follow from that moment on if every other perturbation were switched off.

Usage

```
get_orbital_elements(sim_data, body, parent, mu = NULL, G = NULL)
```

Arguments

sim_data	A tibble output from [simulate_system()].
body	Character id of the orbiting body.
parent	Character id of the body the orbit is measured about.
mu	Gravitational parameter in m^3/s^2 . The default, ‘NULL’, uses GM_{parent} , the same convention as [add_body_keplerian()], so that elements round-trip exactly. Pass ‘G * (parent mass + body mass)’ for the exact two-body relative orbit, which matters when the body’s mass is not negligible (the Moon’s is 1.2% of Earth’s).
G	The gravitational constant, used only when ‘mu’ is ‘NULL’. Defaults to the value recorded by [simulate_system()], or to [gravitational_constant].

Details

The computation follows the standard textbook route. The specific angular momentum $\mathbf{h} = \mathbf{r} \times \mathbf{v}$ gives the inclination ($\cos i = h_z/h$) and, through the node vector $\hat{\mathbf{z}} \times \mathbf{h}$, the longitude of the ascending node. The eccentricity vector $\mathbf{e} = (\mathbf{v} \times \mathbf{h})/\mu - \mathbf{r}/r$ points to periapsis and gives the eccentricity, the argument of periapsis (the angle from the node to $\mathbf{e}$), and the true anomaly (the angle from $\mathbf{e}$ to $\mathbf{r}$). The semi-major axis comes from the vis-viva equation, $a = 1/(2/r - v^2/\mu)$.

Two cases are degenerate and need a convention. For an orbit in the reference plane ($i = 0$) the ascending node is undefined: 'lan' is reported as 0 and 'arg_pe' is measured from the x axis. For a circular orbit ($e = 0$) periapsis is undefined: 'arg_pe' is reported as 0 and 'nu' is measured from the ascending node (or the x axis).

For an unbound orbit ($e \geq 1$) 'a' is negative and 'period' is 'NA'.

Value

A tibble with one row per time step and columns 'time', 'a' (meters), 'e', 'i', 'lan', 'arg_pe', 'nu' (degrees; angles other than 'i' are in $[0, 360)$), and 'period' (seconds).

Examples

```
# Round trip: the elements that went in come back out at t = 0
sim <- create_system() |>
  add_sun() |>
  add_body_keplerian("Mars", mass = mass_mars, parent = "Sun",
                    a = distance_mars_sun, e = 0.0934, i = 1.85,
                    lan = 49.6, arg_pe = 286.5, nu = 120) |>
  simulate_system(time_step = seconds_per_day, duration = seconds_per_day * 687)

elements <- get_orbital_elements(sim, "Mars", "Sun")
elements[1, ]

# Two bodies: a and e are constant to integration error
range(elements$e)

# Earth's eccentricity drifting under Jupiter's pull
perturbed <- create_system() |>
  add_sun() |>
  add_planet("Earth", parent = "Sun") |>
  add_planet("Jupiter", parent = "Sun", nu = 90) |>
  simulate_system(time_step = seconds_per_day, duration = seconds_per_year * 12)

earth <- get_orbital_elements(perturbed, "Earth", "Sun")
plot(earth$time / seconds_per_year, earth$e, type = "l",
     xlab = "Years", ylab = "Osculating eccentricity")
```

load_solar_system	<i>Load a pre-built solar system</i>
-------------------	--------------------------------------

Description

A convenience function that creates a complete solar system with the Sun and all eight planets (plus optionally the Moon and Pluto) using real orbital data. Bodies are placed using Keplerian orbital elements from the JPL DE440 ephemeris (J2000 epoch), giving realistic eccentricities, inclinations, and orbital orientations out of the box.

Usage

```
load_solar_system(moon = TRUE, pluto = TRUE)
```

Arguments

moon	Logical. If 'TRUE' (the default), include the Moon in orbit around Earth.
pluto	Logical. If 'TRUE' (the default), include Pluto.

Details

This is a quick way to get a physically accurate starting point without typing out a dozen [add_body()] calls. The returned system is a normal 'orbit_system' that you can modify further — add bodies, change parameters, or pipe straight into [simulate_system()].

Value

An 'orbit_system' object containing the Sun and planets, ready for simulation.

