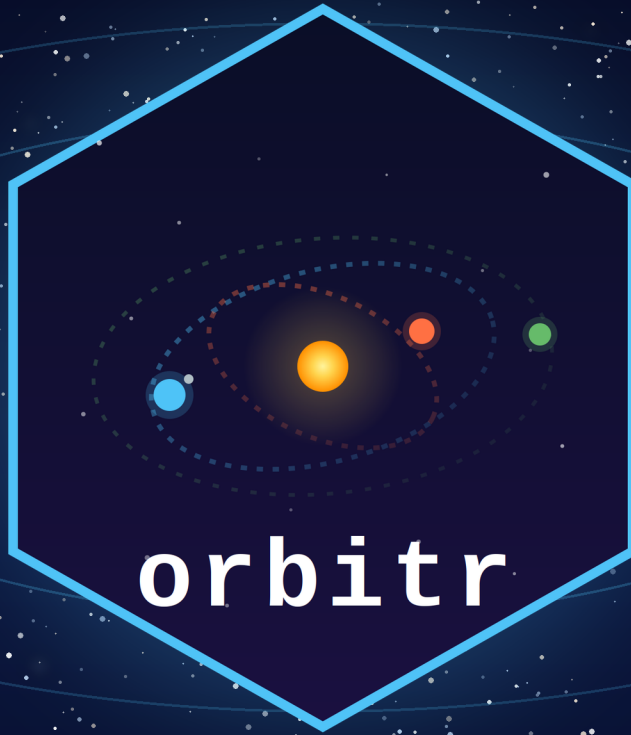




Orbital Mechanics with R

Simulating Planets, Binary Stars, Chaotic Systems,
and the N-Body Problem with orbitr

Dave Rosenman

Copyright © 2026 David P. Rosenman

All rights reserved. No part of this book may be reproduced in any form without written permission from the author, except for brief quotations in reviews.

The orbitr package is free software released under the MIT License; see <https://orbit-r.com>. All code listings in this book may be used freely.

Written for orbitr version 1.0.0.

First edition.

Table of contents

Preface	1
Who this book is for	1
What is in this book	2
What you need	2
A note on versions	3
How the code works	3
Conventions	4
Try it first: the orbitr app	4
Acknowledgments	5
Dedication	5
I. First Orbits	7
1. Why Simulate Gravity?	9
1.1. Three reasons	9
1.2. What a simulation actually is	9
1.3. What orbitr is	10
1.4. What orbitr is not	11
1.5. How to read this book	11
1.6. Setting up	11
2. Four Lines to an Orbit	13
2.1. The Moon in four lines	13
2.2. What each line does	13
2.3. What came back	15
2.4. Where did Earth go?	17
2.5. Real bodies without looking anything up	19
2.6. Looking before you leap	21
2.7. Snapshots and animation	22
2.8. Saving and sharing	23
II. Where the Physics Comes From	25
3. Newton's Law and the N-Body Problem	27
3.1. From the apple to the Moon	27
3.2. The law in vector form	28
3.3. Superposition: the N-body acceleration	28
3.4. Why point masses are enough	29
3.5. Units, G , and how well we know it	30
3.6. The equations of motion as a system	30

Table of contents

3.7. Writing the acceleration yourself	31
3.8. A three-body example	32
3.9. What the engine does differently	33
4. The Two-Body Problem, Solved by Hand	35
4.1. Two bodies become one	35
4.2. Angular momentum is conserved, and the orbit is flat	36
4.3. Energy is conserved	37
4.4. The orbit equation	37
4.5. The geometry of an ellipse	39
4.6. Vis-viva, circular speed, and escape speed	39
4.7. Kepler's third law	40
4.8. Checking everything against the simulation	40
5. Integrators: Euler, Euler-Cromer, and Velocity Verlet	47
5.1. The problem of taking a step	47
5.2. Euler's method	47
5.3. Euler-Cromer	48
5.4. Velocity Verlet	49
5.5. Seeing it in orbitr	52
5.6. Choosing a time step	56
6. Softening: What Happens When Bodies Get Close	59
6.1. The singularity, numerically	59
6.2. Plummer softening	60
6.3. What softening costs	61
6.4. The encounter, softened	62
6.5. When to use it	64
7. From Orbital Elements to State Vectors	65
7.1. Six numbers	65
7.2. Position and velocity in the orbital plane	66
7.3. Three rotations	67
7.4. Implementing it	67
7.5. The inverse problem	69
7.6. The Moon on its real orbit	71
7.7. Where is Mars today? Kepler's equation	72
7.8. Osculating elements	74
III. Doing Astrophysics with orbitr	77
8. The Solar System from JPL Elements	79
8.1. One call	79
8.2. Where the numbers come from	80
8.3. Kepler's third law, verified	80
8.4. How often do Earth and Mars line up?	82
8.5. The Sun wobbles	83
8.6. A long run checked	85

9. Binary Stars and Circumbinary Planets	87
9.1. Two stars	87
9.2. An eccentric binary	88
9.3. Kepler-16	89
9.4. How close is too close?	91
10.Reference Frames	95
10.1. Moving the camera	95
10.2. The geocentric Moon	95
10.3. Retrograde Mars	97
10.4. The Moon's path around the Sun	99
10.5. Two sunsets	99
10.6. A rotating frame, by hand	102
11.Chaos and the Three-Body Problem	105
11.1. What Poincaré proved	105
11.2. Sensitive dependence, measured	105
11.3. Slingshot	107
11.4. Why the Sun, Earth, and Moon are not chaotic	109
11.5. The Lagrange points, and which ones are traps	111
12.Conservation Laws as Diagnostics	115
12.1. Three conserved quantities, for N bodies	115
12.2. Which laws an integrator keeps exactly	116
12.3. Computing all three	117
12.4. Reading the plots	119
12.5. A worked diagnosis: the orbit that looked fine	119
13.Comets and Highly Eccentric Orbits	123
13.1. Halley	123
13.2. Where a fixed step fails	124
13.3. Segmented runs	125
13.4. Hyperbolic visitors	127
13.5. Sungrazers	129
14.Figures for Papers and Talks	131
14.1. <code>plot_orbits()</code> is a ggplot	131
14.2. Your own plots from the tibble	132
14.3. A figure for a paper	134
14.4. Interactive 3D for the screen	137
14.5. Animation	137
14.6. Export and reproducibility	138
IV.Under the Hood	141
15.The C++ Engine	143
15.1. Why compiled code	143
15.2. The kernel, line by line	143
15.3. The R fallback	145
15.4. Where the time goes	146

Table of contents

15.5. The n^2 wall	148
15.6. Changing the physics	149
16.Limits and Extensions	151
16.1. Collisions	151
16.2. Forces other than gravity	152
16.3. Oblateness	156
16.4. General relativity	156
16.5. Better integrators	157
16.6. The roadmap, and how to help	158
16.7. Where to go next	158
References	161
Appendices	165
A. R Basics	165
A.1. Installing R and RStudio	165
A.2. Running code	165
A.3. Packages	166
A.4. Getting help	166
A.5. Functions and arguments	167
A.6. The pipe	167
A.7. Vectors, data frames, and tibbles	168
A.8. The six dplyr verbs the book uses	169
A.9. The ggplot2 the book uses	170
A.10.Where to learn more	171
B. Mathematical Background	173
B.1. Vectors	173
B.2. Derivatives of vectors	175
B.3. Polar coordinates in the plane	175
B.4. Conic sections	176
B.5. Taylor series	177
B.6. Differential equations and the idea of state	177
B.7. Matrices, rotations, and determinants	178
B.8. Logarithms, exponentials, and power laws	179
B.9. Angles	179
B.10.Units and dimensions	179
B.11.Where each tool is used	180
C. Function Quick Reference	181
C.1. Building a system	181
C.2. Running	181
C.3. Analysis	182
C.4. Plotting	182
C.5. Saving	183
C.6. Helpers defined in this book	183
D. Physical Constants	185

About the Author	187
Where to find things	187

Preface

The documentation for `orbitr` lives at <https://orbit-r.com>, and it is free. It will show you how to put a planet around a star in four lines of R, how to load the whole solar system from real JPL data, and how to animate a planet orbiting two suns. If all you want is to drive the package, you may not need this book.

This book is about everything the documentation leaves out. The documentation says that `simulate_system()` uses the Velocity Verlet method by default because it “conserves energy over long timescales.” It does not say *why* Verlet conserves energy when the textbook Euler method does not, or where the method came from (a 1907 paper about the aurora, as it turns out), or how you would check for yourself that it is doing its job. The documentation gives you the circular-orbit speed $v_{\text{circ}} = \sqrt{GM/r}$ as a fact. It does not derive it, and it does not tell you how that one formula, pushed a little further, gives you Kepler’s three laws, the shape of every orbit in the solar system, and the velocity formulas that make the Kepler-16 example work.

So this is two books in one. The first is a physics book that uses `orbitr` the way a lab course uses a bench: every result gets derived on paper and then checked against a simulation you can run yourself. The second is an applications book that uses the physics to do real astrophysics: verifying Kepler’s third law across the solar system, building a binary star from conservation of momentum, watching Mars go retrograde from a geocentric frame, measuring how fast a chaotic triple system forgets its initial conditions.

Who this book is for

Two kinds of readers.

If you know R and are curious about how orbits work, this book will teach you the physics using tools you already have. You will need calculus and vectors, and it helps to have met Newton’s laws once, but you do not need to have taken a mechanics course. Where a derivation needs a fact from mechanics, the fact is stated and explained.

If you are a physics or astronomy student and want to *compute*, this book will show you how to turn the equations in your mechanics course into working simulations, and how to use tidy data tools to analyze what comes out. You do not need to know R at all. Appendix A gets you from nothing to running the book’s code (installing R, packages, the pipe, the handful of `dplyr` and `ggplot2` functions the book uses) in an hour, and points to the free books to read when you want more. Every piece of code in the chapters is explained, and the whole book uses a small, consistent vocabulary.

If the mathematics is the uncertain part, Appendix B collects everything the derivations assume (vectors and their products, derivatives of vectors, polar coordinates, Taylor series, the geometry of an ellipse) in one place, with the specific identities the book uses.

Preface

If you are both, you will find the two halves of the book talk to each other more than you might expect. The best way to understand a symplectic integrator is to watch a non-symplectic one fail; the best way to understand an orbital element is to vary it and look.

What is in this book

Part I, First Orbits, gets you running. Chapter 1 explains what an N-body simulation is and why anyone bothers when Kepler already solved the two-body problem. Chapter 2 walks through the four-line pipeline and the tidy table it produces.

Part II, Where the Physics Comes From, is the heart of the book. Chapter 3 starts from Newton’s law and builds the N-body equations of motion that the package integrates. Chapter 4 solves the two-body problem by hand and derives everything the documentation states as fact: the orbit equation, Kepler’s laws, the vis-viva equation, circular and escape velocity. Chapter 5 derives the three integrators (Euler, Euler-Cromer, Velocity Verlet), shows why only two of them are fit for orbits, and traces the Verlet method back through molecular dynamics, the aurora, and the *Principia*. Chapter 6 explains gravitational softening. Chapter 7 derives the conversion from Keplerian orbital elements to the Cartesian state vectors the engine actually uses.

Part III, Doing Astrophysics with orbitr, is the applications half. Each chapter takes a problem from astronomy, sets it up from the physics in Part II, runs it, and analyzes the result with dplyr and ggplot2.

Part IV, Under the Hood, looks at the C++ engine and at the places where a point-mass Newtonian model stops being enough.

The appendices collect the mathematical background the derivations assume, the R needed to run the book, a quick function reference, and the built-in physical constants with their sources.

What you need

- R, version 4.1 or later, from <https://cran.r-project.org>. The book uses R’s native pipe `|>` everywhere.
- The orbitr package, version 1.0.0 or later, from CRAN (see the box below).
- The dplyr and ggplot2 packages. They are installed automatically with orbitr. A few chapters also use tidyr.
- For the interactive 3D plots and the animations, the plotly, gganimate, and gifski packages. These are optional; the book tells you when it uses them.

An editor that runs R code chunks (RStudio or Positron, from <https://posit.co>) makes it easy to run the book’s examples as you read.

Installing orbitr

This book needs orbitr 1.0.0 or later, which is on CRAN:

```
install.packages("orbitr")
```

`packageVersion("orbitr")` should then report 1.0.0 or higher. The development version, with whatever has been added since, is at <https://github.com/DRosenman/orbitr>; it compiles the package's C++ engine, so on Windows that needs Rtools and on a Mac the Xcode command-line tools. For this book, the CRAN version is the one to use.

The optional packages, in one line:

```
install.packages(c("tidyr", "plotly", "gganimate", "gifski"))
```

A note on versions

This book is written for orbitr 1.0.0, and every figure in it was produced by that version. Version 1.0 is the point at which the package's function names, arguments, and defaults became stable: anything that changes afterwards will be deprecated first, so code in this book should keep running on later versions, possibly with a warning pointing at the replacement. Several of the analysis functions the book uses (`get_orbital_elements()`, `get_energy()`, `conserved_quantities()`, `continue_simulation()`, the barycentric reference frame) were added in 1.0.0 because this book needed them, so it will not run on 0.3.0. The physics in Part II does not depend on the version at all.

How the code works

Every figure and every number in this book was produced by the code shown next to it, when the book was built. Nothing was pasted in from elsewhere. That means two things for you as a reader. First, if you run the code, you should get the same picture. Second, when the text says a simulated period is “about 27 days,” that number came out of the simulation, not out of my head.

Code appears in blocks like this, with output marked by `#>`:

```
sqrt(gravitational_constant * mass_earth / distance_earth_moon)
#> [1] 1018.289
```

Two kinds of output from orbitr do not survive the trip to paper. Interactive 3D plots (`plot_orbits_3d()`, and `plot_orbits()` whenever a body moves in z) are plotly widgets, and animations (`animate_system()`) are GIFs. Throughout the book I use static ggplot2 figures, which are the same in every edition, and pass `three_d = FALSE` wherever a system has three-dimensional motion so that the figure stays flat.

Preface

Where the text discusses an animation or an interactive plot, the printed book shows the call you would make and a static snapshot from `plot_system()` in place of the result. The online edition at <https://book.orbit-r.com> shows them live.

Conventions

All quantities are in SI units: meters, kilograms, seconds. The built-in constants (`mass_earth`, `distance_earth_sun`, `speed_moon`, ...) are all SI, and all of the package's arguments expect SI. Vectors are set in bold, $\mathbf{r}$, and their magnitudes in italic, $r = |\mathbf{r}|$. Unit vectors wear a hat, $\hat{\mathbf{r}} = \mathbf{r}/r$. Dots denote time derivatives: $\dot{\mathbf{r}}$ is velocity and $\ddot{\mathbf{r}}$ is acceleration.

The symbol μ always means the gravitational parameter $\mu = G(M + m)$ of a pair of bodies (or GM when one is much heavier). Some mechanics texts use μ for the reduced mass; this book writes the reduced mass as m_r to avoid the clash.

A few helper functions that the text defines for the derivations (measuring a period from a time series, solving Kepler's equation, converting elements to a state vector by hand) are collected in the file `R/helpers.R` in the book's source, which every chapter loads, so later chapters use them without repeating the definitions. Everything else is a package function.

Boxes marked **Where it comes from** give the history of a method or a formula: who found it, what problem they were working on, and how it ended up in the package. Boxes marked **Key result** collect the formulas you will reuse.

Try it first: the orbitr app

Before any code, it helps to have *seen* an orbit respond to a change: a planet given a little more speed and swinging out into an ellipse, a time step pushed too far and the orbit falling apart. The orbitr app is the package with a control panel on it. It runs in a browser, needs no installation, and does nothing the code in this book cannot do, which is the point: every control in it corresponds to an argument of a function you will meet in Chapter 2, and when you later type `add_body("Moon", mass = mass_moon, x = distance_earth_moon, vy = speed_moon)` you will already know what happens when that `vy` goes up. The app's **Code** tab makes that correspondence explicit: whatever system you have set up, it shows the R code that produces exactly what is on the screen. Set up a system by clicking, read the code, paste it into R, and you have run your first simulation.

The app is at <https://daverosenman.shinyapps.io/orbitr/>, or scan the code below. The online edition of this book at <https://book.orbit-r.com> embeds it in this page, with a guided walkthrough of what to try.



Ten minutes with it is a good investment before Chapter 2. Suggested experiments, in order: pick a preset system and run it as is; change one body's starting speed by ten percent in each direction and run again; shorten and lengthen the time step until something looks wrong; switch the integrator from Verlet to Euler and run the same system. After each one, open the Code tab and see which argument changed. Chapters 4 and 5 explain what you just saw.

Acknowledgments

The physics in this book was shaped by Thornton and Marion's *Classical Dynamics* (Thornton and Marion 2004) and Newman's *Computational Physics* (Newman 2012), the two books I kept open while writing the engine. The N-body methods follow Aarseth (Aarseth 2003).

Dedication

For Kerry, a true academic, with a mind so beautiful it is hard to believe it runs on the laws of physics.

For my parents, who supported me through not one undergraduate degree but two: political science at Temple, and then, when I announced the crazy idea of studying theoretical physics, that one as well.

And for my grandfather, a math teacher, for a million things, among them the algebra and trigonometry he taught me all over again before I could start on physics; and for my grandmother, who powered every one of those sessions with the best Italian food on the planet.

Part I.

First Orbits

1. Why Simulate Gravity?

In 1609 Johannes Kepler published the first two of his three laws of planetary motion, and in 1687 Isaac Newton showed that all three follow from a single inverse-square force law (Kepler 1609; Newton 1687). For one planet going around one star, the problem is *solved*: write down the initial position and velocity, and the orbit is an ellipse whose size, shape, and orientation you can compute by hand, exactly, for all time. Chapter 4 does exactly that.

So why would anyone simulate it?

1.1. Three reasons

Because two is a special number. Add a third body and the exact solution disappears. Newton knew it; he spent years on the Moon’s motion under the combined pull of the Earth and the Sun and reportedly said it was the only problem that ever made his head ache. In 1890 Henri Poincaré proved that the three-body problem has no general solution of the kind Kepler’s ellipses provide, and in doing so discovered what we now call chaos (Poincaré 1890). Our own solar system has one star, eight planets, hundreds of moons, and a great many smaller things, every one of which pulls on every other. The planets follow Kepler’s ellipses *almost*, and the “almost” contains all of the interesting physics: Jupiter tugging Earth’s orbit, resonances in the asteroid belt, the slow precession of Mercury’s perihelion. None of it can be written down in closed form. All of it can be simulated.

Because the solution and the understanding are different things. Even for the two-body problem, knowing that the answer is an ellipse is not the same as knowing what happens when you give a planet 20% too much speed, or launch it at the wrong angle, or put it around a star of twice the mass. A simulation lets you ask those questions in seconds and *see* the answer. Most of the intuition that working astronomers have about orbits was built this way, by changing something and looking.

Because the simulation is the check on the theory. When you derive, on paper, that a body launched at $\sqrt{2}$ times the circular speed should just barely escape, you can set up that exact situation and watch. If the simulation agrees, you understand both the physics and the code a little better. If it disagrees, one of them is wrong, and finding out which is where the learning is. Part II of this book is built around that loop: derive, simulate, compare.

1.2. What a simulation actually is

Here is the whole idea in four sentences. At any instant, every body has a position and a velocity. Newton’s law tells you the acceleration of every body from the positions of all

1. Why Simulate Gravity?

the others. If you know where things are and how fast they are moving right now, you can estimate where they will be a short time Δt later. Do that, then do it again from the new state, and again, and you have an orbit.

The “estimate” is the whole subject of numerical integration, and it is where the craft is. The computer never solves the equations of motion. It takes small steps and hopes the error in each step is small enough, and cancels well enough, that the result after a million steps still means something. Whether that hope is justified depends entirely on *how* you take the step. Chapter 5 derives three ways of doing it, and shows that the most obvious one, the one every introductory text starts with, produces orbits that spiral outward and planets that quietly gain energy from nowhere. The one the package uses by default was invented for a different problem entirely and has a surprisingly deep reason for working.

This discreteness is the one thing to keep in mind every time you look at a simulation result. A simulated orbit is not the true orbit; it is the true orbit plus an error that depends on the step size and the method. Most of the time the error is invisible. Part of learning to simulate is learning when it is not.

1.3. What orbitr is

orbitr is an N-body gravitational simulator written for R (Rosenman 2026). You build a system by adding bodies to it with masses, positions, and velocities, run it forward in time, and get back a table. A few things make it different from the N-body codes that research astronomers use, and they are the reasons this book is possible:

- **The output is a tibble.** One row per body per time step, with columns for time, position, and velocity. There is no special result object to learn. Every tool in the R data-analysis ecosystem (dplyr, ggplot2, plotly, anything that takes a data frame) works on it directly. Most of the analysis in this book is a `filter()`, a `mutate()`, and a `ggplot()`.
- **It is built for the pipe.** A simulation reads top to bottom: create a system, add bodies, simulate, plot.
- **Analysis is built in.** The output can be re-centered on any body or on the system’s center of mass, reduced to orbital elements, checked against the conservation laws, and continued from where it stopped, each with one function.
- **The physics is visible.** Three integration methods are available, so you can compare them. Softening is optional and off by default, so you get exact Newtonian gravity unless you ask otherwise. The gravitational constant is an argument, so you can turn gravity up, down, or off.
- **Real data is built in.** Masses, orbital distances, and speeds for the Sun, the planets, and the Moon are included as constants, and the planets’ orbital elements come from the JPL DE440 ephemeris (Park et al. 2021), so `add_planet("Mars", parent = "Sun")` puts Mars where Mars actually is.
- **The inner loop is compiled.** The pairwise force calculation runs in C++ through Rcpp (Eddelbuettel and François 2011), with a pure-R fallback, so systems of a few dozen bodies simulate in seconds.

1.4. What orbitr is not

It is not an ephemeris. If you need to know where Mars will be on a given date to arc-second precision, use JPL Horizons, which fits decades of radar and spacecraft tracking data and includes effects this package ignores. `orbitr` starts from JPL's orbital elements at one epoch and integrates pure point-mass Newtonian gravity forward. Over a few years the planets land close to where they should; over centuries, small omitted effects accumulate.

The omissions are deliberate, and knowing them tells you what the package can and cannot answer:

- Bodies are point masses. Gravity outside a spherically symmetric body is exactly that of a point mass at its center (Chapter 3 proves this), so for orbital distances this costs nothing. But there is no notion of two bodies touching, no oblateness, no tides.
- Gravity is Newtonian. General relativity's correction to Mercury's perihelion precession, 43 arcseconds per century, is not in the model.
- There are no non-gravitational forces: no atmospheric drag, no radiation pressure, no thrust.
- The time step is fixed. Orbits that pass very close to a massive body need small steps near periapsis and would happily take large ones elsewhere; `orbitr` takes the same step throughout. Chapter 13 shows where this bites and how to handle it.

Chapter 16 returns to each of these, and to the package roadmap.

1.5. How to read this book

If you want to get something running first, go to Chapter 2 now, then come back. Part II can be read straight through as a physics text; the code is there to check the physics, not the other way round. Part III assumes Part II but each chapter stands alone, so you can go straight to binary stars or chaos if that is what you came for.

The most useful habit while reading is to predict before you run. Each chapter's code sets up something whose outcome the physics tells you in advance; say what you expect, then look. When the two disagree, one of them is wrong, and finding out which is where the learning is.

1.6. Setting up

The preface lists what to install and how; the short version is R, `orbitr` 1.0.0 or later, and an editor that runs code chunks. Once that is done, compare your versions with the ones this book was built with:

```
packageVersion("orbitr")
#> [1] '1.0.0'
R.version.string
#> [1] "R version 4.6.1 (2026-06-24 ucrt)"
```

1. *Why Simulate Gravity?*

If `packageVersion("orbitr")` reports something older than 1.0.0, update: the analysis functions this book relies on were added in 1.0.0. Anything newer should run the book's code unchanged; the changelog (`NEWS.md` in the GitHub repository, <https://github.com/D-Rosenman/orbitr>) lists any deprecations.

2. Four Lines to an Orbit

This chapter gets you from nothing to a working simulation, then slows down and looks carefully at each piece. By the end you will know what every argument in the four-line pipeline does, what the table it produces looks like, and one subtle thing the four-line version gets slightly wrong, which turns out to be a preview of a real conservation law.

2.1. The Moon in four lines

```
library(orbitr)

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) |>
  plot_orbits()
```

Earth sits at the origin. The Moon starts 384,400 km away along the x axis, moving at 1,022 m/s in the $+y$ direction, and 28 days later it has gone around once. (The Earth's own path is in there too; it is too small to see at this scale. We will come back to it.)

2.2. What each line does

create_system() makes an empty system. Its one argument is the gravitational constant, G , which defaults to the real value, $6.6743 \times 10^{-11} \text{ m}^3 \text{ kg}^{-1} \text{ s}^{-2}$. You will almost never change it, but it is there, and setting it to zero is a quick way to confirm that a body with no forces on it moves in a straight line.

add_body() adds one body. It needs a name (*id*) and a *mass*; everything else defaults to zero. The position arguments x , y , z are in meters and the velocity arguments v_x , v_y , v_z in meters per second. Earth gets only a mass, so it starts at the origin at rest. The Moon gets a mass, an x position, and a y velocity. The constants `mass_earth`, `mass_moon`, `distance_earth_moon`, and `speed_moon` are exported by the package; Appendix D lists all of them with their sources.

simulate_system() runs the integration. `time_step` is the size of each step in seconds and `duration` is how long to run, also in seconds. Here that is one hour per step for 28 days, or $28 \times 24 = 672$ steps. The defaults are one hour and one year. Two more arguments matter and have sensible defaults: `method`, which picks the integrator ("`verlet`" by default;

2. Four Lines to an Orbit

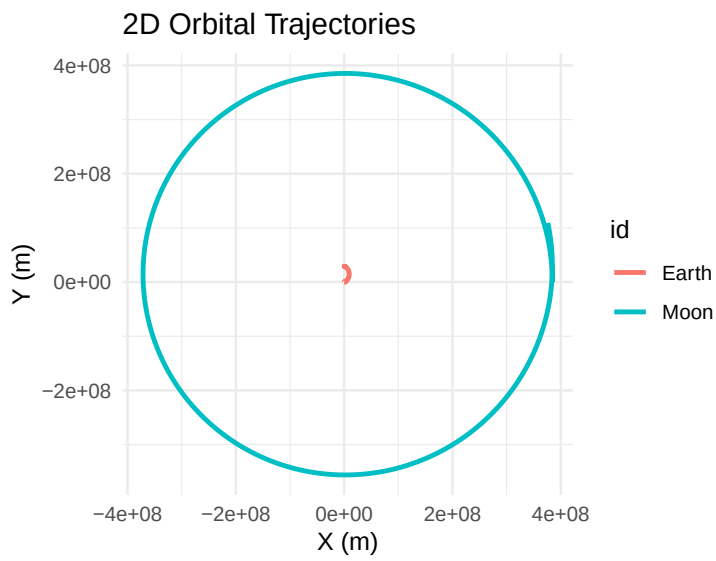


Figure 2.1.: The Moon's orbit around Earth over 28 days, simulated one hour at a time.

Chapter 5), and `softening`, which is zero by default, meaning exact Newtonian gravity (Chapter 6).

`plot_orbits()` draws the path of every body with `ggplot2` and returns the plot. Because it is an ordinary `ggplot` object you can add layers to it with `+`.

The constants `seconds_per_hour`, `seconds_per_day`, and `seconds_per_year` exist because every time argument in the package is in seconds, and nobody wants to type 2419200.

2.3. What came back

Drop the plotting step and keep the result:

```
moon <- 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)
```

```
moon
#> # A tibble: 1,346 x 9
#>   id      mass      x      y      z      vx      vy      vz  time
#>   <chr>   <dbl>   <dbl>   <dbl> <dbl> <dbl>   <dbl> <dbl> <dbl>
#> 1 Earth 5.97e24      0      0      0      0      0      0      0
#> 2 Moon 7.34e22 384400000      0      0      0 1022      0      0
#> 3 Earth 5.97e24    215.      0      0 0.119 0.000571      0 3600
#> 4 Moon 7.34e22 384382520. 3679200      0 -9.71 1022.      0 3600
#> 5 Earth 5.97e24    860.    4.11      0 0.239 0.00229      0 7200
#> 6 Moon 7.34e22 384330083. 7358065.      0 -19.4 1022.      0 7200
#> 7 Earth 5.97e24   1934.   16.5      0 0.358 0.00514      0 10800
#> 8 Moon 7.34e22 384242692. 11036262.      0 -29.1 1022.      0 10800
#> 9 Earth 5.97e24   3438.   41.1      0 0.477 0.00914      0 14400
#> 10 Moon 7.34e22 384120357. 14713454.      0 -38.8 1021.      0 14400
#> # i 1,336 more rows
```

It is a tibble. Each row is one body at one instant. There are 1346 rows: two bodies, times 673 time steps, which is the 672 steps plus the initial state. The columns are the body's `id`, its `mass`, its position `x`, `y`, `z`, its velocity `vx`, `vy`, `vz`, and `time` in seconds from the start.

That is the entire output format. There is nothing else to learn, which is the point: everything you already know how to do with a data frame works. How far is the Moon from the origin at the end of the run?

```
moon |>
  filter(time == max(time)) |>
  mutate(r_km = sqrt(x^2 + y^2 + z^2) / 1e3) |>
  select(id, time, r_km)
#> # A tibble: 2 x 3
```

2. Four Lines to an Orbit

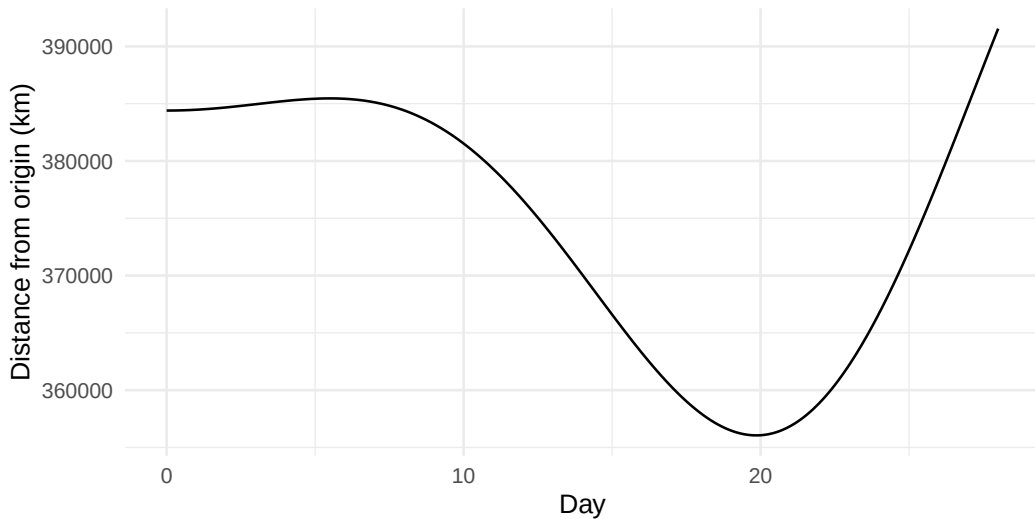


Figure 2.2.: The Moon's distance from the origin over the 28-day run. It is not quite constant: the orbit is slightly elliptical.

```
#>   id      time    r_km
#>   <chr>   <dbl>   <dbl>
#> 1 Earth 2419200 29053.
#> 2 Moon  2419200 391550.
```

How does the Moon's distance change over the month?

```
library(dplyr)
library(ggplot2)

moon |>
  filter(id == "Moon") |>
  mutate(r_km = sqrt(x^2 + y^2 + z^2) / 1e3) |>
  ggplot(aes(time / seconds_per_day, r_km)) +
  geom_line() +
  labs(x = "Day", y = "Distance from origin (km)")
```

The distance dips and recovers: the orbit is an ellipse, not a circle, because 1,022 m/s is not exactly the speed that produces a circle at this distance. The speed that does is $\sqrt{G(M+m)/r}$, which evaluates to

```
sqrt(gravitational_constant * (mass_earth + mass_moon) / distance_earth_moon)
#> [1] 1024.529
```

m/s. Chapter 4 derives that formula and shows how the small gap between it and 1,022 m/s determines the eccentricity of the orbit in Figure 2.2. For now, just notice that the simulation knows about it without being told.

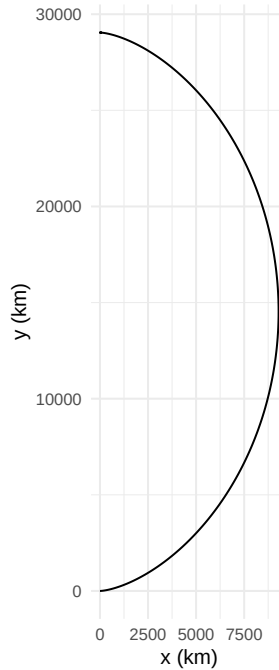


Figure 2.3.: Earth's own motion during the 28-day run. It loops once around the pair's center of mass while the whole system drifts in the $+y$ direction.

2.4. Where did Earth go?

I said Earth's path was too small to see. Let's look at it on its own:

```
moon |>
  filter(id == "Earth") |>
  ggplot(aes(x / 1e3, y / 1e3)) +
  geom_path() +
  coord_equal() +
  labs(x = "x (km)", y = "y (km)")
```

Earth does not stay at the origin. It makes a small loop, as it should, because the Moon pulls on Earth just as Earth pulls on the Moon, and the two orbit their common center of mass. But the loop also *drifts*: after 28 days Earth has moved about

```
drift_km <- moon |>
  filter(id == "Earth", time == max(time)) |>
  mutate(r_km = sqrt(x^2 + y^2) / 1e3) |>
  pull(r_km)
round(drift_km)
#> [1] 29053
```

2. Four Lines to an Orbit

kilometers from where it started. Nothing is wrong with the simulation. The drift is the simulation being *right* about something we got wrong in the setup.

We gave the Moon momentum, $m_{\text{Moon}} \times 1022 \text{ m/s}$ in the $+y$ direction, and gave Earth none. The total momentum of the system is therefore not zero, and since gravity is an internal force, total momentum is conserved: the system's center of mass has to keep moving at a constant velocity forever. That velocity is the Moon's momentum divided by the total mass,

```
v_com <- mass_moon * speed_moon / (mass_earth + mass_moon)
v_com
#> [1] 12.41192
```

about 12 m/s, which over 28 days is $3.0027 \times 10^4 \text{ km}$. That is the drift in Figure 2.3, exactly.

The fix is to give Earth the equal and opposite momentum so the total is zero. Earth's velocity must be $-v_{\text{Moon}} m_{\text{Moon}} / m_{\text{Earth}}$:

```
v_earth <- -speed_moon * mass_moon / mass_earth

moon_fixed <- create_system() |>
  add_body("Earth", mass = mass_earth, vy = v_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)

moon_fixed |>
  filter(id == "Earth") |>
  ggplot(aes(x / 1e3, y / 1e3)) +
  geom_path() +
  coord_equal() +
  labs(x = "x (km)", y = "y (km)")
```

Now Earth's path closes. For a Sun-planet system the same drift exists and is a thousand times smaller, which is why the package's `add_sun()` puts the Sun at rest at the origin and nobody notices. For a binary star, where the two masses are comparable, getting the momentum right is the whole problem, and Chapter 9 builds the Kepler-16 system from exactly this principle.

If you just want to *look* at the Moon going around Earth without thinking about the center of mass, there is a shortcut: re-center the data on Earth after the fact.

```
moon |>
  shift_reference_frame("Earth") |>
  plot_orbits()
```

`shift_reference_frame()` subtracts Earth's position and velocity from every body at every time step, which puts Earth at the origin for the whole run. And if what you want is the drift gone without re-running anything, `shift_reference_frame(moon, "barycenter")` re-centers every step on the pair's center of mass, which is where the system was sitting all along. Chapter 10 is about what you can learn by doing this.

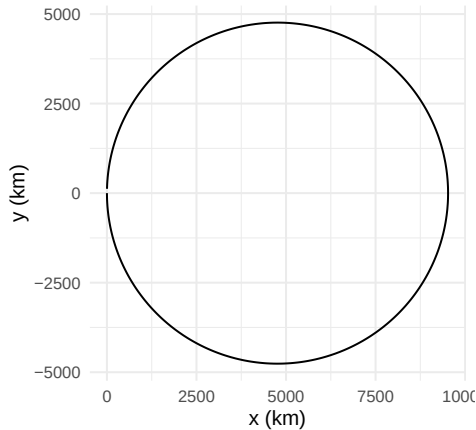


Figure 2.4.: With zero total momentum, Earth’s path closes: a small circle around the center of mass, about 4,700 km from the origin.

2.5. Real bodies without looking anything up

Typing masses, distances, and speeds works for any system you can imagine. For the real solar system there is an easier way. `add_sun()` places the Sun at the origin at rest, and `add_planet()` places a planet using its real orbital elements from the JPL DE440 ephemeris:

```
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(three_d = FALSE)
```

`parent = "Sun"` tells `add_planet()` which body the orbit is around; the parent must already be in the system. Mercury completes about four orbits in the time Earth completes one, and you can see that its orbit is noticeably off-center: Mercury’s eccentricity is 0.21, the largest of any planet.

Why `three_d = FALSE`?

The real planets’ orbits are slightly tilted relative to one another, so once you use `add_planet()` the bodies have z motion. `plot_orbits()` notices and, by default, returns an interactive 3D plotly widget instead of a ggplot. That is the right thing on screen and the wrong thing on paper, so throughout this book I pass `three_d = FALSE` for any system with real orbital elements. On your own screen, leave it off and rotate the view.

2. Four Lines to an Orbit

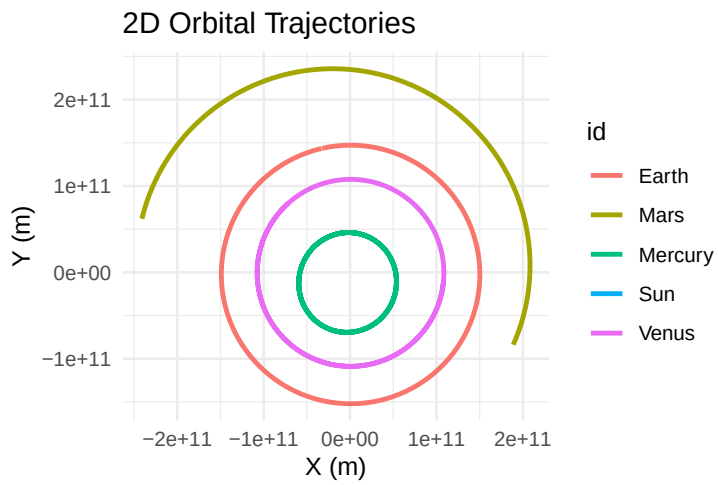


Figure 2.5.: Mercury, Venus, Earth, and Mars for one Earth year, placed from JPL orbital elements.

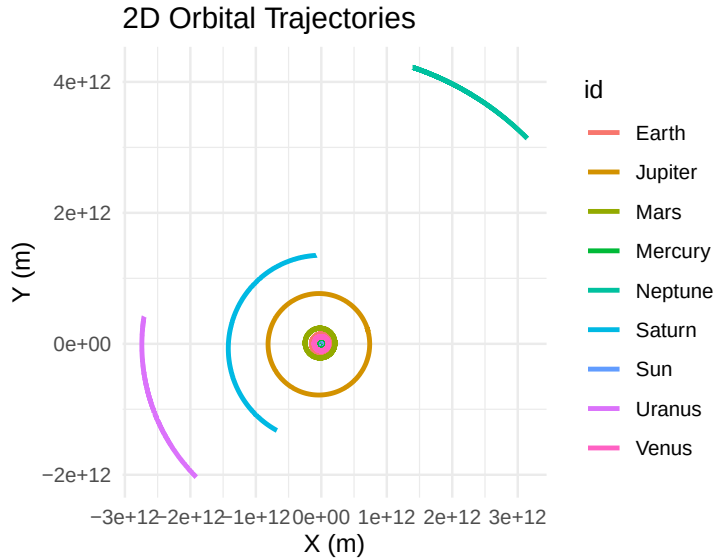


Figure 2.6.: The whole solar system for twelve years. The inner planets are a smudge at this scale; Neptune has completed less than a tenth of an orbit.

`load_solar_system()` builds the Sun, all eight planets, the Moon, and Pluto in one call, and `remove_body()` drops the ones you do not want:

```
load_solar_system() |>
  remove_body(c("Moon", "Pluto")) |>
  simulate_system(time_step = seconds_per_day, duration = seconds_per_year * 12) |>
  plot_orbits(three_d = FALSE)
```

The Moon and Pluto are removed here for different reasons. Pluto's orbit is large and does not change the picture. The Moon is removed because simulating it correctly needs a time step of an hour or so, and a twelve-year run at hourly steps is over a hundred thousand steps. One of the first things you learn about N-body simulation is that the time step is set by the *fastest* orbit in the system, not the one you care about. Chapter 5 explains why.

2.6. Looking before you leap

A system is an object you can inspect before you run it. Printing one shows the gravitational constant and the table of bodies:

2. Four Lines to an Orbit

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

sys
#>                               orbit_system
#> G: 6.6743e-11 (standard)
#> Bodies: 3
#>
#> # A tibble: 3 x 8
#>   id      mass          x          y          z      vx      vy      vz
#>   <chr>   <dbl>      <dbl>      <dbl>      <dbl>   <dbl> <dbl> <dbl>
#> 1 Sun    1.99e30          0          0          0      0      0      0
#> 2 Earth 5.97e24 -32965585121. 143360295956.      0 -29520. -6788.      0
#> 3 Mars  6.42e23 188789951360. -83706961610. -6395443036. 10750. 24226. 243.
```

`get_bodies()` returns that table as a plain tibble, which is useful for checking that a hand-built system has the positions and velocities you intended:

```
get_bodies(sys) |>
  mutate(speed = sqrt(vx^2 + vy^2 + vz^2)) |>
  select(id, speed)
#> # A tibble: 3 x 2
#>   id      speed
#>   <chr>   <dbl>
#> 1 Sun      0
#> 2 Earth 30291.
#> 3 Mars 26505.
```

Earth's initial speed is a little over its mean orbital speed of 29,780 m/s, because `add_planet()` starts each body at periapsis (its closest approach to the Sun), where it moves fastest. The `nu` argument changes the starting point along the orbit; Chapter 7 covers it with the rest of the orbital elements.

2.7. Snapshots and animation

`plot_orbits()` draws whole trajectories. `plot_system()` draws where everything is at one instant, optionally with the trajectories drawn faintly behind:

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

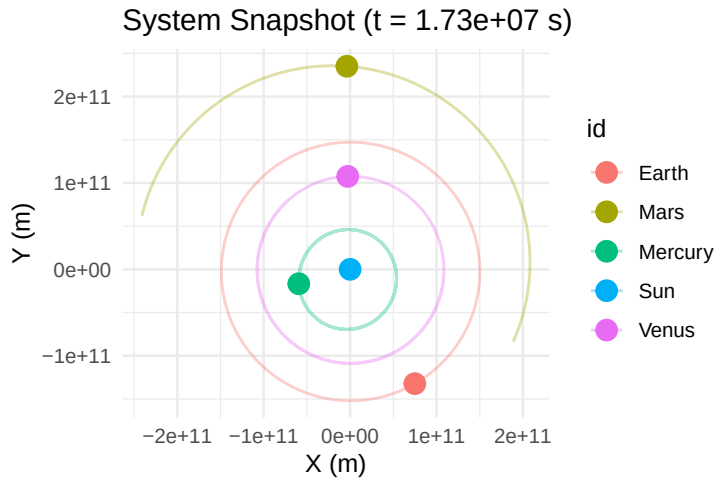


Figure 2.7.: The inner solar system at day 200 of the run, with trails.

