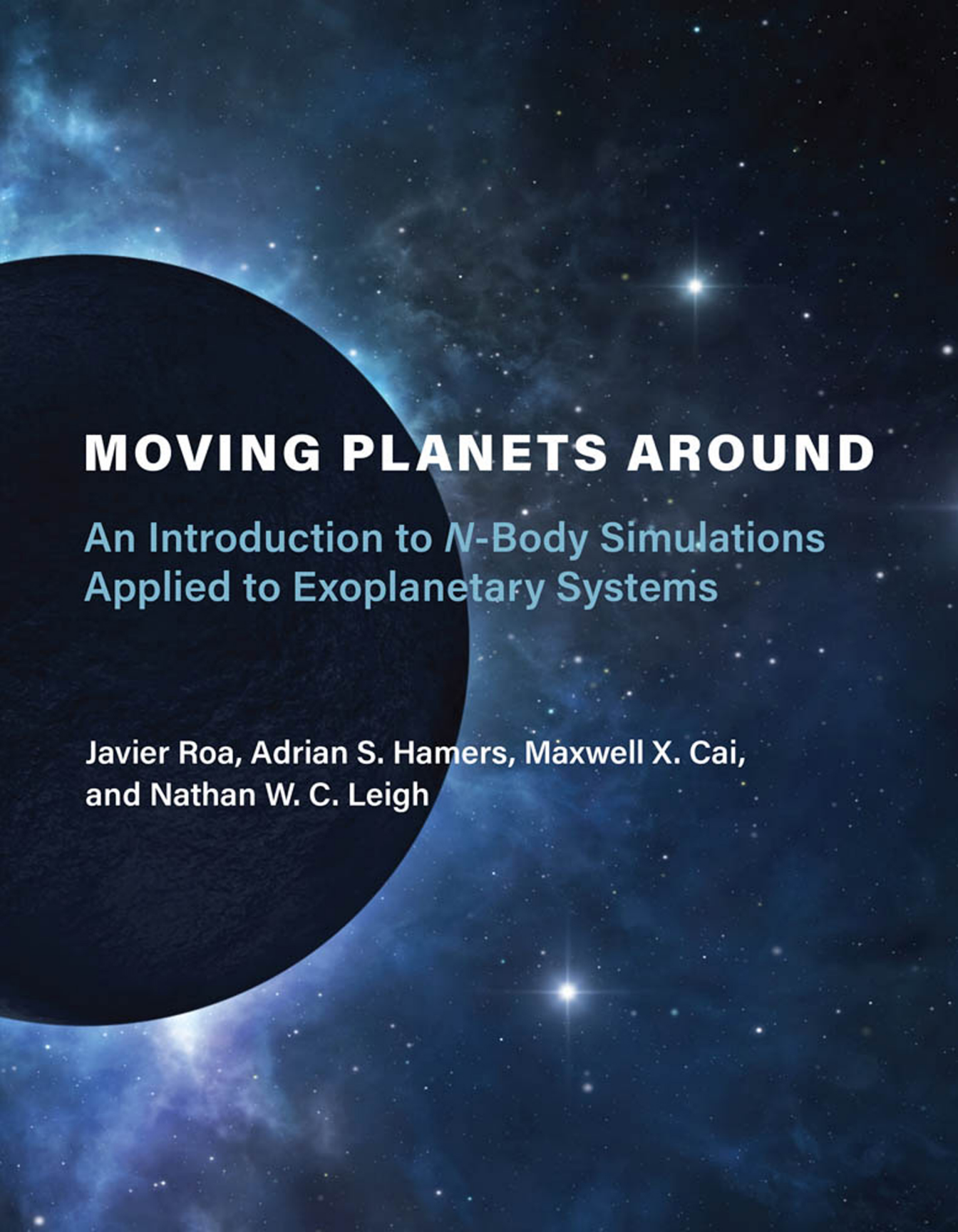

MOVING PLANETS AROUND
An Introduction to N-Body Simulations Applied to Exoplanetary Systems
Javier Roa, Adrian S. Hamers, Maxwell X. Cai, and Nathan W. C. Leigh

Moving Planets Around

Moving Planets Around

An Introduction to *N*-Body Simulations Applied to Exoplanetary Systems

Javier Roa, Adrian S. Hamers, Maxwell X. Cai, and Nathan W. C. Leigh

The MIT Press
Cambridge, Massachusetts
London, England

This book was set in Syntax and Times New Roman by the authors.