Examples

```
# Simulate the full solar system for one year
solar <- load_solar_system() |>
  simulate_system(
    time_step = seconds_per_day,
    duration = seconds_per_year
  )

plot_orbits(solar)

# Just the Sun and planets, no Moon or Pluto
load_solar_system(moon = FALSE, pluto = FALSE)
```

load_system	<i>Load an orbit_system from disk</i>
-------------	---------------------------------------

Description

Restores an 'orbit_system' previously saved with [save_system()].

Usage

```
load_system(path)
```

Arguments

path	File path to an '.rds' file created by [save_system()].
------	---

Value

An 'orbit_system' object.

Examples

```
sys <- create_system() |>
  add_sun() |>
  add_planet("Earth", parent = "Sun")

path <- file.path(tempdir(), "my_system.rds")
save_system(sys, path)
restored <- load_system(path)
```

physical_constants	<i>Physical Constants for Orbital Mechanics</i>
--------------------	---

Description

A curated set of real-world masses and orbital distances for use as convenient starting points in 'orbitr' simulations. All values are in SI units (kilograms and meters).

Usage

```
gravitational_constant

seconds_per_hour

seconds_per_day

seconds_per_year

mass_sun

mass_earth

mass_moon

mass_mars

mass_jupiter

mass_saturn

mass_venus

mass_mercury
```

mass_uranus

mass_neptune

mass_pluto

distance_earth_sun

distance_earth_moon

distance_mars_sun

distance_jupiter_sun

distance_venus_sun

distance_mercury_sun

distance_saturn_sun

distance_uranus_sun

distance_neptune_sun

distance_pluto_sun

speed_earth

speed_moon

speed_mars

speed_jupiter

speed_venus

speed_mercury

speed_saturn

speed_uranus

speed_neptune

speed_pluto

Format

Numeric scalar in kilograms.

Details

`'gravitational_constant'`: Newton's gravitational constant ($6.6743 \times 10^{-11} \text{ m}^3 \text{ kg}^{-1} \text{ s}^{-2}$). Source: CODATA 2018 recommended value. Use this with `'create_system()'` to scale gravity: `'create_system(G = gravitational_constant * 10)'`.

`'seconds_per_hour'`: 3,600 seconds. Convenient for setting `'time_step'` in lunar or close-orbit simulations.

`'seconds_per_day'`: 86,400 seconds. Convenient for setting `'time_step'` in planetary-scale simulations.

`'seconds_per_year'`: 31,557,600 seconds (365.25 days, the Julian year). Convenient for setting `'duration'` in `'simulate_system()'`.

`'mass_sun'`: Mass of the Sun ($1.989 \times 10^{30} \text{ kg}$). Source: IAU 2015 nominal solar mass.

`'mass_earth'`: Mass of the Earth ($5.972 \times 10^{24} \text{ kg}$). Source: IAU 2015 nominal Earth mass.

`'mass_moon'`: Mass of the Moon ($7.342 \times 10^{22} \text{ kg}$). Source: JPL DE440 ephemeris.

`'mass_mars'`: Mass of Mars ($6.417 \times 10^{23} \text{ kg}$). Source: JPL DE440 ephemeris.

`'mass_jupiter'`: Mass of Jupiter ($1.898 \times 10^{27} \text{ kg}$). Source: JPL DE440 ephemeris.

`'mass_saturn'`: Mass of Saturn ($5.683 \times 10^{26} \text{ kg}$). Source: JPL DE440 ephemeris.

`'mass_venus'`: Mass of Venus ($4.867 \times 10^{24} \text{ kg}$). Source: JPL DE440 ephemeris.

`'mass_mercury'`: Mass of Mercury ($3.301 \times 10^{23} \text{ kg}$). Source: JPL DE440 ephemeris.

`'mass_uranus'`: Mass of Uranus ($8.681 \times 10^{25} \text{ kg}$). Source: JPL DE440 ephemeris.

`'mass_neptune'`: Mass of Neptune ($1.024 \times 10^{26} \text{ kg}$). Source: JPL DE440 ephemeris.

`'mass_pluto'`: Mass of Pluto ($1.309 \times 10^{22} \text{ kg}$). Source: JPL DE440 ephemeris. Pluto is a dwarf planet but is included for convenience.

`'distance_earth_sun'`: Semi-major axis of Earth's orbit around the Sun ($1.496 \times 10^{11} \text{ m}$, ~149.6 million km). Earth's actual distance varies between ~147.1 million km (perihelion) and ~152.1 million km (aphelion).

`'distance_earth_moon'`: Semi-major axis of the Moon's orbit around Earth ($3.844 \times 10^8 \text{ m}$, ~384,400 km). The Moon's actual distance varies between ~363,300 km (perigee) and ~405,500 km (apogee).