```
simulate_system(time_step = seconds_per_day, duration = seconds_per_year)

plot_system(inner, time = seconds_per_day * 200, trails = TRUE, three_d = FALSE)
```

The `time` argument is in simulation seconds and snaps to the nearest available step. A sequence of snapshots at different times is how the printed book shows motion. On screen, use the animation:

```
animate_system(inner, fps = 15, duration = 6, trails = TRUE)
```

This samples the run down to `fps * duration` frames and renders a GIF with `gganimate` (which you need to have installed, along with `gifski`). Each body leaves a fading wake. Rendering takes tens of seconds, so get the static plot right first.

2.8. Saving and sharing

`save_system()` writes a system, bodies and all, to an `.rds` file and `load_system()` reads it back, so a carefully tuned setup does not have to be rebuilt:

2. *Four Lines to an Orbit*

```
save_system(sys, "earth-mars.rds")  
sys2 <- load_system("earth-mars.rds")
```

`export_bodies()` writes the body table to CSV, which is the right format for handing initial conditions to someone who is not using R:

```
export_bodies(sys, "earth-mars-bodies.csv")
```

The simulation output is just a tibble, so `readr::write_csv()` or `saveRDS()` handle it like any other data.

Part II.

Where the Physics Comes From

3. Newton's Law and the N-Body Problem

Everything `orbitr` does rests on one equation. This chapter writes it down carefully, in the vector form the code actually uses, and then asks the questions a careful reader should ask about it: why the mass of the body being pulled drops out, why a planet can be treated as a point, what the constant G is and how well it is known, and what the computer does with the equation once it has it. By the end of the chapter you will have written your own version of `orbitr`'s inner loop in plain R and checked it against the package.

3.1. From the apple to the Moon

Newton's law of universal gravitation says that two point masses m_1 and m_2 separated by a distance r attract each other with a force of magnitude

$$F = \frac{G m_1 m_2}{r^2}. \quad (3.1)$$

The force is along the line joining them, and it falls off as the inverse square of the distance.

Where it comes from

The inverse-square law was in the air in the 1670s. Robert Hooke, Edmond Halley, and Christopher Wren all suspected that a force falling off as $1/r^2$ would produce Kepler's ellipses; none of them could prove it. In 1684 Halley asked Newton what orbit such a force would produce. Newton said an ellipse, that he had worked it out years before, and that he had lost the calculation. He redid it, and the redoing grew into the *Principia* (Newton 1687).

The famous test is the Moon. If the same force that makes an apple fall also holds the Moon in orbit, then since the Moon is about 60 Earth radii away, its acceleration toward Earth should be $1/60^2 = 1/3600$ of g . The acceleration needed to keep a body on a circle of radius r with period T is $4\pi^2 r/T^2$, and both numbers were known in Newton's time. Here is the comparison with modern values:

3. Newton's Law and the N-Body Problem

```
R_earth <- 6.371e6                                # m
g        <- 9.81                                   # m/s^2
T_moon   <- 27.32 * seconds_per_day                # sidereal month

g * (R_earth / distance_earth_moon)^2              # inverse-square prediction
#> [1] 0.002694744
4 * pi^2 * distance_earth_moon / T_moon^2          # centripetal, from the orbit
#> [1] 0.002723668
```

They agree to about one percent. The residual is mostly the Moon's own mass: the orbit's period is set by the combined mass, $G(M + m)$, which is 1.2% larger than GM alone. Chapter 4 makes this precise. Newton's own first calculation, in the 1660s, used an inaccurate figure for the size of the Earth and did not agree well; he returned to the problem years later when Picard's survey gave a better one.

3.2. The law in vector form

A simulation needs directions, not just magnitudes. Let $\mathbf{r}_j$ be the position vector of body j . The vector from body j to body k is $\mathbf{r}_k - \mathbf{r}_j$, its length is $r_{jk} = |\mathbf{r}_k - \mathbf{r}_j|$, and the unit vector pointing from j toward k is

$$\hat{\mathbf{r}}_{jk} = \frac{\mathbf{r}_k - \mathbf{r}_j}{r_{jk}}.$$

The force on j due to k points *toward* k (gravity attracts), so

$$\mathbf{F}_{jk} = \frac{G m_j m_k}{r_{jk}^2} \hat{\mathbf{r}}_{jk}. \quad (3.2)$$

Swapping j and k flips the unit vector and nothing else, so $\mathbf{F}_{kj} = -\mathbf{F}_{jk}$: Newton's third law is built in. This is the fact behind the drifting Earth in Chapter 2. Because every internal force comes in an equal and opposite pair, the total momentum of a system of gravitating bodies never changes, no matter how complicated the motion.

3.3. Superposition: the N-body acceleration

Newton's second law gives the acceleration of body j from the net force on it, $m_j \ddot{\mathbf{r}}_j = \sum_k \mathbf{F}_{jk}$. Gravitational forces add as vectors; the pull of the Sun on Earth does not care that the Moon is also pulling. Summing Equation 3.2 over every other body and dividing by m_j ,

$$\ddot{\mathbf{r}}_j = \sum_{k \neq j} \frac{G m_k}{r_{jk}^2} \hat{\mathbf{r}}_{jk} = \sum_{k \neq j} G m_k \frac{\mathbf{r}_k - \mathbf{r}_j}{|\mathbf{r}_k - \mathbf{r}_j|^3}. \quad (3.3)$$

This is the equation. The first form is the one in the package documentation; the second is the one the code uses, because it avoids computing the unit vector separately. The cube in the denominator is one power for turning the difference vector into a unit vector and two for the inverse-square law.

Two things to notice.

The mass of body j has cancelled. The acceleration of a body in a gravitational field does not depend on its own mass. A feather and a hammer fall together; the International Space Station and an astronaut floating inside it follow the same orbit. This is only true because the m_j in Newton’s second law (inertial mass, the resistance to being accelerated) is the same quantity as the m_j in the law of gravitation (gravitational mass, the thing that gravity pulls on). Nothing in Newtonian mechanics requires those to be equal; they just are. The equality has been tested to about one part in 10^{15} (Touboul et al. 2022), and Einstein took it as the starting point of general relativity. For the simulation it means something practical: a body’s mass matters for how it pulls on the others, but a test particle of negligible mass follows exactly the same path it would with any other negligible mass.

The sum is over every pair. With N bodies there are $N(N - 1)$ ordered pairs, and each needs a subtraction, a norm, and a division. Doubling the number of bodies quadruples the work. This $O(N^2)$ scaling is the single fact that dominates the design of every N-body code ever written, and it is why orbitr’s inner loop is in C++ (Chapter 15). For the handful of bodies in this book it is not a problem.

3.4. Why point masses are enough

Earth is not a point. It is a ball 12,742 km across, and every piece of it pulls on the Moon from a slightly different direction and distance. Why is it legitimate to write Equation 3.3 with a single $\mathbf{r}_{\text{Earth}}$?

Theorem 3.1 (Newton’s shell theorem). *A spherically symmetric body attracts an external point mass exactly as if all of its mass were concentrated at its center. A spherically symmetric shell exerts no net gravitational force on a point mass anywhere inside it.*

Newton proved this geometrically in the *Principia* (Book I, Propositions 70 and 71), and he is often said to have delayed publishing the theory of gravitation until he had the proof, since without it the Moon test above is not justified. The modern proof takes three lines if you know Gauss’s law for gravity, which says that the flux of the gravitational field $\mathbf{g}$ through any closed surface is proportional to the mass enclosed:

$$\oint \mathbf{g} \cdot d\mathbf{A} = -4\pi G M_{\text{enc}}.$$

Take the surface to be a sphere of radius r centered on the body. By symmetry, $\mathbf{g}$ points radially and has the same magnitude $g(r)$ everywhere on the sphere, so the integral is $-g(r) \cdot 4\pi r^2$ (negative because the field points inward). Hence $g(r) = GM_{\text{enc}}/r^2$, directed toward the center. Outside the body, M_{enc} is the whole mass, and the field is that of a point mass. Inside a shell, $M_{\text{enc}} = 0$, and the field vanishes.

3. Newton’s Law and the N -Body Problem

So for any body whose mass is arranged in spherical shells, which is a very good description of stars and planets, the point-mass equation is not an approximation at all. It is exact, as long as the bodies do not overlap.

Where it fails is instructive. Earth bulges at the equator by about 21 km because it spins; the extra mass there pulls satellites in low orbit slightly off the point-mass prediction, and the orbits of GPS satellites precess measurably because of it. Astronomers call the leading correction J_2 . Two stars in a close binary raise tides on each other and are not spherical either. `orbitr` ignores all of this, which is the right choice for orbital distances and the wrong one for anything skimming a surface. The package roadmap lists J_2 as a possible addition; Chapter 16 discusses what it would take.

3.5. Units, G , and how well we know it

Everything in `orbitr` is in SI: meters, kilograms, seconds. The gravitational constant is

```
gravitational_constant  
#> [1] 6.6743e-11
```

in $\text{m}^3 \text{kg}^{-1} \text{s}^{-2}$. That is the CODATA 2018 recommended value, $6.67430(15) \times 10^{-11}$ (Tiesinga et al. 2021), and the digits in parentheses are the uncertainty in the last two places: about 2 parts in 10^5 . By the standards of fundamental constants that is terrible. The speed of light is exact by definition; Planck’s constant is now exact by definition; the fine-structure constant is known to a few parts in 10^{10} . G is hard to measure because gravity is so weak that laboratory experiments have to detect the pull of a kilogram-scale mass on another, and because there is no way to shield the apparatus from everything else.

The constant that astronomy actually uses is not G but the product GM for each body, called the gravitational parameter and written μ . For the Sun, $GM_\odot$ is known to about one part in 10^{10} , because it is what you get directly from fitting planetary orbits and spacecraft trajectories, no laboratory required. When the package reports the Sun’s mass as 1.989×10^{30} kg, that number is $GM_\odot$ divided by a poorly known G , and its uncertainty is essentially the uncertainty in G . None of this affects a simulation, because the equations of motion only ever contain G and a mass together. It does mean that if you ever compare a simulated orbital period to a measured one at the fifth significant figure, the discrepancy may be in G , not in the integrator.

3.6. The equations of motion as a system

Equation 3.3 is a second-order differential equation for each of N bodies. A simulation does not work with second-order equations; it works with a *state* and a rule for advancing it. The state of the system at time t is the list of all positions and velocities,

$$(\mathbf{r}_1, \dots, \mathbf{r}_N, \mathbf{v}_1, \dots, \mathbf{v}_N),$$

$6N$ numbers in all. The rule for how the state changes is a pair of first-order equations for each body,

$$\dot{\mathbf{r}}_j = \mathbf{v}_j, \quad \dot{\mathbf{v}}_j = \mathbf{a}_j(\mathbf{r}_1, \dots, \mathbf{r}_N), \quad (3.4)$$

where $\mathbf{a}_j$ is the right-hand side of Equation 3.3. The accelerations depend only on positions, not on velocities. That is a special feature of gravity (and of any force that comes from a potential), and Chapter 5 will show that the best integrators exploit it.

Here is the structure of every N-body code, including this one, in pseudo-code:

```
state <- initial positions and velocities
for each time step:
  a      <- accelerations(state)          # the N-body sum, all pairs
  state <- advance(state, a, dt)         # the integrator
  record state
```

The `accelerations()` function is the physics. The `advance()` function is the numerics. `orbitr` keeps them separate, and so will this book: this chapter is about the first function, Chapter 5 is about the second.

3.7. Writing the acceleration yourself

The best way to be sure what Equation 3.3 says is to implement it. Here it is in plain R, with no attempt at speed:

```
accelerations <- function(bodies, G = gravitational_constant) {
  n <- nrow(bodies)
  pos <- as.matrix(bodies[, c("x", "y", "z")])
  a <- matrix(0, nrow = n, ncol = 3)
  for (j in seq_len(n)) {
    for (k in seq_len(n)) {
      if (j == k) next
      d <- pos[k, ] - pos[j, ]          # vector from j toward k
      r <- sqrt(sum(d^2))                # distance
      a[j, ] <- a[j, ] + G * bodies$mass[k] * d / r^3
    }
  }
  tibble(id = bodies$id, ax = a[, 1], ay = a[, 2], az = a[, 3])
}
```

The double loop runs over all ordered pairs (j, k) with $j \neq k$. For each pair it forms the difference vector, its length, and the term in the sum. `bodies` is any data frame with columns `id`, `mass`, `x`, `y`, `z`, which is exactly what `get_bodies()` returns:

3. Newton's Law and the N-Body Problem

```
em <- create_system() |>
  add_body("Earth", mass = mass_earth) |>
  add_body("Moon", mass = mass_moon, x = distance_earth_moon, vy = speed_moon)

accelerations(get_bodies(em))
#> # A tibble: 2 x 4
#>   id          ax      ay      az
#>   <chr>      <dbl> <dbl> <dbl>
#> 1 Earth  0.0000332      0      0
#> 2 Moon -0.00270      0      0
```

The Moon's acceleration is in the $-x$ direction, toward Earth, with magnitude GM_{Earth}/r^2 :

```
gravitational_constant * mass_earth / distance_earth_moon^2
#> [1] 0.002697483
```

and Earth's is in the $+x$ direction, toward the Moon, smaller by the mass ratio $m_{\text{Moon}}/M_{\text{Earth}} \approx 1/81$. Equal and opposite *forces*; unequal accelerations.

Now a check against the package. The package's output does not include accelerations, but it does include velocities at every step, and over one hour the velocity change of the Moon should be close to $\mathbf{a} \Delta t$:

```
sim <- simulate_system(em, time_step = seconds_per_hour,
  duration = seconds_per_hour)

sim |>
  filter(id == "Moon") |>
  summarise(dvx_per_second = diff(vx) / seconds_per_hour)
#> # A tibble: 1 x 1
#>   dvx_per_second
#>   <dbl>
#> 1 -0.00270
```

That is the x acceleration computed above, to about four significant figures. (It is not exact because the acceleration changes slightly during the hour as the Moon moves; the integrator uses the average of the accelerations at the start and end of the step, which is one of the things that makes it good.)

3.8. A three-body example

With three bodies the sums in Equation 3.3 have two terms each. Put the Sun at the origin, Earth at 1 AU, and the Moon 384,400 km beyond Earth, and ask how hard the Sun pulls on the Moon compared to how hard Earth does:

```
sem <- create_system() |>
  add_sun() |>
  add_body("Earth", mass = mass_earth, x = distance_earth_sun) |>
  add_body("Moon", mass = mass_moon,
    x = distance_earth_sun + distance_earth_moon)

a_sun_on_moon <- gravitational_constant * mass_sun /
  (distance_earth_sun + distance_earth_moon)^2
a_earth_on_moon <- gravitational_constant * mass_earth /
  distance_earth_moon^2

a_sun_on_moon / a_earth_on_moon
#> [1] 2.187709
```

The Sun pulls on the Moon more than twice as hard as Earth does. This surprises almost everyone the first time. The Moon orbits Earth not because Earth's pull is stronger but because Earth and the Moon are *both* falling toward the Sun together, at nearly the same rate; what keeps the Moon bound to Earth is the small *difference* between the Sun's pull on the two of them. Seen from the Sun, the Moon's path is a slightly wobbly ellipse that is everywhere concave toward the Sun. Chapter 10 draws it.

The full acceleration of the Moon is the vector sum of both terms, and our function gets it right with no changes:

```
accelerations(get_bodies(sem)) |>
  filter(id == "Moon")
#> # A tibble: 1 x 4
#>   id      ax      ay      az
#>   <chr>   <dbl> <dbl> <dbl>
#> 1 Moon -0.00860      0      0
```

3.9. What the engine does differently

The function above is correct and slow. Each call does $N(N-1)$ iterations of interpreted R, and a simulation calls it once or twice per step for hundreds of thousands of steps. The package's pure-R fallback vectorizes the same computation with matrix outer products so that R does the loop in compiled code internally; its C++ engine writes the double loop directly in C++ and runs about as fast as the hardware allows. Both compute exactly Equation 3.3, with one small addition: an optional softening length ε that replaces r^3 in the denominator with $(r^2 + \varepsilon^2)^{3/2}$. With $\varepsilon = 0$, the default, there is no difference. Chapter 6 explains what softening is for.

Key results

- The acceleration of body j is $\ddot{\mathbf{r}}_j = \sum_{k \neq j} G m_k \frac{\mathbf{r}_k - \mathbf{r}_j}{|\mathbf{r}_k - \mathbf{r}_j|^3}$. It does not depend on m_j .

3. *Newton's Law and the N-Body Problem*

- Internal forces come in equal and opposite pairs, so total momentum is conserved: the center of mass of any isolated system moves at constant velocity.
- A spherically symmetric body gravitates exactly like a point mass at its center (shell theorem).
- The computational cost grows as N^2 .

4. The Two-Body Problem, Solved by Hand

The package documentation hands you a handful of formulas. The circular-orbit speed is $\sqrt{GM/r}$; escape speed is $\sqrt{2}$ times that; an orbit's period is $2\pi\sqrt{a^3/\mu}$; the distance from the parent at true anomaly ν is $a(1 - e^2)/(1 + e \cos \nu)$. Every one of them is true, and every one of them follows from Newton's law in a few lines of vector algebra. This chapter does the algebra. It is the one place in the book where the derivations come first and the code comes second, and it will repay the effort: these are the formulas you will use to set up every system in Part III.

The strategy is the physicist's standard one. First find the quantities that *do not change* as the bodies move, the conserved quantities. Then use them to pin down the shape of the orbit without ever integrating anything. Along the way, Kepler's three laws drop out as consequences.

4.1. Two bodies become one

Take two point masses, m_1 at $\mathbf{r}_1$ and m_2 at $\mathbf{r}_2$, with nothing else in the universe. From Equation 3.3,

$$\ddot{\mathbf{r}}_1 = Gm_2 \frac{\mathbf{r}_2 - \mathbf{r}_1}{r^3}, \quad \ddot{\mathbf{r}}_2 = Gm_1 \frac{\mathbf{r}_1 - \mathbf{r}_2}{r^3},$$

where $r = |\mathbf{r}_2 - \mathbf{r}_1|$. Subtract the first from the second. Writing $\mathbf{r} = \mathbf{r}_2 - \mathbf{r}_1$ for the *relative* position of body 2 as seen from body 1,

$$\ddot{\mathbf{r}} = -\frac{\mu}{r^3} \mathbf{r}, \quad \mu = G(m_1 + m_2). \quad (4.1)$$

Two coupled equations in six coordinates have become one equation in three. It says the relative position vector accelerates toward the origin as if there were a single fixed mass $m_1 + m_2$ sitting there. The combination $\mu = G(m_1 + m_2)$ is called the gravitational parameter, and it is the only combination of G and the masses that the relative motion knows about.

What happened to the other three coordinates? Add m_1 times the first equation to m_2 times the second: the right-hand sides cancel, so the center of mass $\mathbf{R} = (m_1\mathbf{r}_1 + m_2\mathbf{r}_2)/(m_1 + m_2)$ has $\ddot{\mathbf{R}} = 0$. It moves in a straight line at constant velocity, which is the drift we met in Chapter 2. Given $\mathbf{R}$ and $\mathbf{r}$, the individual positions are

$$\mathbf{r}_1 = \mathbf{R} - \frac{m_2}{m_1 + m_2} \mathbf{r}, \quad \mathbf{r}_2 = \mathbf{R} + \frac{m_1}{m_1 + m_2} \mathbf{r}. \quad (4.2)$$

4. The Two-Body Problem, Solved by Hand

Both bodies trace the same shape of orbit around the center of mass, scaled by the mass ratio. In Chapter 2's fixed Earth-Moon system, Earth's little circle had a radius $m_{\text{Moon}}/(m_{\text{Earth}} + m_{\text{Moon}}) \approx 1/82$ of the Moon's.

When one body is much heavier, $m_1 \gg m_2$, two things happen at once: $\mu \approx Gm_1$, and $\mathbf{r}_1 \approx \mathbf{R}$, so the heavy body sits essentially at the center of mass and the light body orbits it. That is the “fixed central body” picture of the two-body vignette, and the approximation is excellent for the Sun and any planet (the Sun outweighs Jupiter a thousand to one) and merely good for Earth and the Moon, where the correction is 1.2%. From here on the derivation is exact for the relative motion; whether you then set $\mu = GM$ or $\mu = G(M + m)$ is a choice about how much precision you need.

4.2. Angular momentum is conserved, and the orbit is flat

Define the *specific angular momentum* (angular momentum per unit mass) of the relative motion,

$$\mathbf{h} = \mathbf{r} \times \dot{\mathbf{r}}.$$

Differentiate it using the product rule:

$$\dot{\mathbf{h}} = \dot{\mathbf{r}} \times \dot{\mathbf{r}} + \mathbf{r} \times \ddot{\mathbf{r}} = \mathbf{0} + \mathbf{r} \times \left(-\frac{\mu}{r^3}\mathbf{r}\right) = \mathbf{0}.$$

The first term is a vector crossed with itself and the second is $\mathbf{r}$ crossed with a multiple of $\mathbf{r}$; both vanish. So $\mathbf{h}$ is a constant of the motion. This works for *any* force directed along $\mathbf{r}$, not just inverse-square: it is a consequence of the force being central.

Two consequences follow immediately. Since $\mathbf{h}$ is perpendicular to both $\mathbf{r}$ and $\dot{\mathbf{r}}$ at every instant, and $\mathbf{h}$ never changes, the position and velocity are confined forever to the plane perpendicular to $\mathbf{h}$. **Every two-body orbit is planar.** And the magnitude $h = |\mathbf{r} \times \dot{\mathbf{r}}|$ is constant. In a small time dt the position vector sweeps out a thin triangle with sides $\mathbf{r}$ and $\dot{\mathbf{r}} dt$, whose area is half the magnitude of their cross product:

$$\frac{dA}{dt} = \frac{1}{2} |\mathbf{r} \times \dot{\mathbf{r}}| = \frac{h}{2}. \quad (4.3)$$

The line from the central body to the orbiting one sweeps out equal areas in equal times. That is **Kepler's second law**, and it took one line.

4.3. Energy is conserved

Now dot Equation 4.1 with the velocity:

$$\dot{\mathbf{r}} \cdot \ddot{\mathbf{r}} = -\frac{\mu}{r^3} \mathbf{r} \cdot \dot{\mathbf{r}}.$$

The left side is $\frac{d}{dt} \left(\frac{1}{2} \dot{\mathbf{r}} \cdot \dot{\mathbf{r}} \right) = \frac{d}{dt} \left(\frac{1}{2} v^2 \right)$. For the right side, note that $r^2 = \mathbf{r} \cdot \mathbf{r}$, so $2r\dot{r} = 2\mathbf{r} \cdot \dot{\mathbf{r}}$ and $\mathbf{r} \cdot \dot{\mathbf{r}} = r\dot{r}$. Then $-\mu \mathbf{r} \cdot \dot{\mathbf{r}} / r^3 = -\mu \dot{r} / r^2 = \frac{d}{dt} (\mu / r)$. Moving everything to one side,

$$\frac{d}{dt} \left(\frac{v^2}{2} - \frac{\mu}{r} \right) = 0, \quad \varepsilon \equiv \frac{v^2}{2} - \frac{\mu}{r} = \text{const.} \quad (4.4)$$

ε is the *specific orbital energy*: kinetic energy per unit mass plus gravitational potential energy per unit mass. If $\varepsilon < 0$ the body cannot reach $r = \infty$ (it would need negative kinetic energy there), so it is bound. If $\varepsilon \geq 0$ it can, and it escapes.

Chapter 12 turns both of these conservation laws into diagnostics: a simulation that lets ε or $\mathbf{h}$ drift is a simulation with a problem.

4.4. The orbit equation

Here is the step that makes the whole chapter work. Conservation of $\mathbf{h}$ and ε constrains the motion but does not by itself give the shape of the path. For the inverse-square force, and essentially only for it, there is a *third* conserved vector, and it hands you the orbit directly.

Start from the time derivative of $\dot{\mathbf{r}} \times \mathbf{h}$. Since $\mathbf{h}$ is constant,

$$\frac{d}{dt} (\dot{\mathbf{r}} \times \mathbf{h}) = \ddot{\mathbf{r}} \times \mathbf{h} = -\frac{\mu}{r^3} \mathbf{r} \times (\mathbf{r} \times \dot{\mathbf{r}}).$$

Expand the triple product with the identity $\mathbf{a} \times (\mathbf{b} \times \mathbf{c}) = \mathbf{b}(\mathbf{a} \cdot \mathbf{c}) - \mathbf{c}(\mathbf{a} \cdot \mathbf{b})$:

$$\mathbf{r} \times (\mathbf{r} \times \dot{\mathbf{r}}) = \mathbf{r}(\mathbf{r} \cdot \dot{\mathbf{r}}) - \dot{\mathbf{r}} r^2 = \mathbf{r} r \dot{r} - \dot{\mathbf{r}} r^2,$$

so

$$\frac{d}{dt} (\dot{\mathbf{r}} \times \mathbf{h}) = \mu \left(\frac{\dot{\mathbf{r}}}{r} - \frac{\dot{r} \mathbf{r}}{r^2} \right) = \mu \frac{d}{dt} \left(\frac{\mathbf{r}}{r} \right).$$

The last equality is just the quotient rule applied to $\mathbf{r}/r$. Both sides are total time derivatives, so their difference is a constant vector. Divide by μ and call the constant $\mathbf{e}$:

$$\mathbf{e} = \frac{\dot{\mathbf{r}} \times \mathbf{h}}{\mu} - \frac{\mathbf{r}}{r}. \quad (4.5)$$

4. The Two-Body Problem, Solved by Hand

This is the *eccentricity vector* (also called the Laplace-Runge-Lenz vector, in various normalizations). It lies in the orbital plane, because both terms do. To see what it means, take its dot product with $\mathbf{r}$. The first term gives $\mathbf{r} \cdot (\dot{\mathbf{r}} \times \mathbf{h})/\mu = \mathbf{h} \cdot (\mathbf{r} \times \dot{\mathbf{r}})/\mu = h^2/\mu$, using the cyclic property of the scalar triple product; the second gives $-r$. If ν is the angle between $\mathbf{e}$ and $\mathbf{r}$, the dot product is also $er \cos \nu$. So

$$er \cos \nu = \frac{h^2}{\mu} - r \quad \Rightarrow \quad r = \frac{h^2/\mu}{1 + e \cos \nu}. \quad (4.6)$$

This is the **orbit equation**: the distance from the central body as a function of the angle ν measured from the direction of $\mathbf{e}$. The quantity $p = h^2/\mu$ is called the semi-latus rectum, and $e = |\mathbf{e}|$ is the eccentricity. The distance r is smallest when $\cos \nu = 1$, so $\mathbf{e}$ points toward the point of closest approach, the *periapsis*, and ν , the *true anomaly*, is the angle from periapsis. Equation 4.6 is a conic section with one focus at the origin:

Table 4.1.: The four kinds of two-body orbit.

e	ε	shape	$r_{\max}$
0	$-\mu^2/2h^2$	circle	$= r_{\min}$
$0 < e < 1$	< 0	ellipse	finite
1	0	parabola	∞
> 1	> 0	hyperbola	∞

Bound orbits are ellipses with the central body at a focus. That is **Kepler's first law**, and it has now been derived from the inverse-square law, which is the thing Halley asked Newton about in 1684.

The relation between e and ε in the table comes from squaring Equation 4.5. Since $\dot{\mathbf{r}} \perp \mathbf{h}$, $|\dot{\mathbf{r}} \times \mathbf{h}| = v h$, and the cross term is $-2\mathbf{r} \cdot (\dot{\mathbf{r}} \times \mathbf{h})/(\mu r) = -2h^2/(\mu r)$, so

$$e^2 = \frac{v^2 h^2}{\mu^2} - \frac{2h^2}{\mu r} + 1 = 1 + \frac{2\varepsilon h^2}{\mu^2}. \quad (4.7)$$

Negative energy means $e < 1$: bound orbits are ellipses, as the table says.

Where it comes from

Kepler extracted his first two laws from Tycho Brahe's observations of Mars, published in *Astronomia Nova* in 1609 (Kepler 1609), and the third from the periods and distances of all the planets, published in *Harmonices Mundi* in 1619 (Kepler 1619). He had no theory of why; he had tried circles and epicycles for years and the data would not fit. Newton's *Principia* (Newton 1687) showed that all three laws follow from an inverse-square force (Book I, Propositions 11 to 15), using geometry rather than vectors. The vector form here is the standard modern one; Goldstein (Goldstein et al. 2002) and Bate, Mueller, and White (Bate et al. 1971) both present it, and the eccentricity vector is the starting point of every algorithm that turns a position and velocity into orbital elements, including the one in Chapter 7.

4.5. The geometry of an ellipse

An ellipse has a longest diameter, the major axis, of length $2a$; a is the semi-major axis. From Equation 4.6 the nearest and farthest points are at $\nu = 0$ and $\nu = \pi$:

$$r_p = \frac{p}{1+e}, \quad r_a = \frac{p}{1-e}, \quad a = \frac{r_p + r_a}{2} = \frac{p}{1-e^2}.$$

So $p = a(1-e^2)$, and the orbit equation takes the form the package uses:

$$r = \frac{a(1-e^2)}{1+e\cos\nu}, \quad r_p = a(1-e), \quad r_a = a(1+e). \quad (4.8)$$

The semi-minor axis is $b = a\sqrt{1-e^2}$. Two more facts fall out of these. First, given any two-body orbit, you can read off its eccentricity from its extreme distances alone:

$$e = \frac{r_a - r_p}{r_a + r_p}. \quad (4.9)$$

Second, substituting $h^2 = \mu p = \mu a(1-e^2)$ into Equation 4.7 and simplifying (try it; everything but one factor cancels) gives the energy in terms of the size of the orbit alone:

$$\varepsilon = -\frac{\mu}{2a}. \quad (4.10)$$

The energy of an orbit depends only on its semi-major axis, not on its eccentricity. A circle of radius a and a needle-thin ellipse of semi-major axis a have the same energy.

4.6. Vis-viva, circular speed, and escape speed

Put Equation 4.10 into the definition of ε and solve for the speed:

$$v^2 = \mu \left(\frac{2}{r} - \frac{1}{a} \right). \quad (4.11)$$

This is the *vis-viva* equation (Latin for “living force,” an old name for kinetic energy). It gives the speed at any distance r along an orbit of semi-major axis a , and it is the formula you want whenever you know where a body is and need to know how fast it should be going.

Two special cases are the ones the documentation quotes. On a circular orbit, $r = a$ always, so

$$v_{\text{circ}} = \sqrt{\frac{\mu}{r}}. \quad (4.12)$$

4. The Two-Body Problem, Solved by Hand

On a parabolic orbit, $a \rightarrow \infty$ and the $1/a$ term vanishes, so the speed that just barely escapes from distance r is

$$v_{\text{esc}} = \sqrt{\frac{2\mu}{r}} = \sqrt{2} v_{\text{circ}}. \quad (4.13)$$

Now the rules of thumb in the two-body vignette can be made exact. Suppose you place a body at distance r from the central mass and launch it perpendicular to the radius at speed $v = k v_{\text{circ}}$. Then $h = rv$, so $p = h^2/\mu = r^2 v^2/\mu = k^2 r$. The launch point has $\dot{r} = 0$, so it is either periapsis or apoapsis, and Equation 4.6 gives $r = p/(1 \pm e)$, or

$$e = |k^2 - 1|. \quad (4.14)$$

If $k > 1$ the launch point is periapsis and the orbit swings outward; if $k < 1$ it is apoapsis and the orbit dips inward; $k = 1$ is a circle; $k = \sqrt{2}$ gives $e = 1$, a parabola. A planet launched at 1.2 times circular speed has $e = 0.44$; at 1.4 times, $e = 0.96$, a sliver away from escape. We will check both numbers in a moment.

4.7. Kepler's third law

The area of an ellipse is πab . By Equation 4.3 the orbit sweeps area at the constant rate $h/2$, so one full orbit takes $T = 2\pi ab/h$. Substitute $b = a\sqrt{1 - e^2}$ and $h = \sqrt{\mu a(1 - e^2)}$:

$$T = \frac{2\pi a^2 \sqrt{1 - e^2}}{\sqrt{\mu a(1 - e^2)}} = 2\pi \sqrt{\frac{a^3}{\mu}}. \quad (4.15)$$

The eccentricity has cancelled: **Kepler's third law**. The period depends on the semi-major axis and the total mass and on nothing else. Kepler's own statement, that T^2/a^3 is the same number for every planet, holds only because every planet's mass is negligible next to the Sun's, so $\mu \approx GM_{\odot}$ for all of them. The correction for Jupiter is one part in a thousand in μ , half that in T .

This is also the law that weighs things. Measure a moon's period and orbital radius and Equation 4.15 gives you the mass of its planet; this is how the masses in Appendix D were obtained, and it is how we know the mass of the Sun, of Jupiter, and of the black hole at the center of the Milky Way.

4.8. Checking everything against the simulation

Time to see whether the package agrees with the algebra. The derivations above are exact for two bodies; the simulation is approximate; if they match, both the physics and the numerics are in good shape. To keep the comparison honest I will compute every prediction with the full $\mu = G(M + m)$.

4.8.1. The speed sweep

Set up the vignette's star and planet, and launch the planet at several multiples of circular speed:

```
M <- 1e30
m <- 1e24
r0 <- 1e11
mu <- gravitational_constant * (M + m)
v_circ <- sqrt(mu / r0)

make_sim <- function(k, duration = seconds_per_year * 2) {
  create_system() |>
    add_body("Star", mass = M) |>
    add_body("Planet", mass = m, x = r0, vy = k * v_circ) |>
    simulate_system(time_step = seconds_per_hour, duration = duration) |>
    mutate(k = k)
}

sweep <- bind_rows(lapply(c(0.7, 1.0, 1.2, 1.4), make_sim))

sweep |>
  filter(id == "Planet") |>
  ggplot(aes(x, y, color = factor(k))) +
  geom_path() +
  annotate("point", x = 0, y = 0, size = 3) +
  coord_equal() +
  labs(x = NULL, y = NULL, color = expression(v / v[circ]))
```

4.8.2. Measuring eccentricity from a single state

Equation 4.5 is more than a derivation device. It gives the eccentricity from the position and velocity at *any one instant*, no full orbit required. Two helpers make that easy. The first extracts the relative motion of one body about another, the quantity all of this chapter's formulas are about:

```
relative_state <- function(sim, body, center, G = gravitational_constant) {
  ctr <- sim |>
    filter(id == center) |>
    select(time, m_c = mass, x_c = x, y_c = y, z_c = z,
           vx_c = vx, vy_c = vy, vz_c = vz)
  sim |>
    filter(id == body) |>
    inner_join(ctr, by = "time") |>
    mutate(mu = G * (mass + m_c),
           rx = x - x_c, ry = y - y_c, rz = z - z_c,
           ux = vx - vx_c, uy = vy - vy_c, uz = vz - vz_c) |>
```

4. The Two-Body Problem, Solved by Hand

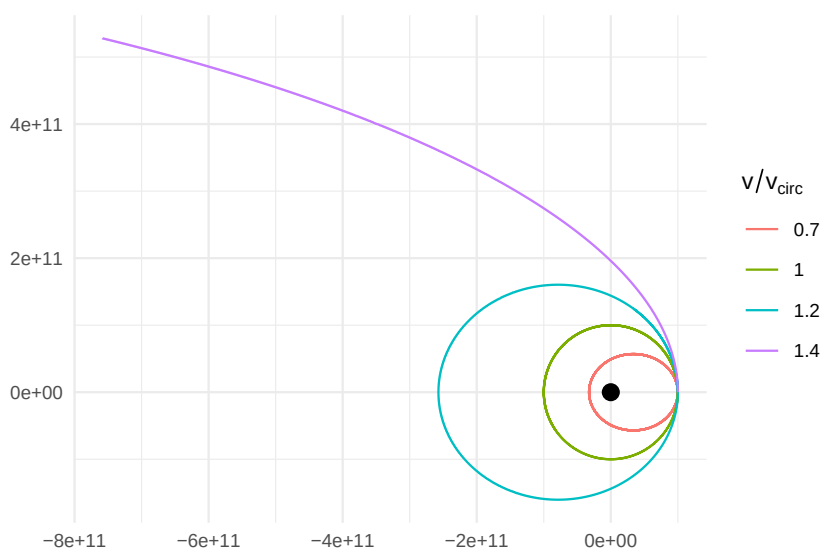


Figure 4.1.: The same launch point, four launch speeds. The circle is $k = 1$; slower launches dip inward from apoapsis, faster ones swing outward from periapsis. The $k = 1.4$ orbit has $e = 0.96$ and a period of nearly a century, so two years show only its beginning.

4.8. Checking everything against the simulation

```
    select(time, mu, rx, ry, rz, ux, uy, uz)
  }
```

The second computes $\mathbf{h} = \mathbf{r} \times \mathbf{u}$ and then $\mathbf{e} = (\mathbf{u} \times \mathbf{h})/\mu - \mathbf{r}/r$ component by component:

```
eccentricity <- function(sim, body, center, G = gravitational_constant) {
  relative_state(sim, body, center, G) |>
  mutate(r = sqrt(rx^2 + ry^2 + rz^2),
         hx = ry * uz - rz * uy,                # h = r x u
         hy = rz * ux - rx * uz,
         hz = rx * uy - ry * ux,
         ex = (uy * hz - uz * hy) / mu - rx / r, # e = (u x h)/mu - r/r
         ey = (uz * hx - ux * hz) / mu - ry / r,
         ez = (ux * hy - uy * hx) / mu - rz / r,
         e = sqrt(ex^2 + ey^2 + ez^2)) |>
  select(time, e, ex, ey, ez)
}
```

Now compare the prediction $e = |k^2 - 1|$ with the eccentricity measured from the simulated state at the *end* of each run, two years of integration away from the initial conditions:

```
sweep |>
  group_by(k) |>
  group_modify(~ eccentricity(.x, "Planet", "Star") |>
    filter(time == max(time))) |>
  ungroup() |>
  mutate(predicted = abs(k^2 - 1)) |>
  select(k, predicted, measured = e)
#> # A tibble: 4 x 3
#>       k predicted    measured
#>   <dbl>    <dbl>    <dbl>
#> 1  0.7      0.51 0.510
#> 2  1.0      0.000000413
#> 3  1.2      0.44 0.440
#> 4  1.4      0.96 0.960
```

The agreement is to many decimal places. And since $\mathbf{e}$ is supposed to be a constant of the motion, we can ask how constant the simulation kept it over the whole two years for the $k = 1.2$ orbit:

```
e_track <- eccentricity(filter(sweep, k == 1.2), "Planet", "Star")
range(e_track$e)
#> [1] 0.4400000 0.4400002
```

The spread between the smallest and largest value is the integrator's error. It is small. Chapter 5 explains why it is small and what it depends on, and shows what the same test looks like with a worse integrator.

4. The Two-Body Problem, Solved by Hand

The package does this computation for you. `get_orbital_elements()` applies exactly the eccentricity-vector and vis-viva formulas of this chapter to every time step and returns the full set of orbital elements, with the semi-major axis and the Kepler's-third-law period alongside the eccentricity:

```
get_orbital_elements(filter(sweep, k == 1.2), "Planet", "Star",
                           mu = gravitational_constant * (M + m)) |>
  summarise(e = mean(e), a = mean(a), period_days = mean(period) / seconds_per_day)
#> # A tibble: 1 x 3
#>       e          a period_days
#>   <dbl>    <dbl>    <dbl>
#> 1 0.440 178571465973.    672.
```

(The `mu` argument is there because, by default, the function uses the parent's mass alone, which is the convention of `add_body_keplerian()` and is exact in the test-particle limit; passing $G(M + m)$ gives the exact two-body elements, which is what the derivations here are about.) I wrote `eccentricity()` out by hand above so that you could see the formula do its work; from here on the book uses the package function, and Chapter 7 derives the rest of what it computes.

4.8.3. Measuring the period

Kepler's third law needs a period. `get_orbital_elements()` reports one, but it is computed from the third law, $T = 2\pi\sqrt{a^3/\mu}$, so using it to check the third law would be circular. An independent measurement means finding when the body has actually gone all the way round. The cleanest way is to track the polar angle of the relative position, unwrap the jump from $+\pi$ to $-\pi$, and interpolate to the moment the total angle reaches 2π :

```
measure_period <- function(sim, body, center = NULL, G = gravitational_constant) {
  if (is.null(center)) {                                # measure about the origin
    rel <- sim |> filter(id == body) |> transmute(time, rx = x, ry = y)
  } else {
    rel <- relative_state(sim, body, center, G)
  }
  theta <- atan2(rel$ry, rel$rx)
  dtheta <- ((diff(theta) + pi) %% (2 * pi)) - pi      # unwrap the jump at +/- pi
  swept <- abs(cumsum(c(0, dtheta)))                  # total angle travelled
  i <- which(swept >= 2 * pi)[1]
  if (is.na(i)) return(NA_real_)                      # the run was too short
  f <- (2 * pi - swept[i - 1]) / (swept[i] - swept[i - 1])
  rel$time[i - 1] + f * (rel$time[i] - rel$time[i - 1])
}
```

For the circular case, Equation 4.15 with $a = r_0$:

4.8. Checking everything against the simulation

```
T_pred <- 2 * pi * sqrt(r0^3 / mu)
T_meas <- measure_period(filter(sweep, k == 1), "Planet", "Star")

c(predicted_days = T_pred / seconds_per_day,
  measured_days = T_meas / seconds_per_day)
#> predicted_days measured_days
#>      281.4900      281.4901
```

For the $k = 1.2$ ellipse the semi-major axis is $a = r_p / (1 - e)$ with $r_p = r_0$ and $e = 0.44$, which makes the orbit larger and the period longer:

```
a_12 <- r0 / (1 - 0.44)
T_pred <- 2 * pi * sqrt(a_12^3 / mu)
T_meas <- measure_period(filter(sweep, k == 1.2), "Planet", "Star")

c(predicted_days = T_pred / seconds_per_day,
  measured_days = T_meas / seconds_per_day)
#> predicted_days measured_days
#>      671.7088      671.7090
```

4.8.4. The Moon, revisited

Chapter 2's four-line Moon started at the semi-major-axis distance of 384,400 km with the mean orbital speed of 1,022 m/s. We can now say exactly what orbit that is. The circular speed at that distance, with the full $\mu = G(M_{\text{Earth}} + m_{\text{Moon}})$, is a little *higher* than 1,022 m/s, so the launch point is apoapsis and the orbit dips inward:

```
mu_em <- gravitational_constant * (mass_earth + mass_moon)
k_moon <- speed_moon / sqrt(mu_em / distance_earth_moon)
e_moon <- abs(k_moon^2 - 1)
a_moon <- distance_earth_moon / (1 + e_moon)      # launch point is apoapsis
T_moon <- 2 * pi * sqrt(a_moon^3 / mu_em)

c(k = k_moon, e = e_moon, a_km = a_moon / 1e3, T_days = T_moon / seconds_per_day)
#>          k          e          a_km          T_days
#> 9.975312e-01 4.931523e-03 3.825136e+05 2.708447e+01
```

Against the simulation:

```
moon <- 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)

measure_period(moon, "Moon", "Earth") / seconds_per_day
#> [1] 27.08531
```

4. The Two-Body Problem, Solved by Hand

The simulation and the formula agree. But the real Moon’s sidereal period is 27.32 days, and the four-line Moon’s is about 27.1. The simulation is not wrong; the *setup* describes a slightly different orbit from the real one. “Semi-major axis” and “mean speed” are two properties of the real elliptical orbit, but using one as a starting distance and the other as a starting speed does not reproduce that ellipse; it produces a nearby one with a smaller a and hence a shorter period. The right way to put a body on its real orbit is to specify the orbit itself, by its orbital elements, and let the package compute a consistent position and velocity. That is what `add_planet()` does and what Chapter 7 derives.

Key results

For the relative motion of two bodies with $\mu = G(m_1 + m_2)$:

- $\mathbf{h} = \mathbf{r} \times \dot{\mathbf{r}}$ and $\varepsilon = v^2/2 - \mu/r$ are constant; the orbit is planar and sweeps equal areas in equal times.
- Orbit equation: $r = a(1 - e^2)/(1 + e \cos \nu)$, with the central body at a focus. $e < 1$ bound, $e \geq 1$ unbound.
- $\varepsilon = -\mu/2a$, independent of e .
- Vis-viva: $v^2 = \mu(2/r - 1/a)$. Circular speed $\sqrt{\mu/r}$; escape speed $\sqrt{2\mu/r}$.
- Launching perpendicular to the radius at k times circular speed gives $e = |k^2 - 1|$.
- Period: $T = 2\pi\sqrt{a^3/\mu}$.
- $e = (r_a - r_p)/(r_a + r_p)$, and $\mathbf{e} = (\dot{\mathbf{r}} \times \mathbf{h})/\mu - \mathbf{r}/r$ from any single state.

5. Integrators: Euler, Euler-Cromer, and Velocity Verlet

Chapter 3 ended with the two halves of every N-body code: a function that computes accelerations from positions, and a function that advances the state by one step. This chapter is about the second one. `simulate_system()` offers three choices for it, through its `method` argument: "euler", "euler_cromer", and "verlet", the default. They differ by a few characters of code and by everything that matters. The textbook method quietly feeds energy into every orbit it touches; the default keeps a planet on a closed ellipse for as long as you care to run it. The reason is not that Verlet is “more accurate” in the usual sense. It is that Verlet respects a geometric property of Newton’s equations that Euler violates, and the goal of this chapter is to make that property concrete enough that you could check for it yourself.

5.1. The problem of taking a step

The state of a body is its position $\mathbf{x}$ and velocity $\mathbf{v}$, and the equations of motion say $\dot{\mathbf{x}} = \mathbf{v}$ and $\dot{\mathbf{v}} = \mathbf{a}(\mathbf{x})$, where the acceleration depends only on positions (Chapter 3). Given the state at time t , we want the state at $t + \Delta t$.

Taylor’s theorem is the whole toolkit. For a smooth trajectory,

$$\begin{aligned}\mathbf{x}(t + \Delta t) &= \mathbf{x} + \mathbf{v} \Delta t + \frac{1}{2} \mathbf{a} \Delta t^2 + \frac{1}{6} \dot{\mathbf{a}} \Delta t^3 + O(\Delta t^4), \\ \mathbf{v}(t + \Delta t) &= \mathbf{v} + \mathbf{a} \Delta t + \frac{1}{2} \dot{\mathbf{a}} \Delta t^2 + O(\Delta t^3),\end{aligned}\tag{5.1}$$

where everything on the right is evaluated at t . We know $\mathbf{x}$ and $\mathbf{v}$, and we can compute $\mathbf{a}$ from $\mathbf{x}$. We do not know $\dot{\mathbf{a}}$ or anything beyond it. Every integrator is a decision about what to do with the terms we cannot compute.

A method whose single-step error is $O(\Delta t^{p+1})$ is said to be of *order* p , because over a fixed total time T there are $T/\Delta t$ steps and the errors accumulate to $O(\Delta t^p)$. Halve the step and a first-order method’s error halves; a second-order method’s error drops by four. That is the usual way to grade integrators, and it will turn out to be only half the story.

5.2. Euler’s method

The simplest decision is to keep only the terms we can compute exactly and drop the rest:

5. Integrators: Euler, Euler-Cromer, and Velocity Verlet

$$\begin{aligned}\mathbf{x}_{n+1} &= \mathbf{x}_n + \mathbf{v}_n \Delta t, \\ \mathbf{v}_{n+1} &= \mathbf{v}_n + \mathbf{a}_n \Delta t.\end{aligned}\tag{5.2}$$

Both updates truncate Equation 5.1 after the first power of Δt , so the local error is $O(\Delta t^2)$ and the method is first order. It is what every introductory text starts with, and it is `method = "euler"`.

It is also wrong for orbits in a way that halving the step does not fix. Consider a body on a circular orbit, where $\mathbf{v} \perp \mathbf{x}$ and $\mathbf{a} \perp \mathbf{v}$. After one Euler step,

$$|\mathbf{x}_{n+1}|^2 = |\mathbf{x}_n|^2 + 2\Delta t \mathbf{x}_n \cdot \mathbf{v}_n + \Delta t^2 |\mathbf{v}_n|^2 = r^2 + v^2 \Delta t^2,$$

so the body ends the step slightly farther out, and

$$|\mathbf{v}_{n+1}|^2 = v^2 + 2\Delta t \mathbf{v}_n \cdot \mathbf{a}_n + \Delta t^2 a^2 = v^2 + a^2 \Delta t^2,$$

so it also ends the step slightly faster. Farther out means higher potential energy; faster means higher kinetic energy. Every single step adds energy to the orbit, in both accounts at once, and there is no step that ever takes any back. The orbit spirals outward, forever. A smaller Δt makes each step's contribution smaller, but there are correspondingly more steps, and the sign never changes.

There is a cleaner way to see what is going on, using the simplest oscillating system there is: a mass on a spring, $\ddot{x} = -\omega^2 x$, with state (x, v) . One Euler step is a linear map,

$$\begin{pmatrix} x_{n+1} \\ v_{n+1} \end{pmatrix} = \underbrace{\begin{pmatrix} 1 & \Delta t \\ -\omega^2 \Delta t & 1 \end{pmatrix}}_{M_E} \begin{pmatrix} x_n \\ v_n \end{pmatrix}, \quad \det M_E = 1 + \omega^2 \Delta t^2.\tag{5.3}$$

The determinant of a linear map is the factor by which it scales areas. The true dynamics of the oscillator carries any region of the (x, v) plane around an ellipse without changing its area, a fact that holds for every Hamiltonian system and is known as Liouville's theorem. Euler's map scales area by $1 + \omega^2 \Delta t^2 > 1$ on every step. The oscillator's energy is proportional to the area of the ellipse it traces in the (x, v) plane, so a map that inflates areas inflates energies. Nothing about that depends on Δt being large.

5.3. Euler-Cromer

Here is a change so small it looks like a typo:

$$\begin{aligned}\mathbf{v}_{n+1} &= \mathbf{v}_n + \mathbf{a}_n \Delta t, \\ \mathbf{x}_{n+1} &= \mathbf{x}_n + \mathbf{v}_{n+1} \Delta t.\end{aligned}\tag{5.4}$$

The velocity is updated first, and the position update uses the *new* velocity. This is `method = "euler_cromer"`, also called the semi-implicit Euler method or symplectic Euler. Its

local error is still $O(\Delta t^2)$: it is a first-order method, no more accurate than plain Euler by the usual grading. But redo the oscillator calculation. Substituting the first line into the second, $x_{n+1} = x_n + (v_n - \omega^2 x_n \Delta t) \Delta t$, so

$$M_{\text{EC}} = \begin{pmatrix} 1 - \omega^2 \Delta t^2 & \Delta t \\ -\omega^2 \Delta t & 1 \end{pmatrix}, \quad \det M_{\text{EC}} = (1 - \omega^2 \Delta t^2) + \omega^2 \Delta t^2 = 1. \quad (5.5)$$

Exactly one. The Euler-Cromer map preserves area, for every Δt , just as the true dynamics does. An integrator with this property is called *symplectic*. (In one spatial dimension, symplectic means area-preserving; in more dimensions it is a slightly stronger condition that reduces to the same idea.) A symplectic integrator cannot have the systematic energy drift that Euler has, because energy drift is area growth. Its energy error oscillates, and for a small enough step it stays bounded forever.

Where it comes from

The method is named for Alan Cromer, who published it in a two-page note in the *American Journal of Physics* in 1981 (Cromer 1981). He reports that the reordering was found by one of his students, who noticed that it made the Euler method work on oscillators when it otherwise would not. The method itself had been known to numerical analysts under other names, but Cromer’s note is where the physics-teaching community learned that the order of two lines of code could be the difference between a stable orbit and an unstable one.

Why does this version work when the other does not? The position update in Equation 5.4 uses $\mathbf{v}_{n+1}$, which is the velocity at the *end* of the step. In effect it averages a forward-looking velocity with a backward-looking acceleration, and the two mistakes cancel in exactly the way needed to preserve area. There is a deeper structure here: the map is the composition of two exact flows, a “kick” that changes only the velocity and a “drift” that changes only the position, and each of those is itself symplectic because it is the exact solution of a Hamiltonian system (a free particle for the drift, a particle frozen in place for the kick). Composing symplectic maps gives a symplectic map. Plain Euler is not a composition of exact flows, and that is what it costs.

Euler-Cromer is fast, needing one acceleration evaluation per step, and is a reasonable choice for a quick preview. But first order is first order. Its positions drift in *phase* along the orbit at a rate proportional to Δt even when the energy is right, so a planet simulated with Euler-Cromer is on the correct ellipse at the wrong time. Something better is needed for real work.

5.4. Velocity Verlet

Go back to Equation 5.1 and keep one more term. The position expansion through Δt^2 uses only quantities we have,

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \mathbf{v}_n \Delta t + \frac{1}{2} \mathbf{a}_n \Delta t^2,$$

5. Integrators: Euler, Euler-Cromer, and Velocity Verlet

with local error $O(\Delta t^3)$. For the velocity, the troublesome term is $\frac{1}{2}\dot{\mathbf{a}}\Delta t^2$. We cannot compute $\dot{\mathbf{a}}$ directly, but having just computed $\mathbf{x}_{n+1}$, we can compute $\mathbf{a}_{n+1} = \mathbf{a}(\mathbf{x}_{n+1})$, and then approximate the derivative by the difference quotient $\dot{\mathbf{a}} \approx (\mathbf{a}_{n+1} - \mathbf{a}_n)/\Delta t$. Substituting,

$$\mathbf{v}_{n+1} = \mathbf{v}_n + \mathbf{a}_n \Delta t + \frac{1}{2}(\mathbf{a}_{n+1} - \mathbf{a}_n)\Delta t = \mathbf{v}_n + \frac{1}{2}(\mathbf{a}_n + \mathbf{a}_{n+1})\Delta t.$$

Together:

$$\begin{aligned}\mathbf{x}_{n+1} &= \mathbf{x}_n + \mathbf{v}_n \Delta t + \frac{1}{2} \mathbf{a}_n \Delta t^2, \\ \mathbf{a}_{n+1} &= \mathbf{a}(\mathbf{x}_{n+1}), \\ \mathbf{v}_{n+1} &= \mathbf{v}_n + \frac{1}{2} (\mathbf{a}_n + \mathbf{a}_{n+1}) \Delta t.\end{aligned}\tag{5.6}$$

This is **Velocity Verlet**, method = "verlet". Position first, then the new acceleration, then the velocity using the average of the old and new accelerations, which is exactly the recipe in the package documentation. It is second order: halving the step divides the error by four.

Second order is nice, but it is not why the method is the default. Redo the oscillator one more time. With $\mathbf{a} = -\omega^2 x$, the position update is $x_{n+1} = (1 - \frac{1}{2}\omega^2 \Delta t^2)x_n + \Delta t v_n$, and substituting into the velocity update gives, after a few lines of algebra,

$$M_{\text{VV}} = \begin{pmatrix} 1 - \frac{1}{2}\omega^2 \Delta t^2 & \Delta t \\ -\omega^2 \Delta t (1 - \frac{1}{4}\omega^2 \Delta t^2) & 1 - \frac{1}{2}\omega^2 \Delta t^2 \end{pmatrix}, \quad \det M_{\text{VV}} = 1.\tag{5.7}$$

(Multiply out the determinant: the $\omega^2 \Delta t^2$ and $\omega^4 \Delta t^4$ terms cancel exactly.) Velocity Verlet is symplectic. It is also *time-reversible*: if you take a step forward and then a step backward with the velocity reversed, you land exactly where you started, which is a symmetry the true equations of motion have and Euler and Euler-Cromer do not. Symplectic, time-reversible, and second order, at the cost of a second acceleration evaluation per step. (orbitr computes $\mathbf{a}_n$ at the top of each step and $\mathbf{a}_{n+1}$ after the position update, so a Verlet step costs two force evaluations to Euler-Cromer's one. Since $\mathbf{a}_{n+1}$ is the next step's $\mathbf{a}_n$, an implementation that kept it would pay for only one; Chapter 15 comes back to this.) For gravitational dynamics, that combination is very hard to beat.

5.4.1. Three names for one method

Velocity Verlet has two other common forms, and recognizing them will save you confusion when reading other codes.

The Størmer, or position, form. Write the Taylor expansion of $\mathbf{x}$ both forward and backward in time and add them. The odd powers of Δt cancel:

$$\mathbf{x}(t + \Delta t) + \mathbf{x}(t - \Delta t) = 2\mathbf{x}(t) + \mathbf{a}(t)\Delta t^2 + O(\Delta t^4),$$

so

$$\mathbf{x}_{n+1} = 2\mathbf{x}_n - \mathbf{x}_{n-1} + \mathbf{a}_n \Delta t^2. \quad (5.8)$$

No velocities appear at all, and the local error in position is $O(\Delta t^4)$, one order better than the derivation of Equation 5.6 suggested. Subtracting consecutive position updates of Equation 5.6 and using the velocity update to eliminate $\mathbf{v}$ (a few lines of algebra worth doing once) shows that velocity Verlet produces exactly the same sequence of positions as Equation 5.8. Velocity Verlet is just the Störmer form with the velocity carried along explicitly, which is convenient because you usually want the velocity.

The leapfrog, or kick-drift-kick, form. Split the velocity update of Equation 5.6 in half, doing the first half before the position update and the second half after:

$$\mathbf{v}_{n+\frac{1}{2}} = \mathbf{v}_n + \frac{1}{2}\mathbf{a}_n \Delta t, \quad \mathbf{x}_{n+1} = \mathbf{x}_n + \mathbf{v}_{n+\frac{1}{2}} \Delta t, \quad \mathbf{v}_{n+1} = \mathbf{v}_{n+\frac{1}{2}} + \frac{1}{2}\mathbf{a}_{n+1} \Delta t. \quad (5.9)$$

Substitute the first line into the second and you recover the position update of Equation 5.6; substitute it into the third and you recover the velocity update. The three forms are algebraically identical. The leapfrog form makes the kick-drift-kick structure explicit, and like Euler-Cromer it is a composition of exact flows, which is the structural reason it is symplectic. Astrophysicists usually say “leapfrog”; chemists say “velocity Verlet”; numerical analysts say “Störmer-Verlet.” They are all talking about Equation 5.6.

Where it comes from

The method has been discovered at least four times, by people working on completely different problems.

The first was Newton. Proposition 1 of Book I of the *Principia* (Newton 1687) proves Kepler’s area law by imagining the central force acting as a sequence of discrete impulses: the body drifts in a straight line, receives a kick toward the center, drifts again. Newton then lets the interval between kicks go to zero. But stop before the limit and what you have is the leapfrog scheme, exactly. Hairer, Lubich, and Wanner make this point in their survey of the method (Hairer et al. 2003), and it is a satisfying one: the very first proof about orbits under gravity was, in effect, a symplectic integrator.

The second was Carl Størmer, a Norwegian mathematician who in 1907 wanted to understand the aurora (Størmer 1907). He computed the paths of charged particles in Earth’s magnetic field, by hand, with a team of assistants, using the position form Equation 5.8. The name “Størmer’s method” (usually spelled Störmer in the numerical-analysis literature) stuck.

The third was Loup Verlet, in 1967 (Verlet 1967), simulating liquid argon as a few hundred Lennard-Jones atoms, one of the first molecular dynamics simulations. He needed a method that would keep the total energy of the fluid constant over long runs, found the position form, and the chemistry community has called it the Verlet algorithm ever since. The velocity form Equation 5.6 was written down by Swope, Andersen, Berens, and Wilson in 1982 (Swope et al. 1982), in a paper about water clusters.

Somewhere in between, Richard Feynman taught the method to Caltech freshmen. Chapter 9 of the first volume of the *Lectures on Physics* (Feynman et al. 1963) integrates a planetary orbit by hand, step by step, with the velocities evaluated at the half-steps, which is the leapfrog form. He does not give it a name.

5. Integrators: Euler, Euler-Cromer, and Velocity Verlet

The reason the same method keeps turning up is the reason it is orbitr's default: anyone who needs long-time stability in a conservative system, and tries the obvious thing first, ends up here.

5.4.2. What “conserves energy” really means

Velocity Verlet does *not* conserve energy exactly. No method with a fixed step can, for a nonlinear force. What it does is better than it sounds. A symplectic integrator applied to a Hamiltonian H turns out to be the *exact* solution of a slightly different Hamiltonian, $\tilde{H} = H + \Delta t^2 H_2 + \Delta t^4 H_4 + \dots$, called the modified or shadow Hamiltonian. The integrator conserves $\tilde{H}$ exactly, step after step (more precisely, over exponentially long times), so the true energy H can only wander as far from its starting value as $\tilde{H}$ differs from H , which is $O(\Delta t^2)$. The energy error oscillates within a band and never leaves it. This is called backward error analysis, and Hairer, Lubich, and Wanner's book is the standard reference (Hairer et al. 2006).

Euler has no shadow Hamiltonian, because it is not symplectic, and so its energy error is free to grow without bound. The difference between a bounded oscillation and a secular drift is the difference between a planet that is still there after a million years and one that is not.

5.5. Seeing it in orbitr

Here is the vignette's comparison. A star and a planet, launched at 30 km/s from 10^{11} m, which is an ellipse with $e \approx 0.35$ and a period of about 535 days. To make the differences obvious I use a time step of six hours rather than the vignette's one hour; with hourly steps the same things happen, more slowly.

```
star_planet <- create_system() |>
  add_body("Star", mass = 1e30) |>
  add_body("Planet", mass = 1e24, x = 1e11, vy = 30000)

run <- function(method, time_step = seconds_per_hour * 6,
  duration = seconds_per_year * 10) {
  star_planet |>
    simulate_system(time_step = time_step, duration = duration,
      method = method) |>
    mutate(method = method)
}

runs <- bind_rows(run("verlet"), run("euler_cromer"), run("euler"))

runs |>
  filter(id == "Planet") |>
  ggplot(aes(x, y, color = method)) +
```

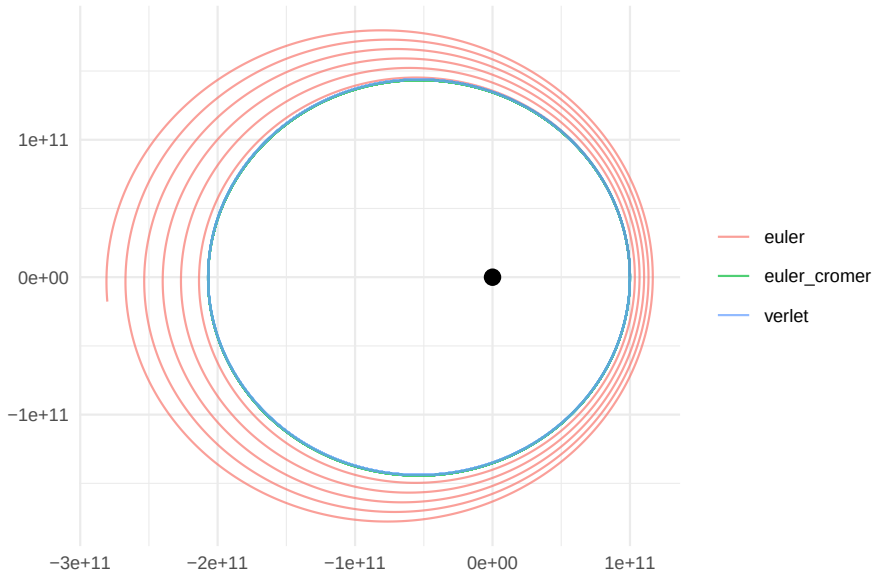


Figure 5.1.: Ten years of the same orbit with three integrators and a six-hour step. Verlet retraces one ellipse; Euler-Cromer stays on an ellipse of the right size but slips in phase; Euler spirals outward.

```
geom_path(alpha = 0.7) +
  annotate("point", x = 0, y = 0, size = 3) +
  coord_equal() +
  labs(x = NULL, y = NULL, color = NULL)
```

The picture is the one in the documentation. Now the number behind it. The total energy of any N-body system is the kinetic energy of every body plus the potential energy of every *pair*,

$$E = \sum_j \frac{1}{2} m_j v_j^2 - \sum_{j < k} \frac{G m_j m_k}{r_{jk}}, \quad (5.10)$$

and both are easy to compute from the output tibble. The potential energy needs every pair at every time step, which is a self-join on `time`:

```
total_energy <- function(sim, G = gravitational_constant) {
  kinetic <- sim |>
    group_by(time) |>
    summarise(KE = sum(0.5 * mass * (vx^2 + vy^2 + vz^2)))
```

5. Integrators: Euler, Euler-Cromer, and Velocity Verlet

```
potential <- sim |>
  select(time, id, mass, x, y, z) |>
  inner_join(sim |> select(time, id2 = id, mass2 = mass,
                        x2 = x, y2 = y, z2 = z),
            by = "time", relationship = "many-to-many") |>
  filter(id < id2) |> # each unordered pair once
  mutate(r = sqrt((x - x2)^2 + (y - y2)^2 + (z - z2)^2)) |>
  group_by(time) |>
  summarise(PE = -sum(G * mass * mass2 / r))

inner_join(kinetic, potential, by = "time") |>
  mutate(E = KE + PE,
         rel_error = (E - first(E)) / abs(first(E)))
}
```

This is what the package's `get_energy()` computes, one row per time step with `kinetic`, `potential`, and `energy` columns (and it reads the softening length from the run, so the potential matches the force law that was actually used). `conserved_quantities()` goes one step further and adds the relative error $(E - E_0)/|E_0|$ as `energy_error`, together with the total momentum and angular momentum, which Chapter 12 uses. Having seen the sum written out once, I will use those from here on.

```
energy <- runs |>
  group_by(method) |>
  group_modify(~ conserved_quantities(.x)) |>
  ungroup()

energy |>
  ggplot(aes(time / seconds_per_year, energy_error)) +
  geom_line() +
  facet_wrap(~ method, scales = "free_y", ncol = 1) +
  labs(x = "Years", y = "Relative energy error")
```

The Euler run gains tens of percent of its binding energy in ten years and would eventually escape. The two symplectic runs show the bounded oscillation that backward error analysis promises: the error rises and falls once per orbit, biggest near periapsis where the planet moves fastest and a fixed step covers the most angle, and comes back to nearly zero each time. It does not accumulate.

5.5.1. Order, measured

The last claim to check is the order. Run Euler and Verlet at four step sizes and record the worst energy error over two years:

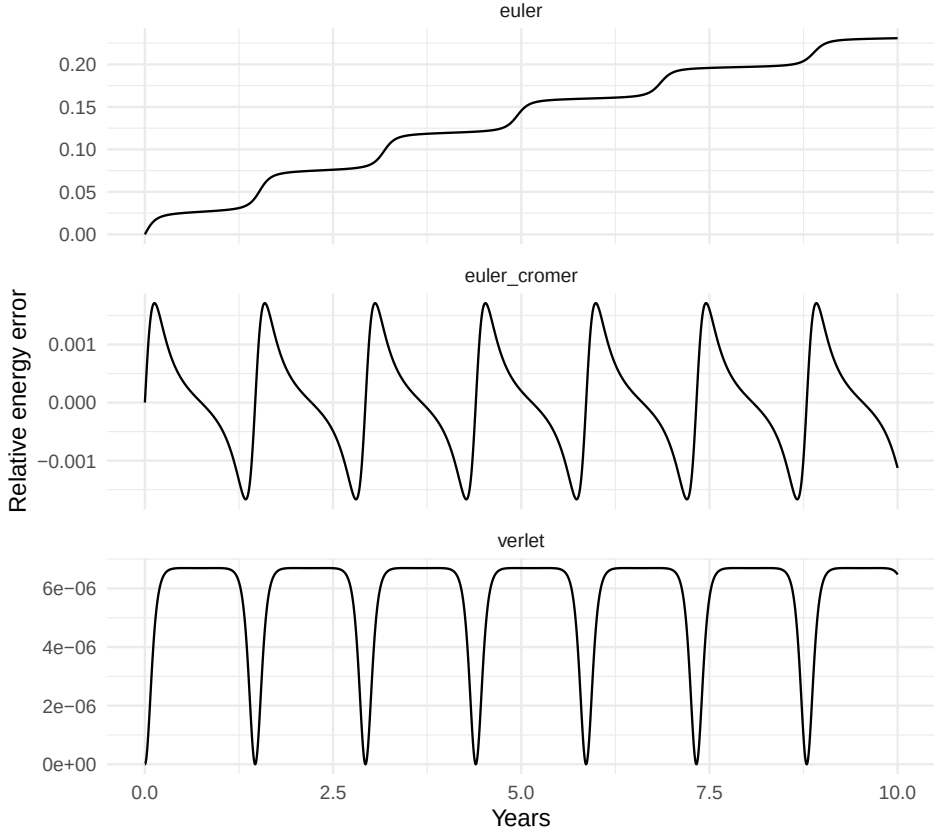


Figure 5.2.: Relative energy error $(E - E_0)/|E_0|$ for the three runs. Note the different vertical scales: Euler's error grows without limit, while the two symplectic methods oscillate within a fixed band, with spikes at each periapsis passage where the step is effectively largest.

5. Integrators: Euler, Euler-Cromer, and Velocity Verlet

```
steps <- seconds_per_day * c(1, 0.5, 0.25, 0.125)

convergence <- bind_rows(lapply(c("euler", "verlet"), function(m) {
  bind_rows(lapply(steps, function(dt) {
    star_planet |>
      simulate_system(time_step = dt, duration = seconds_per_year * 2,
                      method = m) |>
      conserved_quantities() |>
      summarise(max_rel_error = max(abs(energy_error))) |>
      mutate(method = m, step_days = dt / seconds_per_day)
  })))
}))

convergence |>
  group_by(method) |>
  mutate(ratio = lag(max_rel_error) / max_rel_error) |>
  ungroup() |>
  select(method, step_days, max_rel_error, ratio)
#> # A tibble: 8 x 4
#>   method step_days max_rel_error ratio
#>   <chr>      <dbl>          <dbl> <dbl>
#> 1 euler      1          0.238    NA
#> 2 euler      0.5          0.137    1.74
#> 3 euler      0.25          0.0736   1.86
#> 4 euler      0.125          0.0383   1.92
#> 5 verlet    1          0.000107  NA
#> 6 verlet    0.5          0.0000268 4.00
#> 7 verlet    0.25          0.00000669 4.00
#> 8 verlet    0.125          0.00000167 4.00
```

Each halving of the step cuts Euler’s error by about two and Verlet’s by about four: first order and second order, as derived. The ratios are not exactly 2 and 4 because the error has higher-order pieces too, and because the “worst” error for Verlet is set by the periapsis passages, which a coarser step samples differently each time. They are close.

5.6. Choosing a time step

Everything above gives you the tools to choose `time_step` sensibly rather than by habit.

The fastest orbit sets the step. The error in a step depends on how much the acceleration changes during it, which is to say on how large a fraction of an orbit the step covers. In a system with the Sun, Earth, and the Moon, the Moon’s 27-day orbit, not Earth’s year, decides the step. A good rule of thumb is a few hundred to a thousand steps per orbit of the tightest orbit in the system: an hour for the Moon, a day for the planets, as the package defaults suggest.

Periapsis is where it hurts. For an eccentric orbit the body moves fastest at periapsis, and the angular rate there is h/r_p^2 , larger than the mean rate by a factor of roughly $(1+e)/(1-e)$

when e is substantial. A comet with $e = 0.97$ moves through periapsis about sixty times faster than its average, and a step that is fine at aphelion is useless at perihelion. Chapter 13 deals with this.

Stability is not accuracy. For the oscillator, the Verlet map is stable (its eigenvalues stay on the unit circle) as long as $\omega\Delta t < 2$, which is only about three steps per orbit. Beyond that the solution blows up; short of it, the solution is bounded but may be badly wrong. Do not take “it didn’t blow up” as evidence that the step was small enough.

Test by halving. The convergence table is the test. Run the simulation, halve `time_step`, run it again, and compare the quantities you care about: final positions, periods, the energy band. If the answers move by less than you care about, the step was fine. If they move a lot, halve again. This costs a factor of two or three in compute time once and saves you from publishing a numerical artifact.

Key results

- Euler: first order, increases phase-space area every step, energy grows without bound. Educational only.
- Euler-Cromer: first order, symplectic. Energy error bounded, phase error $O(\Delta t)$. Fast previews.
- Velocity Verlet (= Störmer = leapfrog): second order, symplectic, time-reversible. Energy error bounded and $O(\Delta t^2)$. Use it.
- Total energy of an N-body system: $E = \sum_j \frac{1}{2} m_j v_j^2 - \sum_{j < k} G m_j m_k / r_{jk}$.
- Choose the step from the fastest orbit in the system, check it by halving.

6. Softening: What Happens When Bodies Get Close

The inverse-square force has a hole in it. As two point masses approach, $1/r^2$ grows without limit, and at $r = 0$ it is infinite. Nothing in the real universe does this, because nothing in the real universe is a point: two stars that get close enough touch, and from then on gravity is the least of their problems. But the simulation does not know that. `simulate_system()` has an argument, `softening`, whose entire purpose is to paper over the hole. This chapter explains what the argument does to the force law, where the formula comes from, what it costs you when bodies are far apart, and when you should use it. The short version: softening is a model of bodies having size, not a fix for a time step that is too large, and it is off by default for a reason.

6.1. The singularity, numerically

An integrator with a fixed step assumes the acceleration does not change much during a step. Near a close encounter that assumption fails completely. The characteristic time of the encounter is roughly the periapsis distance divided by the periapsis speed, and for a deep encounter that can be minutes while the step is an hour. The integrator then takes one enormous kick in a direction computed from a position that is already wrong, and the result is two bodies flying apart with far more energy than they came in with. Nothing blows up to NaN; it just produces garbage that looks like physics.

Here is a deliberately bad encounter: two Sun-mass stars approaching each other on a path that would bring them within about two-thirds of a solar radius, which for real stars is a collision.

```
M <- 1e30
mu <- gravitational_constant * 2 * M
u <- 1.2e5      # initial relative speed, m/s
b <- 3e9        # initial sideways offset, m

encounter <- function(softening = 0, time_step = 600,
                      duration = 4 * seconds_per_day) {
  create_system() |>
    add_body("A", mass = M, x = -1e10, y = -b / 2, vx = u / 2) |>
    add_body("B", mass = M, x = 1e10, y = b / 2, vx = -u / 2) |>
    simulate_system(time_step = time_step, duration = duration,
                    softening = softening)
}
```

6. Softening: What Happens When Bodies Get Close

Before running anything, Chapter 4 tells us what should happen. From the initial relative state we can get the eccentricity, and from the eccentricity the periapsis distance and the angle through which the stars should be deflected. For a hyperbola, each asymptote makes an angle $\arccos(-1/e)$ with the periapsis direction, so the total deflection is $\delta = 2 \arcsin(1/e)$:

```
start <- encounter(duration = 600)           # one step: just the initial state
e0     <- get_orbital_elements(start, "B", "A", mu = mu)$e[1]   # mu = G(M_A + M_B)
r_p    <- (b * u)^2 / mu / (1 + e0)         # p / (1 + e), with h = b u

c(e = e0, periapsis_solar_radii = r_p / 6.957e8,
  deflection_deg = 2 * asin(1 / e0) * 180 / pi)
#>                e periapsis_solar_radii      deflection_deg
#>                1.0043512          0.6962632      169.3294587
```

The stars should swing around each other through about 169 degrees, nearly reversing direction, and leave with exactly the eccentricity and energy they arrived with. Now run it with a ten-minute step and no softening:

```
bad <- encounter(softening = 0, time_step = 600)

c(e_start = get_orbital_elements(bad, "B", "A", mu = mu)$e[1],
  e_end    = get_orbital_elements(bad, "B", "A", mu = mu)$e[nrow(bad) / 2],
  max_energy_error = max(abs(conserved_quantities(bad)$energy_error)))
#>                e_start          e_end max_energy_error
#>                1.004351      1.005115      21.463194
```

The eccentricity, which should be a constant of the motion, is not even close, and the energy error is enormous. The time step was far too large for the periapsis passage, and the integrator did what integrators do with too large a step: it invented energy. The paths look like a slingshot because that is what a huge spurious kick looks like.

6.2. Plummer softening

The package's remedy replaces the distance in the force law:

$$r \longrightarrow r_{\text{soft}} = \sqrt{r^2 + \varepsilon^2},$$

so the acceleration of body j due to body k becomes

$$\mathbf{a}_{jk} = Gm_k \frac{\mathbf{r}_k - \mathbf{r}_j}{(r_{jk}^2 + \varepsilon^2)^{3/2}}. \quad (6.1)$$

The difference vector in the numerator is untouched; only the r^3 in the denominator is softened. At $r = 0$ the acceleration is now zero, not infinite, and its maximum magnitude, which occurs at $r = \varepsilon/\sqrt{2}$, is about $0.38 Gm_k/\varepsilon^2$.

Where it comes from

In 1911 H. C. Plummer was fitting the observed brightness profiles of globular clusters and found that a density law of the form $\rho(r) \propto (1 + r^2/\varepsilon^2)^{-5/2}$ matched them well (Plummer 1911). The gravitational potential of that density distribution works out to

$$\Phi(r) = -\frac{GM}{\sqrt{r^2 + \varepsilon^2}},$$

a potential that looks exactly like a point mass at large r and flattens out smoothly to a finite value at the center. Differentiating, the acceleration is $-d\Phi/dr = -GM r/(r^2 + \varepsilon^2)^{3/2}$, which is Equation 6.1. So softening is not an ad hoc patch: it replaces each point mass by a small Plummer sphere of scale radius ε , and the resulting force is still exactly the gradient of a potential, which means the softened system still conserves energy (its own softened energy) and angular momentum. When N-body simulation of star clusters began in earnest in the 1960s, Aarseth and others adopted the Plummer form because it was the simplest potential that was both finite at the origin and physically interpretable (Aarseth 2003), and it has been the default ever since.

Figure 6.1 shows what the softened force looks like. Inside ε it falls to zero; outside a few ε it is indistinguishable from the point-mass force.

```
tibble(x = seq(0.05, 5, by = 0.01)) |>
  mutate(point_mass = 1 / x^2,
         softened    = x / (1 + x^2)^1.5) |>
  tidyr::pivot_longer(-x, names_to = "law", values_to = "a") |>
  ggplot(aes(x, a, linetype = law)) +
  geom_line() +
  coord_cartesian(ylim = c(0, 2)) +
  labs(x = expression(r / epsilon), y = "Acceleration", linetype = NULL)
```

6.3. What softening costs

The softened force is weaker than the true force at *every* distance, not just inside ε . The ratio is

$$\frac{a_{\text{soft}}}{a} = \left(1 + \frac{\varepsilon^2}{r^2}\right)^{-3/2} \approx 1 - \frac{3}{2} \frac{\varepsilon^2}{r^2} \quad (r \gg \varepsilon). \quad (6.2)$$

For a circular orbit the period goes as the inverse square root of the force, so softening lengthens every period by a fraction of about $\frac{3}{4} \varepsilon^2/r^2$. Put numbers in. The documentation suggests **softening** = **1e4** (10 km) as reasonable for planetary systems; at Earth's distance from the Sun, $\varepsilon/r \approx 7 \times 10^{-8}$ and the period changes by one part in 10^{14} . You could not detect that with a million years of simulation. At $\varepsilon = 10^9$ m the change at 1 AU is 3×10^{-5} , about a quarter of an hour per year, which starts to matter for precise work. At $\varepsilon = 10^{10}$ m, a bit under a tenth of an AU, Earth's year would be about three days too long.

6. Softening: What Happens When Bodies Get Close

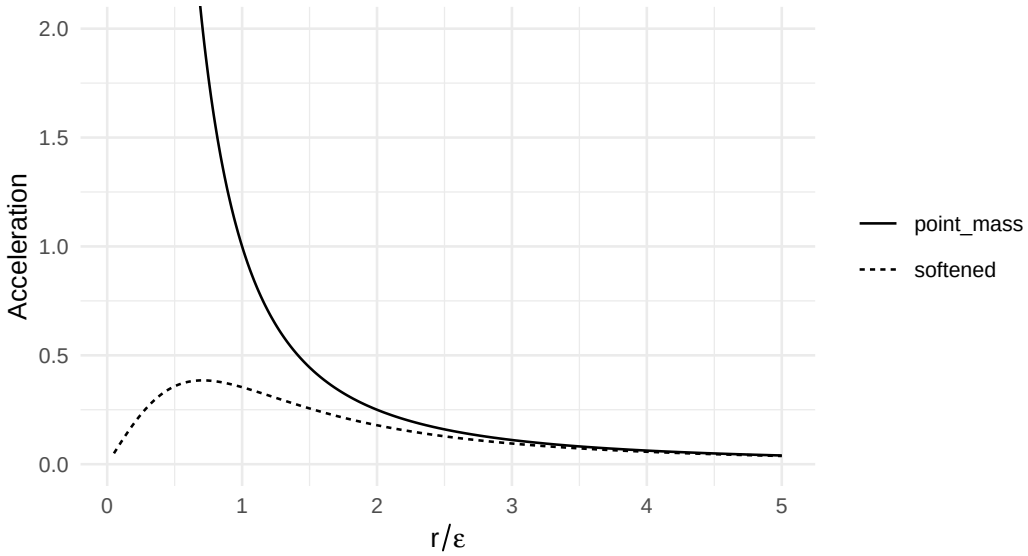


Figure 6.1.: The softened acceleration (solid) compared with the point-mass $1/r^2$ law (dashed), in units where $Gm/\varepsilon^2 = 1$. Beyond $r \approx 3\varepsilon$ the two agree to a few percent.

The lesson is that softening is cheap as long as ε is much smaller than every orbital separation you care about, and it is never free. With `softening = 0`, the default, there is no error of this kind at all, and for planetary systems, where nothing ever gets close to anything, the default is the right choice.

6.4. The encounter, softened

Run the same encounter with the same ten-minute step and a softening length of 3×10^9 m, about four solar radii, which is a crude way of saying “these stars have size”:

```
soft <- encounter(softening = 3e9, time_step = 600)

c(e_start = get_orbital_elements(soft, "B", "A", mu = mu)$e[1],
  e_end    = get_orbital_elements(soft, "B", "A", mu = mu)$e[nrow(soft) / 2],
  max_energy_error = max(abs(conserved_quantities(soft)$energy_error)))
#>          e_start          e_end max_energy_error
#>      1.0043512      1.0047221      0.0140379
```

Two things changed. First, the energy is now conserved to a fraction of a percent, because the softened force varies slowly enough during the encounter for a ten-minute step to follow it. Second, the eccentricity after the encounter is different from the eccentricity before, *and that is correct*: the stars did not interact through a point-mass force, so they are not on the point-mass hyperbola. Softening changed the physics, which is what it is for. The energy diagnostic needs the same softening, so that it computes the potential energy of the force that was actually used. `conserved_quantities()` reads the softening length from

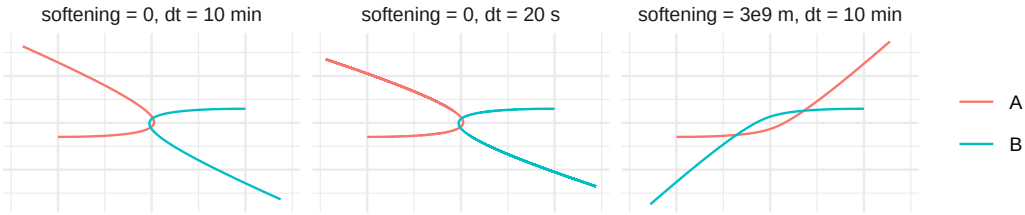


Figure 6.2.: The same two stars in three runs. Left: unsoftened, ten-minute step, a spurious kick at periaapsis. Center: softened to 3×10^9 m, same step, a smooth but physically different encounter. Right: unsoftened, twenty-second step, the true point-mass deflection.

the simulation output and replaces r with $\sqrt{r^2 + \varepsilon^2}$ in the potential automatically; if you compute the energy yourself, you have to do the same.

The third run is the honest one: no softening, and a step small enough for the point-mass physics.

```
fine <- encounter(softening = 0, time_step = 20)

c(e_start = get_orbital_elements(fine, "B", "A", mu = mu)$e[1],
  e_end   = get_orbital_elements(fine, "B", "A", mu = mu)$e[nrow(fine) / 2],
  max_energy_error = max(abs(conserved_quantities(fine)$energy_error)))
#>          e_start          e_end max_energy_error
#>      1.0043512      1.0043512      0.0270577
```

Now the eccentricity survives the encounter and the energy error is small. That cost thirty times as many steps as the softened run, and it is the only one of the three that answers the question “what would two point masses do?”

```
bind_rows(bad |> mutate(run = "softening = 0, dt = 10 min"),
  soft |> mutate(run = "softening = 3e9 m, dt = 10 min"),
  fine |> mutate(run = "softening = 0, dt = 20 s")) |>
  ggplot(aes(x, y, color = id)) +
  geom_path() +
  coord_equal() +
  facet_wrap(~ run) +
  labs(x = NULL, y = NULL, color = NULL) +
  theme(axis.text = element_blank())
```

6.5. When to use it

Use softening when bodies have size and you care about what happens when they meet. Stars in a cluster, a planetesimal disk, anything with many bodies and frequent close passages. Set ε to something like the physical size of the bodies, or to a fraction of the typical spacing, and accept that encounters closer than ε are not being modeled correctly, because they would not be point-mass encounters in reality either. In this regime a modest softening also lets you use a larger time step than the closest passage would otherwise demand, which for a thousand-body system is the difference between a run that finishes and one that does not.

Do not use softening to rescue a bad time step in a few-body system. If two planets, or a comet and the Sun, pass close and the energy jumps, the cure is a smaller step (Chapter 13 shows how to afford one), not a softened force that changes the answer. The only exception is when you have decided that the close approach is a collision and you do not care about the trajectory afterward; then softening is a cheap way to keep the simulation from flinging things to infinity.

Keep ε far below the smallest orbit. Equation 6.2 is the rule: the relative error in the force is about $\frac{3}{2}(\varepsilon/r)^2$, so a softening of one percent of the smallest orbital radius changes periods by about one part in ten thousand. Decide what precision you need and work backward.

Account for it in diagnostics. The softened system conserves the softened energy. `get_energy()` and `conserved_quantities()` take the softening length from the run; if you hand-compute energies, use the softened distance, or you will diagnose a drift that is not there.

Key results

- Softening replaces r^3 in the force denominator with $(r^2 + \varepsilon^2)^{3/2}$. The force is finite everywhere and zero at $r = 0$.
- It is the force of a Plummer sphere of scale radius ε : still conservative, still central.
- Relative error in the force at distance $r \gg \varepsilon$: about $\frac{3}{2}(\varepsilon/r)^2$. Periods lengthen by about $\frac{3}{4}(\varepsilon/r)^2$.
- Softening models bodies having size. It does not fix a time step that is too large for the physics you asked for.

7. From Orbital Elements to State Vectors

`add_body()` wants six numbers: three for position, three for velocity. Astronomers describe orbits with six different numbers, the classical orbital elements, and `add_body_keplerian()` and `add_planet()` translate between the two. Chapter 4 ended with a reason to care: the four-line Moon was on the wrong orbit because a distance and a speed, chosen separately, do not describe the real ellipse. The elements do. This chapter derives the translation, implements it in plain R, checks it against the package, derives the reverse translation, and then uses it to do two things the package does not do on its own: put a planet where it was on a given date, and watch an orbit's elements drift under the pull of a third body.

7.1. Six numbers

A two-body orbit is an ellipse in a plane. Two numbers fix the ellipse's size and shape, three fix the plane's orientation in space and the ellipse's orientation within it, and one says where along the ellipse the body is right now.

- **Semi-major axis, a .** The size. Half the longest diameter, in meters. Sets the period through $T = 2\pi\sqrt{a^3/\mu}$.
- **Eccentricity, e .** The shape. 0 is a circle, values approaching 1 are increasingly stretched.
- **Inclination, i .** The tilt of the orbital plane relative to the reference plane (for the solar system, the ecliptic, the plane of Earth's orbit). 0° is flat; 90° is a polar orbit.
- **Longitude of the ascending node, Ω (`lan` in the package).** Where the tilt points. The orbit crosses the reference plane twice; the *ascending node* is the crossing where the body moves from below to above, and Ω is the angle from the reference x axis to that point, measured in the reference plane.
- **Argument of periapsis, ω (`arg_pe`).** The angle, measured within the orbital plane, from the ascending node to periapsis. It rotates the ellipse within its own plane.
- **True anomaly, ν (`nu`).** The angle from periapsis to the body's current position, measured in the orbital plane in the direction of motion. Chapter 4's orbit equation is written in terms of it.

All angles are in degrees in the package's arguments and in radians in this chapter's formulas. The documentation's Keplerian Elements article shows each element varied in turn, with pictures; what follows is the algebra underneath those pictures.

7.2. Position and velocity in the orbital plane

Set up coordinates in the orbital plane with the x axis pointing toward periapsis and the y axis 90° ahead in the direction of motion. This is the *perifocal* frame. In it the position is simply polar coordinates with the orbit equation Equation 4.8 for the radius:

$$r = \frac{a(1 - e^2)}{1 + e \cos \nu}, \quad \mathbf{r}_{\text{pf}} = (r \cos \nu, r \sin \nu, 0). \quad (7.1)$$

The velocity takes a short derivation. Differentiate the position:

$$\dot{x}_{\text{pf}} = \dot{r} \cos \nu - r \dot{\nu} \sin \nu, \quad \dot{y}_{\text{pf}} = \dot{r} \sin \nu + r \dot{\nu} \cos \nu.$$

We need $\dot{r}$ and $\dot{\nu}$. The angular momentum in polar coordinates is $h = r^2 \dot{\nu}$, so $\dot{\nu} = h/r^2$ and $r \dot{\nu} = h/r$. Using the orbit equation with $p = a(1 - e^2) = h^2/\mu$,

$$r \dot{\nu} = \frac{h}{r} = \frac{h(1 + e \cos \nu)}{p} = \frac{\mu}{h}(1 + e \cos \nu).$$

For $\dot{r}$, differentiate $r = p/(1 + e \cos \nu)$ with respect to time:

$$\dot{r} = \frac{p e \sin \nu \dot{\nu}}{(1 + e \cos \nu)^2} = \frac{r^2 e \sin \nu}{p} \dot{\nu} = \frac{r^2 e \sin \nu}{p} \cdot \frac{h}{r^2} = \frac{h e}{p} \sin \nu = \frac{\mu}{h} e \sin \nu.$$

Substituting both into the velocity components and simplifying (the $e \sin \nu \cos \nu$ terms cancel in $\dot{x}$, and $\sin^2 \nu + \cos^2 \nu = 1$ collapses $\dot{y}$):

$$\dot{x}_{\text{pf}} = -\frac{\mu}{h} \sin \nu, \quad \dot{y}_{\text{pf}} = \frac{\mu}{h} (e + \cos \nu), \quad h = \sqrt{\mu a(1 - e^2)}. \quad (7.2)$$

These are the formulas in the package documentation. They have a nice interpretation: in the perifocal frame the velocity vector traces a *circle* of radius μ/h centered at $(0, \mu e/h)$ as the body goes around. That is Hamilton's hodograph theorem, and it is another way of saying the eccentricity vector is conserved.

At periapsis, $\nu = 0$ and the speed is $(\mu/h)(1 + e)$; at apoapsis, $\nu = \pi$ and it is $(\mu/h)(1 - e)$. Their ratio is $(1 + e)/(1 - e)$, which for Halley's Comet ($e = 0.967$) is about 60. Chapter 5's warning about periapsis is this ratio.

7.3. Three rotations

The perifocal frame is tied to the orbit. The simulation frame is tied to the reference plane. Getting from one to the other takes three rotations, applied to both the position and the velocity vectors, and the only hard part is keeping the order and the signs straight.

Start with the orbit lying in the reference plane with periapsis along $+x$; that is the perifocal frame itself. Then:

1. Rotate about the z axis by ω . Periapsis moves away from the x axis; the x axis is now where the ascending node will be.
2. Rotate about the (new) x axis by i . The orbital plane tilts out of the reference plane, hinged on the node line.
3. Rotate about the (original) z axis by Ω . The node line swings around to its proper direction.

With the standard rotation matrices

$$R_z(\theta) = \begin{pmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{pmatrix}, \quad R_x(\theta) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta \\ 0 & \sin \theta & \cos \theta \end{pmatrix},$$

the combined transformation is

$$\mathbf{r} = R_z(\Omega) R_x(i) R_z(\omega) \mathbf{r}_{\text{pf}}, \quad \mathbf{v} = R_z(\Omega) R_x(i) R_z(\omega) \mathbf{v}_{\text{pf}}. \quad (7.3)$$

Matrices act right to left, so $R_z(\omega)$ is applied first, which matches the list above. The package documentation writes the same product as $R_z(-\Omega)R_x(-i)R_z(-\omega)$; the sign difference is a convention about whether a rotation matrix rotates the vector or the coordinate axes, and the two descriptions produce the same state vector. The code below is the test of that claim.

Finally, the elements describe the orbit *relative to the parent*, so the parent's own position and velocity are added at the end. That is why `parent` must already be in the system, and it is what lets `add_planet("Moon", parent = "Earth")` work when Earth is itself moving around the Sun.

7.4. Implementing it

```
rot_x <- function(theta) {
  co <- cos(theta); si <- sin(theta)
  matrix(c(1, 0, 0, 0, co, si, 0, -si, co), nrow = 3)
}
rot_z <- function(theta) {
  co <- cos(theta); si <- sin(theta)
  matrix(c(co, si, 0, -si, co, 0, 0, 0, 1), nrow = 3)
}
```

7. From Orbital Elements to State Vectors

```
keplerian_to_state <- function(a, e = 0, i = 0, lan = 0, arg_pe = 0,
                              nu = 0, mu) {
  rad <- pi / 180
  i <- i * rad; lan <- lan * rad; w <- arg_pe * rad; nu <- nu * rad

  p <- a * (1 - e^2)
  h <- sqrt(mu * p)
  r <- p / (1 + e * cos(nu))

  r_pf <- c(r * cos(nu), r * sin(nu), 0)
  v_pf <- c(-mu / h * sin(nu), mu / h * (e + cos(nu)), 0)

  Q <- rot_z(lan) %*% rot_x(i) %*% rot_z(w)
  r_in <- as.vector(Q %*% r_pf)
  v_in <- as.vector(Q %*% v_pf)

  tibble(x = r_in[1], y = r_in[2], z = r_in[3],
         vx = v_in[1], vy = v_in[2], vz = v_in[3])
}
```

(matrix() fills column by column, which is why the entries in rot_x() and rot_z() are listed one column at a time.) Now a test against the package, using Mars with all six elements non-trivial:

```
mars <- list(a = distance_mars_sun, e = 0.0934, i = 1.85,
            lan = 49.6, arg_pe = 286.5, nu = 120)

from_package <- create_system() |>
  add_sun() |>
  add_body_keplerian("Mars", mass = mass_mars, parent = "Sun",
                    a = mars$a, e = mars$e, i = mars$i,
                    lan = mars$lan, arg_pe = mars$arg_pe, nu = mars$nu) |>
  get_bodies() |>
  filter(id == "Mars") |>
  select(x, y, z, vx, vy, vz)

by_hand <- keplerian_to_state(a = mars$a, e = mars$e, i = mars$i,
                             lan = mars$lan, arg_pe = mars$arg_pe,
                             nu = mars$nu,
                             mu = gravitational_constant * mass_sun)

bind_rows(package = from_package, by_hand = by_hand, .id = "source")
#> # A tibble: 2 x 7
#>   source      x      y      z    vx    vy    vz
#>   <chr>    <dbl>    <dbl>    <dbl>  <dbl> <dbl> <dbl>
#> 1 package -25114100376. 235578953124. 5549391368. -23180. -512. 559.
#> 2 by_hand -25114100376. 235578953124. 5549391368. -23180. -512. 559.
```

```
max(abs(as.numeric(from_package) - as.numeric(by_hand)) /
     abs(as.numeric(by_hand)))
#> [1] 3.553888e-15
```

The two agree to floating-point precision, which settles the sign convention. One detail worth noticing: the gravitational parameter I passed is $\mu = GM_\odot$, the parent's mass alone, which is what `add_body_keplerian()` uses. For Mars the difference from $G(M_\odot + m_{\text{Mars}})$ is three parts in ten million. For the Moon it is not negligible, and we will come back to it.

7.5. The inverse problem

Going the other way, from a state vector to elements, is the step that tells you what orbit a simulated body is *actually* on, and every piece of it was derived in Chapter 4. Given relative position $\mathbf{r}$ and velocity $\mathbf{v}$:

- $\mathbf{h} = \mathbf{r} \times \mathbf{v}$ is normal to the orbital plane, so $\cos i = h_z/h$.
- The *node vector* $\mathbf{n} = \hat{\mathbf{z}} \times \mathbf{h}$ lies along the line of nodes, pointing to the ascending node, so $\Omega = \text{atan2}(n_y, n_x)$.
- The eccentricity vector $\mathbf{e} = (\mathbf{v} \times \mathbf{h})/\mu - \mathbf{r}/r$ points to periapsis, so ω is the angle from $\mathbf{n}$ to $\mathbf{e}$, and ν is the angle from $\mathbf{e}$ to $\mathbf{r}$, both measured in the orbital plane.
- Vis-viva, solved for a : $a = 1/(2/r - v^2/\mu)$.

The angles in the orbital plane need a sign, which comes from whether the cross product of the two vectors points along $\mathbf{h}$ or against it. Two degenerate cases need a convention: a flat orbit has no ascending node (take $\mathbf{n} = \hat{\mathbf{x}}$, so ω becomes the angle from the x axis to periapsis), and a circular orbit has no periapsis (take $\mathbf{e}$ along $\mathbf{n}$, so ν is measured from the node).