Library of Congress Cataloging-in-Publication Data

Names: Roa, Javier (Javier Roa Vicens), author. | Hamers, Adrian S. (Adrian Sven), 1988- author. | Cai, Maxwell X. (Maxwell Xu), author. | Leigh, Nathan W. C., author.
Title: Moving planets around : an introduction to n-body simulations applied to exoplanetary systems / Javier Roa, Adrian S. Hamers, Maxwell X. Cai, and Nathan W. C. Leigh.
Description: Cambridge, Massachusetts : The MIT Press, [2019] | Includes bibliographical references and index.
Identifiers: LCCN 2019057050 | ISBN 9780262539340 (paperback)
Subjects: LCSH: Extrasolar planets–Orbits–Computer simulation. | Many-body problem.
Classification: LCC QB820 .R63 2019 | DDC 523.2/4–dc23
LC record available at https://lccn.loc.gov/2019057050

Dr. Javier Roa
Jet Propulsion Laboratory
California Institute of Technology[1]
Pasadena, CA 91109, USA

Dr. Adrian S. Hamers
School of Natural Sciences
Institute for Advanced Study
Princeton, NJ 08540, USA

Max Planck Institute for Astrophysics
Karl-Schwarzschild-Str. 1
Garching, D-85741, Germany

Dr. Maxwell X. Cai (蔡翔)
Leiden Observatory
Leiden University
2300 RA Leiden, The Netherlands

SURF Cooperative
1098 XG Amsterdam
The Netherlands

Dr. Nathan W. C. Leigh
Departamento de Astronomía
Facultad de Ciencias Físicas y Matemáticas
Universidad de Concepción
Concepción, Chile

Department of Astrophysics
American Museum of Natural History
New York, NY 10024, USA

[1] This work was done as a private venture and not in the author's capacity as an employee of the Jet Propulsion Laboratory, California Institute of Technology.

Contents

List of Code Snippets

Acknowledgments

This book would not have been possible without the inspiration and support by Prof. Piet Hut (IAS) and Prof. Jun Makino (RIKEN). We are extremely grateful to them for getting us involved in this endeavor and for supporting us along the way by providing feedback on important decisions and even inviting us to RIKEN in Kobe, Japan, to meet in person and to work on the book. We thank Yuko Wakamatsu, Yoshie Yamaguchi, and the other support staff at RIKEN for taking care of many practical details of our unforgettable trip to Japan.

We also thank Piet Hut for writing a foreword in which he puts our new work in perspective of the history of the development of astrophysical computer software and its educational aspects.

Lastly, we thank Prof. Scott Tremaine (IAS) and Prof. Simon Portegies Zwart (Leiden University) for carefully reading our manuscript and for writing statements of recommendation.

Foreword

This book is unique in providing all that you need to start writing your own software for simulating the dynamical evolution of planets, stars, or other heavenly bodies. For decades, the knowledge gathered here between two covers could only be found scattered across the literature, at best, while partly existing only in the form of oral traditions, passed on from teacher to student or just from student to student.

The authors have asked me to write this foreword in order to provide a bit of prehistory, spanning more than three decades, leading up to the writing of this book. It started in 1985, at the Institute for Advanced Study in Princeton, when Joshua Barnes and I were developing new algorithms for stellar dynamics, such as the tree code that could efficiently speed up gravitational interactions from $\mathcal{O}(N^2)$ to $\mathcal{O}(N \log N)$.[1]

At that time, both of us already had years of experience writing, running, and analyzing the results of N-body codes. In conversations, we both deplored the fact that there were no good books or even review articles about developing such codes. As a result, many users (then and even now!) relied on codes that were at least partly treated like black boxes. This carries a double risk: at the side of the code writer or modifier, there may be latent bugs in the code that had not shown up before and cannot easily be traced, and at the side of the code user, it is all too easy to use a black-boxed code outside the range of applicability that the writer had foreseen.