`'distance_mars_sun'`: Semi-major axis of Mars's orbit around the Sun ($2.279 \times 10^{11} \text{ m}$, ~227.9 million km). Mars has a notably eccentric orbit ($e = 0.093$), ranging from ~206.7 million km to ~249.2 million km.

`'distance_jupiter_sun'`: Semi-major axis of Jupiter's orbit around the Sun ($7.785 \times 10^{11} \text{ m}$, ~778.5 million km).

`'distance_venus_sun'`: Semi-major axis of Venus's orbit around the Sun ($1.082 \times 10^{11} \text{ m}$, ~108.2 million km). Venus has the most circular orbit of any planet ($e = 0.007$).

`'distance_mercury_sun'`: Semi-major axis of Mercury's orbit around the Sun ($5.791 \times 10^{10} \text{ m}$, ~57.9 million km). Mercury has the most eccentric planetary orbit ($e = 0.206$), ranging from ~46.0 million km to ~69.8 million km.

'distance_saturn_sun': Semi-major axis of Saturn's orbit around the Sun (1.434×10^{12} m, ~1.434 billion km).

'distance_uranus_sun': Semi-major axis of Uranus's orbit around the Sun (2.871×10^{12} m, ~2.871 billion km).

'distance_neptune_sun': Semi-major axis of Neptune's orbit around the Sun (4.495×10^{12} m, ~4.495 billion km).

'distance_pluto_sun': Semi-major axis of Pluto's orbit around the Sun (5.906×10^{12} m, ~5.906 billion km). Pluto has a highly eccentric orbit ($e = 0.249$), ranging from ~4.437 billion km to ~7.376 billion km.

'speed_earth': Mean orbital speed of Earth around the Sun (29,780 m/s).

'speed_moon': Mean orbital speed of the Moon around Earth (1,022 m/s).

'speed_mars': Mean orbital speed of Mars around the Sun (24,070 m/s).

'speed_jupiter': Mean orbital speed of Jupiter around the Sun (13,060 m/s).

'speed_venus': Mean orbital speed of Venus around the Sun (35,020 m/s).

'speed_mercury': Mean orbital speed of Mercury around the Sun (47,360 m/s).

'speed_saturn': Mean orbital speed of Saturn around the Sun (9,680 m/s).

'speed_uranus': Mean orbital speed of Uranus around the Sun (6,800 m/s).

'speed_neptune': Mean orbital speed of Neptune around the Sun (5,430 m/s).

'speed_pluto': Mean orbital speed of Pluto around the Sun (4,740 m/s).

A Note on "Distance" Constants

Orbital distances are not truly constant. Every orbit is an ellipse, so the separation between two bodies changes continuously. The distances provided here are **semi-major axes** — the average of the closest approach (periapsis) and farthest point (apoapsis). The semi-major axis is the single most characteristic length scale of an elliptical orbit: it determines the orbital period via Kepler's Third Law, and when paired with the circular velocity at that distance, it produces a near-circular orbit that closely approximates the real trajectory.

For example, the Earth-Sun distance varies from about 147.1 million km (perihelion in January) to 152.1 million km (aphelion in July). The semi-major axis of 149.598 million km sits right in the middle and gives the correct orbital period of one year.

plot_orbits

Plot Orbital Trajectories (Smart 2D/3D Dispatch)

Description

Plot Orbital Trajectories (Smart 2D/3D Dispatch)

Usage

```
plot_orbits(sim_data, three_d = NULL)
```

Arguments

sim_data	A tibble output from ‘simulate_system()’
three_d	Logical. If TRUE, forces a 3D plot even for 2D data.

Value

A ‘ggplot’ object (2D) or a ‘plotly’ HTML widget (3D) showing the orbital trajectories of all bodies in the simulation.

plot_orbits_3d	<i>Plot 3D Interactive Orbital Trajectories</i>
----------------	---

Description

Generates an interactive 3D visualization of the orbital system using plotly. You can click, drag to rotate, and scroll to zoom in on the trajectories.

Usage

```
plot_orbits_3d(sim_data)
```

Arguments

sim_data	A tibble containing the simulation output from ‘simulate_system()’.
----------	---

Value

A plotly HTML widget displaying the 3D orbits.