```
state_to_elements <- function(x, y, z, vx, vy, vz, mu) {
  cross <- function(a, b) c(a[2] * b[3] - a[3] * b[2],
                           a[3] * b[1] - a[1] * b[3],
                           a[1] * b[2] - a[2] * b[1])
  norm <- function(a) sqrt(sum(a^2))
  deg <- function(theta) (theta * 180 / pi) %% 360

  r <- c(x, y, z); v <- c(vx, vy, vz)
  h <- cross(r, v)
  n <- cross(c(0, 0, 1), h)
  e_vec <- cross(v, h) / mu - r / norm(r)
  e <- norm(e_vec)
  a <- 1 / (2 / norm(r) - sum(v^2) / mu)
  i <- acos(h[3] / norm(h))

  if (norm(n) < 1e-12 * norm(h)) n <- c(1, 0, 0)      # flat orbit
  e_dir <- if (e > 1e-12) e_vec / e else n / norm(n) # circular orbit
  h_hat <- h / norm(h)
```

7. From Orbital Elements to State Vectors

```

lan    <- atan2(n[2], n[1])
arg_pe <- atan2(sum(cross(n, e_dir) * h_hat), sum(n * e_dir))
nu     <- atan2(sum(cross(e_dir, r) * h_hat), sum(e_dir * r))

tibble(a = a, e = e, i = deg(i), lan = deg(lan),
        arg_pe = deg(arg_pe), nu = deg(nu))
}

```

Round trip:

```

state_to_elements(by_hand$x, by_hand$y, by_hand$z,
                  by_hand$vx, by_hand$vy, by_hand$vz,
                  mu = gravitational_constant * mass_sun)
#> # A tibble: 1 x 6
#>       a      e      i    lan arg_pe    nu
#>   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#> 1 227900000000 0.0934  1.85  49.6  286.  120.

```

Mars's elements come back. Now the question Chapter 4 left open: what orbit was the four-line Moon on?

```

state_to_elements(distance_earth_moon, 0, 0, 0, speed_moon, 0,
                  mu = gravitational_constant * (mass_earth + mass_moon))
#> # A tibble: 1 x 6
#>       a      e      i    lan arg_pe    nu
#>   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#> 1 382513625. 0.00493    0    0    180    180

```

A nearly circular orbit, starting at apoapsis ($\nu = 180^\circ$), with a semi-major axis a little under 384,400 km, exactly as Chapter 4 computed by hand.

This inverse conversion is what `get_orbital_elements()` does, for a whole simulation at once: it takes the position and velocity of a body relative to a parent at every time step and returns the six elements, plus the period. Its default μ is GM_{parent} , matching `add_body_keplerian()`, so the round trip is exact:

```

mars_check <- create_system() |>
  add_sun() |>
  add_body_keplerian("Mars", mass = mass_mars, parent = "Sun",
                    a = mars$a, e = mars$e, i = mars$i,
                    lan = mars$lan, arg_pe = mars$arg_pe, nu = mars$nu) |>
  simulate_system(time_step = seconds_per_day, duration = seconds_per_day)

get_orbital_elements(mars_check, "Mars", "Sun")[1, ]
#> # A tibble: 1 x 8
#>   time      a      e      i    lan arg_pe    nu  period
#>   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#> 1    0 227900000000. 0.0934  1.85  49.6  286.  120. 59330240.

```

The function is `state_to_elements()` with the bookkeeping done: the relative state, the join on time, the degenerate cases, and a `mu` argument when you want the two-body value. From here on the book uses it.

7.6. The Moon on its real orbit

The real Moon's orbit has $a = 384,400$ km, $e = 0.0549$, and $i = 5.15^\circ$ to the ecliptic, and `add_planet("Moon", parent = "Earth")` builds it from those elements. Earth and the Moon are the one pair in the solar system where the secondary's mass is not negligible, and the conversion uses $\mu = GM_{\text{parent}}$, so it is worth checking what orbit the package actually produces. The simulation always uses both masses, so the orbit the Moon follows is a two-body orbit with $\mu = G(M_{\oplus} + m_{\text{Moon}})$, and `get_orbital_elements()` with that μ tells us its true semi-major axis:

```
em <- create_system() |>
  add_body("Earth", mass = mass_earth) |>
  add_planet("Moon", parent = "Earth")

em_sim <- simulate_system(em, time_step = seconds_per_hour,
                          duration = seconds_per_day * 30)
mu_em <- gravitational_constant * (mass_earth + mass_moon)

actual <- get_orbital_elements(em_sim, "Moon", "Earth", mu = mu_em)[1, ]
actual
#> # A tibble: 1 x 8
#>   time      a      e      i      lan arg_pe      nu      period
#>   <dbl>    <dbl> <dbl> <dbl> <dbl> <dbl>    <dbl>    <dbl>
#> 1      0 379258886. 0.0421 5.15 125.    318. 1.01e-13 2310295.

c(a_ratio = actual$a / 3.844e8,
  T_predicted_days = 2 * pi * sqrt(actual$a^3 / mu_em) / seconds_per_day)
#>      a_ratio T_predicted_days
#>      0.9866256      26.7395241
```

The ratio is not 1. The velocity the conversion assigned is right for $\mu = GM_{\oplus}$ but slightly too small for the μ the simulation actually runs with, so the Moon settles onto a slightly smaller, faster orbit, with a period a couple of percent short of the real 27.32 days. The simulation confirms it:

```
measure_period(em_sim, "Moon", "Earth") / seconds_per_day
#> [1] 26.74042
```

If you need the Moon's period right to better than a percent, build it with `keplerian_to_state()` using the combined μ and pass the result to `add_body()`; `measure_period()` then returns the real 27.32 days. For every other body in the solar system the question does not arise: the heaviest planet is a thousandth of the Sun, and the error is a twentieth of a percent in its period.

7.7. Where is Mars today? Kepler's equation

The elements fix the orbit, and ν fixes the position on it, but ν is not what an ephemeris gives you. An ephemeris gives you the *time of perihelion passage* or, equivalently, the *mean anomaly* M at some epoch, and converting a time into a true anomaly is a problem Kepler himself posed and could not solve in closed form. Nobody can; it has to be done numerically. The package starts every body at $\nu = 0$ by default and lets you set `nu` yourself, so if you want the planets where they were on a particular date, this is the calculation you have to do.

Define the mean anomaly as the angle a body would have moved through if it went around uniformly:

$$M = \frac{2\pi}{T} (t - t_p) = \sqrt{\frac{\mu}{a^3}} (t - t_p),$$

where t_p is the time of periapsis. The link between M and ν goes through a third angle, the *eccentric anomaly* E , defined geometrically: draw the circle of radius a that circumscribes the ellipse, project the body's position straight up (perpendicular to the major axis) onto that circle, and E is the angle of that projected point from the center of the ellipse. Standard ellipse geometry then gives

$$r = a(1 - e \cos E), \quad \tan \frac{\nu}{2} = \sqrt{\frac{1+e}{1-e}} \tan \frac{E}{2}. \quad (7.4)$$

The area swept from periapsis, computed in the auxiliary circle and scaled by b/a , comes out to $\frac{1}{2}ab(E - e \sin E)$. Kepler's second law says the area swept is also $\pi ab \cdot (t - t_p)/T = \frac{1}{2}ab M$. Equating them:

$$M = E - e \sin E. \quad (7.5)$$

This is **Kepler's equation**. Given M , it has to be solved for E , and since E appears both bare and inside a sine there is no algebraic solution. Newton's method does it in a handful of iterations for any $e < 1$:

```
solve_kepler <- function(M, e, tol = 1e-12) {
  E <- if (e < 0.8) M else pi          # a safe starting guess
  repeat {
    dE <- (E - e * sin(E) - M) / (1 - e * cos(E))
    E <- E - dE
    if (abs(dE) < tol) break
  }
  E
}

true_anomaly_at <- function(t_since_periapsis, a, e, mu) {
  M <- sqrt(mu / a^3) * t_since_periapsis
  E <- solve_kepler(M %% (2 * pi), e)
```

7.7. Where is Mars today? Kepler's equation

```
nu <- 2 * atan2(sqrt(1 + e) * sin(E / 2), sqrt(1 - e) * cos(E / 2))
(nu * 180 / pi) %% 360
}
```

Where it comes from

Kepler stated the equation in *Astronomia Nova* (Kepler 1609) and described solving it by guesswork, apologizing to the reader for not having anything better. Newton, in the *Principia*, gave a geometric iteration for it that is essentially the one coded above; the method now named for him was first published for this very problem. The hunt for closed-form or series solutions occupied mathematicians for two centuries and produced, among other things, the Bessel functions, which Bessel introduced in 1824 to write E as a Fourier series in M . In practice everyone just iterates.

Check it against the simulation. Mars starts at perihelion, so $t_p = 0$; where should it be 100 days later, and where is it?

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

mu_sun <- gravitational_constant * mass_sun
predicted <- true_anomaly_at(100 * seconds_per_day,
                             a = distance_mars_sun, e = 0.0934, mu = mu_sun)

simulated <- get_orbital_elements(mars_sim, "Mars", "Sun") |>
  filter(time == max(time)) |>
  pull(nu)

c(predicted_nu = predicted, simulated_nu = simulated,
  mean_anomaly = (sqrt(mu_sun / distance_mars_sun^3) * 100 *
    seconds_per_day) * 180 / pi)
#> predicted_nu simulated_nu mean_anomaly
#>      61.51723      61.53228      52.42521
```

The true anomaly runs ahead of the mean anomaly by several degrees, because Mars moves fastest near perihelion, and the simulation agrees with Kepler's equation to the precision of the daily step. To place a planet on a given date, then, you need its elements and its time of perihelion passage from an ephemeris, compute ν with `true_anomaly_at()`, and pass it as `nu`. Appendix D notes that the package's elements are at the J2000 epoch; for the outer planets, whose elements change slowly, that is good for decades, and for everything the perihelion time is the one extra number you need.

7.8. Osculating elements

In a two-body system the elements are constants. In any other system they are not, and the elements `get_orbital_elements()` reports at some instant describe the ellipse the body *would* follow if every other perturbation switched off at that instant. These are called *osculating* elements (from the Latin for “kissing”: the ellipse that touches the true path at that point), and watching them change is how astronomers describe perturbations.

Here is Earth with and without Jupiter, over 24 years:

```
earth_alone <- create_system() |>
  add_sun() |>
  add_planet("Earth", parent = "Sun") |>
  simulate_system(time_step = seconds_per_day, duration = seconds_per_year * 24)

earth_jupiter <- 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 * 24)

bind_rows(
  get_orbital_elements(earth_alone, "Earth", "Sun") |> mutate(system = "Sun + Earth"),
  get_orbital_elements(earth_jupiter, "Earth", "Sun") |> mutate(system = "Sun + Earth + 
) |>
  ggplot(aes(time / seconds_per_year, e, color = system)) +
  geom_line() +
  labs(x = "Years", y = "Osculating eccentricity", color = NULL)
```

The changes are small, of order 10^{-4} , and they are real: Jupiter’s pull on Earth is a few parts in 10^5 of the Sun’s, and over the long term it drives a slow oscillation of Earth’s eccentricity with a period of about 100,000 years that is one of the Milankovitch cycles behind the ice ages. A 24-year run shows only the fast part. The two-body description does not fail here; it just stops being exact, and the elements become a *description* of the motion rather than a prediction of it. The package’s `add_planet()` elements are the osculating elements at J2000, and the simulation takes it from there.

Key results

- Perifocal position: $r = a(1 - e^2)/(1 + e \cos \nu)$, $(r \cos \nu, r \sin \nu)$. Perifocal velocity: $(\mu/h)(-\sin \nu, e + \cos \nu)$ with $h = \sqrt{\mu a(1 - e^2)}$.
- Inertial state: $R_z(\Omega)R_x(i)R_z(\omega)$ applied to both, plus the parent’s state.
- Inverse: i from $\mathbf{h}$, Ω from $\hat{\mathbf{z}} \times \mathbf{h}$, ω and ν from $\mathbf{e}$, a from vis-viva.
- Time to position: $M = \sqrt{\mu/a^3}(t - t_p)$, solve $M = E - e \sin E$, then $\tan(\nu/2) = \sqrt{(1 + e)/(1 - e)} \tan(E/2)$.
- In an N-body system the elements are osculating: they describe the instantaneous ellipse and drift under perturbation.

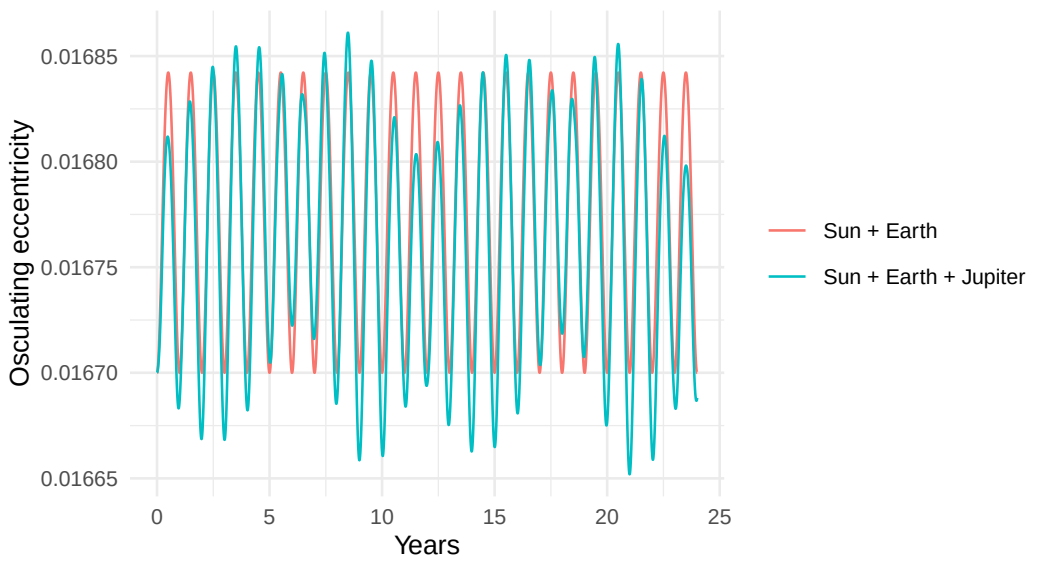


Figure 7.1.: Earth's osculating eccentricity over 24 years, alone (flat line) and with Jupiter in the system. The wiggles repeat each time Jupiter laps Earth, every 1.09 years, on top of a slower drift.

Part III.

Doing Astrophysics with orbitr

8. The Solar System from JPL Elements

Part II built the tools. Part III uses them. This chapter takes the one system everyone knows, asks where the package's numbers for it come from, and then uses a single long run to do three pieces of real astronomy: verify Kepler's third law across all eight planets and weigh the Sun with it, measure how often Earth and Mars line up, and watch the Sun wobble under Jupiter's pull in the way that lets astronomers find planets around other stars.

8.1. One call

```
solar <- load_solar_system(moon = FALSE, pluto = FALSE) |>
  simulate_system(time_step = seconds_per_day, duration = seconds_per_year * 170)

solar |> count(id)
#> # A tibble: 9 x 2
#>   id          n
#>   <chr>    <int>
#> 1 Earth    62093
#> 2 Jupiter  62093
#> 3 Mars     62093
#> 4 Mercury  62093
#> 5 Neptune  62093
#> 6 Saturn   62093
#> 7 Sun      62093
#> 8 Uranus   62093
#> 9 Venus    62093
```

Nine bodies for 170 years at a daily step: a little over half a million rows, which takes a minute or so (Chapter 15 explains where the time goes). The run is long because Neptune's period is 165 years and I want to measure it. The Moon is left out because a daily step is far too coarse for it (Chapter 5), and Pluto because it would need another century.

Every planet starts at perihelion, which is the package's default, so the planets begin lined up along their individual perihelion directions rather than where they were on any real date. Chapter 7 showed how to fix that with Kepler's equation if you need to; for the measurements below it does not matter.

8.2. Where the numbers come from

The elements behind `add_planet()` are from JPL's DE440 planetary ephemeris (Park et al. 2021). An ephemeris is not a theory; it is a fit. JPL integrates the full equations of motion of the solar system, including general relativity, the oblateness of the Sun and planets, hundreds of asteroids, and the Moon's own rotation, and adjusts the initial conditions and masses until the model matches every observation they have: centuries of optical astrometry, decades of radar ranging to the inner planets, laser ranging to the Moon, and tracking data from every spacecraft that has flown past or orbited a planet. The result is positions good to well under a kilometer for the inner planets, and the masses in Appendix D are by-products of the same fit.

What the package takes from DE440 is the *osculating* elements of each planet at one instant, the J2000 epoch (noon on January 1, 2000, in Terrestrial Time), in the ecliptic frame of that date. Those elements describe the ellipse each planet was on at that instant. From there `orbitr` integrates point-mass Newtonian gravity, nothing else. Over the years and decades the planets' orbits stay very close to the real ones; over centuries the effects JPL includes and `orbitr` does not (relativity for Mercury, the asteroids for Mars) accumulate to differences you could detect with a telescope but not by eye.

8.3. Kepler's third law, verified

Chapter 4 derived $T = 2\pi\sqrt{a^3/\mu}$ for one planet. Kepler's version, that T^2/a^3 is the same for all planets, is the statement that μ is the same for all of them, which it is to the extent that every planet's mass is negligible against the Sun's. Let's measure it. For each planet, the osculating elements at the start give a , and `measure_period()` gives T :

```
planets <- c("Mercury", "Venus", "Earth", "Mars",
            "Jupiter", "Saturn", "Uranus", "Neptune")

kepler3 <- bind_rows(lapply(planets, function(p) {
  get_orbital_elements(solar, p, "Sun", mu = gravitational_constant * mass_sun)[1, ] |>
    mutate(planet = p,
           T_days = measure_period(solar, p, "Sun") / seconds_per_day)
})) |>
  select(planet, a, e, T_days)

kepler3
#> # A tibble: 8 x 4
#>   planet      a      e  T_days
#>   <chr>    <dbl> <dbl>   <dbl>
#> 1 Mercury 5.79e10 0.206    88.2
#> 2 Venus   1.08e11 0.00680   225.
#> 3 Earth   1.50e11 0.0167   365.
#> 4 Mars    2.28e11 0.0934   687.
#> 5 Jupiter 7.78e11 0.0489  4329.
#> 6 Saturn  1.43e12 0.0565 10786.
```

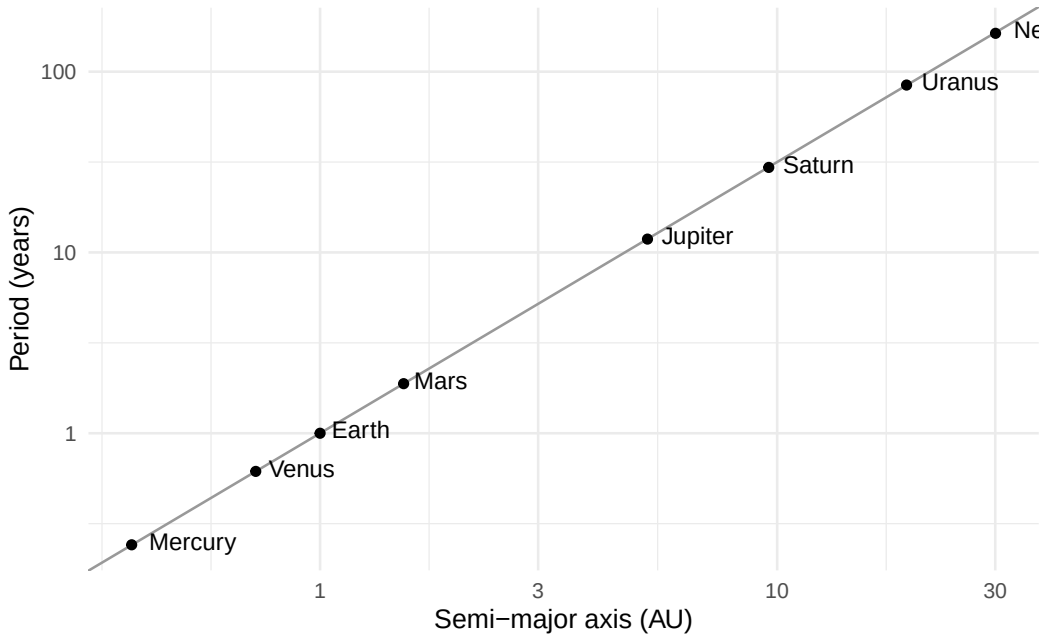


Figure 8.1.: Orbital period against semi-major axis for the eight planets, from the simulation. The line has slope exactly $3/2$.

```
#> 7 Uranus 2.87e12 0.0457 30795.
#> 8 Neptune 4.49e12 0.0113 59514.
```

Mercury's 88 days and Neptune's 165 years come out of the same run. On a log-log plot, T against a should be a straight line of slope $3/2$:

```
fit <- lm(log(T_days) ~ log(a), data = kepler3)
```

```
kepler3 |>
```

```
ggplot(aes(a / distance_earth_sun, T_days / 365.25)) +
  geom_abline(intercept = 0, slope = 1.5, color = "grey60") + # T = a^(3/2) in AU
  geom_point() +
  geom_text(aes(label = planet), hjust = -0.2, vjust = 0.3, size = 3) +
  scale_x_log10() + scale_y_log10() +
  labs(x = "Semi-major axis (AU)", y = "Period (years)")
```

```
coef(fit)[["log(a)"]]
```

```
#> [1] 1.498288
```

That is the exponent. Now the constant. Rearranging, $\mu = 4\pi^2 a^3 / T^2$, and μ/G is a mass:

8. The Solar System from JPL Elements

```
kepler3 |>
  mutate(M_implied = 4 * pi^2 * a^3 / (T_days * seconds_per_day)^2 /
          gravitational_constant,
          M_over_sun = M_implied / mass_sun) |>
  select(planet, M_over_sun)
#> # A tibble: 8 x 2
#>   planet M_over_sun
#>   <chr>   <dbl>
#> 1 Mercury 0.994
#> 2 Venus   0.999
#> 3 Earth   1.000
#> 4 Mars    1.000
#> 5 Jupiter 1.00
#> 6 Saturn  1.01
#> 7 Uranus  0.994
#> 8 Neptune 1.02
```

Every planet weighs the Sun. The answers agree to better than a part in a thousand, and the one that disagrees most is Jupiter, because what the period really measures is $M_{\odot} + m_{\text{planet}}$, and Jupiter is a thousandth of a solar mass. Kepler had no way to know that his third law was slightly different for each planet; the differences were far below the precision of Tycho's data. We can see them in a table.

Where it comes from

This is exactly how the masses in Appendix D were determined, with one twist. For a planet with moons, you watch a moon: its period and distance give $G(M_{\text{planet}} + m_{\text{moon}})$ directly. Jupiter's mass was known to a fraction of a percent by the 18th century from the Galilean moons, long before anyone had a good value of G , which is why astronomers habitually quote GM rather than M . For Mercury and Venus, which have no moons, the mass had to wait for spacecraft: the bending of a spacecraft's path during a flyby is a two-body hyperbola, and Chapter 4's formulas give the mass from the deflection.

8.4. How often do Earth and Mars line up?

Mars is closest to Earth, and brightest in our sky, at *opposition*, when Earth passes between Mars and the Sun, so that the two planets have the same heliocentric longitude. How often does that happen? Earth laps Mars, and the time between lappings, the *synodic period*, follows from the angular rates. Earth's longitude advances at $2\pi/T_E$ per unit time and Mars's at $2\pi/T_M$, so the difference advances at $2\pi(1/T_E - 1/T_M)$, and one full lap takes

$$\frac{1}{T_{\text{syn}}} = \frac{1}{T_E} - \frac{1}{T_M}. \quad (8.1)$$

With the measured periods:

```
T_E <- kepler3$T_days[kepler3$planet == "Earth"]
T_M <- kepler3$T_days[kepler3$planet == "Mars"]
1 / (1 / T_E - 1 / T_M)
#> [1] 780.2493
```

About 780 days, or two years and seven weeks. Now from the data directly. Compute each planet's heliocentric longitude, unwrap it so it grows continuously, and take the difference:

```
longitude <- function(sim, body) {
  relative_state(sim, body, "Sun") |>
    transmute(time, lambda = unwrap(atan2(ry, rx)))
}

lap <- inner_join(longitude(solar, "Earth"), longitude(solar, "Mars"),
  by = "time", suffix = c("_earth", "_mars")) |>
  mutate(lead = lambda_earth - lambda_mars)
```

The lead grows by 2π per synodic period, so its average slope gives T_{syn} directly:

```
slope <- coef(lm(lead ~ time, data = lap))["time"]
2 * pi / slope / seconds_per_day
#> [1] 780.2043
```

The two agree. But the lead does not grow uniformly. The times when it passes a multiple of 2π are the oppositions:

```
oppositions <- lap |>
  filter(floor(lead / (2 * pi)) != lag(floor(lead / (2 * pi)))) |>
  pull(time) / seconds_per_day

head(diff(oppositions), 8)
#> [1] 783 809 799 775 767 764 767 777
```

The gaps between successive oppositions swing from about 764 to 810 days. The reason is Mars's eccentricity: it moves faster near perihelion and slower near aphelion, so Earth catches it sooner or later depending on where in its orbit the catching happens. Oppositions near Mars's perihelion are also the close ones, when Mars is at its largest and brightest in our sky; they recur every 15 to 17 years, and the synodic period alone does not tell you that. The simulation does.

8.5. The Sun wobbles

The Sun is not fixed. Chapter 4's center-of-mass analysis applies to every pair, and Jupiter, a thousandth of a solar mass at 5.2 AU, pulls the Sun around a circle of radius about 0.005 AU, a little more than the Sun's own radius, once every 11.9 years. The Sun's speed on that

8. The Solar System from JPL Elements

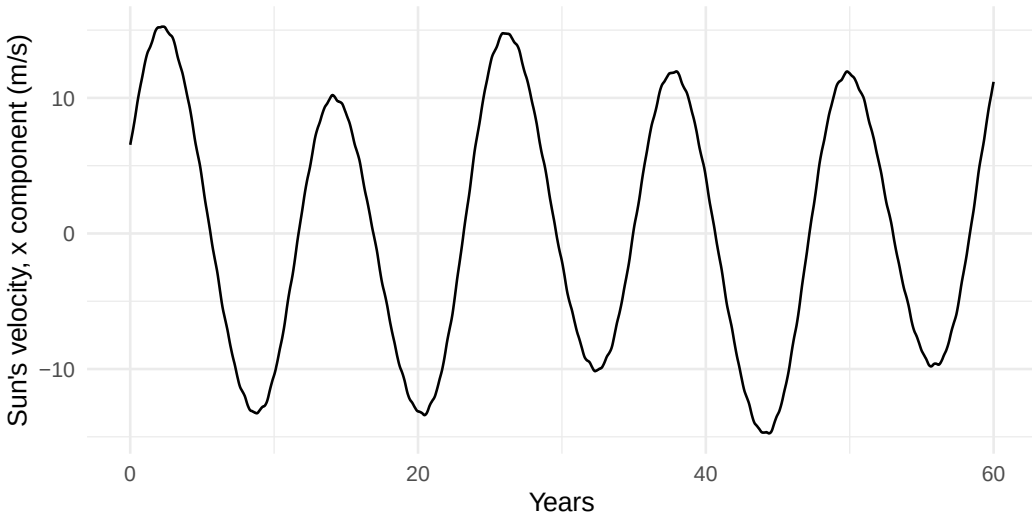


Figure 8.2.: The Sun's velocity relative to the solar system's center of mass, x component, over 60 years. The 12-year, 12 m/s oscillation is Jupiter; the slower modulation on top of it is Saturn.

circle is a dozen meters per second. That number is the foundation of the radial-velocity method for finding planets around other stars: you cannot see the planet, but a spectrograph can see the star's velocity along your line of sight change by meters per second as the planet goes around.

`load_solar_system()` starts the Sun at rest at the origin, so the system's center of mass drifts slowly (Chapter 2 again). To see the wobble alone, look at the Sun in the barycentric frame, where the center of mass is at rest by construction:

```
sun_wobble <- solar |>
  shift_reference_frame("barycenter") |>
  filter(id == "Sun")

sun_wobble |>
  filter(time < seconds_per_year * 60) |>
  ggplot(aes(time / seconds_per_year, vx)) +
  geom_line() +
  labs(x = "Years", y = "Sun's velocity, x component (m/s)")
```

Twelve meters per second is a brisk walking pace, measured on a star. For a distant observer sitting in the plane of the solar system, that is the signal the Sun would show. Modern spectrographs reach below one meter per second, which is enough to find Saturn (about 3 m/s) and, with patience, Neptune. Earth's contribution is nine centimeters per second, which is why finding true Earth twins this way is still at the edge of what is possible.

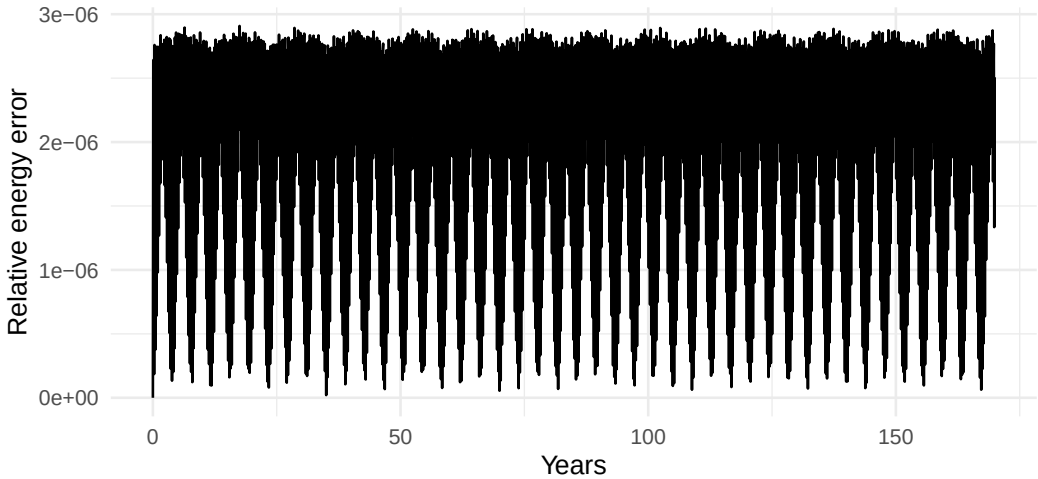


Figure 8.3.: Relative energy error of the 170-year solar system run, sampled every 30 days. The oscillation is Mercury’s orbit; there is no drift.

8.6. A long run checked

A 170-year integration is long enough that it is worth asking whether the integrator held up. The total energy of a nine-body system needs every pair at every time step, which for 62,000 steps and 36 pairs is more rows than we need; sampling every 30 days is plenty for a drift check:

```
solar |>
  filter(time %% (seconds_per_day * 30) == 0) |>
  conserved_quantities() |>
  ggplot(aes(time / seconds_per_year, energy_error)) +
  geom_line() +
  labs(x = "Years", y = "Relative energy error")
```

The error oscillates at the level set by Mercury, the tightest orbit, and does not grow, which is Chapter 5’s promise kept over 62,000 steps. If you double the duration, the band stays the same width. If you double the step, it gets four times wider. Both are worth doing once to see.

Key results

- The package’s planets are osculating J2000 elements from JPL DE440, integrated with Newtonian point-mass gravity.
- Kepler’s third law holds across all eight planets to better than a part in a thousand; the residual is each planet’s own mass.
- Synodic period: $1/T_{\text{syn}} = 1/T_1 - 1/T_2$. Earth-Mars: about 780 days, varying by $\pm 3\%$ because of Mars’s eccentricity.

8. *The Solar System from JPL Elements*

- The Sun moves at about 12 m/s because of Jupiter; this is the radial-velocity exoplanet signal.

9. Binary Stars and Circumbinary Planets

Most stars are not alone. Roughly half of Sun-like stars have a companion, and the “central body at rest at the origin” picture that works for the solar system fails completely for a pair of comparable masses: both stars move, both move a lot, and the only thing that stays put is the point between them. The Kepler-16 example in the package documentation sets up a real binary with a planet around it, using four velocity formulas stated without derivation. This chapter derives them from Chapter 4, checks each one against the simulation, extends them to an eccentric binary, and then asks the question the discoverers of Kepler-16b had to answer: how close to a pair of stars can a planet orbit and survive?

9.1. Two stars

Take stars of mass m_A and m_B a distance d apart on circular orbits. Chapter 4 reduced this to a single relative orbit with gravitational parameter $\mu = G(m_A + m_B)$, so the relative speed on a circle of radius d is $u = \sqrt{G(m_A + m_B)/d}$. Each star’s actual orbit is around the center of mass, and from Equation 4.2 the two stars sit on opposite sides of it at distances

$$r_A = \frac{m_B}{m_A + m_B} d, \quad r_B = \frac{m_A}{m_A + m_B} d, \quad (9.1)$$

so the heavier star makes the smaller circle. The speeds scale the same way, $v_A = u m_B/(m_A + m_B)$ and $v_B = u m_A/(m_A + m_B)$, and substituting u :

$$v_A = \sqrt{\frac{G m_B^2}{(m_A + m_B) d}}, \quad v_B = \sqrt{\frac{G m_A^2}{(m_A + m_B) d}}. \quad (9.2)$$

These are the formulas in the documentation. Note what they say: the star you are computing the speed *of* appears only in the denominator, inside the total mass; the numerator has the mass of the *other* star. A light companion barely moves the heavy primary. The momenta $m_A v_A$ and $m_B v_B$ are equal, so with the stars on opposite sides moving in opposite directions, the total momentum is zero and the center of mass stays at the origin. The period, from Kepler’s third law with $a = d$, is $T = 2\pi\sqrt{d^3/G(m_A + m_B)}$ for both stars; they must have the same period, since they are always opposite each other.

A function that builds any circular binary from these formulas:

9. Binary Stars and Circumbinary Planets

```
add_binary <- function(system, m_A, m_B, d, ids = c("A", "B")) {  
  r_A <- d * m_B / (m_A + m_B)  
  r_B <- d * m_A / (m_A + m_B)  
  v_A <- sqrt(gravitational_constant * m_B^2 / ((m_A + m_B) * d))  
  v_B <- sqrt(gravitational_constant * m_A^2 / ((m_A + m_B) * d))  
  system |>  
    add_body(ids[1], mass = m_A, x = r_A, vy = v_A) |>  
    add_body(ids[2], mass = m_B, x = -r_B, vy = -v_B)  
}
```

Check it on a lopsided pair, a Sun-mass star with a companion of a tenth of the mass, one AU apart:

```
pair <- create_system() |>  
  add_binary(m_A = mass_sun, m_B = 0.1 * mass_sun, d = distance_earth_sun) |>  
  simulate_system(time_step = seconds_per_hour * 6, duration = seconds_per_year * 2)  
  
pair |>  
  group_by(id) |>  
  summarise(r_min_AU = min(sqrt(x^2 + y^2)) / distance_earth_sun,  
            r_max_AU = max(sqrt(x^2 + y^2)) / distance_earth_sun)  
#> # A tibble: 2 x 3  
#>   id      r_min_AU r_max_AU  
#>   <chr>      <dbl>   <dbl>  
#> 1 A          0.0909    0.0909  
#> 2 B          0.909     0.909
```

Each star stays on a circle (minimum and maximum radius agree), the radii are in the ratio 1 to 10 as Equation 9.1 says, and they add to one AU. The period should be shorter than a year by the factor $\sqrt{1.1}$, because the total mass is $1.1 M_{\odot}$:

```
c(predicted = 365.25 / sqrt(1.1),  
   measured = measure_period(pair, "B", "A") / seconds_per_day)  
#> predicted measured  
#> 348.2522 348.2169
```

For a binary the natural frame is the one centered on the pair's center of mass, and `shift_reference_frame(sim, "barycenter")` puts every time step in it. In that frame the center of mass sits at the origin at rest, and a body's distance from the origin is its distance from the barycenter, which is what a circumbinary planet orbits. `measure_period()` with no `center` argument measures a body's period about the origin, which is how the next section uses it.

9.2. An eccentric binary

Real binaries are rarely circular. Kepler-16's two stars have $e = 0.16$. The same reduction handles it: pick the relative orbit, then split it between the stars by the mass ratio. If the

relative orbit has semi-major axis a_{bin} and eccentricity e , start the stars at periapsis, where the separation is $d_p = a_{\text{bin}}(1 - e)$ and the relative speed, from vis-viva, is

$$u_p = \sqrt{\frac{\mu}{a_{\text{bin}}} \cdot \frac{1+e}{1-e}}, \quad \mu = G(m_A + m_B). \quad (9.3)$$

Then $v_A = u_p m_B / (m_A + m_B)$ and $v_B = u_p m_A / (m_A + m_B)$ as before, and the stars are placed at $d_p m_B / (m_A + m_B)$ and $d_p m_A / (m_A + m_B)$ on either side of the origin.

```
add_eccentric_binary <- function(system, m_A, m_B, a_bin, e, ids = c("A", "B")) {
  M <- m_A + m_B
  d_p <- a_bin * (1 - e)
  u_p <- sqrt(gravitational_constant * M / a_bin * (1 + e) / (1 - e))
  system |>
    add_body(ids[1], mass = m_A, x = d_p * m_B / M, vy = u_p * m_B / M) |>
    add_body(ids[2], mass = m_B, x = -d_p * m_A / M, vy = -u_p * m_A / M)
}
```

The eccentricity vector from Chapter 4 checks that this produced the orbit it was supposed to:

```
ecc_pair <- create_system() |>
  add_eccentric_binary(m_A = mass_sun, m_B = 0.5 * mass_sun,
    a_bin = distance_earth_sun, e = 0.4) |>
  simulate_system(time_step = seconds_per_hour * 6, duration = seconds_per_year)

get_orbital_elements(ecc_pair, "B", "A", mu = gravitational_constant * 1.5 * mass_sun)
summarise(e_min = min(e), e_max = max(e))
#> # A tibble: 1 x 2
#>   e_min e_max
#>   <dbl> <dbl>
#> 1 0.400 0.400
```

9.3. Kepler-16

Kepler-16b was the first planet confirmed to orbit two stars, found in 2011 by the Kepler spacecraft when it was seen transiting both of them (Doyle et al. 2011). The system is a K-type star of $0.69 M_{\odot}$ and an M-type star of $0.20 M_{\odot}$ in a 41-day orbit with semi-major axis 0.224 AU and eccentricity 0.16, and a Saturn-mass planet ($0.33 M_{\text{Jup}}$) orbiting the pair every 229 days at 0.705 AU, on a nearly circular orbit in the same plane. The documentation builds it with a circular binary; here it is with the real eccentricity.

The planet is far enough out that the two stars act on it almost like a single star of their combined mass at the barycenter. So the planet goes on a circle of radius 0.705 AU around the origin at the circular speed for mass $m_A + m_B$:

9. Binary Stars and Circumbinary Planets

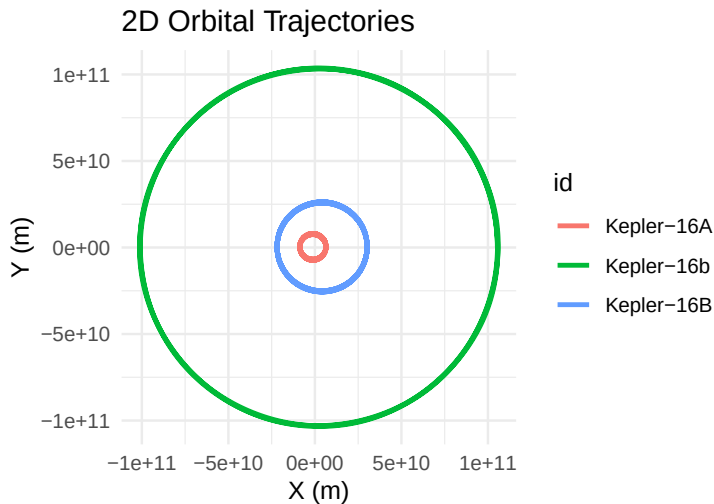


Figure 9.1.: Kepler-16 for two years. The two stars orbit their common center of mass every 41 days while the planet circles the pair every 229 days.

```
AU <- distance_earth_sun
m_A <- 0.69 * mass_sun
m_B <- 0.20 * mass_sun
m_p <- 0.333 * mass_jupiter

r_planet <- 0.7048 * AU
v_planet <- sqrt(gravitational_constant * (m_A + m_B) / r_planet)

kepler16 <- create_system() |>
  add_eccentric_binary(m_A, m_B, a_bin = 0.2243 * AU, e = 0.159,
    ids = c("Kepler-16A", "Kepler-16B")) |>
  add_body("Kepler-16b", mass = m_p, x = r_planet, vy = v_planet) |>
  simulate_system(time_step = seconds_per_hour, duration = seconds_per_year * 2)

plot_orbits(kepler16)
```

The planet's orbit is not a two-body orbit about anything, but about the barycenter it is very nearly one, and the barycentric frame lets us measure it. The predictions are Kepler's third law with the *combined* mass of all three bodies, and a period of 229 days for the planet and 41 days for the stars:

```

k16 <- shift_reference_frame(kepler16, "barycenter")
mu_total <- gravitational_constant * (m_A + m_B + m_p)

c(planet_predicted_days = 2 * pi * sqrt(r_planet^3 / mu_total) / seconds_per_day,
  planet_measured_days   = measure_period(k16, "Kepler-16b") / seconds_per_day,
  binary_measured_days   = measure_period(k16, "Kepler-16B", "Kepler-16A") /
                                seconds_per_day)
#> planet_predicted_days planet_measured_days binary_measured_days
#>                229.01798                220.79718                41.12564

```

Both match the published periods. The planet is not *exactly* on a circle, because the two stars' pull is not exactly that of a point at the barycenter; the difference is a small periodic wobble at the binary's period, and you can see it by plotting the planet's distance from the barycenter over time. That wobble is real and it is how the planet's orbit was characterized from the transit timing.

Where it comes from

The transit method finds planets by the dip in a star's brightness when the planet crosses in front of it. For Kepler-16 the light curve showed the planet crossing *both* stars, and the two stars eclipsing each other, so the geometry of the whole system could be reconstructed from timing alone. The paper includes a diagram of the three orbits that looks almost exactly like Figure 9.1. Working out the masses needed a full three-body fit, because the planet's orbit is noticeably non-Keplerian: the stars' motion makes the planet's transit times and durations vary noticeably from one transit to the next, and the size of the variation fixes the mass ratio.

9.4. How close is too close?

A planet orbiting a binary can only survive if it is far enough out that the two stars act like one. Too close, and the alternating pulls of the stars as they go around pump energy into the planet's orbit until it is ejected or thrown into one of the stars. The boundary is not sharp, and it depends on the binary's mass ratio and eccentricity, but it is roughly two to four times the binary's separation. Holman and Wiegert (Holman and Wiegert 1999) ran thousands of three-body simulations and fitted a formula for the critical semi-major axis:

$$\frac{a_{\text{crit}}}{a_{\text{bin}}} = 1.60 + 5.10 e - 2.22 e^2 + 4.12 q - 4.27 eq - 5.09 q^2 + 4.61 e^2 q^2, \quad q = \frac{m_B}{m_A + m_B}. \quad (9.4)$$

For Kepler-16, $q = 0.22$ and $e = 0.16$ give $a_{\text{crit}} \approx 2.9 a_{\text{bin}}$, or 0.65 AU. The planet is at 0.705 AU, nine percent outside the line. It is one of the closest-in circumbinary planets known, and it sits almost exactly where the stability boundary says a planet can sit.

We can test the boundary ourselves. Keep the real stars and drop a test planet at a range of distances, from well inside the critical radius to well outside, for fifteen years each:

9. Binary Stars and Circumbinary Planets

```
a_bin <- 0.2243 * AU

trial <- function(ratio) {
  r <- ratio * a_bin
  create_system() |>
    add_eccentric_binary(m_A, m_B, a_bin = a_bin, e = 0.159,
                        ids = c("Kepler-16A", "Kepler-16B")) |>
    add_body("planet", mass = 1e20, x = r,
            vy = sqrt(gravitational_constant * (m_A + m_B) / r)) |>
    simulate_system(time_step = seconds_per_hour * 3,
                  duration = seconds_per_year * 15) |>
    shift_reference_frame("barycenter") |>
    filter(id == "planet") |>
    mutate(r = sqrt(x^2 + y^2 + z^2),
          energy = (vx^2 + vy^2 + vz^2) / 2 -
                  gravitational_constant * (m_A + m_B + 1e20) / r) |>
    summarise(ratio = ratio,
              r_min = min(r) / a_bin, r_max = max(r) / a_bin,
              bound_at_end = last(energy) < 0)
}

stability <- bind_rows(lapply(c(1.5, 2.0, 2.5, 2.8, 3.1, 3.5, 4.0), trial))
stability
#> # A tibble: 7 x 4
#>   ratio r_min r_max bound_at_end
#>   <dbl> <dbl> <dbl> <lgl>
#> 1  1.5 0.0661 1459. FALSE
#> 2  2  0.0174 3528. FALSE
#> 3  2.5 1.80 13.1 TRUE
#> 4  2.8 2.55 2.90 TRUE
#> 5  3.1 2.90 3.18 TRUE
#> 6  3.5 3.33 3.57 TRUE
#> 7  4  3.87 4.05 TRUE
```

A planet that survives keeps its distance from the barycenter within a narrow band around where it started and stays bound. One that does not either wanders far inward and outward or ends with positive energy. The transition in the table sits near the Holman-Wiegert value of 2.9, which is remarkable agreement for a formula fitted to a different set of simulations with a different integrator. Fifteen years is not forever, and a planet that looks stable here might be ejected after a thousand; the published fit is for runs of ten thousand binary periods. But the lesson is already visible: there is a hard inner edge to where planets can live around a binary, and Kepler-16b lives right at it.