In addition, Josh and I were frustrated by the clumsy way in which N-body codes were used back then. There was no standard, well-documented software environment in which to develop and test new codes or modify and improve old ones. As a result, just about any astrophysicist working in stellar dynamics had cobbled together a grab bag of programs and scripts to run their own codes, with very little compatibility among them.

In order to address both problems, we developed our own toolbox of small codes that could be combined, using the modularity of the Unix operating system. We used the Unix pipes system to let the user choose a module for setting up initial conditions, from which

[1] Barnes, J. & Hut, P., 1986, *Nature* 324, 446–449.

the data would flow to an integrator that would be used for the actual run, and from there on to one or more analysis programs. A popular article about our approach was published soon afterward in *Scientific American* by Gerald Sussman and me.[2]

We quickly attracted other users for our toolbox-based environment. During the next few years, our software forked into a more focused branch and a more general branch with wrappers containing programs provided by others. Josh continued to develop the first branch, as the original version, Zeno.[3] Peter Teuben, who had later joined our efforts, maintained a publicly supported version, Nemo.[4]

Meanwhile, I started to design a third version, Starlab, with a different underlying data structure. While Zeno and Nemo were very efficient for conducting experiments in galactic dynamics, through the use of homogeneous arrays to store particle data, they lacked the ability to simulate small-N systems such as star clusters. In order to model the role of binary stars, for example, a more flexible data format was called for, allowing individual stars to be treated separately, each in a different way.

I also used this extra freedom to develop a flexible way of reporting interesting events, through what I called a "story mechanism," in which stars undergoing unusual interactions could scribble their own reports. These were implemented as ascii strings, with data structures that could be designed on the fly, as a kind of forerunner of scripting languages, all written in C. In this way, I laid the groundwork for Starlab, with inspiration from helpful conversations with Jun Makino, whom I was visiting during my first sabbatical, at Tokyo University.

After returning to Princeton, Steve McMillan joined me in the development, and soon afterward we met Simon Portegies Zwart, who offered his stellar evolution code to be integrated with the dynamics part of the new environment. By that time, we switched to C++, mainly to use the convenience of data encapsulation, from vector notation, to carrying stellar evolution data around. With Jun Makino joining us as well, the four of us spent a decade to extend Starlab into a workhorse for simulating star cluster evolution.

In the spring of 2001, Jun Makino and I began to develop a vision for a simpler and more educational version of Starlab while hiking in the desert near Tucson before attending one of the first astrophysics conferences on virtual observatories. We worked on two parallel approaches: first to write an introduction to stellar dynamics, from scratch, and second to write an extension in which we introduced very simple, almost cartoon-like versions of stellar evolution.

---

[2] Hut, P. & Sussman, G. J., 1987, *Scientific American* 255, 145–153.

[3] http://www.ifa.hawaii.edu/~barnes/zeno/index.html

[4] https://bima.astro.umd.edu/nemo/index_old.html

In the fall of 2001, we started to work on the introduction, which we finished in 2003 as an online book *Moving Stars Around*, written in C++, followed a few years later by a version with the same title, written in RUBY, a scripting language much like PYTHON.[5]

In the summer of 2002, we produced our first cartoon version of stellar evolution, during the inaugural MODEST meeting in the Museum for Natural History in New York City.[6] MODEST stands for MOdeling DEnse STellar systems.[7]

A central theme of the MODEST initiative was to develop a multipurpose software environment for astrophysical applications in which different existing numerical codes are incorporated into a single framework. Using STARLAB, compiled as a single code, as our taking-off point, the first steps toward such an environment were taken in Amsterdam in 2006, at the MODEST-6d meeting. The environment was called MUSE, for MUltiscale MUltiphysics Simulation Environment.

MUSE, in turn, led to AMUSE, for Astrophysical Multipurpose Software Environment, developed at Leiden Observatory by Simon Portegies Zwart and coworkers, supported by a multimillion Euro grant from the Dutch government. The AMUSE kickoff workshop was held there in the fall of 2009, followed by various other meetings. Further development remains ongoing, and the package is well maintained, with a friendly staff happy to help users who run into problems.[8]

A very clear and helpful introduction to AMUSE can be found in a recent book.[9] In many ways, their book is complementary to the current book. Where this book teaches you to write and understand individual codes in detail, the AMUSE book teaches you to set up larger experiments by using individual codes as gray or black boxes that the user can combine by using the tools in the AMUSE environment.