Examples

```
create_system() |>
  add_body("Earth", mass = mass_earth) |>
  add_body("Moon", mass = mass_moon,
           x = distance_earth_moon, vy = speed_moon, vz = 150) |>
  simulate_system(time_step = seconds_per_hour, duration = seconds_per_day * 30) |>
  plot_orbits_3d()
```

plot_system	<i>Plot System Snapshot at a Single Time (Smart 2D/3D Dispatch)</i>
-------------	---

Description

Plots the position of every body in the system at a single time step, optionally with the full orbital trajectories drawn faintly behind. This is the snapshot counterpart to [plot_orbits()], which draws full trajectories.

Usage

```
plot_system(sim_data, time = NULL, trails = FALSE, three_d = NULL)
```

Arguments

sim_data	A tibble output from [simulate_system()].
time	Time (in simulation seconds) to snapshot. Defaults to the last time step. The function snaps to the closest available time in the data.
trails	Logical. If 'TRUE' (the default), the full orbit paths are drawn faintly behind the snapshot points. Set 'FALSE' for a pure snapshot showing only the body positions at the chosen time.
three_d	Logical. If 'TRUE', forces a 3D plot even for planar data.

Details

If any body has non-zero motion in the Z dimension (or 'three_d = TRUE'), [plot_system_3d()] is used; otherwise a 2D 'ggplot2' plot is returned.

Value

A 'ggplot' object (2D) or a 'plotly' HTML widget (3D).

Examples

```
sim <- create_system() |>
  add_sun() |>
  add_body("Earth", mass = mass_earth, x = distance_earth_sun, vy = speed_earth) |>
  simulate_system(time_step = seconds_per_day, duration = seconds_per_year)

# Final state with faint orbit trails
plot_system(sim)

# State at day 100, no trails
plot_system(sim, time = seconds_per_day * 100, trails = FALSE)
```

plot_system_3d	<i>Plot 3D Interactive System Snapshot at a Single Time</i>
----------------	---

Description

The 3D counterpart to [plot_system()]. Draws every body's position at a chosen time as a sphere in an interactive plotly scene, optionally with the full orbital trajectories shown faintly behind.

Usage

```
plot_system_3d(sim_data, time = NULL, trails = FALSE)
```

Arguments

sim_data	A tibble output from [simulate_system()].
time	Time (in simulation seconds) to snapshot. Defaults to the last time step. Snaps to the closest available time in the data.
trails	Logical. If 'TRUE' (the default), the full orbit paths are drawn faintly behind the snapshot points.

Value

A 'plotly' HTML widget.

Examples

```
create_system() |>
  add_body("Earth", mass = mass_earth) |>
  add_body("Moon", mass = mass_moon,
           x = distance_earth_moon, vy = speed_moon, vz = 100) |>
  simulate_system(time_step = seconds_per_hour, duration = seconds_per_day * 30) |>
  plot_system_3d()
```

print.orbit_system	<i>Print an orbit_system</i>
--------------------	------------------------------

Description

Displays a compact, human-readable summary of an 'orbit_system' showing the gravitational constant and a tibble of body states.

Usage

```
## S3 method for class 'orbit_system'
print(x, ...)
```

Arguments

x	An ‘orbit_system’ object.
...	Additional arguments (ignored).

Value

‘x’, invisibly.

Examples

```
create_system() |>
  add_sun() |>
  add_planet("Earth", parent = "Sun") |>
  add_planet("Mars", parent = "Sun")
```

remove_body

Remove one or more bodies from the system

Description

Drops bodies by name from an ‘orbit_system’. This is the counterpart to [add_body()] — use it to strip out bodies you no longer need before simulating, or to prune a system built with [load_solar_system()].

Usage

```
remove_body(system, id)
```

Arguments

system	An ‘orbit_system’ object.
id	A character vector of body names to remove. All names must exist in the system.

Details

““ load_solar_system() |> remove_body(c("Pluto", "Moon")) |> simulate_system(time_step = seconds_per_day, duration = seconds_per_year) ““

Value

The updated ‘orbit_system’ with the specified bodies removed.

Examples

```
# Remove a single body
create_system() |>
  add_sun() |>
  add_planet("Earth", parent = "Sun") |>
  add_planet("Mars", parent = "Sun") |>
  remove_body("Mars")

# Remove multiple bodies from the full solar system
load_solar_system() |>
  remove_body(c("Pluto", "Moon")) |>
  simulate_system(time_step = seconds_per_day, duration = seconds_per_year) |>
  plot_orbits()
```

save_system	<i>Save an orbit_system to disk</i>
-------------	-------------------------------------

Description

Saves the full ‘orbit_system’ object (bodies, forces, and time) to an ‘.rds’ file so it can be restored later with [load_system()]. This preserves everything — the gravitational constant, body states, and class — exactly as it was.