Key results

- A circular binary: stars at $d m_B / M$ and $d m_A / M$ from the barycenter, speeds $\sqrt{G m_B^2 / (M d)}$ and $\sqrt{G m_A^2 / (M d)}$, $M = m_A + m_B$. Zero total momentum.

- An eccentric binary: use the periapsis separation $a(1 - e)$ and periapsis speed $\sqrt{(\mu/a)(1 + e)/(1 - e)}$, split by mass ratio.
- A circumbinary planet far from the pair orbits the barycenter as if the stars were one body of mass $m_A + m_B$.
- Planets inside about 2 to 4 binary separations (Equation 9.4) are unstable.

10. Reference Frames

A simulation has to put its origin somewhere, and the place that is convenient for setting up a system is often the wrong place for understanding it. The solar system is easiest to build with the Sun at the origin; the history of astronomy is largely the story of people who stood on Earth and tried to make sense of what they saw from there. `shift_reference_frame()` lets you stand anywhere. This chapter uses it to reproduce the three observations that defined pre-Copernican astronomy (the Moon's orbit, the retrograde loops of Mars, and the fact that the planets' loops are tied to the Sun's position), then turns it on a system no one has ever stood in, Kepler-16b, and finally builds by hand the one kind of frame the package does not provide: a rotating one.

10.1. Moving the camera

`shift_reference_frame(sim, "Earth")` subtracts Earth's position and velocity from every body at every time step:

$$\mathbf{r}'_i = \mathbf{r}_i - \mathbf{r}_{\text{Earth}}, \quad \mathbf{v}'_i = \mathbf{v}_i - \mathbf{v}_{\text{Earth}}.$$

Earth lands at the origin with zero velocity for the whole run; everything else is now described relative to Earth. The transformation is applied after the simulation, to the output tibble, and it changes no physics: every force was computed in the original frame, and the shifted data is the same motion seen from a different place.

One subtlety is worth stating because the vignette does not. The new frame is not inertial. Earth accelerates (it is orbiting the Sun), so a frame attached to it is an accelerated frame, and if you tried to *do dynamics* in it, Newton's second law would need fictitious forces added. For *looking* at a trajectory that has already been computed, none of that matters. The shifted coordinates are exactly what an observer riding on Earth would record, fictitious forces and all, because the observer does not compute anything; they just watch. The same holds for a frame attached to any body, and later in the chapter for a rotating frame.

10.2. The geocentric Moon

Build the Sun, Earth, and Moon heliocentrically, as the documentation does, by giving the Moon Earth's position and velocity plus its own:

10. Reference Frames

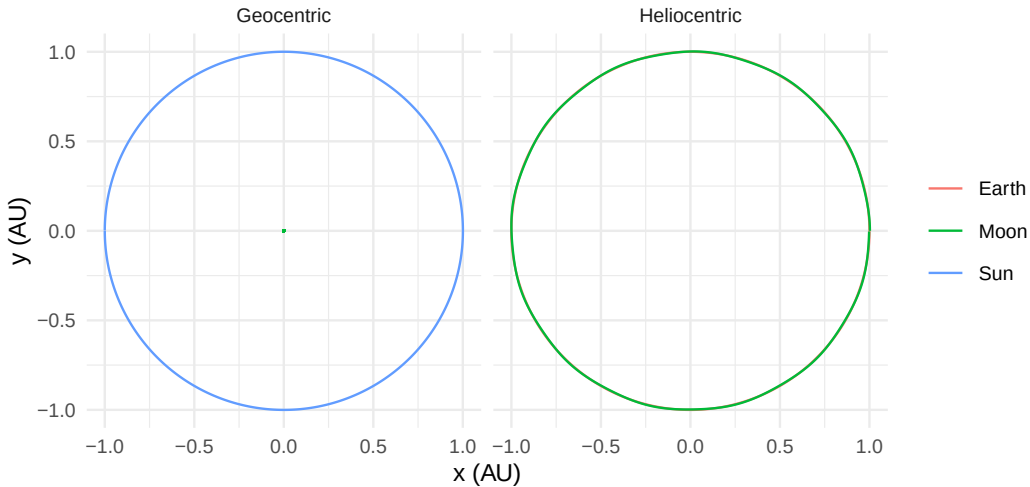


Figure 10.1.: The same year seen from the Sun (left) and from Earth (right). In the geocentric frame the Moon's thirteen orbits are visible and the Sun circles once.

```
sem <- 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)
```

In the heliocentric frame the Moon's path and Earth's are indistinguishable: 384,000 km is a quarter of a percent of an AU. In Earth's frame the Moon is a clean ellipse and the Sun sweeps a wide circle once a year:

```
helio <- sem |> mutate(frame = "Heliocentric")
geo <- sem |> shift_reference_frame("Earth") |> mutate(frame = "Geocentric")

bind_rows(helio, geo) |>
  ggplot(aes(x / distance_earth_sun, y / distance_earth_sun, color = id)) +
  geom_path() +
  coord_equal() +
  facet_wrap(~ frame) +
  labs(x = "x (AU)", y = "y (AU)", color = NULL)
```

The shifted velocities are as useful as the positions. The Moon's speed relative to Earth, read directly from the shifted v_x , v_y , v_z , oscillates once per orbit, faster at perigee and slower at apogee, which is Kepler's second law as a time series:

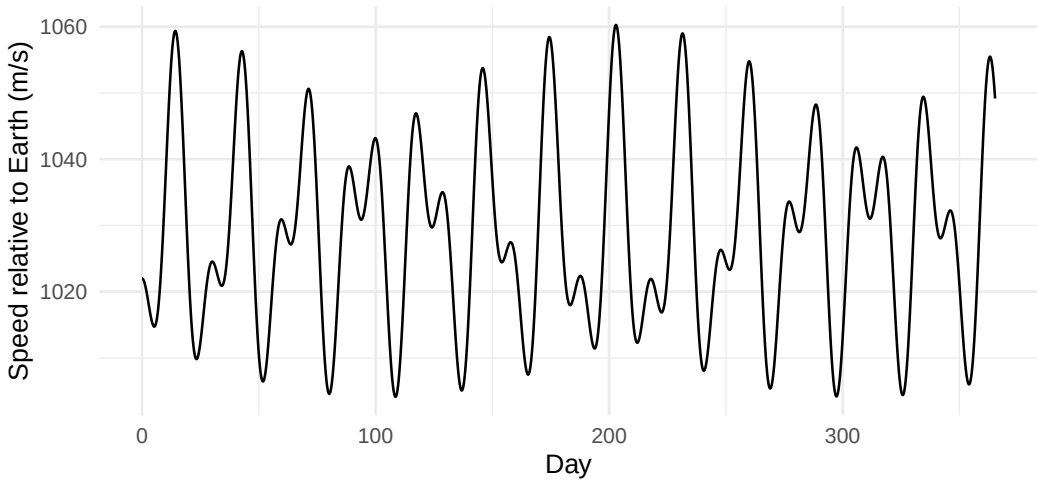


Figure 10.2.: The Moon's speed relative to Earth. The monthly oscillation is the Moon's eccentricity; the slow envelope is the Sun's perturbation.

```
geo |>
  filter(id == "Moon") |>
  mutate(speed = sqrt(vx^2 + vy^2 + vz^2)) |>
  ggplot(aes(time / seconds_per_day, speed)) +
  geom_line() +
  labs(x = "Day", y = "Speed relative to Earth (m/s)")
```

10.3. Retrograde Mars

For most of the year Mars drifts eastward against the stars. Every 26 months it slows, stops, moves *backward* for about two months, stops again, and resumes. Ptolemy explained the loops with epicycles; Copernicus explained them as the view from a moving platform. Here is the Copernican explanation, run:

```
sem_mars <- create_system() |>
  add_sun() |>
  add_planet("Earth", parent = "Sun") |>
  add_planet("Mars", parent = "Sun", nu = 60) |>
  simulate_system(time_step = seconds_per_day, duration = seconds_per_year * 2.2)

mars_geo <- sem_mars |>
  shift_reference_frame("Earth", keep_center = FALSE) |>
  filter(id == "Mars")

mars_geo |>
  ggplot(aes(x / distance_earth_sun, y / distance_earth_sun,
```

10. Reference Frames

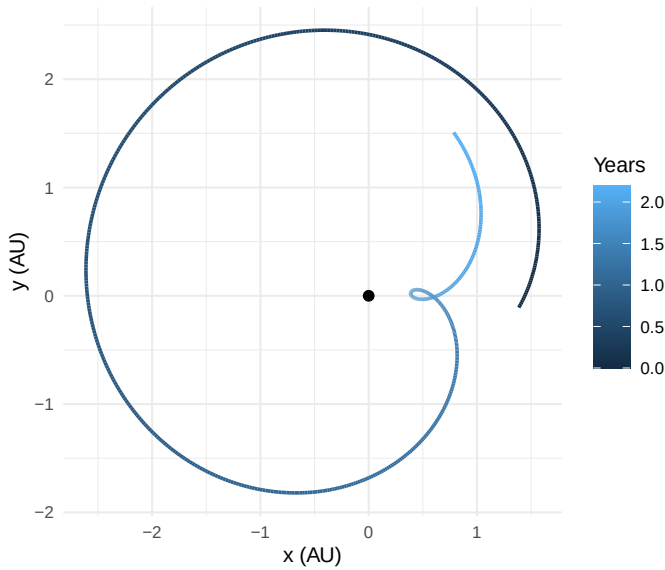


Figure 10.3.: Mars seen from Earth over 2.2 years. The loop is retrograde motion: Earth, on the faster inner orbit, overtakes Mars, and for a few weeks around opposition Mars appears to move backward.

```

    color = time / seconds_per_year)) +
geom_path(linewidth = 0.8) +
annotate("point", x = 0, y = 0, size = 2) +
coord_equal() +
labs(x = "x (AU)", y = "y (AU)", color = "Years")

```

The loop is real and it is nothing but geometry. We can measure it. Mars's geocentric longitude is the angle of its position vector in Earth's frame; it increases when Mars moves direct and decreases when it moves retrograde:

```

retro <- mars_geo |>
  mutate(longitude = unwrap(atan2(y, x)),
         direction = if_else(c(0, diff(longitude)) < 0, "retrograde", "direct"))

retro |>
  filter(direction == "retrograde") |>
  summarise(starts_day = min(time) / seconds_per_day,
            ends_day = max(time) / seconds_per_day,
            duration_days = (max(time) - min(time)) / seconds_per_day)
#> # A tibble: 1 x 3
#>   starts_day ends_day duration_days

```

```
#>          <dbl>    <dbl>          <dbl>
#> 1          598      659           61
```

About ten weeks of backward motion, which is what the almanacs list, and the midpoint is the opposition, when Earth passes directly between Mars and the Sun. The retrograde interval is *always* centered on opposition. Ptolemy's model had to build that coincidence in by hand, by tying the epicycle's rotation to the Sun's; Copernicus's gets it for free, because the loop *is* Earth's motion.

10.4. The Moon's path around the Sun

Here is a puzzle that trips up almost everyone. From Earth, the Moon goes around us in a circle. From the Sun, what does its path look like? The tempting answer is a wavy line, looping back on itself as the Moon swings around Earth. The actual answer is that the Moon's heliocentric path is *everywhere concave toward the Sun*: it never loops, never even straightens out, and looks like a slightly wobbly circle. Chapter 3 gave the reason: the Sun's pull on the Moon is more than twice Earth's, so the Moon's acceleration always points mostly at the Sun, and a path whose acceleration always points toward a point is always curving toward that point.

The simulation lets us check the sign of the curvature directly. A curve bends toward the side its acceleration points to, and the direction of bending is given by the sign of $\mathbf{v} \times \mathbf{a}$. For a path that always bends toward the origin, $(\mathbf{v} \times \mathbf{a})_z$ never changes sign. Estimate $\mathbf{a}$ by differencing the velocity:

```
sem |>
  filter(id == "Moon") |>
  mutate(ax = c(NA, diff(vx)) / seconds_per_hour,
         ay = c(NA, diff(vy)) / seconds_per_hour,
         curvature_sign = sign(vx * ay - vy * ax)) |>
  count(curvature_sign)
#> # A tibble: 2 x 2
#>   curvature_sign     n
#>         <dbl> <int>
#> 1           1  8766
#> 2          NA     1
```

Every single step curves the same way. Compare Earth seen from the Moon's frame in the next exercise: there the Sun's effect cancels (both bodies fall toward it together) and the loop is a loop.

10.5. Two sunsets

Kepler-16b is a real planet with two suns. From its surface, what does the sky do? Rebuild the system from Chapter 9 and shift to the planet:

10. Reference Frames

```
AU <- distance_earth_sun
m_A <- 0.69 * mass_sun
m_B <- 0.20 * mass_sun
m_p <- 0.333 * mass_jupiter

add_eccentric_binary <- function(system, m_A, m_B, a_bin, e, ids) {
  M <- m_A + m_B
  d_p <- a_bin * (1 - e)
  u_p <- sqrt(gravitational_constant * M / a_bin * (1 + e) / (1 - e))
  system |>
    add_body(ids[1], mass = m_A, x = d_p * m_B / M, vy = u_p * m_B / M) |>
    add_body(ids[2], mass = m_B, x = -d_p * m_A / M, vy = -u_p * m_A / M)
}

r_p <- 0.7048 * AU
kepler16 <- create_system() |>
  add_eccentric_binary(m_A, m_B, a_bin = 0.2243 * AU, e = 0.159,
    ids = c("Kepler-16A", "Kepler-16B")) |>
  add_body("Kepler-16b", mass = m_p, x = r_p,
    vy = sqrt(gravitational_constant * (m_A + m_B) / r_p)) |>
  simulate_system(time_step = seconds_per_hour, duration = seconds_per_year)

from_planet <- kepler16 |>
  shift_reference_frame("Kepler-16b", keep_center = FALSE)

from_planet |>
  ggplot(aes(x / AU, y / AU, color = id)) +
  geom_path(linewidth = 0.6) +
  annotate("point", x = 0, y = 0, size = 2) +
  coord_equal() +
  labs(x = "x (AU)", y = "y (AU)", color = NULL)
```

The spirograph pattern is two motions combined: the stars circle the planet once per planetary year (which is the planet's own orbit, seen from the planet), and they circle each other every 41 days on top of it. The quantity an inhabitant would notice is the angle between the two suns in the sky. With both stars' positions relative to the planet, that is one arccosine:

```
star_a <- from_planet |> filter(id == "Kepler-16A") |> select(time, xa = x, ya = y, za = z)
star_b <- from_planet |> filter(id == "Kepler-16B") |> select(time, xb = x, yb = y, zb = z)

inner_join(star_a, star_b, by = "time") |>
  mutate(cos_sep = (xa * xb + ya * yb + za * zb) /
    (sqrt(xa^2 + ya^2 + za^2) * sqrt(xb^2 + yb^2 + zb^2)),
    separation_deg = acos(cos_sep) * 180 / pi) |>
  ggplot(aes(time / seconds_per_day, separation_deg)) +
  geom_line() +
  labs(x = "Day", y = "Separation of the two suns (degrees)")
```

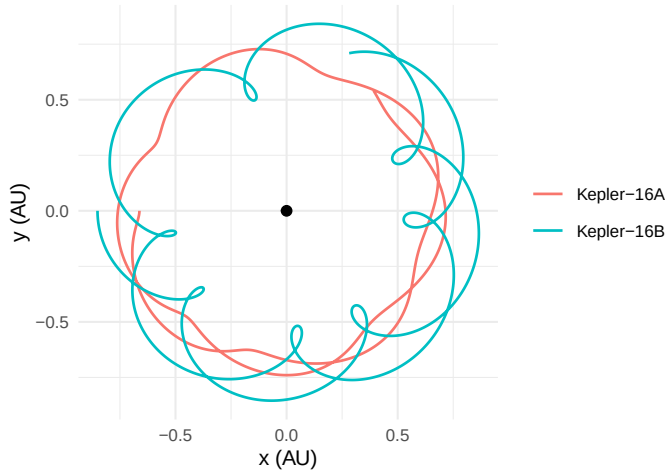


Figure 10.4.: Kepler-16’s two stars as seen from the planet over one year. Each star’s path is the binary’s 41-day orbit wound around the planet’s 229-day orbit.

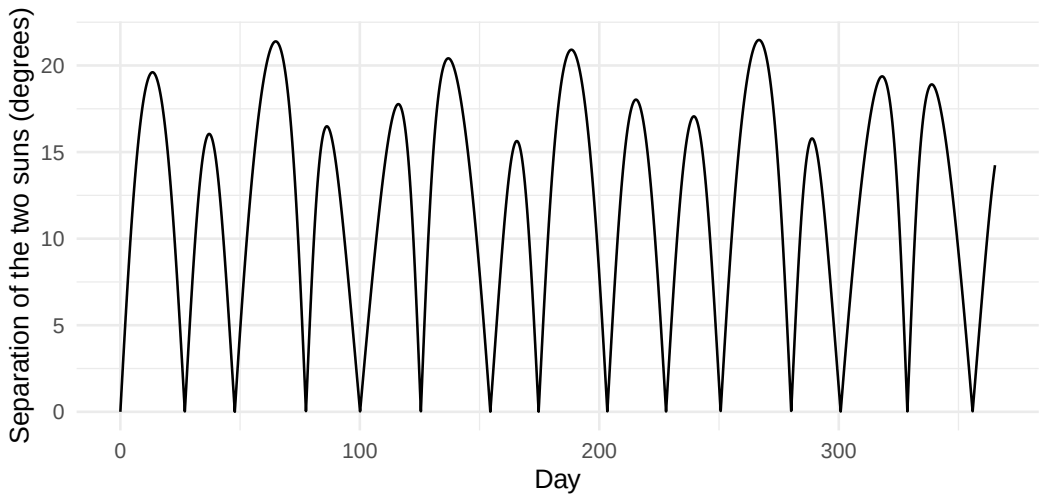


Figure 10.5.: Angular separation of Kepler-16’s two stars as seen from the planet. They swing apart and together every 41 days, never reaching more than about twenty degrees apart.

10. Reference Frames

Twice every 41 days the two suns line up (separation near zero, one eclipsing the other) and twice they stand as far apart as they ever get, about forty Moon-widths. A double sunset on Kepler-16b would be two suns a hand's width apart going down together, with the gap between them changing noticeably from one week to the next.

10.6. A rotating frame, by hand

Every frame so far has been a translation: the origin moves, the axes do not turn. There is one more kind of frame that matters in orbital mechanics, and it is the one in which a binary star, or the Sun and Jupiter, stand *still*: a frame that rotates with them. In a rotating frame the dynamics picks up centrifugal and Coriolis terms, and the five famous equilibrium points of the three-body problem, the Lagrange points, are the places where those terms balance gravity. The package has no rotating-frame function, but `dplyr` does not need one. If the frame rotates at angular rate ω about the z axis, the co-rotating coordinates are

$$x' = x \cos \omega t + y \sin \omega t, \quad y' = -x \sin \omega t + y \cos \omega t, \quad (10.1)$$

which just undoes the rotation that has accumulated by time t .

Set up the Sun and Jupiter as a circular binary about their barycenter (Chapter 9's `add_binary()`), and a small body at the L_4 point, which sits at the third corner of an equilateral triangle with the two masses, 60 degrees ahead of Jupiter. A body exactly at L_4 moving at exactly the frame's rotation speed stays there forever; nudge it slightly, and it wanders around L_4 on a *tadpole* orbit. That is where the Trojan asteroids live.

```
add_binary <- function(system, m_A, m_B, d, ids) {
  r_A <- d * m_B / (m_A + m_B); r_B <- d * m_A / (m_A + m_B)
  v_A <- sqrt(gravitational_constant * m_B^2 / ((m_A + m_B) * d))
  v_B <- sqrt(gravitational_constant * m_A^2 / ((m_A + m_B) * d))
  system |>
    add_body(ids[1], mass = m_A, x = r_A, vy = v_A) |>
    add_body(ids[2], mass = m_B, x = -r_B, vy = -v_B)
}

d <- distance_jupiter_sun
omega <- sqrt(gravitational_constant * (mass_sun + mass_jupiter) / d^3)

# Jupiter sits at angle 180 degrees moving counterclockwise; L4 is 60 degrees
# ahead of it, at the apex of the equilateral triangle below the x axis.
sun_x <- d * mass_jupiter / (mass_sun + mass_jupiter)
jup_x <- -d * mass_sun / (mass_sun + mass_jupiter)
L4 <- c((sun_x + jup_x) / 2, -sqrt(3) / 2 * d)
p <- L4 * 1.02 # nudged 2% outward
v <- omega * c(-p[2], p[1]) # rigid co-rotation

trojan <- create_system() |>
  add_binary(mass_sun, mass_jupiter, d, ids = c("Sun", "Jupiter")) |>
```

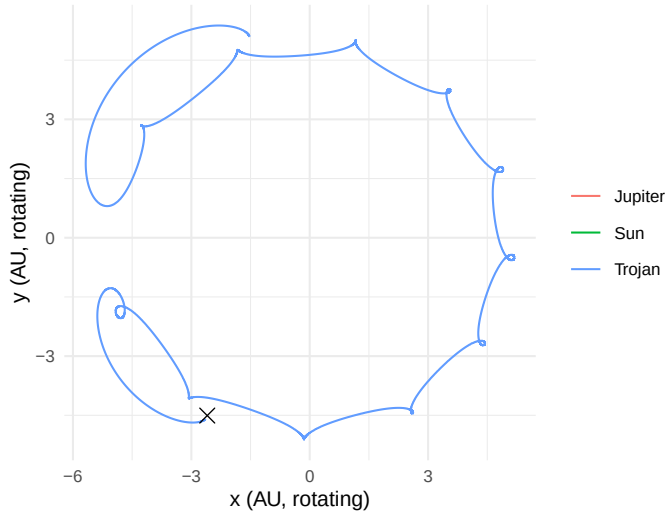


Figure 10.6.: The Sun-Jupiter system in the frame rotating with Jupiter, over 150 years. The Sun and Jupiter are fixed points; the nudged Trojan librates around L_4 on a tadpole orbit.

```
add_body("Trojan", mass = 1e15, x = p[1], y = p[2], vx = v[1], vy = v[2]) |>
simulate_system(time_step = seconds_per_day, duration = seconds_per_year * 150)

trojan |>
mutate(theta = omega * time,
        xr = x * cos(theta) + y * sin(theta),
        yr = -x * sin(theta) + y * cos(theta)) |>
ggplot(aes(xr / AU, yr / AU, color = id)) +
geom_path() +
annotate("point", x = L4[1] / AU, y = L4[2] / AU, shape = 4, size = 3) +
coord_equal() +
labs(x = "x (AU, rotating)", y = "y (AU, rotating)", color = NULL)
```

In the inertial frame the Trojan's path is an unremarkable near-circle at Jupiter's distance. In the rotating frame it is a closed loop around a point in empty space, held there by nothing but the combined gravity of the Sun and Jupiter in a frame where that gravity has to fight the centrifugal force. The $\times$ marks L_4 . Jupiter has about ten thousand known Trojans doing exactly this, and Earth has two. Chapter 11 looks at the other Lagrange points, which are not so hospitable.

Key results

- `shift_reference_frame()` is a Galilean translation applied to the output: same physics, different observer. The frame it produces is accelerated, which is fine for looking and not for computing forces.
- Retrograde motion is the view of an outer planet from a faster inner one; the retrograde interval is centered on opposition.
- The Moon's heliocentric path is always concave toward the Sun, because the Sun's pull on it exceeds Earth's.
- Rotating frame: $x' = x \cos \omega t + y \sin \omega t$, $y' = -x \sin \omega t + y \cos \omega t$. In the frame co-rotating with a circular binary, the Lagrange points are fixed and L_4 , L_5 are stable.

11. Chaos and the Three-Body Problem

Chapter 4 solved the two-body problem completely, on paper. Add one more body and the paper runs out: there is no formula, and for most starting configurations there is not even a stable outcome, only a brief dance and then an ejection. The package documentation shows one such triple and says that it is chaotic. This chapter measures the chaos: how fast two nearly identical triples diverge, what a slingshot ejection does to the energy budget, and why some three-body systems (ours, for instance) are perfectly well behaved. It ends with the five points where a third body can sit still, and why three of them are traps.

11.1. What Poincaré proved

Where it comes from

In 1885 King Oscar II of Sweden offered a prize for a proof that the solar system is stable. Henri Poincaré did not prove it. He showed instead that the three-body problem, even in a restricted form, has orbits so tangled that no formula of the usual kind could ever describe them, and in the course of correcting an error in his own prize essay he found the first example of what is now called a homoclinic tangle: trajectories that wind around each other infinitely often, so that where a body ends up depends on where it started with a sensitivity no measurement can satisfy (Poincaré 1890). He later wrote of the resulting figure that he would not even attempt to draw it. The phenomenon was named *chaos* seventy years later, when computers made it possible to draw.

“No general solution” does not mean the motion cannot be computed. It means the computation cannot be shortcut. Kepler’s ellipse lets you find a planet’s position a million years from now in one step; for three bodies, the only way to find out what happens is to follow them step by step, and the simulation is the solution.

11.2. Sensitive dependence, measured

Here is the documentation’s triple: three Sun-mass stars at the corners of a triangle, with velocities that are nearly but not quite symmetric.

11. Chaos and the Three-Body Problem

```
triple <- function(dx = 0) {
  create_system() |>
    add_body("Star A", mass = 1e30, x = 1e11 + dx, y = 0,
             vx = 0, vy = 15000) |>
    add_body("Star B", mass = 1e30, x = -5e10, y = 8.66e10,
             vx = -12990, vy = -7500) |>
    add_body("Star C", mass = 1e30, x = -5e10, y = -8.66e10,
             vx = 14000, vy = -8000) |>
    simulate_system(time_step = seconds_per_hour, duration = seconds_per_year * 3)
}

run_1 <- triple()
run_2 <- triple(dx = 1)      # Star A starts one meter to the right
```

One meter, in a system a hundred billion meters across: a part in 10^{11} , far below any conceivable measurement of a real star's position. Track how far apart the two runs drift:

```
divergence <- inner_join(
  run_1 |> select(time, id, x, y, z),
  run_2 |> select(time, id, x2 = x, y2 = y, z2 = z),
  by = c("time", "id")) |>
  mutate(d = sqrt((x - x2)^2 + (y - y2)^2 + (z - z2)^2)) |>
  group_by(time) |>
  summarise(separation = sqrt(sum(d^2)))

divergence |>
  ggplot(aes(time / seconds_per_year, separation)) +
  geom_line() +
  scale_y_log10() +
  labs(x = "Years", y = "Separation between runs (m)")
```

The separation grows *exponentially*: a straight line on a log axis. That is the signature of chaos, and the slope of the line is its rate. If the separation grows as $\delta(t) = \delta_0 e^{t/\tau}$, then τ is the *Lyapunov time*, the time for a small error to grow by a factor of e . Fit it on the part of the curve that is still growing, before the separation saturates at the size of the system:

```
growth <- divergence |>
  filter(separation > 1e3, separation < 1e9)

tau <- 1 / coef(lm(log(separation) ~ time, data = growth))["time"]
tau / seconds_per_day
#> [1] 92.63501
```

A Lyapunov time of a few weeks. Every few weeks the uncertainty in the system's state grows by a factor of nearly three; in a year it has grown by a factor of $e^{365/\tau}$, which is enormous. After the point where the two curves saturate, the runs have nothing to do with each other: they are two different triples that happen to share a history.

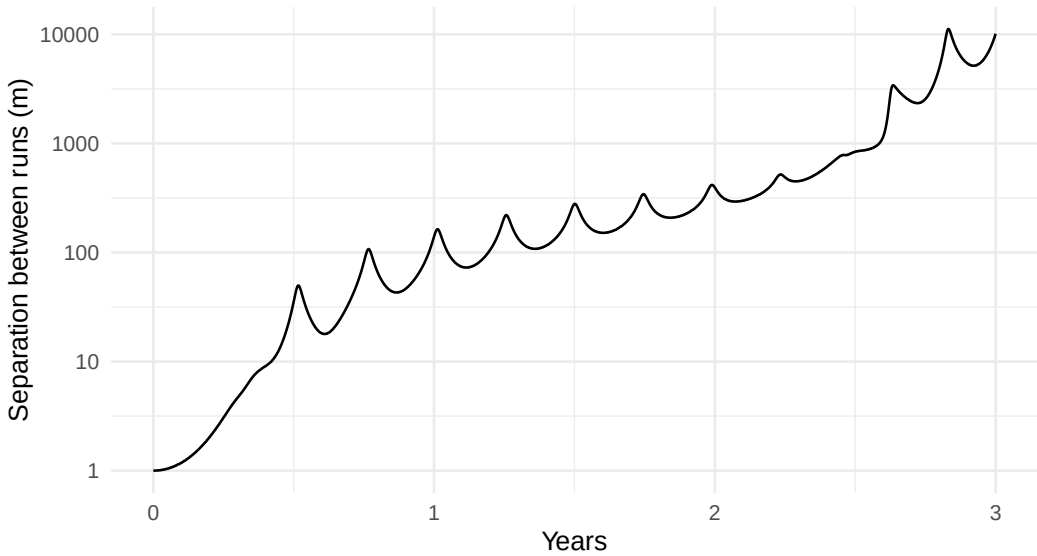


Figure 11.1.: Distance in configuration space between two runs of the same triple whose only difference is one meter in one star’s starting position. The vertical axis is logarithmic: the straight-line section is exponential growth.

This has a practical consequence that is easy to miss. Beyond the Lyapunov horizon, *the simulation is not predicting this system*. The integrator’s own rounding errors, at the 10^{-16} level, are amplified just like the one-meter nudge, so two runs on different computers, or the same run with a different time step, will disagree in detail after a year or two. What the simulation does predict is the *kind* of thing that happens: the statistics of the dance, the fact of the ejection, roughly when. For chaotic systems, those are the only questions with answers.

11.3. Slingshot

```
plot_orbits(run_1)
```

The documentation says one star gets flung out. Let’s make “flung out” precise. A body has escaped a system when its energy relative to the rest, kinetic energy in the center-of-mass frame plus its potential energy with every other body, is positive: it has enough speed to climb out of the potential well of the others and keep going. Here is that bookkeeping at a single instant:

```
body_energies <- function(snapshot, G = gravitational_constant) {
  M   <- sum(snapshot$mass)
  v_cm <- c(sum(snapshot$mass * snapshot$vx),
            sum(snapshot$mass * snapshot$vy),
```

11. Chaos and the Three-Body Problem

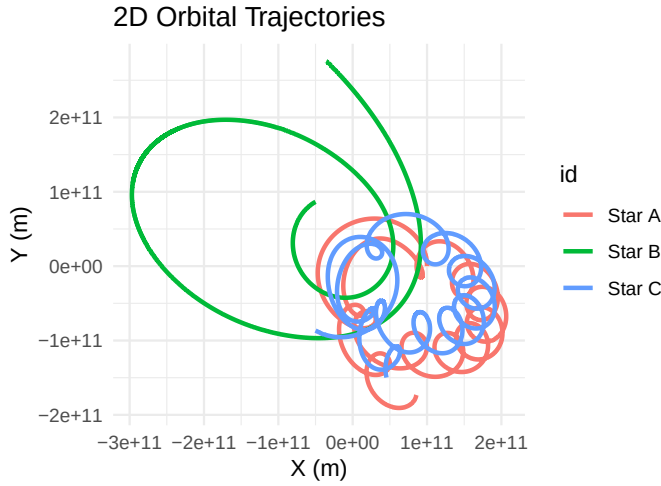


Figure 11.2.: Three years of the chaotic triple. The static picture is hard to read; the energy table below says what happened.

```

      sum(snapshot$mass * snapshot$vz)) / M
n <- nrow(snapshot)
E <- numeric(n)
for (i in seq_len(n)) {
  v_rel <- c(snapshot$vx[i], snapshot$vy[i], snapshot$vz[i]) - v_cm
  KE <- 0.5 * snapshot$mass[i] * sum(v_rel^2)
  PE <- 0
  for (j in seq_len(n)[-i]) {
    r_ij <- sqrt((snapshot$x[i] - snapshot$x[j])^2 +
                  (snapshot$y[i] - snapshot$y[j])^2 +
                  (snapshot$z[i] - snapshot$z[j])^2)
    PE <- PE - G * snapshot$mass[i] * snapshot$mass[j] / r_ij
  }
  E[i] <- KE + PE
}
tibble(id = snapshot$id, energy = E,
       r_from_cm = sqrt((snapshot$x - sum(snapshot$mass * snapshot$x) / M)^2 +
                        (snapshot$y - sum(snapshot$mass * snapshot$y) / M)^2))
}

bind_rows(
  run_1 |> filter(time == min(time)) |> body_energies() |> mutate(when = "start"),
  run_1 |> filter(time == max(time)) |> body_energies() |> mutate(when = "end")
)

```

```

) |>
  arrange(id, when)
#> # A tibble: 6 x 4
#>   id      energy    r_from_cm when
#>   <chr>    <dbl>    <dbl> <chr>
#> 1 Star A -1.31e39 166443322355. end
#> 2 Star A -6.56e38 100000000000. start
#> 3 Star B -1.93e38 299192659248. end
#> 4 Star B -6.55e38 99997799976. start
#> 5 Star C -1.03e39 134679821974. end
#> 6 Star C -6.47e38 99997799976. start

```

At the start all three stars are bound to the system. At the end, a star whose energy is positive is leaving and never coming back, and it is the one farthest from the center of mass. Where did its energy come from? Total energy is conserved (check it with `conserved_quantities()`), so if one star gained enough to escape, the other two lost the same amount: the pair left behind is on a tighter, more negative-energy orbit than any pair was at the start. This is the *slingshot*, and it is the basic mechanism of stellar dynamics. In a star cluster, three-body encounters harden binaries and eject single stars, and over millions of years that is how clusters lose members and how their cores contract. We can look at the binary that remains:

```

final <- run_1 |> filter(time == max(time)) |> body_energies()
bound <- final |> filter(energy < 0) |> pull(id)
bound
#> [1] "Star A" "Star B" "Star C"

if (length(bound) == 2) {
  last_year <- run_1 |> filter(time > max(time) - seconds_per_year)
  get_orbital_elements(last_year, bound[2], bound[1],
                        mu = gravitational_constant * 2e30) |>
    filter(time == max(time)) |>
    mutate(a_AU = a / distance_earth_sun) |>
    select(a_AU, e)
}

```

The survivors' orbit is a two-body ellipse, and from here on, unless the third star comes back, Chapter 4 describes them exactly. A typical outcome of a three-body breakup is a binary with a substantial eccentricity: the third body did not leave politely.

11.4. Why the Sun, Earth, and Moon are not chaotic

If three bodies are chaotic, how does the Moon manage? The Sun-Earth-Moon system is a three-body system, and the Moon has been in orbit for four billion years. The answer is *hierarchy*: the Moon is close to Earth and both are far from the Sun, so the Moon's orbit is a two-body orbit about Earth with a small perturbation, and Earth's is a two-body

11. Chaos and the Three-Body Problem

orbit about the Sun with a small perturbation. Chaos needs the three bodies to interact as equals.

How close is close enough? Think about it in the frame rotating with Earth around the Sun. A satellite of Earth is held by Earth's gravity, $Gm_{\oplus}/r^2$, and pulled away by the difference between the Sun's gravity at its position and at Earth's, plus the centrifugal effect of the rotation. Along the Sun-Earth line, that *tidal* acceleration works out to $3GM_{\odot}r/R^3$, where R is Earth's distance from the Sun (two parts from the Sun's gradient, one from the centrifugal term). The two balance at

$$r_H = R \left(\frac{m_{\oplus}}{3M_{\odot}} \right)^{1/3}, \quad (11.1)$$

the *Hill radius*. Inside it, Earth wins; outside it, the Sun does. For Earth it is

```
r_H <- distance_earth_sun * (mass_earth / (3 * mass_sun))^(1 / 3)
c(r_H_km = r_H / 1e3, in_lunar_distances = r_H / distance_earth_moon)
#>           r_H_km in_lunar_distances
#>      1.496418e+06      3.892866e+00
```

about four times the Moon's distance. The Moon sits at a quarter of the Hill radius, which is comfortably inside. Simulations of the restricted three-body problem show that orbits going the same way as the planet (prograde) are stable out to roughly half the Hill radius, and retrograde ones somewhat farther. Test it with two hypothetical moons, one at $0.3r_H$ and one at $1.0r_H$:

```
hill_test <- create_system() |>
  add_sun() |>
  add_body("Earth", mass = mass_earth, x = distance_earth_sun, vy = speed_earth) |>
  add_body("Inner", mass = 1e20, x = distance_earth_sun + 0.3 * r_H,
    vy = speed_earth + sqrt(gravitational_constant * mass_earth / (0.3 * r_H))) |>
  add_body("Outer", mass = 1e20, x = distance_earth_sun + 1.0 * r_H,
    vy = speed_earth + sqrt(gravitational_constant * mass_earth / r_H)) |>
  simulate_system(time_step = seconds_per_hour, duration = seconds_per_year * 5)

hill_test |>
  shift_reference_frame("Earth", keep_center = FALSE) |>
  filter(id != "Sun") |>
  mutate(r = sqrt(x^2 + y^2 + z^2) / r_H) |>
  ggplot(aes(time / seconds_per_year, r, color = id)) +
  geom_line() +
  scale_y_log10() +
  labs(x = "Years", y = "Distance from Earth (Hill radii)", color = NULL)
```

The inner moon's distance stays in a narrow band; the outer one wanders off within months and is thereafter a planet in its own right, orbiting the Sun near Earth. The Hill radius is the boundary of a planet's domain, and it explains a great deal: why the giant planets, with enormous Hill spheres, have dozens of moons each; why Mercury, deep in the Sun's gravity,

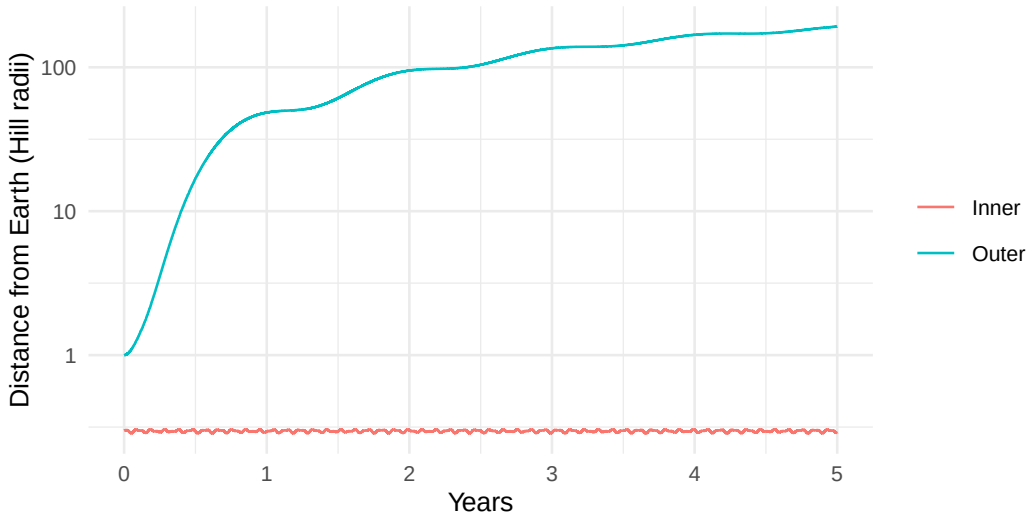


Figure 11.3.: Two test moons of Earth, started on circular orbits at 0.3 and 1.0 Hill radii. The inner one stays; the outer one is stripped by the Sun within the first orbit.

has none; and why the Moon's orbit, though perturbed by the Sun in a hundred measurable ways, is in no danger.

11.5. The Lagrange points, and which ones are traps

Chapter 10 found the L_4 point, where a small body can orbit with Jupiter indefinitely. There are five such points in the rotating frame of any circular binary. L_1 lies between the two bodies, L_2 beyond the smaller one, L_3 beyond the larger one, and L_4 and L_5 at the two equilateral positions. Only the last two are stable, and only when the mass ratio is small. L_1 , L_2 , and L_3 are *saddle points*: a body placed exactly there stays, but any displacement along one direction grows exponentially.

L_1 and L_2 are at approximately one Hill radius from the smaller body, toward and away from the larger one; Equation 11.1 is in fact the leading-order position of those two points. Put a test body at the Sun-Earth L_1 , moving so that it would co-rotate with Earth, and watch:

```
d      <- distance_earth_sun
omega <- sqrt(gravitational_constant * (mass_sun + mass_earth) / d^3)
M      <- mass_sun + mass_earth

earth_x <- d * mass_sun / M          # Earth's distance from the barycenter
L1      <- c(earth_x - r_H, 0)
v_L1    <- omega * c(-L1[2], L1[1])

l1_test <- create_system() |>
  add_body("Sun", mass = mass_sun, x = -d * mass_earth / M,
```

11. Chaos and the Three-Body Problem

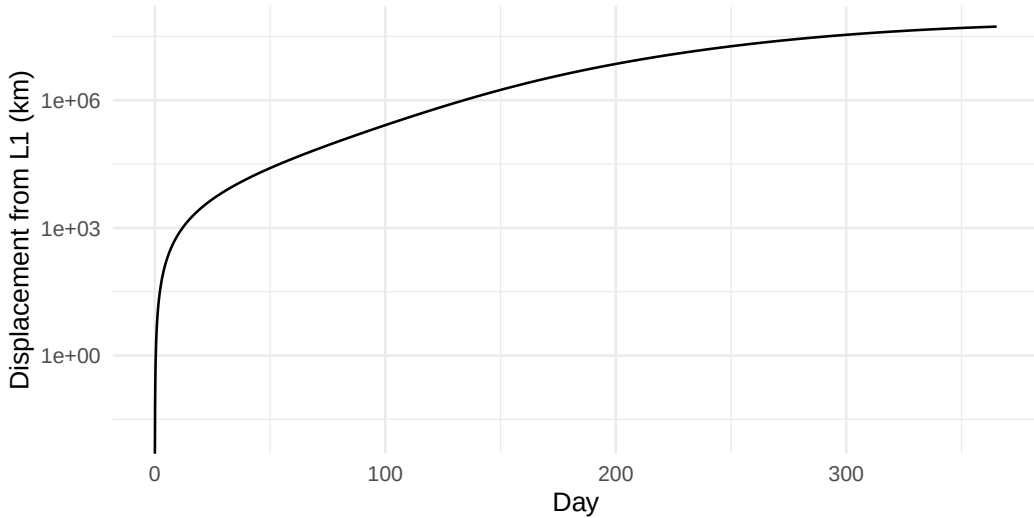


Figure 11.4.: A probe placed at the Sun-Earth L_1 point. Its displacement from L_1 in the rotating frame grows exponentially with an e-folding time of a few weeks.

```

      vy = -omega * d * mass_earth / M) |>
add_body("Earth", mass = mass_earth, x = earth_x, vy = omega * earth_x) |>
add_body("Probe", mass = 1e3, x = L1[1], vx = v_L1[1], vy = v_L1[2]) |>
simulate_system(time_step = seconds_per_hour, duration = seconds_per_year)

l1_test |>
  filter(id == "Probe") |>
  mutate(theta = omega * time,
         xr = x * cos(theta) + y * sin(theta),
         yr = -x * sin(theta) + y * cos(theta),
         displacement = sqrt((xr - L1[1])^2 + (yr - L1[2])^2)) |>
  ggplot(aes(time / seconds_per_day, displacement / 1e3)) +
  geom_line() +
  scale_y_log10() +
  labs(x = "Day", y = "Displacement from L1 (km)")

l1_growth <- l1_test |>
  filter(id == "Probe") |>
  mutate(theta = omega * time,
         xr = x * cos(theta) + y * sin(theta),
         yr = -x * sin(theta) + y * cos(theta),
         displacement = sqrt((xr - L1[1])^2 + (yr - L1[2])^2)) |>
  filter(time > seconds_per_day * 20, time < seconds_per_day * 120)

1 / coef(lm(log(displacement) ~ time, data = l1_growth))["time"] / seconds_per_day
#> [1] 19.98168

```

11.5. The Lagrange points, and which ones are traps

The e-folding time is a few weeks. (The probe does not start exactly at L_1 , because Equation 11.1 is only the first term of the exact position, so it begins drifting at once; a body placed with perfect precision would still be dislodged by the Moon, the other planets, and sunlight.) This is why the spacecraft parked at L_1 and L_2 , SOHO, Gaia, JWST, do not sit at the point but fly *halo orbits* around it, and fire their thrusters every few weeks to stay on them. The instability is slow enough to manage and fast enough that it can never be ignored.

Key results

- Nearby trajectories of a chaotic system separate as $e^{t/\tau}$; τ is the Lyapunov time. Beyond a few τ , a simulation predicts the statistics of the motion, not the motion.
- A body has escaped when its energy relative to the rest of the system is positive. Three-body breakups eject one body and leave a tighter, more eccentric binary. Total energy is conserved throughout.
- Hierarchical triples are stable. The Hill radius $r_H = R(m/3M)^{1/3}$ bounds a planet's domain; prograde satellites survive out to about half of it.
- L_1, L_2, L_3 are unstable saddle points with e-folding times of weeks (Sun-Earth); L_4, L_5 are stable for small mass ratios.

12. Conservation Laws as Diagnostics

A simulation can be wrong in ways that are invisible in a plot. An orbit that is an ellipse of the right size, traced at the right speed, can still be rotating slowly in its plane for no physical reason, and nothing about the picture will tell you. What *will* tell you is a set of quantities that the real system keeps constant and the simulated one may not. Chapter 5 used total energy this way; this chapter adds total momentum and total angular momentum, derives why all three are conserved for any number of bodies, explains which of them a good integrator conserves *exactly* and which only approximately, and turns them into a single function you can run on any simulation output. It ends with a worked diagnosis of a simulation that passes the energy test and is still wrong.

12.1. Three conserved quantities, for N bodies

Chapter 4 derived conservation of energy and angular momentum for two bodies. The N -body versions follow from two properties of gravity: it is *pairwise antisymmetric* ($\mathbf{F}_{jk} = -\mathbf{F}_{kj}$) and *central* ($\mathbf{F}_{jk}$ lies along $\mathbf{r}_k - \mathbf{r}_j$).