Fast-forwarding to the fall of 2016, I had some conversations about few-body dynamics with Adrian Hamers and Javier Roa at IAS in Princeton. When we talked about various aspects of software development, I mentioned my work with Jun Makino on an ambitious project called the *Art of Computational Science*, ACS for short, that Jun and I had started in 2003.[10] I added that we had lost momentum after a few years, for various reasons, although our website was still being used by many visitors, either for the codes presented there or the educational texts that we provided.

---

[5] http://www.artcompsci.org/#msa

[6] Hut, P., Shara, M. M., Aarseth, S. J., Klessen, R. S., Lombardi, J. C., Makino, J., McMillan, S., Pols, O. R., Teuben, P. J. & Webbink, R. F., 2003, *New Astronomy* 8, 337–370.

[7] http://www.manybody.org/modest/

[8] http://amusecode.org/

[9] Portegies Zwart, S. & McMillan, S., 2019, *Astrophysical Recipes: The Art of AMUSE*, Iop Publishing Ltd.

[10] http://www.artcompsci.org/

I especially mentioned the *Moving Stars Around* books, in their C++ and RUBY versions, mentioned above. I also made a casual remark that it would be nice if someone would write something similar, in a more modern way, and actually carry it to completion in the form of a published book. To my happy surprise, they showed great interest in doing so and immediately started scheming as to how that could be done.

When I next talked with Jun Makino, he invited Adrian and Javier and me to visit him in Kobe, in the summer of 2017, to discuss together what such a newer version of our project might look like. By the time we all met, Maxwell Cai and Nathan Leigh had joined as well, and the six of us had a great time discussing the options for a new book.

Jun and I strongly felt that it would be best for them to start from scratch and only use the few ingredients of our original online book that they found useful. We suggested the use of a dialogue as a pedagogical device and also the inclusion of a discussion about debugging, something that is typically left out in books about software. But other than that, all the ideas presented in the current book are those of the authors.

Indeed, the current book has incorporated many great improvements over the original text. Instead of stars, the particles now represent exoplanets, a very timely and important topic of both observational and theoretical studies. Instead of RUBY, the scripting language used is PYTHON, by now the obvious choice for a widespread and well-supported language, suitable for numerical work. The actual integrators presented and developed in the book from scratch also go well beyond the much more elementary algorithms used in the original version and are among the best that are currently used at the cutting edge of research using simulations of exoplanet dynamics.

Most important, the current book includes a realistic student research project, leading to an actual publication in the scientific literature; see the last five chapters of the book. This was a great idea, since there is no better way to teach something than to show a full example of how to actually do something, from beginning to end. I was stunned when they handed me the draft of their manuscript, since I had never seen anything like that in a textbook.

When I asked them whether they were thinking of publishing their research article and, if so, whether they would do so under the names of the protagonists in the book, Alice and Bob, they smiled. They told me they had already submitted the paper[11] but after some discussion had decided to use their own names, in order not to create confusion about the identity of the mysterious Alice and Bob authors.

So in this book, you will find an abundance of original work, from a how-to manual of designing and implementing state-of-the-art simulation codes, up to and including a how-to manual for publishing your first article. The proof of the pudding may be in the eating, but

[11] Hamers, A. S., Cai, M. X., Roa, J. & Leigh, N., 2018, *Monthly Notices of the Royal Astronomical Society* 480, 3800–3811.

the proof of a piece of research is in the publishing—for real students reading this book as well as for the fictional students who set the example in the last few chapters of this book.

When Jun Makino and I invited the authors to visit us in Kobe two years ago, we never expected to see such a great book growing so quickly out of the small seed that we handed them to get started. Congratulations, Javier, Adrian, Maxwell, and Nathan!

Piet Hut
Institute for Advanced Study
Princeton, NJ
March 2019

Prologue

Exciting times are upon us. In the past decade, the number of planets known to be orbiting stars other than our Sun has skyrocketed. The numbers are presently in the thousands and rising steadily. The Universe, and our place in it, will never look the same.

Observations of these extrasolar planets have revealed countless treasures. It turns out that our own Solar System is not unique; multiplanet architectures exist around other stars as well. The properties of these planets are diverse, ranging from fiery hot rocky worlds to icy gas giants. The planetary zoo takes all kinds.