Usage

```
save_system(system, path)
```

Arguments

system	An ‘orbit_system’ object.
path	File path to save to. Should end in ‘.rds’.

Value

‘system’, invisibly.

Examples

```
sys <- create_system() |>
  add_sun() |>
  add_planet("Earth", parent = "Sun")

save_system(sys, file.path(tempdir(), "my_system.rds"))
```

shift_reference_frame *Shift the coordinate reference frame of the simulation*

Description

Recalculates the positions and velocities of all bodies relative to a specific target body, or to the system's center of mass. This effectively "anchors the camera" to the chosen point, placing it at the origin (0, 0, 0) for all time steps.

Usage

```
shift_reference_frame(sim_data, center_id, keep_center = TRUE)
```

Arguments

sim_data	A tidy 'tibble' containing the output from 'simulate_system()'.
center_id	The character string ID of the body to use as the new origin, or "'barycenter'" to use the system's center of mass (the mass-weighted mean position and velocity of all bodies at each time step).
keep_center	Logical. Should the central body remain in the dataset (it will have 0 for all coordinates) or be removed? Default is 'TRUE'. Ignored when 'center_id = "barycenter"'.

Details

The shift is a Galilean transformation: at every time step the chosen point's position and velocity are subtracted from every body. No physics changes; the same forces and accelerations produced the data, and you are only choosing where to stand when you look at it.

The barycentric frame is the natural one for binary stars and any other system where no single body dominates. In it the total momentum is zero and the center of mass sits at the origin for the whole run, which removes the slow drift you get when a system is built with one body at rest but nonzero total momentum (for example, a planet given an orbital velocity around a star that was not given the balancing recoil).

If a body in the system is itself named "'barycenter'", that body is used as the center rather than the center of mass.

Value

A tidy 'tibble' with updated 'x', 'y', 'z', 'vx', 'vy', and 'vz' columns.

Examples

```
# Simulate Sun-Earth-Moon
orbit_data <- create_system() |>
  add_sun() |>
  add_body("Earth", mass = mass_earth, x = distance_earth_sun, vy = speed_earth) |>
  add_body("Moon", mass = mass_moon, x = distance_earth_sun + distance_earth_moon,
```

```

        vy = speed_earth + speed_moon) |>
    simulate_system(time_step = seconds_per_hour, duration = seconds_per_year)

# Shift view to Earth and plot
orbit_data |>
  shift_reference_frame(center_id = "Earth") |>
  plot_orbits()

# The Sun started at rest with Jupiter in orbit: the pair's center of mass
# drifts, because the total momentum is not zero. The barycentric frame
# removes the drift and shows the Sun's own small orbit.
sun_jupiter <- create_system() |>
  add_sun() |>
  add_planet("Jupiter", parent = "Sun") |>
  simulate_system(time_step = seconds_per_day, duration = seconds_per_year * 12)

sun_jupiter |>
  shift_reference_frame("barycenter") |>
  plot_orbits(three_d = FALSE)

```

simulate_system

Simulate kinematics for an orbitr system

Description

Propagates the physical state of an ‘orbit_system’ through time using numerical integration. This engine supports multiple mathematical methods, defaulting to the energy-conserving Velocity Verlet algorithm to ensure highly stable orbital trajectories.

Usage

```

simulate_system(
  system,
  time_step = seconds_per_hour,
  duration = seconds_per_year,
  method = "verlet",
  softening = 0,
  use_cpp = TRUE
)

```

Arguments

system	An ‘orbit_system’ object created by ‘create_system()’.
time_step	The time increment per frame in seconds (default 3600s / 1 hour). For planetary orbits around a star, daily steps (‘86400’) are usually sufficient. For lunar-scale or tighter orbits, hourly steps (‘3600’) work well.
duration	Total simulation time in seconds (default 31557600s / 1 year).

method	The numerical integration method: "verlet" (default), "euler_cromer", or "euler".
softening	A small distance (in meters) added to prevent numerical singularities when bodies pass very close to each other. The gravitational distance is computed as $\sqrt{r^2 + \text{softening}^2}$ instead of r . Default is 0 (no softening). A value like $1e4$ (10 km) is reasonable for planetary simulations.
use_cpp	Logical. If 'TRUE' (default), uses the compiled C++ acceleration engine for better performance. Falls back to vectorized R if the C++ code is not available.