Momentum. $\mathbf{P} = \sum_j m_j \mathbf{v}_j$, and

$$\dot{\mathbf{P}} = \sum_j m_j \mathbf{a}_j = \sum_j \sum_{k \neq j} \mathbf{F}_{jk} = \sum_{j < k} (\mathbf{F}_{jk} + \mathbf{F}_{kj}) = \mathbf{0}.$$

Every force appears twice with opposite signs. The center of mass moves in a straight line at constant velocity, which is Chapter 2's drifting Earth.

Angular momentum. $\mathbf{L} = \sum_j m_j \mathbf{r}_j \times \mathbf{v}_j$, and since $\mathbf{v}_j \times \mathbf{v}_j = \mathbf{0}$,

$$\dot{\mathbf{L}} = \sum_j \mathbf{r}_j \times m_j \mathbf{a}_j = \sum_{j < k} (\mathbf{r}_j \times \mathbf{F}_{jk} + \mathbf{r}_k \times \mathbf{F}_{kj}) = \sum_{j < k} (\mathbf{r}_j - \mathbf{r}_k) \times \mathbf{F}_{jk} = \mathbf{0},$$

because $\mathbf{F}_{jk}$ is parallel to $\mathbf{r}_j - \mathbf{r}_k$. Antisymmetry collapses the two terms of each pair into one; centrality kills it.

Energy. $E = \sum_j \frac{1}{2} m_j v_j^2 - \sum_{j < k} G m_j m_k / r_{jk}$ (Equation 5.10). The kinetic term's rate of change is $\sum_j m_j \mathbf{v}_j \cdot \mathbf{a}_j = \sum_{j < k} \mathbf{F}_{jk} \cdot (\mathbf{v}_j - \mathbf{v}_k)$, again by antisymmetry. With $\mathbf{F}_{jk} = G m_j m_k (\mathbf{r}_k - \mathbf{r}_j) / r_{jk}^3$ and the identity $(\mathbf{r}_k - \mathbf{r}_j) \cdot (\mathbf{v}_k - \mathbf{v}_j) = r_{jk} \dot{r}_{jk}$ from Chapter 4,

$$\mathbf{F}_{jk} \cdot (\mathbf{v}_j - \mathbf{v}_k) = -\frac{G m_j m_k \dot{r}_{jk}}{r_{jk}^2},$$

12. Conservation Laws as Diagnostics

and the potential term's rate of change is $\sum_{j < k} G m_j m_k \dot{r}_{jk} / r_{jk}^2$, which cancels it pair by pair. With softening (Chapter 6), replace r_{jk} by $\sqrt{r_{jk}^2 + \varepsilon^2}$ in the potential and the same cancellation goes through.

Where it comes from

Each of the three laws is the shadow of a symmetry. Emmy Noether proved in 1918 that every continuous symmetry of a physical system's equations gives a conserved quantity: invariance under translation in space gives momentum, under rotation gives angular momentum, under translation in *time* gives energy. The N-body equations have all three symmetries, so they have all three laws.

A numerical integrator does not have all three. A fixed-step scheme respects translations and rotations perfectly: shift or rotate every position and velocity, run a step, and you get the shifted or rotated result, exactly. But it does not respect translation in time, because time comes in lumps of Δt . Noether's theorem then says exactly what to expect: a well-constructed integrator conserves momentum and angular momentum to rounding error, but energy only approximately. That is what the next section finds.

12.2. Which laws an integrator keeps exactly

Look at the integrators of Chapter 5 as compositions of *kicks* (velocity updates from accelerations computed at fixed positions) and *drifts* (position updates at fixed velocities).

A kick changes $\mathbf{P}$ by $\Delta t \sum_j m_j \mathbf{a}_j = \Delta t \sum_j \mathbf{F}_j = \mathbf{0}$ and $\mathbf{L}$ by $\Delta t \sum_j \mathbf{r}_j \times m_j \mathbf{a}_j = \mathbf{0}$, by the same pairwise arguments as above, *exactly*, for any Δt . A drift changes neither: $\mathbf{P}$ does not involve positions, and $\mathbf{L}$ changes by $\Delta t \sum_j m_j \mathbf{v}_j \times \mathbf{v}_j = \mathbf{0}$. So any method that is a sequence of kicks and drifts, which is Euler-Cromer (kick, drift) and Velocity Verlet (half kick, drift, half kick), conserves momentum and angular momentum to machine precision no matter how large the step.

Plain Euler is not a kick followed by a drift; it updates positions with the *old* velocities and velocities with the *old* accelerations simultaneously. Momentum still survives, but angular momentum picks up $\Delta t^2 \sum_j m_j \mathbf{v}_j \times \mathbf{a}_j$ per step, which does not vanish. So the three laws grade the three methods differently:

Table 12.1.: What each integrator conserves (exact means to rounding error).

	Euler	Euler-Cromer	Verlet
Momentum	exact	exact	exact
Angular momentum	$O(\Delta t)$ drift	exact	exact
Energy	$O(\Delta t)$ drift	bounded, $O(\Delta t)$	bounded, $O(\Delta t^2)$

The table is the key to reading diagnostics. A momentum or angular momentum error that grows past rounding in a Verlet run is not a time-step problem; it is a bug, in your setup or in the code. An energy error is a time-step problem (or an integrator problem), and its *shape* says which.

12.3. Computing all three

The package has one function per quantity, and each is a single grouped sum over the output tibble. Chapter 5 wrote out the energy; momentum and angular momentum are shorter still, and this is all `get_momentum()` and `get_angular_momentum()` do:

```
sim |>
  group_by(time) |>
  summarise(px = sum(mass * vx), py = sum(mass * vy), pz = sum(mass * vz))

sim |>
  group_by(time) |>
  summarise(Lx = sum(mass * (y * vz - z * vy)),
            Ly = sum(mass * (z * vx - x * vz)),
            Lz = sum(mass * (x * vy - y * vx)))
```

`conserved_quantities()` joins the three into one tidy time series and adds the relative errors. For energy that is $(E - E_0)/|E_0|$ as before. Momentum is often exactly zero at the start, so its error is measured against the total $\sum_j m_j |\mathbf{v}_j|$ at $t = 0$ instead; angular momentum against $|\mathbf{L}_0|$. On Chapter 5's star and planet with the three methods:

```
star_planet <- create_system() |>
  add_body("Star", mass = 1e30) |>
  add_body("Planet", mass = 1e24, x = 1e11, vy = 30000)

runs <- bind_rows(lapply(c("verlet", "euler_cromer", "euler"), function(m) {
  star_planet |>
    simulate_system(time_step = seconds_per_hour * 6,
                    duration = seconds_per_year * 10, method = m) |>
    conserved_quantities() |>
    mutate(method = m)
})))

runs |>
  select(time, method, energy_error, momentum_error, angular_momentum_error) |>
  tidyr::pivot_longer(c(energy_error, momentum_error, angular_momentum_error),
                     names_to = "quantity", values_to = "error") |>
  mutate(quantity = sub("_error$", "", quantity)) |>
  ggplot(aes(time / seconds_per_year, error)) +
  geom_line() +
  facet_grid(quantity ~ method, scales = "free_y") +
  labs(x = "Years", y = "Relative error")
```

Every panel in the table above has its picture here. The momentum row is noise at the 10^{-16} level for all three methods. The angular momentum row is noise for Verlet and Euler-Cromer and a steady climb for Euler. The energy row is Chapter 5's figure again.

12. Conservation Laws as Diagnostics

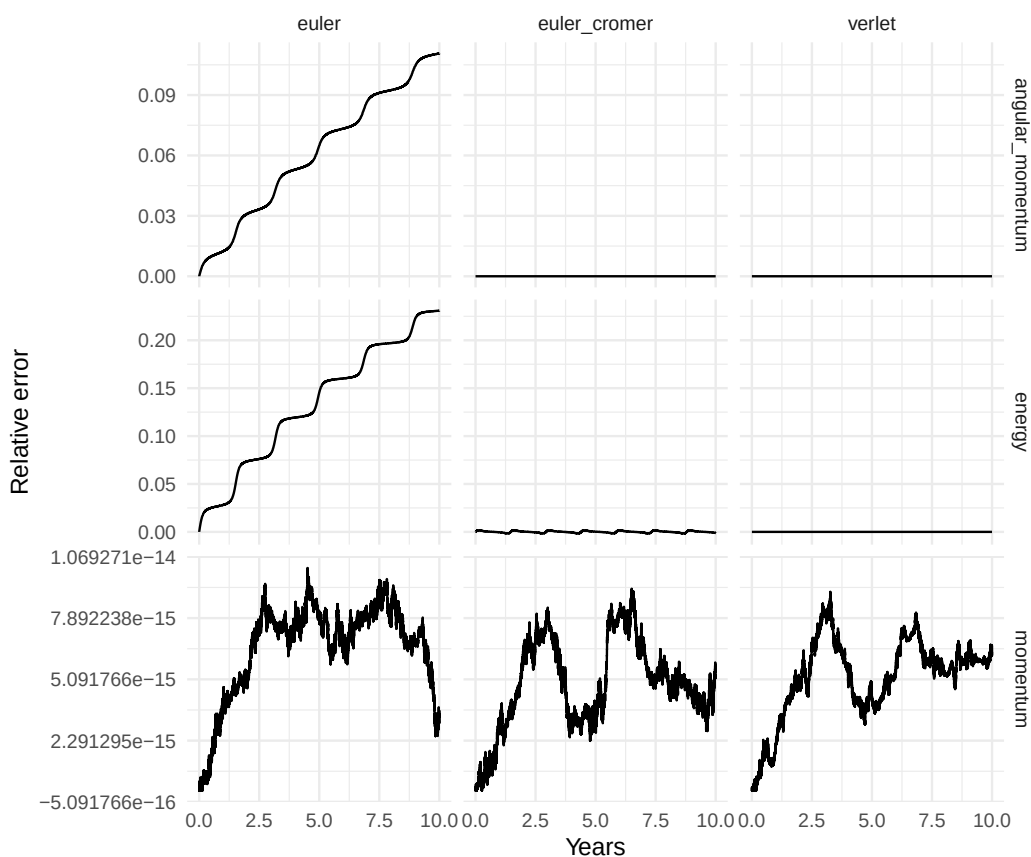


Figure 12.1.: Relative errors in energy, momentum, and angular momentum for ten years of the same orbit with three integrators. Momentum is conserved to rounding by all three; angular momentum by the two symplectic methods; energy by none, but Verlet's error stays in a narrow band. Note the free vertical scales.

12.4. Reading the plots

Four shapes cover nearly everything you will see in an energy diagnostic.

A narrow band, oscillating with the orbit. Healthy. The band's width is the integrator's $O(\Delta t^2)$ error and halving the step should shrink it by four. Nothing to do.

A wide band, still oscillating. The step is too large for the tightest or most eccentric orbit in the system. The run is probably not catastrophically wrong, but the next section shows that “bounded energy error” is a weaker guarantee than it sounds. Halve the step and compare.

A steady drift. Either the wrong integrator (Euler), or a step so large that the symplectic integrator's shadow-Hamiltonian guarantee has broken down, which happens when Δt approaches the periapsis timescale of some orbit (Chapter 13). Check `method`, then halve the step.

A sudden jump. A close encounter. Two bodies passed each other faster than the step could follow and the integrator handed one of them a spurious kick (Chapter 6). Everything after the jump is on a different orbit from everything before. Either the encounter needs a smaller step (Chapter 13's segmented runs) or it is a collision in disguise and the simulation should not continue past it.

A fifth shape is an apparent drift that is not real: you ran with `softening` and computed the energy without it. The softened force conserves the softened energy; `conserved_quantities()` reads the softening length from the run, so this only bites if you compute the energy by hand or override the argument.

12.5. A worked diagnosis: the orbit that looked fine

Mercury's perihelion precesses by 574 arcseconds per century, of which 531 come from the other planets and 43 from general relativity. Suppose you wanted to measure the planetary part with `orbitr`. The first question is what the integrator does to Mercury's perihelion when there are *no* other planets, where the correct answer is zero precession. Two bodies, ten years, with daily and five-day steps:

```
mercury_run <- function(step_days) {
  create_system() |>
  add_sun() |>
  add_planet("Mercury", parent = "Sun") |>
  simulate_system(time_step = seconds_per_day * step_days,
                  duration = seconds_per_year * 10)
}

perihelion_angle <- function(sim) {
  # longitude of perihelion = node + argument of perihelion, unwrapped so it
  # can be tracked through 360 degrees
  get_orbital_elements(sim, "Mercury", "Sun") |>
  mutate(omega_deg = unwrap((lan + arg_pe) * pi / 180) * 180 / pi) |>
```

12. Conservation Laws as Diagnostics

```
      select(time, omega_deg)
    }

runs_m <- bind_rows(lapply(c(5, 2, 1), function(s) {
  sim <- mercury_run(s)
  perihelion_angle(sim) |>
    mutate(step_days = s,
           energy_error = conserved_quantities(sim)$energy_error)
}))
```

(The longitude of perihelion, $\Omega + \omega$, is the quantity to track rather than ω alone, because for Mercury's small inclination the node is poorly defined and the two angles can trade against each other while their sum stays put.)

```
bind_rows(
  runs_m |> group_by(step_days) |>
    transmute(time, panel = "Perihelion direction (deg)",
              value = omega_deg - first(omega_deg)),
  runs_m |> group_by(step_days) |>
    transmute(time, panel = "Relative energy error", value = energy_error)
) |>
  ungroup() |>
  ggplot(aes(time / seconds_per_year, value, color = factor(step_days))) +
  geom_line() +
  facet_wrap(~ panel, scales = "free_y") +
  labs(x = "Years", y = NULL, color = "Step (days)")
```

The energy error is bounded at every step size, exactly as advertised. And the perihelion of the five-day run rotates through tens of degrees in a decade, for no physical reason at all. The daily run rotates too, more slowly. This is *numerical precession*: the integrator's shadow Hamiltonian is not Kepler's, and the difference, though it conserves energy, does not conserve the eccentricity vector. It is the single most common artifact in orbital simulation, and the energy test is blind to it.

Measure the rate at each step:

```
rates <- runs_m |>
  group_by(step_days) |>
  summarise(deg_per_year = coef(lm(omega_deg ~ I(time / seconds_per_year)))[[2]])
rates
#> # A tibble: 3 x 2
#>   step_days deg_per_year
#>   <dbl>      <dbl>
#> 1         1      -2.18
#> 2         2     -8.49
#> 3         5    -45.1
```

Halving the step cuts the rate by about four: the artifact is $O(\Delta t^2)$, like everything else about Verlet. That scaling is also the cure, because it lets you extrapolate. The real

12.5. A worked diagnosis: the orbit that looked fine

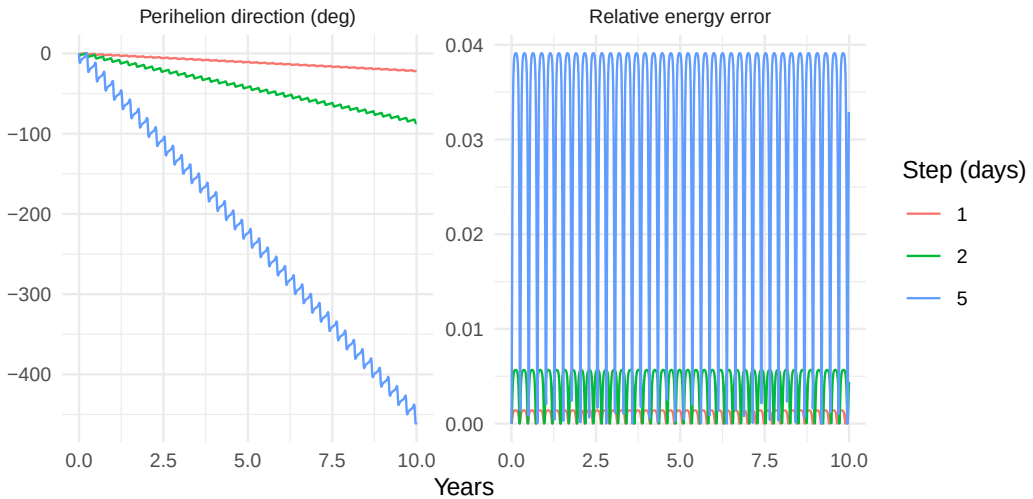


Figure 12.2.: Left: Mercury’s perihelion direction in a two-body run, which should be a flat line, at three step sizes. Right: the energy error of the same runs, which is bounded in every case. Bounded energy did not prevent the orbit from rotating.

planetary precession is 531” per century, or 0.0015° per year. To get the artifact ten times smaller than the signal you would need a step of about

```
rate_1d <- rates$deg_per_year[rates$step_days == 1]
target <- 0.0015 / 10
sqrt(target / abs(rate_1d)) * 24          # hours
#> [1] 0.1992759
```

hours. That is a smaller step than anyone would guess from looking at the orbit, and it is what Chapter 8’s last exercise was getting at when it asked what limits the precision. The general lesson is worth stating plainly: an energy diagnostic tells you the integrator is stable. It does not tell you the integrator is accurate. For accuracy, find a quantity the true dynamics conserves that the integrator does not, the eccentricity vector is the usual one, and watch that.

Key results

- N -body gravity conserves $\mathbf{P}$, $\mathbf{L}$, and E because it is pairwise antisymmetric, central, and derived from a potential.
- Kick-drift integrators (Euler-Cromer, Verlet) conserve $\mathbf{P}$ and $\mathbf{L}$ to rounding error at any step size; a drift in either means a bug. Energy is conserved only approximately, and the shape of its error says what is wrong.
- Bounded energy error does not imply an accurate orbit. Numerical precession of the periapsis is $O(\Delta t^2)$ and invisible to the energy test; track the eccentricity vector.

13. Comets and Highly Eccentric Orbits

Everything in the solar system chapter was nearly circular, and nearly circular orbits are easy: the speed hardly changes, so one time step is as good as another. A comet is the opposite case. Halley's Comet spends seventy years crawling through the outer solar system and a few months sprinting around the Sun at sixty times the speed, and a step that is wasteful at aphelion is hopeless at perihelion. This chapter sets Halley up from its orbital elements, shows exactly how and where a fixed step fails on it, and then builds the tool the package does not have: a way to take small steps only when they are needed. It closes with two orbits that are not ellipses at all.

13.1. Halley

Halley's Comet has semi-major axis 17.8 AU, eccentricity 0.967, and an inclination of 162° , which is to say it orbits the Sun *backward*, tilted 18° from the ecliptic. Its perihelion is 0.586 AU, inside Venus's orbit; its aphelion is 35 AU, beyond Neptune's. Chapter 7's conversion handles all of that:

```
AU      <- distance_earth_sun
mu_sun  <- gravitational_constant * mass_sun

halley <- list(a = 17.834 * AU, e = 0.96714, i = 162.26,
              lan = 58.42, arg_pe = 111.33)

halley_system <- create_system() |>
  add_sun() |>
  add_body_keplerian("Halley", mass = 2.2e14, parent = "Sun",
                    a = halley$a, e = halley$e, i = halley$i,
                    lan = halley$lan, arg_pe = halley$arg_pe, nu = 180)
```

`nu = 180` starts the comet at aphelion, where it is moving slowest. Vis-viva gives the speeds at both ends of the orbit and Kepler's third law the period:

```
q <- halley$a * (1 - halley$e)      # perihelion distance
Q <- halley$a * (1 + halley$e)      # aphelion distance

c(v_perihelion_km_s = sqrt(mu_sun * (2 / q - 1 / halley$a)) / 1e3,
  v_aphelion_km_s   = sqrt(mu_sun * (2 / Q - 1 / halley$a)) / 1e3,
  period_years      = 2 * pi * sqrt(halley$a^3 / mu_sun) / seconds_per_year)
```

13. Comets and Highly Eccentric Orbits

```
#> v_perihelion_km_s    v_aphelion_km_s    period_years
#>      54.577537      0.911688      75.305411
```

Fifty-five kilometers per second at perihelion, under one at aphelion: the ratio $(1+e)/(1-e)$ from Chapter 7. The timescale that matters for the integrator is the perihelion distance divided by the perihelion speed,

```
tau_peri <- q / sqrt(mu_sun * (2 / q - 1 / halley$a))
tau_peri / seconds_per_day
#> [1] 18.59175
```

about 19 days. By Chapter 5's rule a step should be a small fraction of that, so a day is marginal and a week is wrong. Meanwhile the orbit takes 75 years, and a one-day step over 75 years is 27,000 steps for a passage that occupies a few hundred of them.

13.2. Where a fixed step fails

Run one full orbit at four step sizes and watch the energy:

```
T_halley <- 2 * pi * sqrt(halley$a^3 / mu_sun)

halley_runs <- bind_rows(lapply(c(1, 2, 4, 8), function(step_days) {
  halley_system |>
    simulate_system(time_step = seconds_per_day * step_days,
                    duration = T_halley) |>
    conserved_quantities() |>
    mutate(step_days = step_days)
})))

halley_runs |>
  ggplot(aes(time / seconds_per_year, energy_error)) +
  geom_line() +
  facet_wrap(~ step_days, scales = "free_y",
             labeller = labeller(step_days = function(s) paste(s, "day step"))) +
  labs(x = "Years", y = "Relative energy error")
```

The shape is instructive. For the first 37 years the error is essentially zero at every step size, because far from the Sun the acceleration changes slowly and even an 8-day step is tiny compared with the local timescale of years. Then, in the few weeks around perihelion, each run makes all of its error at once. For the 1-day step the kick is small and the comet comes out on nearly the right orbit. For the 8-day step the kick is a substantial fraction of the comet's binding energy, which means the comet leaves perihelion on a *different orbit*, with a different period, and every subsequent apparition is at the wrong time.

The error is $O(\Delta t^2)$ in the usual way, as the ratios between panels show, but that is cold comfort when the only way to shrink it is to spend 27,000 steps on an orbit that needs a thousand. Two of the three possible responses are bad.

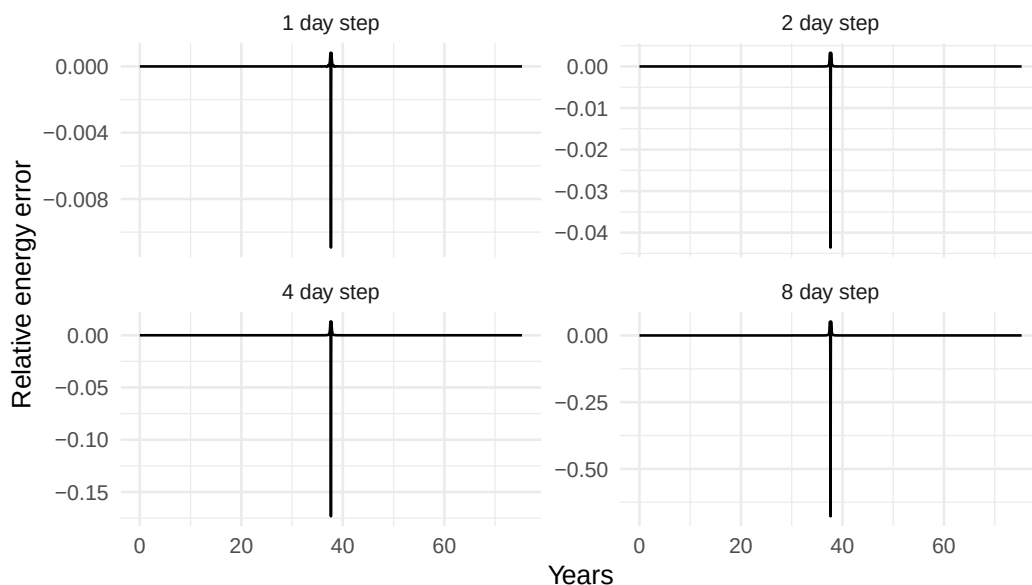


Figure 13.1.: Relative energy error over one 75-year orbit of Halley’s Comet, started at aphelion, at four step sizes. Nothing happens for 37 years; then the comet passes perihelion and the integrator with the 8-day step hands it a large spurious kick.

Softening is not the answer. It would cap the force near the Sun, which is exactly the force that is correct. Halley does not hit the Sun. Softening would change the orbit to a wrong one at every perihelion, smoothly and forever.

A uniform small step works but wastes effort. A 6-hour step is safely inside the perihelion timescale and costs 110,000 steps per orbit. For one comet that is fine, a minute or two of computing. For a thousand comets, or a comet with a period of a thousand years, it is not.

13.3. Segmented runs

The right response is to take small steps near perihelion and large steps elsewhere. `simulate_system()` integrates with one fixed step, but nothing stops you from running it several times, because the last row of a simulation tibble is a complete state. `continue_simulation()` does the bookkeeping: it rebuilds the system from the final time step (`system_from_simulation()` does that part on its own, if you want to edit the system between segments), runs it onward with whatever step you give it, and appends the new rows with `time` continuing from where the old run ended. The gravitational constant, integrator, and softening carry over from the previous segment unless you say otherwise.

Each segment is an ordinary Verlet run with all of Chapter 5’s guarantees; only at the switches is there any disturbance, and it is a single step’s worth.

13. Comets and Highly Eccentric Orbits

When to switch? Say we want the small step inside 5 AU. Kepler's equation, run in reverse, gives the time from perihelion to any true anomaly, and the true anomaly at $r = 5$ AU follows from the orbit equation, $\cos \nu = (p/r - 1)/e$ (the inverse-direction helper `time_since_periapsis()` is in `R/helpers.R`):

```
p      <- halley$a * (1 - halley$e^2)
nu_5AU <- acos((p / (5 * AU) - 1) / halley$e) * 180 / pi
t_inside <- time_since_periapsis(nu_5AU, halley$a, halley$e, mu_sun)

c(nu_at_5AU_deg = nu_5AU, years_inside_5AU_each_way = t_inside / seconds_per_year)
#>                nu_at_5AU_deg years_inside_5AU_each_way
#>                142.71047                1.02473
```

Halley spends about a year on each side of perihelion inside 5 AU, and 73 years outside. Three segments, then: coarse from aphelion to the 5 AU crossing, fine through perihelion, coarse back out:

```
coarse <- seconds_per_day * 8
fine   <- seconds_per_hour * 6
t_out  <- T_halley / 2 - t_inside          # aphelion to the inbound crossing

segmented <- halley_system |>
  simulate_system(time_step = coarse, duration = t_out) |>
  continue_simulation(time_step = fine, duration = 2 * t_inside) |>
  continue_simulation(time_step = coarse, duration = t_out)

n_distinct(segmented$time)
#> [1] 6339
```

A few thousand steps instead of 110,000, and the energy:

```
bind_rows(
  conserved_quantities(segmented) |> mutate(run = "Segmented: 8 d / 6 h / 8 d"),
  halley_runs |> filter(step_days == 8) |> mutate(run = "Uniform 8 d")
) |>
  ggplot(aes(time / seconds_per_year, energy_error, color = run)) +
  geom_line() +
  labs(x = "Years", y = "Relative energy error", color = NULL)

c(segmented = max(abs(conserved_quantities(segmented)$energy_error)),
  uniform_8_day = max(abs(filter(halley_runs, step_days == 8)$energy_error)),
  uniform_1_day = max(abs(filter(halley_runs, step_days == 1)$energy_error)))
#>      segmented uniform_8_day uniform_1_day
#> 0.0003968142 0.6776209618 0.0109308137
```

The segmented run beats the uniform daily run on accuracy at a fraction of its cost. The idea generalizes: for a comet with many perihelion passages, loop over them; for a system

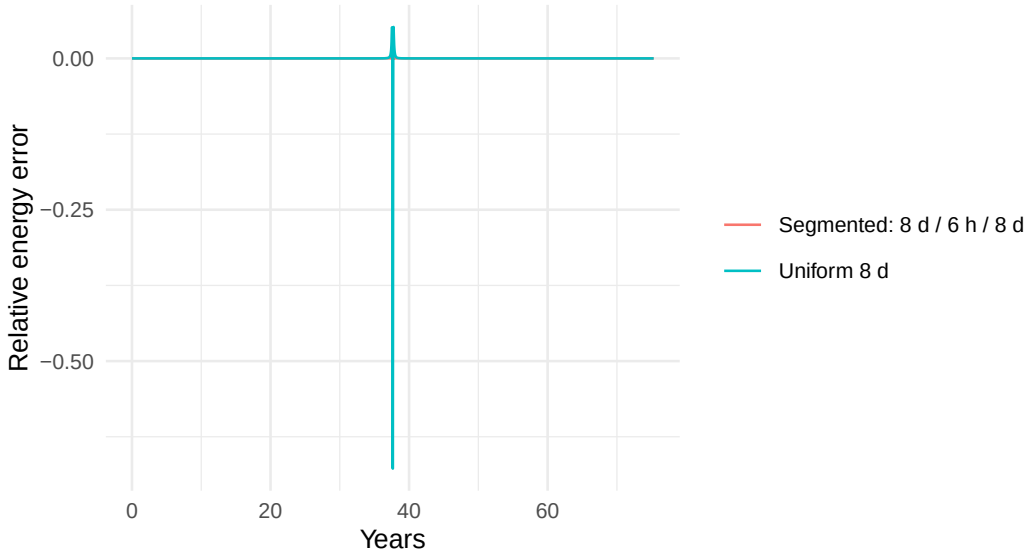


Figure 13.2.: Energy error for the segmented run (8-day steps outside 5 AU, 6-hour steps inside) compared with a uniform 8-day run. The segmented run resolves the perihelion passage at a small fraction of the cost of a uniform fine step.

with one body that needs fine steps and many that do not, you still pay for the fine steps on every body (the package integrates them all together), which is the limitation that true adaptive-step codes remove. Chapter 16 says why they are hard to do without losing Verlet’s symplectic guarantee.

Where it comes from

Halley’s Comet is the reason we know comets are periodic at all. Edmond Halley, using Newton’s new method, computed orbits for two dozen historical comets and noticed that the ones of 1531, 1607, and 1682 had nearly the same elements. He predicted a return in 1758, sixteen years after his own death, and when it came the question of whether comets obeyed gravity was settled. Alexis Clairaut refined the prediction to within a month by computing the perturbations of Jupiter and Saturn, by hand, over the preceding orbit, which was the first serious numerical integration in celestial mechanics and took him and two assistants most of a year.

13.4. Hyperbolic visitors

In 2017 an object was found on an orbit with $e = 1.2$. It was not bound to the Sun at all; it had come from interstellar space and was on its way back out. ‘Oumuamua passed perihelion at 0.255 AU and was gone within months.

The same six elements describe a hyperbola, with one twist: its semi-major axis is *negative*. The periapsis distance $a(1-e)$ stays positive, since both factors flip sign, and with $a < 0$ and

13. Comets and Highly Eccentric Orbits

$e > 1$ the semi-latus rectum $p = a(1 - e^2)$ is positive, $h = \sqrt{\mu p}$ is real, and Equation 7.1 and Equation 7.2 work unchanged. `add_body_keplerian()` accepts $e > 1$ when `a` is negative, provided the starting true anomaly is within the asymptotes, $|\nu| < \arccos(-1/e)$ (it refuses $e = 1$ exactly, where a would be infinite):

```
e_ou <- 1.1995
q_ou <- 0.2553 * AU
a_ou <- -q_ou / (e_ou - 1)                                # negative for a hyperbola

c(asymptote_deg = acos(-1 / e_ou) * 180 / pi,
  deflection_deg = 2 * asin(1 / e_ou) * 180 / pi)
#> asymptote_deg deflection_deg
#>      146.4787      112.9574

oumuamua <- create_system() |>
  add_sun() |>
  add_body_keplerian("Oumuamua", mass = 1e10, parent = "Sun",
    a = a_ou, e = e_ou, i = 122.7, lan = 24.6,
    arg_pe = 241.8, nu = -140) |>
  simulate_system(time_step = seconds_per_hour, duration = seconds_per_year * 2.4)

rel <- relative_state(oumuamua, "Oumuamua", "Sun") |>
  mutate(r = sqrt(rx^2 + ry^2 + rz^2), v = sqrt(ux^2 + uy^2 + uz^2),
    energy = v^2 / 2 - mu / r)

c(r_start_AU = first(rel$r) / AU, r_end_AU = last(rel$r) / AU,
  v_max_km_s = max(rel$v) / 1e3,
  v_perihelion_predicted = sqrt(mu_sun * (1 + e_ou) / q_ou) / 1e3,
  energy_min = min(rel$energy), energy_max = max(rel$energy))
#>      r_start_AU      r_end_AU      v_max_km_s
#>      6.921416e+00      1.061724e+01      8.743576e+01
#> v_perihelion_predicted      energy_min      energy_max
#>      8.743615e+01      3.466985e+08      3.467151e+08
```

The specific energy is positive and constant, the maximum speed is the vis-viva prediction at perihelion ($v_p^2 = \mu(1 + e)/q$, from $a \rightarrow$ negative in Equation 4.11), and the eccentricity at the end is the eccentricity at the start:

```
range(get_orbital_elements(oumuamua, "Oumuamua", "Sun")$e)
#> [1] 1.199492 1.199501
```

The object came in at 26 km/s relative to the Sun and left at 26 km/s, deflected through the angle $2\arcsin(1/e)$ of about 113 degrees. (The real 'Oumuamua showed a small extra acceleration away from the Sun that gravity alone cannot produce, most likely outgassing, which is a non-gravitational force of the kind Chapter 16 discusses.)

13.5. Sungrazers

The Kreutz family of comets, hundreds of which have been seen by solar observatories, have perihelia around 0.005 AU, about one solar radius. Vis-viva at that distance gives

```
R_sun <- 6.957e8
sqrt(2 * mu_sun / R_sun) / 1e3
#> [1] 617.7664
```

kilometers per second, over six hundred, which is the escape speed from the Sun’s surface, since these comets arrive on nearly parabolic orbits. Most of them do not survive. The simulation would carry them through without comment: the Sun is a point, the comet is a point, and two points can pass within a solar radius, or within a meter, and the integrator will either resolve it (with a step of minutes) or produce the spurious kick of Chapter 6. The package roadmap’s **radius** argument and collision handling are the fix; until then, a Kreutz comet is a case where the physics the package implements and the physics that happens part company, and the simulation’s answer should be read as “it would have hit.”

Key results

- A fixed step must resolve the perihelion timescale q/v_p , where $v_p^2 = \mu(1+e)/q$; for Halley that is weeks, for a sungrazer minutes.
- A too-large step makes its whole error in one perihelion kick and leaves the comet on a different orbit.
- `continue_simulation()` lets you change the step between segments; Kepler’s equation tells you when to switch. Softening is never the fix.
- Hyperbolic orbits ($e > 1$, $a < 0$) work in `add_body_keplerian()`; the visitor leaves with the speed it came with, deflected by $2 \arcsin(1/e)$.

14. Figures for Papers and Talks

The plotting functions in `orbitr` are built to get you a picture in one line so that you can get back to the physics. That is the right design for exploring and the wrong one for publishing, where the picture needs the units, labels, colors, and size that your journal or your slides demand. The good news is that there is no gap between the two: `plot_orbits()` returns a `ggplot` you can keep modifying, and everything else is a tibble you can plot however you like. This chapter is the short, practical one. It collects the plotting patterns the rest of the book has been using, adds the ones a paper or a talk needs, and says where the interactive and animated output fits.

14.1. `plot_orbits()` is a `ggplot`

Anything you can add to a `ggplot` you can add to the output of `plot_orbits()`. The most common fixes are a marker for the central body, which is otherwise invisible because it barely moves; axis labels in sensible units; and a theme:

```
AU <- distance_earth_sun

inner <- 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(inner, three_d = FALSE) +
  annotate("point", x = 0, y = 0, size = 3) +
  scale_x_continuous(labels = function(m) m / AU) +
  scale_y_continuous(labels = function(m) m / AU) +
  labs(x = "x (AU)", y = "y (AU)", color = NULL)
```

The scale functions relabel the axes without touching the data, which keeps `coord_equal()` honest. If you would rather convert the data, do it before plotting, as the next section does.

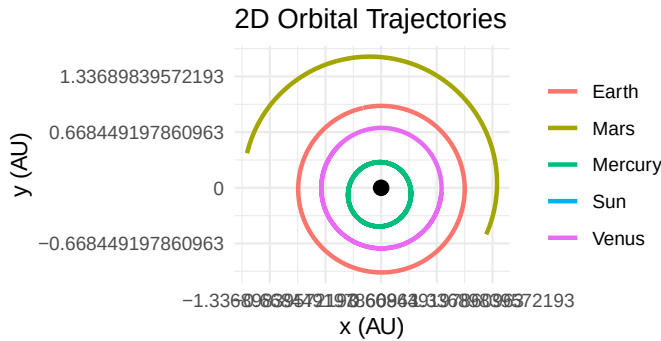


Figure 14.1.: The default plot with three additions: a marker for the Sun, axis labels, and a scale in AU.

14.2. Your own plots from the tibble

The built-in plot draws one path per body. Everything else is a `mutate()` and a `ggplot()`. Four patterns cover most needs.

Color by time, to show direction and speed along a path:

```
create_system() |>
  add_sun() |>
  add_planet("Earth", parent = "Sun") |>
  add_planet("Mars", parent = "Sun", nu = 60) |>
  simulate_system(time_step = seconds_per_day, duration = seconds_per_year * 2.2) |>
  shift_reference_frame("Earth", keep_center = FALSE) |>
  filter(id == "Mars") |>
  ggplot(aes(x / AU, y / AU, color = time / seconds_per_year)) +
  geom_path(linewidth = 0.9) +
  scale_color_viridis_c() +
  coord_equal() +
  labs(x = "x (AU)", y = "y (AU)", color = "Years")
```

Time series of any derived quantity: distance, speed, energy, an orbital element. The pattern is always `mutate()` the quantity, then `aes(time, quantity)`:

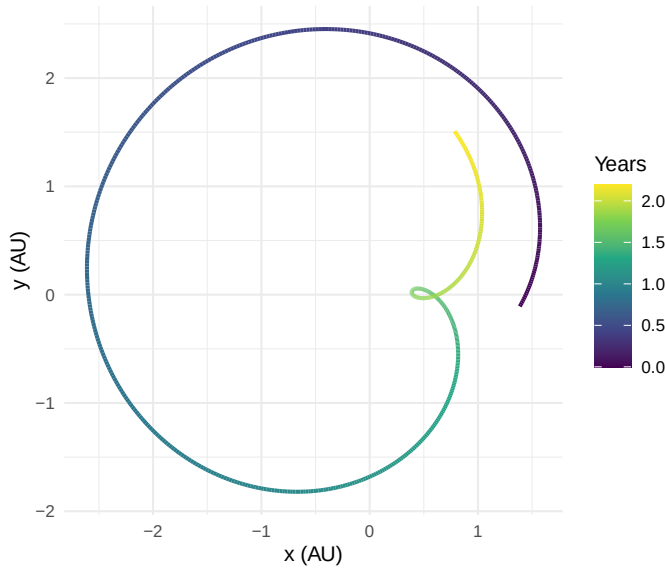


Figure 14.2.: Mars seen from Earth over two years, colored by time. The color gradient makes the direction of motion and the retrograde loop readable without an animation.

```
create_system() |>
  add_sun() |>
  add_planet("Mercury", parent = "Sun") |>
  simulate_system(time_step = seconds_per_hour * 6, duration = seconds_per_day * 88)
  shift_reference_frame("Sun", keep_center = FALSE) |>
  transmute(day = time / seconds_per_day,
            `Distance (AU)` = sqrt(x^2 + y^2 + z^2) / AU,
            `Speed (km/s)` = sqrt(vx^2 + vy^2 + vz^2) / 1e3) |>
  tidyr::pivot_longer(-day) |>
  ggplot(aes(day, value)) +
  geom_line() +
  facet_wrap(~ name, scales = "free_y", ncol = 1) +
  labs(x = "Day", y = NULL)
```

Small multiples with `facet_wrap()`, which is how most of this book's comparisons were drawn: give each run a label column with `mutate(run = "...")`, `bind_rows()` them, and facet by it. Chapter 5's integrator comparison and Chapter 6's encounter figure are both this pattern.

Snapshots in a grid. `plot_system()` draws one instant. For a sequence of instants in a paper, where an animation is not an option, overlay the full paths in grey and facet the positions by time:

```
days <- c(0, 90, 180, 270)
```

14. Figures for Papers and Talks

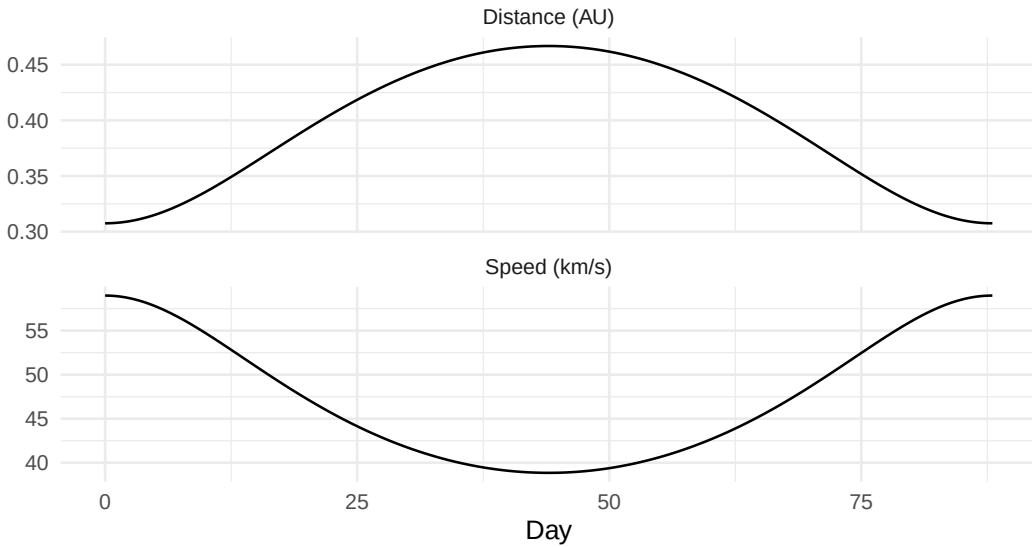


Figure 14.3.: Mercury’s heliocentric distance and speed over one orbit, as time series. Two panels from one tibble via `pivot_longer()`.

```
snapshots <- bind_rows(lapply(days, function(d) {
  inner |>
    filter(abs(time - d * seconds_per_day) < seconds_per_day / 2) |>
    mutate(day = paste("Day", d))
}))

ggplot() +
  geom_path(data = inner, aes(x / AU, y / AU, group = id), color = "grey80") +
  geom_point(data = snapshots, aes(x / AU, y / AU, color = id), size = 2) +
  annotate("point", x = 0, y = 0, size = 2) +
  coord_equal() +
  facet_wrap(~ day) +
  labs(x = "x (AU)", y = "y (AU)", color = NULL)
```

14.3. A figure for a paper

A journal figure has a few requirements that an exploratory plot does not: a fixed physical size, a font size that will still be legible after the figure is shrunk to a column, a color scheme that survives greyscale printing and colorblind readers, and vector output. Here is a recipe:

```
star_planet <- create_system() |>
  add_body("Star", mass = 1e30) |>
  add_body("Planet", mass = 1e24, x = 1e11, vy = 30000)
```

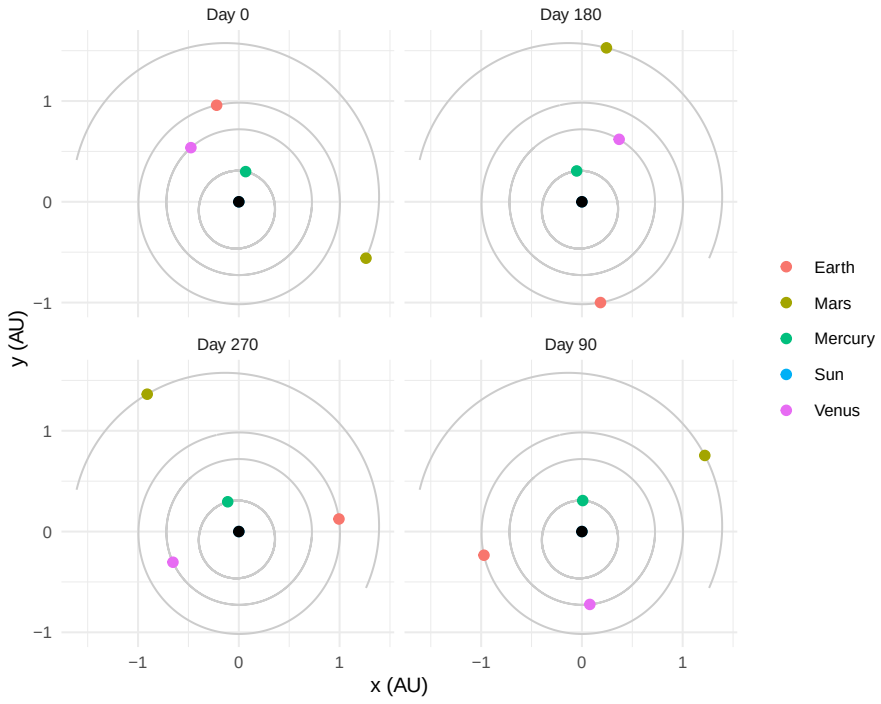


Figure 14.4.: The inner solar system at four instants, with the full year's paths behind. A static substitute for an animation.

14. Figures for Papers and Talks

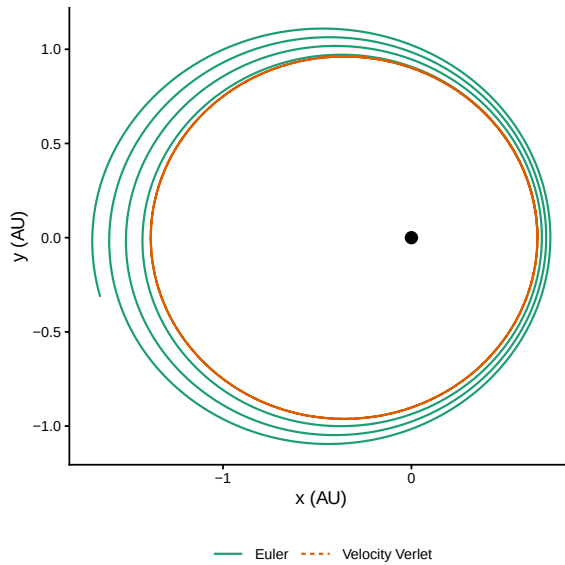


Figure 14.5.: A publication-style version of Chapter 5's integrator comparison: colorblind-safe palette, distinct line types so the figure works in greyscale, a classic theme, and labels in the units a reader expects.

```
comparison <- bind_rows(lapply(c("verlet", "euler"), function(m) {  
  star_planet |>  
    simulate_system(time_step = seconds_per_hour * 6,  
                    duration = seconds_per_year * 6, method = m) |>  
    filter(id == "Planet") |>  
    mutate(method = c(verlet = "Velocity Verlet", euler = "Euler")[m])  
}))
```

```
paper_figure <- comparison |>  
  ggplot(aes(x / AU, y / AU, color = method, linetype = method)) +  
  geom_path(linewidth = 0.5) +  
  annotate("point", x = 0, y = 0, size = 2.5) +  
  scale_color_brewer(palette = "Dark2") +  
  coord_equal() +  
  labs(x = "x (AU)", y = "y (AU)", color = NULL, linetype = NULL) +  
  theme_classic(base_size = 9) +  
  theme(legend.position = "bottom")
```

```
paper_figure
```

Save it at the size it will be printed, as a vector file:

```
ggsave("integrators.pdf", paper_figure, width = 3.4, height = 3.0, units = "in")
ggsave("integrators.png", paper_figure, width = 3.4, height = 3.0, units = "in",
      dpi = 600)
```

A single-column figure in most journals is about 3.4 inches wide; set `base_size` so that the text is 7 to 9 points at that size. The PDF is what you submit; the PNG is for the slide.