Planets orbit their host stars due to the attractive force of gravity; it is the most massive body in the system that sets the stage for the evolution that unfolds, since the acceleration due to gravity imparted by the host star at a given distance is independent of the specifics of the orbiting body. Its trajectory through space and time can be calculated explicitly with spectacular precision, thanks to pioneering efforts by Newton and Kepler. This so-called two-body problem describes the orbit of a given planet about its host star.

Consider observing an exoplanet using some state-of-the-art telescope. If the planet passes in front of the star along the line-of-sight to your telescope, then you will observe a temporary dip in the light emitted from the star. If you wait long enough, you will observe another very similar dip. The time between these two dips in the observed emission of the star corresponds to one year in the frame of reference of the planet. This time scale, which corresponds to one revolution of the planet about its host star, can be calculated using Kepler's laws.

If a third body is randomly thrown into the mix, all hell breaks loose. A solution to the *general* "three-body problem" has evaded researchers for centuries. Analytic solutions have only been found for small subsets of the total phase space. Typically, these are idealized cases that remain dynamically stable for long periods of time, narrowly escaping a chaotic and often violent fate.

In the *restricted* three-body problem, for example, one of the particles is assumed to be massless. This approximation works quite well for the Earth–Moon–Sun system. But can such a simplified approach be applied to the myriad of extrasolar planetary architectures

we now know exist? Nature is rarely so accommodating. As we will learn, the observations suggest that in this regard, these far-off worlds are not going to cooperate.

In this book, we will study the role of gravity in dictating the time evolution of extrasolar multiplanet systems. This is one subset of the more general problem of N bodies moving in the mutual gravitational field. As you might imagine, the N-body problem is a nightmare for pen-and-paper physicists. The rest of us are not as defenseless. Computers have revolutionized the field of gravitational dynamics over the past few decades. Such a powerful weapon can be brought to bear against the horde of alien worlds currently being discovered. We will lay waste to the parameter space, using machines to model the dynamics of these planets with deadly accuracy.

The Solar System in a Box

Our goal is to study planetary orbital dynamics. While an effectively infinite number of options are before us, most do not come along with the desired accuracy and precision. To achieve this last goal, we will need computers. We will create a software package, dubbed an "N-body integrator," designed to calculate the orbits of planets as they move through time and space. Given a planetary architecture and a set of initial positions and velocities for the planets, the software will accurately predict the long-term system evolution. It will be able to address questions like: Is the system stable over millions and even billions of years? Can planets orbit within the habitable zones of their host stars? If so, how many planets? What conditions are needed for asteroids to collide with planets? To be captured as moons? And so on.

In this book, we will learn how to design a modern gravitational N-body integrator from scratch. We will peek under the hood, to fully appreciate the sometimes subtle issues that need to be addressed in order to ensure the development of a fast and reliable code. Here are a few things you will pick up along the way:

- How to lay the foundations of a flexible software environment, designed to be easily adapted and evolved
- How to apply the software you develop to an infinite number of astrodynamics problems, equipped with the knowledge needed to ensure that your calculations yield results consistent with the laws of physics
- How to develop other completely independent large-scale software environments, whether or not they have anything to do with planets and physics
- How to move inanimate objects using only your mind

Okay, we might have lied about the telekinetic powers. You'll have to read the book to find out for sure, but the rest is absolutely true.

Writing Your Own *N*-Body Integrator

N-body integrators have a rich and interesting history. As we will learn, there is often a right integrator for the job. To see this, let's dive right in. Our goal is to evolve planetary systems forward through time, with gravity being the only source of acceleration. Given a planetary system, all we have to do is write a computer program to calculate the gravitational acceleration imparted to every particle by every other particle in the system. If we start with some choice for the initial positions and velocities of the planets, we just need to hurl Newton's laws at the problem and perform this calculation. Using the result, we move the particles forward through space and time by just a little bit. Given their new positions and velocities, we repeat the process, iterating forward in time using some appropriately defined time step. For obvious reasons, we will choose this time step to be much smaller than the orbital periods of the planets.