Value

A tidy 'tibble' containing the physical state (time, id, mass, x, y, z, vx, vy, vz) of every body at every time step. The run's settings are recorded as attributes ("G", "softening", "method", "time_step"), which [get_energy()], [conserved_quantities()], and [continue_simulation()] use as defaults.

Examples

```
my_universe <- create_system() |>
  add_body("Earth", mass = mass_earth) |>
  add_body("Moon", mass = mass_moon, x = distance_earth_moon, vy = speed_moon) |>
  simulate_system(time_step = seconds_per_hour, duration = seconds_per_day * 28)
```

system_from_simulation

Rebuild an orbit_system from a simulation snapshot

Description

Takes the state of every body at one time step of a simulation and returns a new 'orbit_system' with those positions and velocities as its initial conditions. This is the bridge from a finished run back to a system you can modify (add a body, remove one, change a velocity) and simulate again.

Usage

```
system_from_simulation(sim_data, time = NULL, G = NULL)
```

Arguments

sim_data	A tibble output from [simulate_system()].
time	The simulation time (in seconds) of the snapshot to use. Defaults to the last time step. Snaps to the closest available time.
G	The gravitational constant for the new system. Defaults to the value recorded by [simulate_system()] in the "G" attribute of 'sim_data', or to [gravitational_constant] if that is absent.

Value

An ‘orbit_system’ whose bodies have the snapshot’s positions and velocities.

Examples

```
sim <- create_system() |>
  add_sun() |>
  add_planet("Earth", parent = "Sun") |>
  simulate_system(time_step = seconds_per_day, duration = seconds_per_day * 100)

# Where everything was on day 100, as a system ready to simulate again
later <- system_from_simulation(sim)
later
```

Index

`add_body`, 2
`add_body_keplerian`, 3
`add_planet`, 5
`add_sun`, 7
`animate_system`, 8
`animate_system_3d`, 9

`conserved_quantities`, 10
`continue_simulation`, 11
`create_system`, 13

`distance_earth_moon`
 (`physical_constants`), 20
`distance_earth_sun`
 (`physical_constants`), 20
`distance_jupiter_sun`
 (`physical_constants`), 20
`distance_mars_sun` (`physical_constants`),
 20
`distance_mercury_sun`
 (`physical_constants`), 20
`distance_neptune_sun`
 (`physical_constants`), 20
`distance_pluto_sun`
 (`physical_constants`), 20
`distance_saturn_sun`
 (`physical_constants`), 20
`distance_uranus_sun`
 (`physical_constants`), 20
`distance_venus_sun`
 (`physical_constants`), 20

`export_bodies`, 13

`get_angular_momentum` (`get_momentum`), 16
`get_bodies`, 14
`get_energy`, 15
`get_momentum`, 16
`get_orbital_elements`, 17
`gravitational_constant`
 (`physical_constants`), 20

`load_solar_system`, 18
`load_system`, 19

`mass_earth` (`physical_constants`), 20
`mass_jupiter` (`physical_constants`), 20
`mass_mars` (`physical_constants`), 20
`mass_mercury` (`physical_constants`), 20
`mass_moon` (`physical_constants`), 20
`mass_neptune` (`physical_constants`), 20
`mass_pluto` (`physical_constants`), 20
`mass_saturn` (`physical_constants`), 20
`mass_sun` (`physical_constants`), 20
`mass_uranus` (`physical_constants`), 20
`mass_venus` (`physical_constants`), 20

`physical_constants`, 20
`plot_orbits`, 23
`plot_orbits_3d`, 24
`plot_system`, 25
`plot_system_3d`, 26
`print.orbit_system`, 26

`remove_body`, 27

`save_system`, 28
`seconds_per_day` (`physical_constants`), 20
`seconds_per_hour` (`physical_constants`),
 20
`seconds_per_year` (`physical_constants`),
 20
`shift_reference_frame`, 29
`simulate_system`, 30
`speed_earth` (`physical_constants`), 20
`speed_jupiter` (`physical_constants`), 20
`speed_mars` (`physical_constants`), 20
`speed_mercury` (`physical_constants`), 20
`speed_moon` (`physical_constants`), 20
`speed_neptune` (`physical_constants`), 20
`speed_pluto` (`physical_constants`), 20
`speed_saturn` (`physical_constants`), 20

`speed_uranus(physical_constants)`, [20](#)
`speed_venus(physical_constants)`, [20](#)
`system_from_simulation`, [31](#)