Two things to watch. Orbit plots need `coord_equal()`, or circles become ellipses and the eye reads an eccentricity that is not there. And when the data span orders of magnitude, as a comet's distance does, a logarithmic axis (`scale_y_log10()`) is usually right, as it was in Chapter 11.

14.4. Interactive 3D for the screen

Any system with motion out of the plane dispatches to `plotly` automatically, and `plot_orbits_3d()` and `plot_system_3d()` force it:

```
load_solar_system() |>
  remove_body("Moon") |>
  simulate_system(time_step = seconds_per_day, duration = seconds_per_year * 30) |>
  plot_orbits_3d()
```

You can rotate, zoom, and hover for the time stamp. For a talk, the widget can be saved as a self-contained HTML file and opened in any browser:

```
p <- plot_orbits_3d(sim)
htmlwidgets::saveWidget(p, "solar-system-3d.html", selfcontained = TRUE)
```

When you need more than the convenience function offers, build the `plotly` figure from the tibble. The documentation's custom-visualization article colors the Moon's path by speed with a `plot_ly()` call; the same idea works for any quantity you can `mutate()`. For a static image of a 3D view, the usual route is a screenshot at the orientation you want, or `plotly::save_image()`, which needs the `kaleido` Python package installed through `reticulate`.

14.5. Animation

`animate_system()` samples the run into `fps * duration` frames and renders a GIF with `gganimate`:

```
anim <- animate_system(inner, fps = 20, duration = 8, trails = TRUE)
anim
```

14. Figures for Papers and Talks

```
gganimate::anim_save("inner-solar-system.gif", anim)
```

Rendering takes tens of seconds, so get the static version right first, and keep `fps * duration` modest: 160 frames is plenty for a slide. A GIF loops forever and works in every slide program, which is usually what you want. If you need an MP4, or control over the frame labels, build the animation yourself from the tibble with `gganimate`, which is a few lines:

```
library(gganimate)

frames <- inner |> filter(time %% (seconds_per_day * 5) == 0) # every 5th day

p <- ggplot(frames, aes(x / AU, y / AU, color = id)) +
  geom_point(size = 3) +
  coord_equal() +
  labs(title = "Day {round(frame_time / 86400)}", x = "x (AU)", y = "y (AU)") +
  transition_time(time) +
  shadow_wake(wake_length = 0.1)

animate(p, fps = 15, nframes = 73, renderer = av_renderer("inner.mp4"))
```

For a 3D animation the `plotly` route (`animate_system_3d()`) renders instantly and gives a play button and a time slider, but it only lives in a browser.

14.6. Export and reproducibility

A figure in a paper should be reproducible from a script. Keep three things together: the script that builds the system, the version of the package, and the simulation output.

```
save_system(system, "kepler16-setup.rds") # the initial conditions
saveRDS(sim, "kepler16-run.rds") # the output tibble
readr::write_csv(sim, "kepler16-run.csv") # for collaborators not using R
writeLines(as.character(packageVersion("orbitr")), "orbitr-version.txt")
```

The output tibble is often large (half a million rows for Chapter 8's run) and the setup is tiny, and the output can always be regenerated from the setup with the same package version, so the setup and the version are the things to archive. Record the time step and the method too; they are not stored in the output. A reader who has the `.rds` setup file, the version number, and the `simulate_system()` call can reproduce every figure in this book.

Key results

- `plot_orbits()` returns a `ggplot`; add layers, scales, and themes with `+`.

- Everything else is `mutate()` then `ggplot()`: color by time, time series, `facet_wrap()` for comparisons and for snapshot grids.
- For print: `coord_equal()`, a colorblind-safe palette plus line types, `theme_classic()`, fixed size via `ggsave()`, vector PDF.
- Interactive 3D (`plotly`) and animation (`gganimate`) are for screens; save widgets as HTML and animations as GIF.
- Archive the setup (`save_system()`), the package version, and the `simulate_system()` call.

Part IV.

Under the Hood

15. The C++ Engine

Chapter 3 wrote the N-body acceleration in plain R and warned that it was slow. This chapter looks at what the package does instead: a forty-line C++ function that computes the same sum, the vectorized R version that stands in when it is unavailable, and the R loop that calls one or the other at every step. The most useful thing in the chapter is not the C++. It is the measurement of where the time actually goes, which is not where most people expect, and what that means for how big a system you can run.

15.1. Why compiled code

R is an interpreted language, and a `for` loop in R pays an interpretive overhead on every iteration: each `+`, each index, each assignment goes through the evaluator. For the double loop in Chapter 3's `accelerations()`, that is a few microseconds per pair, which for ten bodies and a hundred thousand steps is minutes, and for a hundred bodies is hours. C++ compiles the same loop to machine instructions that run a hundred or a thousand times faster.

Rcpp (Eddelbuettel and François 2011) is the bridge. You write a C++ function that takes R vectors as arguments and returns an R object, mark it with a comment, and Rcpp generates the glue that lets R call it as if it were an R function. The package's kernel lives in `src/engine.cpp` and is exported to R as `calc_acceleration_cpp()`.

15.2. The kernel, line by line

Here it is, in full. It is released under the MIT license with the rest of the package, and it is short enough to read in one sitting:

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
List calc_acceleration_cpp(NumericVector x, NumericVector y, NumericVector z,
                           NumericVector mass, double G,
                           double softening = 0.0) {

  int n = x.size();
  NumericVector ax(n), ay(n), az(n);

  // If no gravity or only one body, return zeros
```

15. The C++ Engine

```

if (G == 0 || n <= 1) {
    return List::create(Named("ax") = ax, Named("ay") = ay, Named("az") = az);
}

double eps2 = softening * softening;

for (int j = 0; j < n; j++) {
    for (int k = 0; k < n; k++) {
        if (j != k) {

            // Distance components (direction from j toward k)
            double dx = x[k] - x[j];
            double dy = y[k] - y[j];
            double dz = z[k] - z[j];

            // Total distance with optional softening to prevent 1/r^2 singularity
            double r = sqrt(dx*dx + dy*dy + dz*dz + eps2);

            if (r > 0) {
                // Scalar acceleration: a = G * M_k / r^2
                double a = G * mass[k] / (r*r);

                // Decompose into directional components
                ax[j] += a * (dx / r);
                ay[j] += a * (dy / r);
                az[j] += a * (dz / r);
            }
        }
    }
}

return List::create(Named("ax") = ax, Named("ay") = ay, Named("az") = az);
}

```

Walk through it against Equation 3.3.

The signature takes the three position vectors, the mass vector, G , and the softening length. `NumericVector` is Rcpp's wrapper for an R numeric vector; it is passed without copying, so handing a thousand positions to the kernel costs nothing. `// [[Rcpp::export]]` is the marker that tells Rcpp to generate the R-callable wrapper.

The early return handles $G = 0$ (Chapter 2's zero-gravity sandbox) and a lone body.

eps2 is ε^2 , computed once. The double loop runs over all ordered pairs (j, k) , $j \neq k$, exactly like Chapter 3's R version. For each pair it forms the difference vector pointing from j toward k and the softened distance $r = \sqrt{dx^2 + dy^2 + dz^2 + \varepsilon^2}$, which is Chapter 6's r_{soft} . The scalar $a = Gm_k/r^2$ times the unit vector $(dx, dy, dz)/r$ is $Gm_k(\mathbf{r}_k - \mathbf{r}_j)/r^3$, the second form of Equation 3.3, accumulated into body j 's acceleration. With $\varepsilon = 0$ and no overlapping bodies, this is exact Newtonian gravity; the $(r > 0)$ guard only matters if two bodies sit at the same point with no softening.

The return value is a list of three vectors, which R receives as a named list with `$ax`, `$ay`, `$az`. That is the whole engine. There is no physics in the package that is not in this function.

One thing the kernel does *not* do: it computes each pair twice, once as (j, k) and once as (k, j) . Newton's third law says the two are equal and opposite, so a loop over $j < k$ that adds the force to one body and subtracts it from the other would do half the work. The package does not bother, because, as the benchmarks below show, for the systems it is built for the kernel is not where the time goes.

15.3. The R fallback

If the compiled code is not available, `simulate_system()` uses a pure-R version built on `outer()`:

```
dx_mat <- outer(x, x, "-") * -1          # mat[j, k] = x[k] - x[j]
dy_mat <- outer(y, y, "-") * -1
dz_mat <- outer(z, z, "-") * -1
r_mat  <- sqrt(dx_mat^2 + dy_mat^2 + dz_mat^2 + softening^2)
diag(r_mat) <- Inf                      # a body does not pull on itself
mass_mat <- matrix(mass, nrow = n, ncol = n, byrow = TRUE)
a_mat <- G * mass_mat / r_mat^2
ax <- rowSums(a_mat * (dx_mat / r_mat))
ay <- rowSums(a_mat * (dy_mat / r_mat))
az <- rowSums(a_mat * (dz_mat / r_mat))
```

This is the same computation with the double loop replaced by $n \times n$ matrices. `outer()` builds the matrix of all pairwise differences in one call, the diagonal is set to infinity so that the self-term contributes zero, and `rowSums()` does the sum over k . Every operation is vectorized, so the loop still happens in compiled code inside R's own internals; the cost is that each step allocates half a dozen $n \times n$ matrices. For small n it is nearly as fast as the C++ kernel; for large n the allocations dominate.

The two engines should agree to rounding error, and do:

```
cluster <- function(n, seed = 1) {
  set.seed(seed)
  sys <- create_system()
  for (i in seq_len(n)) {
    sys <- add_body(sys, paste0("star", i), mass = mass_sun,
                     x = rnorm(1, sd = 1e13), y = rnorm(1, sd = 1e13), z = rnorm(1, sd = 1e13),
                     vx = rnorm(1, sd = 1e3), vy = rnorm(1, sd = 1e3), vz = rnorm(1, sd = 1e3))
  }
  sys
}

sys20 <- cluster(20)
with_cpp <- simulate_system(sys20, time_step = seconds_per_day,
```

15. The C++ Engine

```
duration = seconds_per_day * 50, use_cpp = TRUE)
without <- simulate_system(sys20, time_step = seconds_per_day,
                           duration = seconds_per_day * 50, use_cpp = FALSE)

max(abs(with_cpp$x - without$x) / abs(without$x))
#> [1] 0
```

15.4. Where the time goes

The kernel is C++, but the loop that calls it is R. Here is the shape of `simulate_system()`, stripped to its skeleton:

```
steps <- seq(0, duration, by = time_step)
results <- vector("list", length(steps))

for (i in seq_along(steps)) {
  state$time <- steps[i]
  results[[i]] <- state # record this step
  if (i == length(steps)) break
  accels <- calc_acceleration(state) # kernel call 1
  state$x <- state$x + state$vx * dt + 0.5 * accels$ax * dt^2 # (and y, z)
  new_accels <- calc_acceleration(state) # kernel call 2 (Verlet only)
  state$vx <- state$vx + 0.5 * (accels$ax + new_accels$ax) * dt # (and vy, vz)
}

dplyr::bind_rows(results)
```

`state` is a tibble with one row per body. Every step does two kernel calls, six column assignments on that tibble, and one list assignment; at the end, `bind_rows()` stacks as many tibbles as there were steps, and the run's settings (`G`, `softening`, `method`, `time_step`) are attached to the result as attributes, which is how `conserved_quantities()` and `continue_simulation()` know them without being told. For a handful of bodies the kernel call takes microseconds, and everything else takes longer than it does. Measure it:

```
time_run <- function(n, use_cpp, steps = 400) {
  sys <- cluster(n)
  elapsed <- system.time(
    simulate_system(sys, time_step = seconds_per_day,
                   duration = seconds_per_day * steps,
                   use_cpp = use_cpp, softening = 1e10)
  )["elapsed"]
  tibble(n = n, engine = if (use_cpp) "C++" else "R",
        ms_per_step = elapsed / steps * 1e3)
}

bench <- bind_rows(lapply(c(2, 5, 10, 20, 50, 100, 200), function(n) {
```

```

  bind_rows(time_run(n, TRUE), time_run(n, FALSE))
}))
bench
#> # A tibble: 14 x 3
#>       n engine ms_per_step
#>   <dbl> <chr>      <dbl>
#> 1     2 C++        0.550
#> 2     2 R          0.800
#> 3     5 C++        0.450
#> 4     5 R          0.750
#> 5    10 C++        0.475
#> 6    10 R          0.600
#> 7    20 C++        0.450
#> 8    20 R          0.775
#> 9    50 C++        0.550
#> 10   50 R          1.30
#> 11  100 C++        0.85
#> 12  100 R          3.12
#> 13  200 C++        1.72
#> 14  200 R         10.5

bench |>
  ggplot(aes(n, ms_per_step, color = engine)) +
  geom_line() + geom_point() +
  scale_x_log10() + scale_y_log10() +
  labs(x = "Number of bodies", y = "Time per step (ms)", color = NULL)

```

Read the C++ curve from the left. For two bodies and for twenty, the time per step is about the same: that is the fixed cost of the R loop, the tibble updates, and the call into the kernel, and it is independent of n . The kernel's n^2 term is invisible until it grows to match that overhead, somewhere around fifty bodies, after which the curve bends up with slope 2 on the log-log plot: doubling n quadruples the time. The R fallback has a higher floor, because `outer()` and friends allocate matrices on every call, and reaches the n^2 regime sooner.

The other large cost is at the end. `bind_rows()` on sixty thousand small tibbles, which is what Chapter 8's run produced, is slow, because each tibble has to be checked and copied:

```

one_step <- get_bodies(cluster(9))
pieces <- replicate(10000, one_step, simplify = FALSE)
system.time(bind_rows(pieces))["elapsed"]
#> [1] 0.14

```

Scale that by six for a 60,000-step run. For long runs of small systems, assembling the output tibble takes about as long as integrating it. The lesson for a user is simple: the number of *recorded* steps is what you pay for, and it is also what sets the size of the output. A simulation that records every hour for a century produces 876,000 rows per body whether or not it needed to.

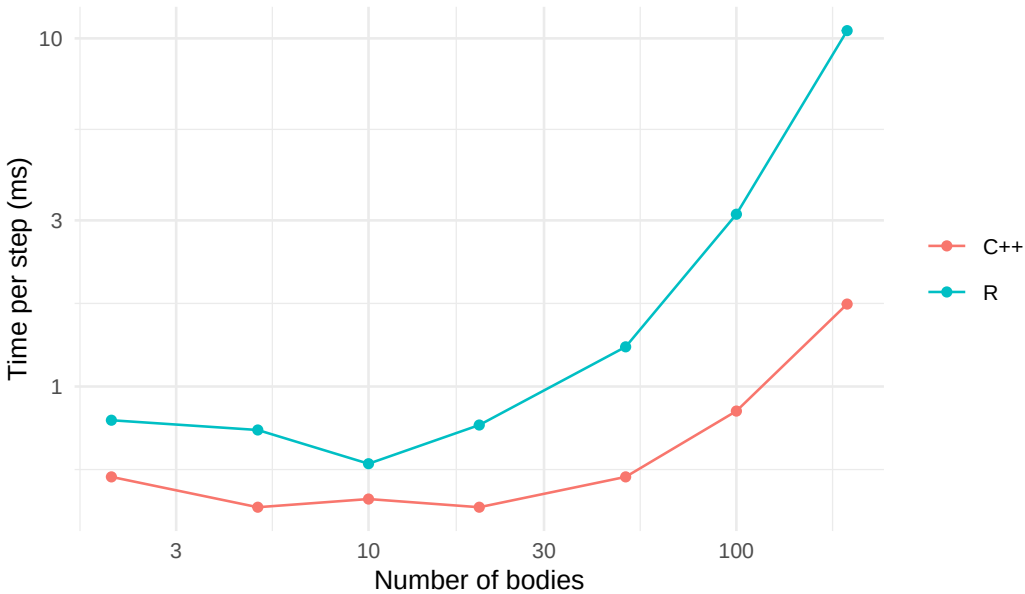


Figure 15.1.: Milliseconds per integration step against the number of bodies, for the C++ kernel and the R fallback, on the machine this book was built on. Both axes are logarithmic. The flat part at small n is the R loop's overhead; the rising part is the n^2 kernel.

Where it comes from

This split, a compiled inner kernel driven by an interpreted outer loop, is how almost all scientific computing in R and Python is done, and the trade-off is always the same. The interpreted layer is where the flexibility lives: the tibble output, the three integrators selectable by a string, the softening argument. The compiled layer is where the n^2 lives. The design is right as long as the compiled part dominates for the problems you care about. *orbiter*'s design assumption is a few to a few dozen bodies, and for those the R layer is the bottleneck, which is a sign that the kernel is doing its job.

15.5. The n^2 wall

Extrapolate the C++ curve. If the kernel costs cn^2 per evaluation, the $n = 200$ point gives c , and

```
c_kernel <- bench |> filter(engine == "C++", n == 200) |> pull(ms_per_step) / 200^2
tibble(n = c(1e3, 1e4, 1e5),
       ms_per_step = c_kernel * n^2,
       hours_per_10000_steps = ms_per_step * 1e4 / 3.6e6)
#> # A tibble: 3 x 3
#>       n ms_per_step hours_per_10000_steps
```

#>	<dbl>	<dbl>	<dbl>
#> 1	1000	43.1	0.120
#> 2	10000	4312.	12.0
#> 3	100000	431250.	1198.

A thousand bodies is comfortable. Ten thousand is a long evening for a short run. A hundred thousand is out of the question, and a galaxy has a hundred billion stars. Research codes get around the wall in three ways:

- **Tree codes** (Barnes and Hut 1986) group distant bodies into cells and use each cell's center of mass instead of its members, reducing the cost to $O(n \log n)$ at the price of a controllable approximation. Chapter 3's shell theorem is the justification: from far away, a clump of mass acts like a point.
- **The fast multipole method** (Greengard and Rokhlin 1987) takes the idea further, expanding the potential of each cell in a series, and reaches $O(n)$.
- **Hardware.** The pairwise sum is embarrassingly parallel, and GPUs compute billions of pair interactions per second. Specialized hardware for exactly this sum, the GRAPE boards of the 1990s and 2000s, was for a time the fastest computing in the world for anything.

orbitr does none of these, and it should not. Each of them is a large piece of engineering whose approximation error has to be understood as carefully as the integrator's, and none of them changes what a user of this package wants: a tidy table of where a dozen bodies went. The tidy table is itself the deeper limit. At $n = 10^4$ bodies and 10^4 steps the output is 10^8 rows, about seven gigabytes in memory, and the data frame would fail before the kernel became the problem. Big N-body simulation does not keep every position at every step; it writes snapshots to disk and analyzes them later. That is a different kind of tool.

15.6. Changing the physics

Because every force the package computes is in one function, adding a force is a small change to that function. A drag term proportional to velocity would need the velocities passed in (three more `NumericVector` arguments) and three lines inside the outer loop; Earth's oblateness would need Earth identified and a dozen lines; the relativistic correction of Chapter 16, a few more. None of it is hard, and all of it is on the roadmap. If you want to try, the package is on GitHub, `Rcpp::sourceCpp()` compiles a modified kernel in seconds, and `simulate_system()`'s `use_cpp` argument is the switch.

Key results

- The entire physics is `calc_acceleration_cpp()`: a double loop over ordered pairs computing $Gm_k(\mathbf{r}_k - \mathbf{r}_j)/r_{\text{soft}}^3$. The R fallback is the same sum with `outer()` matrices.
- For fewer than about fifty bodies, the R loop and the output assembly cost more than the kernel. The number of recorded steps, not the number of bodies, is what you pay for.

15. The C++ Engine

- Beyond that, cost grows as n^2 ; a thousand bodies is fine, a hundred thousand needs a different algorithm and a different output format.

16. Limits and Extensions

Chapter 1 listed what `orbitr` leaves out: bodies have no size, gravity is Newtonian, there are no forces but gravity, and the step is fixed. This chapter returns to each omission with the physics of what is missing, the size of the effect, and what it would take to add it. Some of the additions are on the package roadmap. Two of them you can do today with the tools this book has built, and the chapter shows how. The rest are described precisely enough that if you decide to implement one, you will know what you are implementing.

16.1. Collisions

Two point masses can pass through each other. Nothing in the equations of motion objects, and if the step is small enough the integrator will carry them straight through with energy conserved. In reality two planets that pass within the sum of their radii collide. The package has no notion of radius, so collisions are something you detect *after* the run, by looking.

A helper that finds the closest approach of every pair over a simulation is a self-join on `time`, like Chapter 5's hand-written energy sum:

```
closest_approach <- function(sim) {
  sim |>
    select(time, id, x, y, z) |>
    inner_join(sim |> select(time, id2 = id, x2 = x, y2 = y, z2 = z),
              by = "time", relationship = "many-to-many") |>
    filter(id < id2) |>
    mutate(r = sqrt((x - x2)^2 + (y - y2)^2 + (z - z2)^2)) |>
    group_by(id, id2) |>
    slice_min(r, n = 1) |>
    ungroup() |>
    select(id, id2, time, r)
}
```

Run it on Chapter 11's chaotic triple, where the stars came within a fraction of an AU of one another during the slingshot:

```
triple <- create_system() |>
  add_body("Star A", mass = 1e30, x = 1e11, y = 0, vx = 0, vy = 15000)
  add_body("Star B", mass = 1e30, x = -5e10, y = 8.66e10, vx = -12990, vy = -7500)
  add_body("Star C", mass = 1e30, x = -5e10, y = -8.66e10, vx = 14000, vy = -8000)
  simulate_system(time_step = seconds_per_hour, duration = seconds_per_year * 3)
```

16. Limits and Extensions

```
closest_approach(triple) |>
  mutate(solar_radii = r / 6.957e8, day = time / seconds_per_day)
#> # A tibble: 3 x 6
#>   id      id2      time      r solar_radii  day
#>   <chr>   <chr>    <dbl>    <dbl>    <dbl> <dbl>
#> 1 Star A Star B  9288000 73615522151.    106.   108.
#> 2 Star A Star C  82933200 20904109167.    30.0   960.
#> 3 Star B Star C 10519200 61636240576.    88.6   122.
```

If any pair's closest approach is less than the sum of their radii (two solar radii, for Sun-like stars), the simulation went somewhere reality would not have, and everything after that time is fiction. The check costs seconds and belongs at the end of every run involving close encounters.

A real collision model needs a rule for what happens next. The usual choice in planet-formation codes is a perfectly inelastic merger: the two bodies become one with the summed mass, the center-of-mass position, and the momentum-conserving velocity $\mathbf{v} = (m_1\mathbf{v}_1 + m_2\mathbf{v}_2)/(m_1 + m_2)$. Momentum is conserved; kinetic energy is not, and the difference is the energy that goes into cratering and heat. The roadmap's `radius` argument on `add_body()` is the hook for this, and `system_from_simulation()`, `remove_body()`, `add_body()`, and `continue_simulation()` are enough to do it by hand today: stop at the collision time, rebuild the system from that snapshot, replace the two bodies with their merger, and continue.

16.2. Forces other than gravity

Every force the package knows is in the kernel of Chapter 15. Three others matter in practice, and they behave very differently.

16.2.1. Radiation pressure

Sunlight carries momentum, and a small body absorbing it feels a force directed away from the Sun that falls off as $1/r^2$, exactly like gravity. For a body of mass m and cross-section A at distance r , $F = L_\odot A/(4\pi r^2 c)$, where $L_\odot$ is the Sun's luminosity. Because it has the same r -dependence as gravity, radiation pressure does not change the *shape* of the orbit equation at all; it just reduces the effective gravitational parameter:

$$\mu_{\text{eff}} = GM_\odot (1 - \beta), \quad \beta = \frac{F_{\text{rad}}}{F_{\text{grav}}} = \frac{L_\odot A}{4\pi c GM_\odot m}. \quad (16.1)$$

β depends on A/m , so it is tiny for a planet and large for dust. For a sphere of density ρ and radius s , $A/m = 3/(4\rho s)$, which works out to $\beta \approx 0.57/(\rho s)$ with ρ in g/cm^3 and s in microns: a micron-sized grain, rocky or icy, feels sunlight at a sizable fraction of gravity.

Because the correction is just a rescaled μ , you can simulate it today with the tool already in `create_system()`. For a test particle in orbit around the Sun alone, setting $G \rightarrow G(1 - \beta)$

is exact. A grain released from a parent body on a circular orbit keeps the parent's speed $v = \sqrt{GM/r}$ but sees only μ_{eff} ; by Chapter 4's rule, $e = |k^2 - 1|$ with $k^2 = v^2/(\mu_{\text{eff}}/r) = 1/(1 - \beta)$, so

$$e = \frac{\beta}{1 - \beta},$$

and the grain is unbound ($e \geq 1$) when $\beta \geq 1/2$. Dust released from a comet with $\beta > 0.5$ leaves the solar system; that is what the dust tail is.

```
grain <- function(beta) {
  create_system(G = gravitational_constant * (1 - beta)) |>
  add_sun() |>
  add_body("grain", mass = 1e-12, x = distance_earth_sun, vy = speed_earth) |>
  simulate_system(time_step = seconds_per_day, duration = seconds_per_year * 3) |>
  filter(id == "grain") |>
  mutate(beta = beta)
}

bind_rows(grain(0), grain(0.3), grain(0.6)) |>
  ggplot(aes(x / distance_earth_sun, y / distance_earth_sun,
             color = factor(beta))) +
  geom_path() +
  annotate("point", x = 0, y = 0, size = 3) +
  coord_equal() +
  labs(x = "x (AU)", y = "y (AU)", color = expression(beta))
```

The trick only works when the Sun is the only massive body. With planets present, rescaling G would weaken their gravity too, and a proper implementation adds the radiation force to the grain alone, inside the kernel.

16.2.2. Drag

Atmospheric drag on a satellite is $\mathbf{F} = -\frac{1}{2}\rho(r) C_D A v \mathbf{v}$, opposing the velocity and growing with the square of the speed and with the air density, which falls off exponentially with altitude. It is dissipative: it removes energy, and since $\varepsilon = -\mu/2a$, removing energy shrinks the orbit, which lowers the satellite into denser air, which increases the drag. The decay accelerates until re-entry. The International Space Station loses about two kilometers of altitude a month to this and is periodically boosted.

Drag depends on velocity, which the kernel does not receive, so adding it is a signature change as well as a few lines. But there is a way to approximate it without touching the kernel, and it is a real numerical technique: *operator splitting*. Run gravity for a short interval with `simulate_system()`, then apply the drag as an instantaneous change to the velocity, then run gravity again. Chapter 13's `continue_simulation()` makes the loop a few lines. Here is a toy satellite in an atmosphere with an artificially short damping time, so that a year of decay fits in a figure:

16. Limits and Extensions

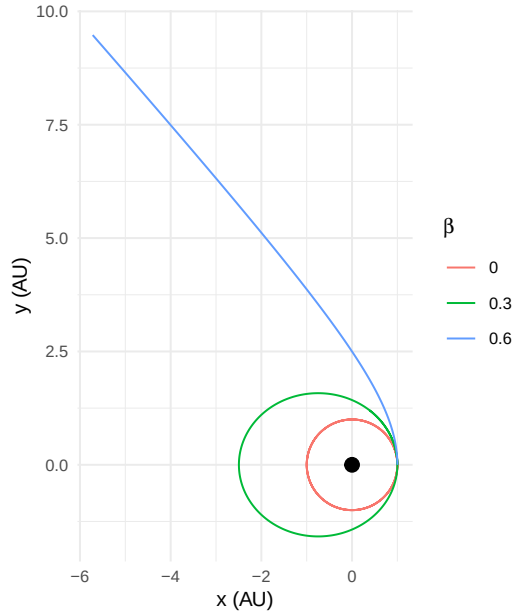


Figure 16.1.: Three dust grains released from Earth's orbit at Earth's speed, with radiation-pressure parameters β of 0, 0.3, and 0.6, simulated by rescaling G . The last is unbound.

```
kick_drag <- function(sim, body, tau, dt) {
  last <- sim$time == max(sim$time) & sim$id == body
  f <- exp(-dt / tau)                                # velocity damping over the interval
  sim$vx[last] <- sim$vx[last] * f
  sim$vy[last] <- sim$vy[last] * f
  sim$vz[last] <- sim$vz[last] * f
  sim
}

sat <- create_system() |>
  add_body("Earth", mass = mass_earth) |>
  add_body("Sat", mass = 1e3, x = 8e6, vy = sqrt(gravitational_constant * mass_earth / 8
  simulate_system(time_step = 60, duration = seconds_per_day)

for (day in 1:40) {
  sat <- sat |>
    kick_drag("Sat", tau = 200 * seconds_per_day, dt = seconds_per_day) |>
    continue_simulation(time_step = 60, duration = seconds_per_day)
}

sat |>
  filter(id == "Sat") |>
  mutate(r_km = sqrt(x^2 + y^2) / 1e3) |>
  ggplot(aes(time / seconds_per_day, r_km)) +
```

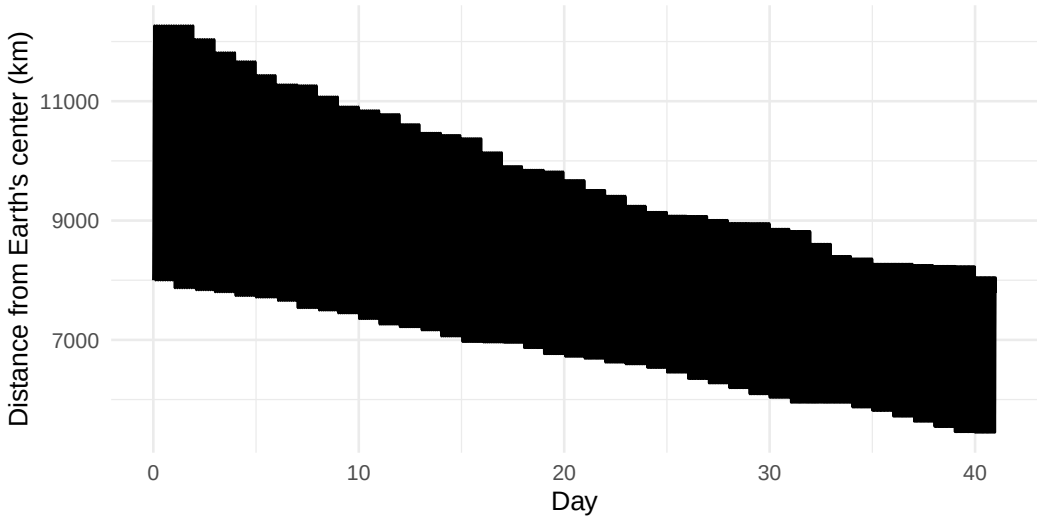


Figure 16.2.: Orbital decay under a toy drag law applied by operator splitting: gravity for a day, a small velocity kick, gravity for a day. Energy leaves the orbit at every kick and the orbit shrinks.

```
geom_line() +
labs(x = "Day", y = "Distance from Earth's center (km)")
```

Splitting is first-order accurate in the interval between kicks, so make the interval short compared with the drag timescale. The toy applies the same fractional kick wherever the satellite happens to be, which is what a body moving through a uniform medium feels. Real atmospheric drag is concentrated at periapsis, where the air is densest, and that changes the shape of the decay: periapsis stays nearly fixed while apoapsis is pulled down, the orbit circularizes, and only then does the whole thing spiral in. The same loop produces that behavior if the kick depends on altitude through a density profile, $\tau(r) = \tau_0 \exp((r - r_0)/H)$ with a scale height H of about 60 km, applied every few minutes instead of every day.

16.2.3. Thrust

A rocket burn is a kick too, and a short one is well modeled as an instantaneous velocity change: a Δv . The Hohmann transfer between two circular orbits is two kicks, one at periapsis of the transfer ellipse and one at apoapsis, with magnitudes that follow directly from vis-viva: the transfer ellipse has $a = (r_1 + r_2)/2$, and each Δv is the difference between the circular speed and the transfer-orbit speed at that radius. Splitting with `continue_simulation()` does it exactly.

16.3. Oblateness

Earth is not a sphere; it is an ellipsoid 21 km fatter at the equator than at the poles, and the extra mass around the equator pulls a satellite slightly off the point-mass orbit. The leading correction to the potential is

$$\Phi(r, \theta) = -\frac{GM}{r} \left[1 - J_2 \left(\frac{R}{r} \right)^2 \frac{3 \cos^2 \theta - 1}{2} \right], \quad (16.2)$$

where θ is the angle from the rotation axis, R is the equatorial radius, and J_2 is a dimensionless number that measures the bulge: 1.083×10^{-3} for Earth, 14.7×10^{-3} for Jupiter, 0.2×10^{-3} for the Moon. The force is no longer central, so angular momentum about the center is not conserved, and the orbit's elements drift: the node Ω regresses and the perigee ω advances, at rates that average to

$$\dot{\Omega} = -\frac{3}{2} n J_2 \left(\frac{R}{a} \right)^2 \frac{\cos i}{(1 - e^2)^2}, \quad \dot{\omega} = \frac{3}{4} n J_2 \left(\frac{R}{a} \right)^2 \frac{5 \cos^2 i - 1}{(1 - e^2)^2}, \quad (16.3)$$

with $n = 2\pi/T$ the mean motion. For the International Space Station:

```
R_earth <- 6.378e6
J2      <- 1.083e-3
a_iss   <- R_earth + 4.2e5
i_iss   <- 51.6 * pi / 180
n_iss   <- sqrt(gravitational_constant * mass_earth / a_iss^3)

node_rate <- -1.5 * n_iss * J2 * (R_earth / a_iss)^2 * cos(i_iss)
node_rate * seconds_per_day * 180 / pi      # degrees per day
#> [1] -4.952777
```

Five degrees a day. The ISS's orbital plane swings all the way around in about ten weeks, which anyone who tracks its passes knows. A satellite at an inclination near 98° has $\dot{\Omega}$ of exactly $+0.9856^\circ$ per day, one full turn per year, so its orbital plane keeps a fixed angle to the Sun: that is a Sun-synchronous orbit, and it is how every Earth-observing satellite that needs constant lighting is flown. All of it comes from the second term in Equation 16.2, and all of it is absent from a point-mass simulation. Adding it to the kernel means giving one body an axis and a J_2 and adding the gradient of the correction term to the acceleration of everything near it.

16.4. General relativity

Newtonian gravity gets Mercury's perihelion precession wrong by 43 arcseconds per century, and the discrepancy was known for half a century before Einstein explained it. The general-relativistic correction to the acceleration of a test body around a mass M , to first post-Newtonian order, is

$$\mathbf{a}_{\text{GR}} = \frac{GM}{c^2 r^3} \left[\left(\frac{4GM}{r} - v^2 \right) \mathbf{r} + 4(\mathbf{r} \cdot \mathbf{v}) \mathbf{v} \right]. \quad (16.4)$$

Its size relative to Newtonian gravity is of order v^2/c^2 , which for Mercury is 2×10^{-8} . Small, but it accumulates: the perihelion advances by

$$\Delta\omega = \frac{6\pi GM}{c^2 a(1-e^2)} \quad (16.5)$$

per orbit, and Mercury makes 415 orbits a century:

```
c_light <- 2.998e8
a_merc  <- distance_mercury_sun
e_merc  <- 0.2056
per_orbit <- 6 * pi * gravitational_constant * mass_sun / (c_light^2 * a_merc * (1 - e_merc^2))
orbits_per_century <- 100 * seconds_per_year / (2 * pi * sqrt(a_merc^3 / (gravitational_constant * mass_sun)))

per_orbit * orbits_per_century * 180 / pi * 3600      # arcseconds per century
#> [1] 42.99535
```

Forty-three arcseconds. Equation 16.4 depends on velocity, like drag, so it needs the same kernel signature change; it is about ten lines of C++, and the roadmap lists it as an opt-in argument. Chapter 12 explained why measuring the result would be harder than implementing it: the numerical precession at any practical step size is larger than the relativistic one, and you would need the step small enough, or a smarter integrator, to see 43 arcseconds under the artifact.

16.5. Better integrators

Velocity Verlet with a fixed step is the right default and the wrong tool for two situations this book has met: a highly eccentric orbit, where the step should shrink near periapsis, and a long integration where second-order accuracy is not enough.

Adaptive steps solve the first problem and create a new one. If you change Δt based on the current state, the integrator is no longer symplectic (the map from one step to the next is no longer a fixed symplectic map), and the bounded-energy guarantee of Chapter 5 is lost: the energy drifts secularly, just as Euler's does, though more slowly. The fix is to choose the step *time-symmetrically*, so that the step size at each point is the same whether you arrive from the past or the future; Hut, Makino, and McMillan showed how to do that with an implicit iteration (Hut et al. 1995), and their time-symmetric leapfrog is the ancestor of the block time steps that every modern star-cluster code uses. Chapter 13's segmented runs are the crude version: a few fixed-step symplectic segments with a small disturbance at each switch.

Higher order solves the second problem, and the symplectic way to do it is composition. Yoshida showed in 1990 that three Verlet steps of sizes $w_1 \Delta t$, $w_0 \Delta t$, $w_1 \Delta t$ with

16. Limits and Extensions

$$w_1 = \frac{1}{2 - 2^{1/3}} \approx 1.351, \quad w_0 = -\frac{2^{1/3}}{2 - 2^{1/3}} \approx -1.702$$

compose to a *fourth*-order symplectic method (Yoshida 1990): halve the step and the error drops by sixteen. The middle step is backward in time, which looks absurd and is fine; it cancels the third-order error of the two forward steps. The same construction yields sixth and eighth order. (The package integrates only forward, so this is a kernel-and-loop change, not a trick with `continue_simulation()`.)

Non-symplectic high order is the other route. IAS15 (Rein and Spiegel 2015), the default integrator of the REBOUND code, is a fifteenth-order adaptive scheme whose errors are below double-precision rounding for a billion orbits; it gives up symplecticity and gets away with it by being so accurate that there is no drift to speak of. For a few bodies over a long time it is the state of the art. If you find yourself needing it, that is the point at which to graduate from *orbitr* to a research code, and you will understand what the research code is doing.

16.6. The roadmap, and how to help

The package roadmap, in the documentation under Roadmap, lists what is being considered: a `radius` argument and collision handling; drag, radiation pressure, J_2 , and the post-Newtonian correction as optional forces; a progress bar for long runs; and built-in conservation diagnostics. Three earlier items, Keplerian elements, the solar system loader, and saving and loading systems, have already shipped, which is some evidence that the list is not decorative.

If you want something that is not there, or want to build something that is, the package is on GitHub at <https://github.com/DRosenman/orbitr>. Open an issue to describe what you need; a reproducible example, which in *orbitr* is usually five lines, helps more than anything. If you implement a feature, every chapter of this book is a test: the conservation laws of Chapter 12, the two-body predictions of Chapter 4, the Kepler’s-law check of Chapter 8. A pull request that comes with the check that shows the physics is right is the kind that gets merged.

16.7. Where to go next

You now know what the simulation does and why. That is a different kind of knowledge from knowing how to drive it, and it transfers. Every orbital-mechanics code, from the ones that fly spacecraft to the ones that model galaxy formation, is built from the same parts: a force law, an integrator, a set of conserved quantities to watch, and a long list of things left out. Some of them use the same integrator as *orbitr*. All of them share its limits in one form or another, and the person who understands the limits is the person who gets the right answer out of the code.

Thornton and Marion (Thornton and Marion 2004) for the mechanics, Murray and Dermott (Murray and Dermott 1999) for the solar system, Hairer, Lubich, and Wanner (Hairer et al. 2006) for the integrators, and Aarseth (Aarseth 2003) for the N-body methods will take you

further than this book did on each of its four parts. The package will keep changing. The physics will not.

Key results

- Collisions: detect after the run with `closest_approach()`; a merger conserves momentum, not kinetic energy.
- Radiation pressure rescales μ by $(1 - \beta)$; grains with $\beta > 1/2$ escape. Drag and thrust are velocity kicks, which operator splitting with `continue_simulation()` can apply today.
- J_2 regresses the node at $-\frac{3}{2}nJ_2(R/a)^2 \cos i$; five degrees a day for the ISS.
- The post-Newtonian correction advances Mercury's perihelion by $6\pi GM/(c^2 a(1 - e^2))$ per orbit: 43 arcseconds per century.
- Adaptive steps break symplecticity unless chosen time-symmetrically; higher order comes from Yoshida's composition of Verlet steps.

References

- Aarseth, Sverre J. 2003. *Gravitational N-Body Simulations: Tools and Algorithms*. Cambridge University Press.
- Barnes, Josh, and Piet Hut. 1986. “A Hierarchical $O(N \log N)$ Force-Calculation Algorithm.” *Nature* 324: 446–49. <https://doi.org/10.1038/324446a0>.
- Bate, Roger R., Donald D. Mueller, and Jerry E. White. 1971. *Fundamentals of Astrodynamics*. Dover.
- Cromer, Alan. 1981. “Stable Solutions Using the Euler Approximation.” *American Journal of Physics* 49 (5): 455–59. <https://doi.org/10.1119/1.12478>.
- Doyle, Laurance R., Joshua A. Carter, Daniel C. Fabrycky, et al. 2011. “Kepler-16: A Transiting Circumbinary Planet.” *Science* 333 (6049): 1602–6. <https://doi.org/10.1126/science.1210923>.
- Eddelbuettel, Dirk, and Romain François. 2011. “Rcpp: Seamless R and C++ Integration.” *Journal of Statistical Software* 40 (8): 1–18. <https://doi.org/10.18637/jss.v040.i08>.
- Feynman, Richard P., Robert B. Leighton, and Matthew Sands. 1963. *The Feynman Lectures on Physics, Volume i*. Addison-Wesley.
- Goldstein, Herbert, Charles Poole, and John Safko. 2002. *Classical Mechanics*. 3rd ed. Addison-Wesley.
- Greengard, Leslie, and Vladimir Rokhlin. 1987. “A Fast Algorithm for Particle Simulations.” *Journal of Computational Physics* 73 (2): 325–48. [https://doi.org/10.1016/0021-9991\(87\)90140-9](https://doi.org/10.1016/0021-9991(87)90140-9).
- Hairer, Ernst, Christian Lubich, and Gerhard Wanner. 2003. “Geometric Numerical Integration Illustrated by the Störmer–Verlet Method.” *Acta Numerica* 12: 399–450. <https://doi.org/10.1017/S0962492902000144>.
- Hairer, Ernst, Christian Lubich, and Gerhard Wanner. 2006. *Geometric Numerical Integration: Structure-Preserving Algorithms for Ordinary Differential Equations*. 2nd ed. Springer.
- Holman, Matthew J., and Paul A. Wiegert. 1999. “Long-Term Stability of Planets in Binary Systems.” *The Astronomical Journal* 117 (1): 621–28. <https://doi.org/10.1086/300695>.
- Hut, Piet, Jun Makino, and Steve McMillan. 1995. “Building a Better Leapfrog.” *The Astrophysical Journal* 443: L93–96. <https://doi.org/10.1086/187844>.

References

- Kepler, Johannes. 1609. *Astronomia Nova*. G. Voegelinus.
- Kepler, Johannes. 1619. *Harmonices Mundi*. J. Planck.
- Murray, Carl D., and Stanley F. Dermott. 1999. *Solar System Dynamics*. Cambridge University Press.
- Newman, Mark. 2012. *Computational Physics*. CreateSpace.
- Newton, Isaac. 1687. *Philosophiæ Naturalis Principia Mathematica*. Royal Society.
- Park, Ryan S., William M. Folkner, James G. Williams, and Dale H. Boggs. 2021. “The JPL Planetary and Lunar Ephemerides DE440 and DE441.” *The Astronomical Journal* 161 (3): 105. <https://doi.org/10.3847/1538-3881/abd414>.
- Plummer, H. C. 1911. “On the Problem of Distribution in Globular Star Clusters.” *Monthly Notices of the Royal Astronomical Society* 71 (5): 460–70. <https://doi.org/10.1093/mnras/71.5.460>.
- Poincaré, Henri. 1890. “Sur Le Problème Des Trois Corps Et Les équations de La Dynamique.” *Acta Mathematica* 13: 1–270. <https://doi.org/10.1007/BF02392506>.
- Rein, Hanno, and David S. Spiegel. 2015. “IAS15: A Fast, Adaptive, High-Order Integrator for Gravitational Dynamics, Accurate to Machine Precision over a Billion Orbits.” *Monthly Notices of the Royal Astronomical Society* 446 (2): 1424–37. <https://doi.org/10.1093/mnras/stu2164>.
- Rosenman, David P. 2026. *Orbitr: A Tidy Physics Engine for Building and Visualizing Orbital Simulations in R*. <https://orbit-r.com>.
- Størmer, Carl. 1907. “Sur Les Trajectoires Des Corpuscules électrisés Dans l’espace Sous l’action Du Magnétisme Terrestre, Avec Application Aux Aurores Boréales.” *Archives Des Sciences Physiques Et Naturelles* 24: 5–18, 113–58, 221–47.
- Swope, William C., Hans C. Andersen, Peter H. Berens, and Kent R. Wilson. 1982. “A Computer Simulation Method for the Calculation of Equilibrium Constants for the Formation of Physical Clusters of Molecules: Application to Small Water Clusters.” *The Journal of Chemical Physics* 76 (1): 637–49. <https://doi.org/10.1063/1.442716>.
- Thornton, Stephen T., and Jerry B. Marion. 2004. *Classical Dynamics of Particles and Systems*. 5th ed. Brooks/Cole.
- Tiesinga, Eite, Peter J. Mohr, David B. Newell, and Barry N. Taylor. 2021. “CODATA Recommended Values of the Fundamental Physical Constants: 2018.” *Reviews of Modern Physics* 93 (2): 025010. <https://doi.org/10.1103/RevModPhys.93.025010>.
- Touboul, Pierre, Gilles Métris, Manuel Rodrigues, et al. 2022. “MICROSCOPE Mission: Final Results of the Test of the Equivalence Principle.” *Physical Review Letters* 129 (12): 121102. <https://doi.org/10.1103/PhysRevLett.129.121102>.

- Verlet, Loup. 1967. “Computer ‘Experiments’ on Classical Fluids. I. Thermodynamical Properties of Lennard-Jones Molecules.” *Physical Review* 159 (1): 98–103. <https://doi.org/10.1103/PhysRev.159.98>.
- Yoshida, Haruo. 1990. “Construction of Higher Order Symplectic Integrators.” *Physics Letters A* 150 (5–7): 262–68. [https://doi.org/10.1016/0375-9601\(90\)90092-3](https://doi.org/10.1016/0375-9601(90)90092-3).

A. R Basics

This appendix is for the reader who has never opened R. It covers the minimum needed to run and modify the code in this book: installing the software, running code, installing packages, and the small vocabulary of functions the chapters use over and over. It is deliberately brief. R is a large language and this book uses a narrow slice of it; the last section says where to go when you want the rest, and every book listed there is free online.

If you already use R, skip this. If you have used Python or MATLAB, skim it; most of what differs is in the sections on the pipe and on tibbles.

A.1. Installing R and RStudio

Two installs, in this order.

R is the language. Download it from <https://cran.r-project.org>: pick your operating system, then the “base” distribution on Windows or the `.pkg` installer on a Mac, and run the installer with its defaults. On Linux, use your distribution’s package manager (`sudo apt install r-base` on Ubuntu).

RStudio is the editor most people use, and the one this book assumes. Download the free desktop version from <https://posit.co/download/rstudio-desktop/>. (Positron, from the same company, is a newer alternative; either works.) When you open RStudio you will see four panes. The two that matter now are the **Console** at the bottom left, where you type a line of R and get an answer, and the **Source** pane above it, where you edit files.

Try the console. Type `2 + 2` next to the `>` prompt and press Enter:

```
2 + 2
#> [1] 4
```

The `[1]` means “this is the first element of the result,” which matters once results have more than one element, and they usually do.

A.2. Running code

Three ways, from quickest to most reusable.

- **Console.** Type and press Enter. Good for checking something.

A. R Basics

- **Script.** File → New File → R Script gives you a text file ending in `.R`. Type lines of code; press Ctrl+Enter (Cmd+Enter on a Mac) to run the line the cursor is on, or select several lines and press it to run them together. Save the file and you can rerun everything tomorrow.
- **Quarto or R Markdown document.** A document that mixes text and code “chunks,” which is what this book is written in. File → New File → Quarto Document. Each chunk has a small green arrow to run it. This is the right format for a lab report, because the figures and numbers are regenerated from the code every time you render.

Lines starting with `#` are comments and are ignored. Assignment uses `<-`: `x <- 3` stores the value 3 under the name `x`, and typing `x` afterward prints it. (`=` also works for assignment; `<-` is the convention and the book uses it.)