We're done. Let's implement our program and see what happens. What could go wrong? It's not like we've been called upon to diffuse a bomb. Don't worry, that doesn't happen until chapter 7. You will be comforted to learn that the worst thing that can happen here is that you will confuse your computer. It will forgive you. Most likely, you will choose your time step to be too small, and the computer will take forever to move planets around. Or, you will choose a time step that is too big, and you will get the wrong answer.

To understand why these are such easy mistakes to make, even if you know what you are doing, consider a single moon orbiting a single planet that in turn orbits a single star. In this case, it makes the most sense to choose our time step to be much shorter than the time it takes for the moon to orbit its host planet once. But, if the moon orbits the planet thousands of times in the time it takes the planet to orbit the star once, then the program will run very slowly. The problem gets even worse if we throw in one additional planet orbiting very far from the star with an orbital period thousands of times that of the inner moon-hosting planet. Here, we require millions of time steps before the outer planet orbits the host star a single time.

In the first half of this book, we will develop a fully functional *N*-body integrator. We emphasize that all of the code developed in this book is freely available on GitHub.[1] Along the way, we will introduce a few modern integration schemes, including adaptive time stepping. We will develop a comprehensive understanding of not only the techniques but also their reasons for being useful to implement. We will delve into the inner workings of gravity integrators, how to write them, how to use them, and when to avoid them.

This highlights an important concept in scientific research: the "black box." Practically, this is every bit as ominous as it sounds. It refers to a research tool designed by somebody else and used directly in research without knowing exactly how it works. The researcher

[1] https://github.com/MovingPlanetsAround

understands what the tool needs to be given as input and what the tool will provide as output. But the rest is a mystery. The inner machinery that makes the tool perform is not well understood, and this can lead to incorrect or unphysical results that the user might not be equipped to catch. In this book, we will avoid "black boxes" entirely. As already discussed, the nitty-gritty details of using gravity integrators correctly demands a comprehensive understanding of their inner machinery. This is exactly what the reader will walk away with: a "white box" that can be applied with a concrete knowledge of exactly how to arrive at physically robust solutions.

In the second half of this book, we will shift our focus to using a state-of-the-art integration scheme to conduct real astrophysics research. We will explain how to select such an integrator from the vast number of options available to us, treating the "present" as a free parameter. In other words, we go to great lengths to ensure that the reader is familiar with the inner mechanics of N-body integration schemes, empowering her or him to work with whatever tools are available in the literature, independent of the precise time the reader discovers this book. This spares the reader from having to toil with the optimization of the code, since this job has often already been done for us. It will facilitate conducting actual research with minimal access to computational facilities. A laptop is all you will need to develop and test the code, as well as a small cluster to carry out production-quality simulations.

We will address a particularly compelling question: how might our Solar System be different if the Sun were but one member of a binary star system? In particular, we will simulate the long-term dynamical stability of extrasolar *circumbinary planets*, identified by the *Kepler* satellite. These are planets that orbit binary star systems (two stars orbiting their common center of mass) as if they were isolated single stars. We will consider a special case of the four-body problem, namely the fate of potential exomoons that could be orbiting these circumbinary planets.

By the end of the book, you will be fully equipped to run your N-body integrator for the purpose of solving some of the most dazzling astrophysical puzzles of our (or your) time, including the fates of exomoons that could be orbiting existing circumbinary planets. Our ultimate goal is to write a real research paper, for submission to a peer-reviewed astrophysical journal. In fact, the scientific results presented in the book have been published by Hamers et al. (2018). This is no small task, but we will get you there. We will walk you through it step-by-step.

Last but not least, we rely heavily on the Socratic method in this book to communicate to the reader its contents. That is, we will usually adopt a dialogue format to help the reader better understand the realities of delving into conducting research with computers. We sometimes resort to humor in an attempt to make the book more engaging. We hope you'll have as much fun reading this book as we did writing it.

1 A Bestiary of Planets

What pertinent astrophysics-related questions might you tackle at the end of this book? Let us begin with a brief overview of the relevant astrophysics background. The list of interesting problems is literally endless. Imagination is the only limit.

Picture an alien world. Given the unfathomable number of planets in the Universe, the odds are that something like your planet actually exists. Somewhere out there, your world is floating through space. It orbits a distant star, just waiting to be discovered.