```
x <- 3
x * 2
#> [1] 6
```

Everything in R is case-sensitive: `mass_earth` and `Mass_Earth` are different names, and only the first one exists.

A.3. Packages

R’s base installation is small on purpose. Almost everything useful, including `orbitr`, comes as a **package**, which you install once and load each session.

```
install.packages("orbitr")      # once, downloads and installs (needs internet)
library(orbitr)                 # every session, makes the functions available
```

`install.packages()` takes the name in quotes; `library()` takes it bare (quotes work there too). If you call a function from a package you have not loaded, R says **could not find function "simulate_system"**, and the fix is `library(orbitr)`. Installing `orbitr` also installs `dplyr` and `ggplot2`, which the book uses for analysis and plotting; load them the same way. The preface’s “What you need” section has the full list, including how to get the development version from GitHub.

Two more things about packages. `packageVersion("orbitr")` tells you which version you have; this book needs 1.0.0 or later. And `orbitr::simulate_system()`, with the double colon, calls a function from a package without loading the whole package, which is why you will see `tidyr::pivot_longer()` in a few places in the book for a package that is used only once.

A.4. Getting help

`?simulate_system` opens the help page for a function: what it does, every argument and its default, and runnable examples at the bottom. This is the single most useful habit to form. Every function in `orbitr` has one, and Appendix C is a condensed index of them. For

a package as a whole, its website (<https://orbit-r.com> for `orbitr`) has longer articles than the help pages.

When something goes wrong, read the error message from the top; the first line usually says what R could not do, and the rest is where it was when it failed. The three errors new users hit most are `could not find function` (load the package), `object 'x' not found` (you have not created `x` yet, or spelled it differently), and `unexpected symbol` or `unexpected ')` (a typo: a missing comma, an extra bracket, mismatched quotes). RStudio's editor underlines the last kind in red before you run it.

A.5. Functions and arguments

A function call is a name followed by parentheses with the inputs, called arguments, inside:

```
sqrt(16)
#> [1] 4
round(3.14159, digits = 2)
#> [1] 3.14
```

Arguments can be matched by position or by name. `round(3.14159, 2)` and `round(3.14159, digits = 2)` are the same call. This book names arguments almost everywhere, because `add_body("Moon", mass = mass_moon, x = distance_earth_moon, vy = speed_moon)` is readable and the positional version is not. Arguments with defaults can be left out: `add_body()` has `x`, `y`, `z`, `vx`, `vy`, and `vz` all defaulting to 0, so you only mention the ones that are not zero.

Writing your own function is the same idea in reverse:

```
circular_speed <- function(mass, r, G = gravitational_constant) {
  sqrt(G * mass / r)
}

circular_speed(mass_earth, distance_earth_moon)
#> [1] 1018.289
```

The body runs with the arguments filled in, and the last value computed is returned. The book defines a dozen small functions like this; every one of them follows this pattern.

A.6. The pipe

Most of the book's code is written as a chain:

```
create_system() |>
  add_sun() |>
  add_planet("Earth", parent = "Sun") |>
  simulate_system(time_step = seconds_per_day, duration = seconds_per_year) |>
  plot_orbits()
```

A. R Basics

The operator `|>` is the **pipe**. It takes the value on its left and passes it as the *first argument* of the function on its right, so `x |> f(y)` means `f(x, y)`. The chain above is exactly

```
plot_orbits(simulate_system(add_planet(add_sun(create_system()), "Earth", parent = "Sun",
                                         time_step = seconds_per_day, duration = seconds_per_year)))
```

which says the same thing inside out and is why nobody writes it that way. Read a pipe top to bottom as a sequence of steps: create a system, add the Sun, add Earth, simulate, plot. Each function in `orbitr` takes the system or the simulation output as its first argument precisely so that it can sit in a chain like this.

You will see `%>%` in older code and in some of the books below; it is the same idea from the `magrittr` package, from before the pipe was built into R. `|>` needs R 4.1 or later, which is why the preface asks for it.

A.7. Vectors, data frames, and tibbles

A **vector** is a sequence of values of one type, built with `c()`:

```
speeds <- c(1000, 1022, 1044)
speeds * 2
#> [1] 2000 2044 2088
speeds[2]
#> [1] 1022
```

Arithmetic applies to every element at once, with no loop, and `[2]` picks the second element. Most of R works this way, which is why the book's code has so few loops.

A **data frame** is a table: columns are vectors, all the same length, each with a name. A **tibble** is the modern data frame, from the `tidyverse`, with a tidier print method and fewer surprises. Everything `orbitr` returns is a tibble. The output of `simulate_system()` has one row per body per time step and the columns `time`, `id`, `mass`, `x`, `y`, `z`, `vx`, `vy`, `vz`:

```
sim <- 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 * 2)
```



```
sim
#> # A tibble: 98 x 9
#>   id      mass      x      y      z      vx      vy      vz  time
#>   <chr>   <dbl>   <dbl>   <dbl> <dbl>   <dbl>   <dbl> <dbl> <dbl>
#> 1 Earth 5.97e24      0      0      0      0      0      0      0
#> 2 Moon 7.34e22 384400000      0      0      0    1022      0      0
#> 3 Earth 5.97e24    215.      0      0    0.119 0.000571      0    3600
#> 4 Moon 7.34e22 384382520. 3679200      0    -9.71  1022.      0    3600
#> 5 Earth 5.97e24    860.    4.11      0    0.239 0.00229      0    7200
```

A.8. The six dplyr verbs the book uses

```
#> 6 Moon 7.34e22 384330083. 7358065. 0 -19.4 1022. 0 7200
#> 7 Earth 5.97e24 1934. 16.5 0 0.358 0.00514 0 10800
#> 8 Moon 7.34e22 384242692. 11036262. 0 -29.1 1022. 0 10800
#> 9 Earth 5.97e24 3438. 41.1 0 0.477 0.00914 0 14400
#> 10 Moon 7.34e22 384120357. 14713454. 0 -38.8 1021. 0 14400
#> # i 88 more rows
```

Printing a tibble shows the first ten rows and the column types (`<chr>` is text, `<dbl>` is a number). `nrow(sim)` and `names(sim)` give the row count and the column names; `sim$x` pulls one column out as a plain vector; `sim[sim$id == "Moon", ]` picks rows by a condition. The dplyr verbs in the next section do the same things more readably, and the book uses them instead.

A.8. The six dplyr verbs the book uses

dplyr is the tidyverse package for working with tables, and the book uses six of its functions. Each takes a tibble as its first argument (so it sits in a pipe) and returns a tibble.

```
library(dplyr)
```

```
sim |>
  filter(id == "Moon") |> # keep rows where this is true
  mutate(r_km = sqrt(x^2 + y^2 + z^2) / 1e3) |> # add a column computed from others
  select(time, r_km) |> # keep only these columns
  arrange(desc(r_km)) |> # sort rows (desc = largest first)
  slice_head(n = 3) # the first few rows
#> # A tibble: 3 x 2
#>   time      r_km
#>   <dbl>   <dbl>
#> 1 172800 384673.
#> 2 169200 384662.
#> 3 165600 384652.
```

The one that takes a moment to learn is `summarise()` with `group_by()`: split the table into groups, then collapse each group to one row.

```
sim |>
  group_by(id) |>
  summarise(max_speed = max(sqrt(vx^2 + vy^2 + vz^2)),
            n_steps = n())
#> # A tibble: 2 x 3
#>   id      max_speed n_steps
#>   <chr>    <dbl>   <int>
#> 1 Earth      5.68      49
#> 2 Moon     1022      49
```

A. R Basics

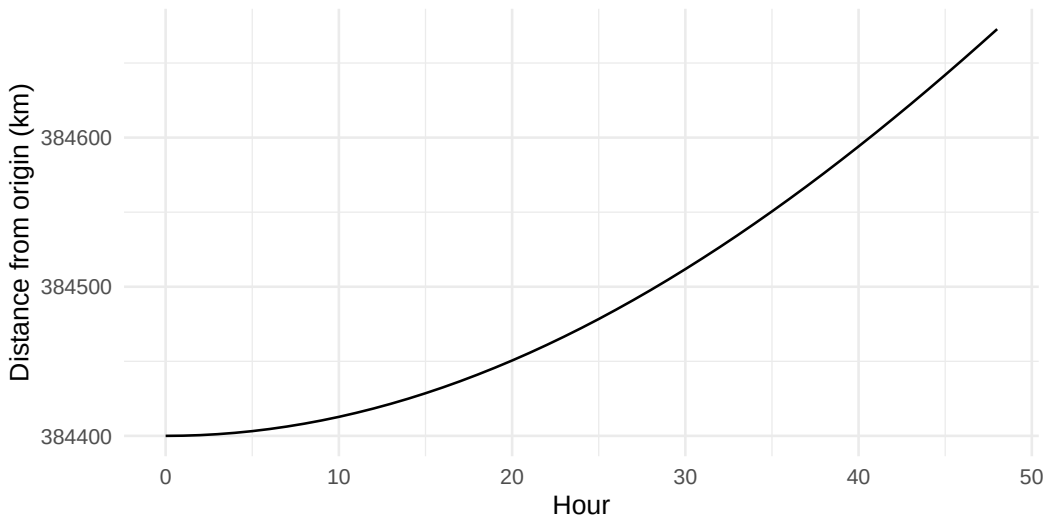
Inside any of these, column names are used bare, without quotes or `sim$`: `filter(id == "Moon")`, not `filter(sim$id == "Moon")`. Comparisons use `==` for equals, `!=` for not equals, and `&`, `|` for and, or. That is nearly all the dplyr in the book; `inner_join()`, which lines up two tables on a shared column, appears in a few chapters and is explained where it is used.

A.9. The ggplot2 the book uses

ggplot2 builds a plot in layers. You say which columns map to which visual properties (`aes()`), add a geometry layer that draws them, and keep adding with `+`:

```
library(ggplot2)

sim |>
  filter(id == "Moon") |>
  mutate(r_km = sqrt(x^2 + y^2 + z^2) / 1e3) |>
  ggplot(aes(x = time / seconds_per_hour, y = r_km)) +
  geom_line() +
  labs(x = "Hour", y = "Distance from origin (km)")
```



Note the switch from `|>` to `+` once the plot starts: ggplot2 predates the pipe and uses `+` to add layers. The geometries the book uses are `geom_line()` (a line in order of x), `geom_path()` (a line in the order of the rows, which is what an orbit needs), `geom_point()`, and `annotate("point", ...)` for a single marker. `aes(color = id)` colors by a column and adds a legend automatically. `facet_wrap(~ method)` splits one plot into panels, one per value of a column. `coord_equal()` makes one meter on the x axis the same length as one meter on y , without which circles come out as ellipses, and `scale_y_log10()` gives a logarithmic axis. `labs()` sets the labels and `theme_minimal()` the look. `plot_orbits()` returns a ggplot built this way, so anything here can be added to it.

A.10. Where to learn more

All of these are free to read online, and all are good.

- **R for Data Science**, by Hadley Wickham, Mine Çetinkaya-Rundel, and Garrett Grolemund (<https://r4ds.hadley.nz>). The standard introduction to the tidyverse: dplyr, ggplot2, tibbles, and the way of working this book assumes. If you read one other book, read this one; the first few chapters are enough.
- **Hands-On Programming with R**, by Garrett Grolemund (<https://rstudio-education.github.io/hopr/>). Shorter and more about the language itself: vectors, functions, loops, the things R4DS skips. Good if you want to understand *why* R behaves as it does.
- **ggplot2: Elegant Graphics for Data Analysis**, by Hadley Wickham, Danielle Navarro, and Thomas Lin Pedersen (<https://ggplot2-book.org>). The full account of the plotting system, from the people who wrote it. Read it when `plot_orbits()` stops being enough.
- **R Graphics Cookbook**, by Winston Chang (<https://r-graphics.org>). Recipes: “I want a plot that looks like this.” The fastest route to a specific figure.
- **Advanced R**, by Hadley Wickham (<https://adv-r.hadley.nz>). Not needed for this book. Read it when you start wanting to know what a function *is*.
- The **Posit cheat sheets** (<https://posit.co/resources/cheatsheets/>). One page each for dplyr, ggplot2, and RStudio; worth printing.

For the physics side, the references in the chapters and the Further Reading section at <https://orbit-r.com> do the same job.

B. Mathematical Background

This appendix collects the mathematics the rest of the book takes for granted. It is not a course; it is a reference, written for a reader who has met calculus and vectors before and wants the specific facts the derivations lean on, in the notation the book uses, with a pointer to where each one earns its keep. If you can read it without surprises, you have everything you need. If a section is new to you, it is short enough to learn from, and the chapter it feeds will make it concrete.

The book assumes you are comfortable with algebra, with trigonometry (sines, cosines, radians), and with the derivative and integral of a function of one variable. Everything beyond that is here.

B.1. Vectors

A **vector** is a quantity with magnitude and direction: a position, a velocity, a force. The book sets vectors in bold, $\mathbf{r}$, and writes their components as $\mathbf{r} = (x, y, z)$. The **magnitude** (length) is

$$r = |\mathbf{r}| = \sqrt{x^2 + y^2 + z^2},$$

in *italic*, and the **unit vector** in the direction of $\mathbf{r}$ wears a hat, $\hat{\mathbf{r}} = \mathbf{r}/r$, with $|\hat{\mathbf{r}}| = 1$. Vectors add component by component, and multiplying a vector by a number scales its length (and reverses it if the number is negative). The difference $\mathbf{r}_k - \mathbf{r}_j$ is the vector *from* $\mathbf{r}_j$ *to* $\mathbf{r}_k$; its magnitude is the distance between the two points. That one fact is the whole content of Chapter 3's force law.

B.1.1. The dot product

The **dot product** of two vectors is a number:

$$\mathbf{a} \cdot \mathbf{b} = a_x b_x + a_y b_y + a_z b_z = |\mathbf{a}| |\mathbf{b}| \cos \theta, \quad (\text{B.1})$$

where θ is the angle between them. It is commutative ($\mathbf{a} \cdot \mathbf{b} = \mathbf{b} \cdot \mathbf{a}$), distributes over addition, and is zero when the vectors are perpendicular. Two uses recur throughout the book. First, a vector dotted with itself is its squared length, $\mathbf{r} \cdot \mathbf{r} = r^2$, which is how the book gets from vector equations to scalar ones (Chapter 4's energy derivation). Second, the dot product with a unit vector picks out the component along that direction: $\mathbf{v} \cdot \hat{\mathbf{r}}$ is the part of the velocity that points radially.

B. Mathematical Background

B.1.2. The cross product

The **cross product** of two vectors is a vector:

$$\mathbf{a} \times \mathbf{b} = (a_y b_z - a_z b_y, a_z b_x - a_x b_z, a_x b_y - a_y b_x). \quad (\text{B.2})$$

Its magnitude is $|\mathbf{a}||\mathbf{b}|\sin\theta$, which is the area of the parallelogram the two vectors span (so $\frac{1}{2}|\mathbf{a} \times \mathbf{b}|$ is the area of the triangle; Chapter 4 uses this for Kepler’s second law). Its direction is perpendicular to both $\mathbf{a}$ and $\mathbf{b}$, by the right-hand rule: curl the fingers of your right hand from $\mathbf{a}$ toward $\mathbf{b}$ and the thumb points along $\mathbf{a} \times \mathbf{b}$. The properties the book uses:

- It is **anticommutative**: $\mathbf{b} \times \mathbf{a} = -\mathbf{a} \times \mathbf{b}$.
- Any vector crossed with itself, or with a multiple of itself, is zero: $\mathbf{a} \times \mathbf{a} = \mathbf{0}$, $\mathbf{a} \times (c\mathbf{a}) = \mathbf{0}$. This is why angular momentum is conserved under a central force (Chapter 4).
- It distributes over addition and obeys the product rule under differentiation (next section).

Angular momentum, $\mathbf{r} \times \mathbf{v}$, is the cross product you will see most. The pattern of indices in Equation B.2 is worth memorizing once: each component uses the other two coordinates, cyclically ($x \rightarrow y \rightarrow z \rightarrow x$), and the code in Chapters 4, 7, and 12 is a direct transcription of it.

B.1.3. Two triple-product identities

Chapter 4’s derivation of the orbit equation uses two identities for three vectors.

The **scalar triple product** $\mathbf{a} \cdot (\mathbf{b} \times \mathbf{c})$ is the volume of the parallelepiped spanned by the three, and it is unchanged by cycling the vectors:

$$\mathbf{a} \cdot (\mathbf{b} \times \mathbf{c}) = \mathbf{b} \cdot (\mathbf{c} \times \mathbf{a}) = \mathbf{c} \cdot (\mathbf{a} \times \mathbf{b}). \quad (\text{B.3})$$

It is zero if any two of the vectors are parallel or if all three lie in a plane. The book uses it to turn $\mathbf{r} \cdot (\mathbf{v} \times \mathbf{h})$ into $\mathbf{h} \cdot (\mathbf{r} \times \mathbf{v}) = h^2$.

The **vector triple product** expands as

$$\mathbf{a} \times (\mathbf{b} \times \mathbf{c}) = \mathbf{b}(\mathbf{a} \cdot \mathbf{c}) - \mathbf{c}(\mathbf{a} \cdot \mathbf{b}), \quad (\text{B.4})$$

the “BAC minus CAB” rule. Note that the result lies in the plane of $\mathbf{b}$ and $\mathbf{c}$, and that the parentheses matter: $(\mathbf{a} \times \mathbf{b}) \times \mathbf{c}$ is a different vector. Both identities can be verified by writing out components with Equation B.2; it is tedious once and then you never doubt them again.

B.2. Derivatives of vectors

A vector that changes with time is differentiated component by component: if $\mathbf{r}(t) = (x(t), y(t), z(t))$ then $\dot{\mathbf{r}} = (\dot{x}, \dot{y}, \dot{z})$. The book uses Newton's dot for time derivatives: $\dot{\mathbf{r}}$ is velocity, $\ddot{\mathbf{r}}$ is acceleration. Velocity is also written $\mathbf{v}$ and acceleration $\mathbf{a}$ when the context is clear.

The product rules work for every kind of product, as long as the order of the factors is kept in the cross product:

$$\frac{d}{dt}(f\mathbf{a}) = \dot{f}\mathbf{a} + f\dot{\mathbf{a}}, \quad \frac{d}{dt}(\mathbf{a} \cdot \mathbf{b}) = \dot{\mathbf{a}} \cdot \mathbf{b} + \mathbf{a} \cdot \dot{\mathbf{b}}, \quad \frac{d}{dt}(\mathbf{a} \times \mathbf{b}) = \dot{\mathbf{a}} \times \mathbf{b} + \mathbf{a} \times \dot{\mathbf{b}}. \quad (\text{B.5})$$

Two consequences do a lot of work in Chapter 4.

The derivative of a length. Differentiating $r^2 = \mathbf{r} \cdot \mathbf{r}$ gives $2r\dot{r} = 2\mathbf{r} \cdot \dot{\mathbf{r}}$, so

$$\dot{r} = \frac{\mathbf{r} \cdot \dot{\mathbf{r}}}{r} = \hat{\mathbf{r}} \cdot \dot{\mathbf{r}}. \quad (\text{B.6})$$

The rate at which the distance changes is the radial component of the velocity. Note that $\dot{r}$ (the derivative of the length) is not $|\dot{\mathbf{r}}|$ (the length of the derivative, the speed); on a circular orbit the first is zero and the second is not.

The derivative of a unit vector. By the quotient rule and Equation B.6,

$$\frac{d}{dt} \left(\frac{\mathbf{r}}{r} \right) = \frac{\dot{\mathbf{r}}}{r} - \frac{\mathbf{r}\dot{r}}{r^2} = \frac{\dot{\mathbf{r}}}{r} - \frac{\mathbf{r}(\hat{\mathbf{r}} \cdot \dot{\mathbf{r}})}{r^3}. \quad (\text{B.7})$$

This is the expression that the vector triple product produces in Chapter 4's eccentricity-vector derivation, and recognizing it is the step that makes the derivation close.

A function of a vector, like the potential $-\mu/r$, is differentiated through the chain rule: $\frac{d}{dt}(-\mu/r) = \mu\dot{r}/r^2$, and then Equation B.6.

B.3. Polar coordinates in the plane

Two-body orbits are planar (Chapter 4 proves it), so most of the book's geometry lives in a plane, where a point can be given either by Cartesian coordinates (x, y) or by its distance r from the origin and its angle θ from the x axis:

$$x = r \cos \theta, \quad y = r \sin \theta, \quad r = \sqrt{x^2 + y^2}, \quad \theta = \text{atan2}(y, x).$$

The orbit equation of Chapter 4 is a relation between r and θ . To differentiate it you need the velocity in polar form. Define the unit vectors $\hat{\mathbf{r}} = (\cos \theta, \sin \theta)$, pointing outward, and $\hat{\boldsymbol{\theta}} = (-\sin \theta, \cos \theta)$, pointing in the direction of increasing θ . Unlike the Cartesian unit

B. Mathematical Background

vectors, these rotate as the point moves: $\dot{\hat{\mathbf{r}}} = \dot{\theta} \hat{\boldsymbol{\theta}}$ and $\dot{\hat{\boldsymbol{\theta}}} = -\dot{\theta} \hat{\mathbf{r}}$. Differentiating $\mathbf{r} = r\hat{\mathbf{r}}$ with the product rule,

$$\mathbf{v} = \dot{r} \hat{\mathbf{r}} + r\dot{\theta} \hat{\boldsymbol{\theta}}. \quad (\text{B.8})$$

The velocity has a radial part, $\dot{r}$, and a transverse part, $r\dot{\theta}$, and the speed squared is $\dot{r}^2 + r^2\dot{\theta}^2$. The angular momentum per unit mass is $h = |\mathbf{r} \times \mathbf{v}| = r \cdot r\dot{\theta} = r^2\dot{\theta}$, and because h is constant for a central force this gives $\dot{\theta} = h/r^2$ directly, which Chapter 7 uses to turn the orbit equation into the velocity formulas. In a time dt the position vector sweeps out a thin triangle of area $\frac{1}{2}r \cdot r\dot{\theta} dt$, so the area swept per unit time is $h/2$: Kepler's second law in one line.

B.4. Conic sections

Every two-body orbit is a conic section: the curve you get by slicing a cone with a plane. Chapter 4 derives this; here is the geometry it needs.

An **ellipse** is the set of points whose distances to two fixed points, the **foci**, add up to a constant, $2a$. It has a longest diameter $2a$ (the major axis) and a shortest $2b$ (the minor axis); a is the **semi-major axis** and b the **semi-minor axis**. The foci lie on the major axis at a distance $c = \sqrt{a^2 - b^2}$ from the center, and the **eccentricity** is $e = c/a$, between 0 (a circle, both foci at the center) and 1 (a degenerate line). The relations the book uses:

$$b = a\sqrt{1 - e^2}, \quad \text{area} = \pi ab, \quad r_{\min} = a(1 - e), \quad r_{\max} = a(1 + e), \quad (\text{B.9})$$

where $r_{\min}$ and $r_{\max}$ are the nearest and farthest distances from a focus (periapsis and apoapsis, when the focus is a star). With the origin at one focus and ν the angle from the direction of the nearest point, the ellipse is

$$r(\nu) = \frac{a(1 - e^2)}{1 + e \cos \nu} = \frac{p}{1 + e \cos \nu}, \quad (\text{B.10})$$

where $p = a(1 - e^2)$ is the **semi-latus rectum**, the distance from the focus to the curve at $\nu = 90^\circ$. Check it: $\nu = 0$ gives $a(1 - e)$ and $\nu = \pi$ gives $a(1 + e)$. This polar form is the one that falls out of Newton's law, and it is why the orbiting body is at a focus rather than at the center.

The same equation with $e = 1$ describes a **parabola**, which reaches infinity at $\nu = \pm 180^\circ$, and with $e > 1$ a **hyperbola**, which reaches infinity at the **asymptotes** $\nu = \pm \arccos(-1/e)$; between them the denominator is positive and the curve is a single open branch. For a hyperbola the semi-major axis is conventionally negative, $a < 0$, so that $p = a(1 - e^2) > 0$ and the periapsis distance $a(1 - e) > 0$ come out right from the same formulas. The angle between the two asymptotes, which is how much a flyby bends a trajectory, is $2 \arcsin(1/e)$. Chapter 13 uses all of this.

B.5. Taylor series

If a function is smooth, its value a short distance ahead can be built from its derivatives at the current point:

$$f(t + \Delta t) = f(t) + f'(t) \Delta t + \frac{f''(t)}{2} \Delta t^2 + \frac{f'''(t)}{6} \Delta t^3 + \dots \quad (\text{B.11})$$

The general term is $f^{(n)}(t) \Delta t^n / n!$. For the position of a body this reads $\mathbf{r}(t + \Delta t) = \mathbf{r} + \mathbf{v}\Delta t + \frac{1}{2}\mathbf{a}\Delta t^2 + \dots$, which is where every integrator in Chapter 5 comes from: an integrator is a decision about where to cut the series off and what to do about the terms you cut.

The **big-O** notation summarizes the size of what was cut. Writing “ $+O(\Delta t^3)$ ” means the omitted terms are no larger than a constant times Δt^3 when Δt is small. The practical meaning is in the scaling: an $O(\Delta t^3)$ error shrinks by a factor of eight when the step is halved, an $O(\Delta t^2)$ error by four. If a method’s error over one step is $O(\Delta t^{p+1})$, then over a fixed time interval, which takes $1/\Delta t$ steps, the accumulated error is $O(\Delta t^p)$, and the method is said to be of order p . Chapter 5 measures these exponents directly.

Three special cases of Equation B.11 appear so often that it is worth knowing them by sight, with x small:

$$(1 + x)^n \approx 1 + nx, \quad \sin x \approx x, \quad e^x \approx 1 + x. \quad (\text{B.12})$$

The first, the **binomial approximation**, is how Chapter 6 estimates the cost of softening: $(1 + \varepsilon^2/r^2)^{-3/2} \approx 1 - \frac{3}{2}\varepsilon^2/r^2$. The last is how a small growth rate turns into exponential growth over many steps: multiplying by $(1 + x)$ a total of N times gives $(1 + x)^N \approx e^{Nx}$.

B.6. Differential equations and the idea of state

Newton’s second law for one body, $\ddot{\mathbf{r}} = \mathbf{a}(\mathbf{r})$, is a **second-order differential equation**: it involves the second derivative of the unknown function $\mathbf{r}(t)$. To solve it you need two pieces of starting information, the position and the velocity at some instant, because the equation only tells you how the velocity *changes*. Those two pieces are the **state**, and the whole of Chapter 3’s “simulation loop” is the observation that a second-order equation can be rewritten as two first-order ones,

$$\dot{\mathbf{r}} = \mathbf{v}, \quad \dot{\mathbf{v}} = \mathbf{a}(\mathbf{r}),$$

each of which says how one part of the state changes given the current state. Any system of higher-order equations can be reduced to first order this way, and numerical methods work on the first-order form.

A **solution in closed form** is a formula for $\mathbf{r}(t)$. The simplest example is the harmonic oscillator, $\ddot{x} = -\omega^2 x$, whose solution is $x(t) = A \cos \omega t + B \sin \omega t$, with A and B fixed by the initial position and velocity. Chapter 5 uses this equation as its test case because it is

B. Mathematical Background

exactly solvable, so an integrator's mistakes can be measured against the truth. The two-body problem has a closed-form solution too (Chapter 4). The three-body problem does not (Chapter 11), and the simulation is the only way to find $\mathbf{r}(t)$.

A **conserved quantity** (or first integral) is a function of the state that does not change along any solution, like the oscillator's energy $\frac{1}{2}\dot{x}^2 + \frac{1}{2}\omega^2 x^2$. Finding one is the standard trick for learning about a solution without solving: Chapter 4 finds three for the two-body problem and reads the shape of the orbit off them.

B.7. Matrices, rotations, and determinants

A **matrix** is a grid of numbers; a 2×2 or 3×3 matrix acts on a vector to produce another vector, by the rule “each component of the output is the dot product of a row with the input”:

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} ax + by \\ cx + dy \end{pmatrix}.$$

Applying one matrix after another is the same as applying their **product**, and the product is read right to left: $M_2 M_1$ applies M_1 first. Matrix multiplication is not commutative, which is why the order of the three rotations in Chapter 7 matters.

A **rotation** of the plane counterclockwise by θ is the matrix

$$R(\theta) = \begin{pmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{pmatrix},$$

and the three-dimensional rotations about the z and x axes are this block embedded in the 3×3 identity:

$$R_z(\theta) = \begin{pmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{pmatrix}, \quad R_x(\theta) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta \\ 0 & \sin \theta & \cos \theta \end{pmatrix}. \quad (\text{B.13})$$

Rotations preserve lengths and angles. Their inverse is their transpose ($R^{-1} = R^T$), so undoing a rotation costs nothing. The sign convention here, that $R(\theta)$ rotates the *vector* counterclockwise by θ , is the one the package uses; some texts instead rotate the coordinate axes, which is the same matrix with $-\theta$.

The **determinant** of a 2×2 matrix is $ad - bc$. Its geometric meaning is the thing Chapter 5 is built on: a matrix with determinant D scales every area by a factor $|D|$ (every volume, for 3×3 matrices). A rotation has determinant 1, as it should. A map that preserves area has determinant exactly 1, and a map with determinant greater than 1 inflates every region it touches. Chapter 5 computes the determinant of each integrator's one-step map and finds $1 + \omega^2 \Delta t^2$ for Euler and exactly 1 for Euler-Cromer and Verlet; that single number is the difference between an orbit that spirals and one that does not.

One more fact about 2×2 matrices, used in a Chapter 5 exercise: the eigenvalues λ satisfy $\lambda^2 - (\text{trace})\lambda + \det = 0$. When $\det = 1$ the two eigenvalues multiply to 1, and they are a

complex-conjugate pair on the unit circle, meaning the map rotates without growing, exactly when $|\text{trace}| < 2$. That is where the stability condition $\omega\Delta t < 2$ comes from.

B.8. Logarithms, exponentials, and power laws

If $y = Cx^k$, then $\log y = \log C + k \log x$: a power law is a straight line on a plot with logarithmic axes, and its slope is the exponent. Chapter 8 checks Kepler’s third law this way (the slope is $3/2$) and Chapter 15 reads the n^2 cost of the force calculation off a log-log benchmark. Chapter 5’s convergence tables are the same idea in a table: halving Δt and watching the error fall by 2^p measures p .

If a quantity grows as $y = y_0 e^{t/\tau}$, then $\log y$ is a straight line against t with slope $1/\tau$; τ is the **e-folding time**, the time for the quantity to grow by a factor of $e \approx 2.718$. Chapter 11 measures the Lyapunov time of a chaotic triple, and the instability time of a Lagrange point, from exactly such a slope. The natural logarithm is the one that makes this work; R’s `log()` is natural by default, and `log10()` is for the axes.

B.9. Angles

The book’s formulas use **radians**; the package’s arguments use **degrees**, because that is how orbital elements are published. The conversion is π radians = 180° , so multiply degrees by $\pi/180$ to get radians, and the book’s code does this at the top of every function that takes an angle.

The angle of a point (x, y) from the x axis is `atan2(y, x)`, not `atan(y/x)`: the two-argument form knows which quadrant the point is in and returns a value in $(-\pi, \pi]$. The jump from $+\pi$ to $-\pi$ as a body crosses the negative x axis is an artifact of that range, and several chapters **unwrap** a sequence of angles, adding 2π at each jump so the angle grows continuously and a body’s total travel can be measured. The helper `unwrap()` in `R/helpers.R` does this with one line: take successive differences, wrap each into $(-\pi, \pi]$, and accumulate.

B.10. Units and dimensions

Everything in the book is in SI units: meters, kilograms, seconds. The only reason to mention it is that **dimensional analysis** is the cheapest error check there is, and the book uses it silently all the time. G has units $\text{m}^3 \text{kg}^{-1} \text{s}^{-2}$, so GM has units $\text{m}^3 \text{s}^{-2}$, GM/r has units $\text{m}^2 \text{s}^{-2}$, and $\sqrt{GM/r}$ has units of m s^{-1} : a speed, as the circular-orbit formula says it should be. $\sqrt{a^3/GM}$ has units of seconds, so $2\pi\sqrt{a^3/\mu}$ is a time, as Kepler’s third law requires. If you derive a formula and the units do not come out right, the formula is wrong; if they do, it is at worst wrong by a dimensionless factor like 2π , which is the kind of error a simulation will show you immediately.

A related habit is estimating **orders of magnitude** before computing. Earth’s orbital speed is $\sqrt{GM_\odot/r} \approx \sqrt{1.3 \times 10^{20}/1.5 \times 10^{11}} \approx \sqrt{10^9} \approx 3 \times 10^4$ m/s. Knowing that number is about 30 km/s before you run anything is what lets you notice when a simulation hands you 300.

B.11. Where each tool is used

Tool	Used in
Dot and cross products, triple-product identities	Chapters 3, 4, 7, 12
Derivatives of r and of $\mathbf{r}/r$	Chapter 4
Polar coordinates, $h = r^2\dot{\theta}$	Chapters 4, 7
Ellipse geometry and the polar conic	Chapters 4, 7, 13
Taylor series, big-O, binomial approximation	Chapters 5, 6
State form of differential equations	Chapters 3, 5
Rotation matrices and determinants	Chapters 5, 7, 10
Log-log slopes and e-folding times	Chapters 5, 8, 11, 15
<code>atan2</code> and unwrapping	Chapters 4, 8, 10, 12

C. Function Quick Reference

Every exported function in `orbitr` 1.0.0, grouped by role. Positions are in meters, velocities in m/s, masses in kg, times in seconds. Full documentation for each is at <https://orbitr.com/reference/>.

C.1. Building a system

Function	What it does	Key arguments
<code>create_system()</code>	Start an empty system	<code>G</code> (default <code>gravitational_constant</code>)
<code>add_body()</code>	Add a body by Cartesian state	<code>id</code> , <code>mass</code> , <code>x</code> , <code>y</code> , <code>z</code> , <code>vx</code> , <code>vy</code> , <code>vz</code> (all default 0)
<code>add_sun()</code>	Add the Sun at rest at the origin	<code>mass</code> (default <code>mass_sun</code>); position and velocity overridable
<code>add_body_keplerian()</code>	Add a body by orbital elements; $e > 1$ with $a < 0$ for a hyperbola	<code>id</code> , <code>mass</code> , <code>a</code> , <code>e</code> , <code>i</code> , <code>lan</code> , <code>arg_pe</code> , <code>nu</code> (degrees), <code>parent</code>
<code>add_planet()</code>	Add a named solar-system body from JPL elements	<code>name</code> , <code>parent</code> , <code>nu</code> ; any element or <code>mass</code> overridable
<code>load_solar_system()</code>	Sun, eight planets, Moon, Pluto	<code>moon = TRUE</code> , <code>pluto = TRUE</code>
<code>remove_body()</code>	Drop bodies by name	<code>id</code> (vector)
<code>get_bodies()</code>	The body table as a tibble	
<code>system_from_simulation()</code>	Rebuild a system from one time step of a run	<code>time</code> (default: the last step), <code>G</code>

C.2. Running

Function	What it does	Key arguments
<code>simulate_system()</code>	Integrate forward in time	<code>time_step</code> (default 3600), <code>duration</code> (default one year), <code>method</code> ("verlet", "euler_cromer", "euler"), <code>softening</code> (default 0), <code>use_cpp</code> (default TRUE)

C. Function Quick Reference

Function	What it does	Key arguments
<code>continue_simulation()</code>	Run on from a simulation's last state, appending rows	<code>time_step</code> , <code>duration</code> ; <code>method</code> , <code>softening</code> , <code>G</code> default to the previous segment's

Output of `simulate_system()`: a tibble with one row per body per time step and columns `time`, `id`, `mass`, `x`, `y`, `z`, `vx`, `vy`, `vz`. The run's `G`, `softening`, `method`, and `time_step` are attached as attributes.

C.3. Analysis

Function	What it does	Key arguments
<code>shift_reference_frame()</code>	Re-center the output on one body, or on the center of mass	<code>center_id</code> (a body id or "barycenter"), <code>keep_center</code> = TRUE
<code>get_orbital_elements()</code>	Osculating <code>a</code> , <code>e</code> , <code>i</code> , <code>lan</code> , <code>arg_pe</code> , <code>nu</code> , <code>period</code> of a body about a parent, at every time step	<code>body</code> , <code>parent</code> , <code>mu</code> (default GM_{parent})
<code>get_energy()</code>	Total kinetic, potential, energy at every time step	<code>G</code> , <code>softening</code> (default: as recorded in the run)
<code>get_momentum()</code>	Total linear momentum <code>px</code> , <code>py</code> , <code>pz</code>	
<code>get_angular_momentum()</code>	Total angular momentum <code>Lx</code> , <code>Ly</code> , <code>Lz</code>	
<code>conserved_quantities()</code>	All of the above joined, plus <code>energy_error</code> , <code>momentum_error</code> , <code>angular_momentum_error</code>	<code>G</code> , <code>softening</code>

C.4. Plotting

Function	What it does	Key arguments
<code>plot_orbits()</code>	Full trajectories; ggplot in 2D, plotly if any <code>z</code> motion	<code>three_d</code>
<code>plot_orbits_3d()</code>	Interactive 3D trajectories	
<code>plot_system()</code>	Snapshot at one time	<code>time</code> , <code>trails</code> , <code>three_d</code>
<code>plot_system_3d()</code>	Interactive 3D snapshot	<code>time</code> , <code>trails</code>
<code>animate_system()</code>	GIF animation (ganimate) or plotly if 3D	<code>fps</code> , <code>duration</code> , <code>trails</code> , <code>three_d</code>
<code>animate_system_3d()</code>	Interactive plotly animation	<code>fps</code> , <code>duration</code> , <code>trails</code>

Function	What it does	Key arguments
----------	--------------	---------------

C.5. Saving

Function	What it does
<code>save_system()</code> / <code>load_system()</code>	Write and read a system as <code>.rds</code>
<code>export_bodies()</code>	Write the body table as CSV

C.6. Helpers defined in this book

These are not part of the package. Each is defined in full where it first appears; those marked with an asterisk are also collected in `R/helpers.R` in the book's source, which every chapter loads. Several of them do by hand what a package function does in one call, and are kept in the text for the derivation.

Helper	Defined in	Purpose	Package equivalent
<code>accelerations()</code>	Chapter 3	N-body accelerations in plain R	the C++ kernel
<code>relative_state()</code> *	Chapter 4	Relative position and velocity of one body about another	<code>shift_reference_frame()</code>
<code>eccentricity()</code> *	Chapter 4	Eccentricity vector at every time step	<code>get_orbital_elements()</code>
<code>measure_period()</code> *	Chapter 4	Orbital period measured from the time series (independent of Kepler's third law)	
<code>total_energy()</code>	Chapter 5	The energy sum written out	<code>get_energy()</code>
<code>keplerian_to_state()</code> *	Chapter 7	Orbital elements to a Cartesian state vector	inside <code>add_body_keplerian()</code>
<code>state_to_elements()</code> *	Chapter 7	Cartesian state vector to osculating elements	<code>get_orbital_elements()</code>
<code>solve_kepler()</code> *, <code>true_anomaly_at()</code> *	Chapter 7	Kepler's equation; true anomaly at a given time since periapsis	

C. Function Quick Reference

Helper	Defined in	Purpose	Package equivalent
<code>unwrap()</code> *	Chapter 8	Make a sequence of angles continuous past $\pm\pi$	
<code>add_binary()</code> , <code>add_eccentric_binary()</code>	Chapter 9	Place two bodies on a circular or eccentric mutual orbit with zero total momentum	
<code>body_energies()</code>	Chapter 11	Each body's energy relative to the rest of the system	
<code>time_since_periapsis()</code>	Chapter 13	Kepler's equation in reverse: time from periapsis to a true anomaly	
<code>closest_approach()</code>	Chapter 16	Minimum separation of every pair over a run	
<code>kick_drag()</code>	Chapter 16	Apply a velocity kick to one body at the last time step	

D. Physical Constants

orbitr exports the constants below, all in SI units. The masses and semi-major axes are from the JPL DE440 ephemeris (Park et al. 2021); the gravitational constant is the CODATA 2018 value (Tiesinga et al. 2021); the time constants are exact (a year here is the Julian year of 365.25 days). The table is generated from the installed package, so it always matches the version the book was built with.

The “distance” constants are semi-major axes, not current separations. Every orbit is an ellipse, and the semi-major axis is the one length that determines its period (Chapter 4). Paired with the mean orbital speed it gives a near-circular orbit that closely approximates the real one; for the real ellipse, use `add_planet()` or `add_body_keplerian()` (Chapter 7).

Table D.1.: Constants exported by orbitr, read from the installed package.

Constant	Value	Unit
gravitational_constant	6.6743e-11	$\text{m}^3 \text{kg}^{-1} \text{s}^{-2}$
seconds_per_hour	3.6000e+03	s
seconds_per_day	8.6400e+04	s
seconds_per_year	3.1558e+07	s
mass_sun	1.9890e+30	kg
mass_mercury	3.3010e+23	kg
mass_venus	4.8670e+24	kg
mass_earth	5.9720e+24	kg
mass_moon	7.3420e+22	kg
mass_mars	6.4170e+23	kg
mass_jupiter	1.8980e+27	kg
mass_saturn	5.6830e+26	kg
mass_uranus	8.6810e+25	kg
mass_neptune	1.0240e+26	kg
mass_pluto	1.3090e+22	kg
distance_mercury_sun	5.7910e+10	m
distance_venus_sun	1.0820e+11	m
distance_earth_sun	1.4960e+11	m
distance_earth_moon	3.8440e+08	m
distance_mars_sun	2.2790e+11	m
distance_jupiter_sun	7.7850e+11	m
distance_saturn_sun	1.4340e+12	m
distance_uranus_sun	2.8710e+12	m
distance_neptune_sun	4.4950e+12	m
distance_pluto_sun	5.9060e+12	m
speed_mercury	4.7360e+04	m/s

D. Physical Constants

Table D.1.: Constants exported by orbitr, read from the installed package.

Constant	Value	Unit
speed_venus	3.5020e+04	m/s
speed_earth	2.9780e+04	m/s
speed_moon	1.0220e+03	m/s
speed_mars	2.4070e+04	m/s
speed_jupiter	1.3060e+04	m/s
speed_saturn	9.6800e+03	m/s
speed_uranus	6.8000e+03	m/s
speed_neptune	5.4300e+03	m/s
speed_pluto	4.7400e+03	m/s

About the Author

Dave Rosenman is a software engineer and data scientist. He holds a B.S. in Physics from The College of New Jersey and an earlier degree in Political Science from Temple University, and is working toward an M.S. in Computer Science at Georgia Tech, specializing in artificial intelligence. He is currently an Infrastructure and Operations Engineer at Optum.

He began his career in R at the New Jersey Department of Environmental Protection, where he was the lead programmer on the *2016 New Jersey Integrated Water Quality Assessment Report*, the state's assessment of its surface waters under the Clean Water Act. He built the application, in R, that downloaded and assessed the water quality data behind the report, and is listed as its co-author. Since then his work has mostly combined SQL Server with R and Python.

He is the author of `orbitr`, the R package this book is built on, and of other open-source packages, among them `sqlserverconnect`, an R package on CRAN for connecting R to SQL Server, and `polars_mssql`, a Python package on PyPI for working with SQL Server data in Polars. He lives in Newtown, Pennsylvania, with his wife, Kerry.

Where to find things

- **orbitr documentation:** <https://orbit-r.com> — the package reference, the articles, and the interactive app. Start here for how any function works.
- **GitHub:** <https://github.com/DRosenman> — the package source (`orbitr`), this book's source (`orbitr-book`), and his other open-source work. Bug reports and pull requests for either go there.
- **Personal site:** <https://daverosenman.com> — projects, writing, and contact.

If you use the book in a course, or build something with the package, he would like to hear about it.