In short, planets are common in our Universe. Cool, dim stars host planets. Binary pairs of stars, orbiting each other's center of mass, host planets. Even Proxima Centauri, the Sun's nearest neighbor, hosts planets. But where does it end? It might very well turn out that nearly every star hosts a planet. Only time, and endless hours spent staring through a telescope, will tell.

The evidence so far suggests that planets are to the astrophysical kingdom what beetles are to the animal kingdom. The number of species might as well be infinite. There exist super-Earths, mini-Neptunes, and even hot-Jupiters. These last planets orbit so close to their host star that they themselves have surface temperatures of thousands of degrees. They can even end up tidally locked to their parent star, such that the same side of the planet is always facing its host. Here, day and night are permanently imprinted on the planet's surface.

Planets have now been found to orbit within the habitable zone of their host star. This "Goldilocks" zone corresponds to the critical distance from the host star where the temperature is just right for water to exist in liquid form. If life exists elsewhere in the Universe, this is as good a place to look as any.

And the list goes on. Despite the vast emptiness of space, or perhaps because of it, Earth and its seven siblings have countless cousins.

1.1 Many Moons

Massive planets often host many moons. Jupiter, for example, is thought to have over seventy moons. As far as the orbital dynamics are concerned, these moons are close analogs

of planets; just replace the host star with a massive planet and away you go. Important and interesting differences exist, of course.

Moons tend to orbit much closer to the surface of their host planets than do planets orbiting their host stars. This alone serves as a catalyst for a number of interesting astrophysical phenomena. For instance, the differential force of gravity becomes so strong that moons can be stretched and squeezed as they traverse their orbits. This is called tidal heating, and the net result is a deposition of energy in the interior of the moon. In the case of Io, a particularly compelling moon of Jupiter, tidal forces catalyze active volcanoes on the surface. The energy deposited within cannot be contained, and it escapes in a violent display of volcanism.

1.2 Multiple Star Systems

Orbital dynamics can extend beyond the domain of planets. Multistar systems are also known to exist. In the Milky Way galaxy, there are as many binary star systems as there are isolated single stars. Many of these binaries, in turn, are orbited by a third star. Many of these triples are orbited by a fourth star, and so on. The hierarchy effectively continues into the star cluster regime. Like planets, gravitationally bound collections of stars are common in the Universe.

Multiple star systems are factories for exotic astrophysical phenomena. In a binary, if the gravitational force exerted by one binary companion at the other companion's surface is sufficiently large relative to the surface gravity of the star, accretion can proceed from one star onto the other. Streams of hot hydrogen-rich matter flow from the surface of one star and are transferred onto the other. If the accretor happens to be a compact object, such as a white dwarf or neutron star, high-energy photons are produced when the mass transfer stream strikes the surface of the dead star. These photons are typically observed at ultraviolet and X-ray wavelengths and can reveal the properties of the donor and accretor alike.

1.3 Dead Stars

Planets orbit dead stars. Pulsars are neutron stars with rotation periods as short as milliseconds. These dense balls of degenerate neutron-rich matter are the leftover cores of once massive stars that have exhausted their nuclear fuel. Their densities are absurd; the mass of several suns is crammed into a volume the size of Manhattan. They harbor extreme magnetic fields and induce strong tidal forces. And yet, planets have been found orbiting pulsars.

Several billion years from now, when the Sun finally exhausts its nuclear fuel, it will shed its outer envelope. The act will leave exposed a hot, dense core called a white dwarf. What

will happen to the Sun's planets when it dies? They might survive. Or they might not. For example, to survive the giant phases of stellar evolution (the so-called red and asymptotic giant phases), during which time the Sun will grow in radius to become hundreds of times larger than it is now, planets must be far enough away from the host star that they avoid engulfment by its expanding outer boundary. However, planetary orbits also expand in response to the mass loss of the Sun. They can also shrink due to strong (but uncertain) tidal effects from the Sun as it expands. Hence, the answer to the question of what planets will survive the evolution of the Sun depends on several factors, most of which are still difficult to answer and not yet clear. Nevertheless, the Earth and Mars may be engulfed, but its relatively wide orbit implies that Jupiter will likely survive.

Recent observations of the surfaces of white dwarfs have revealed the presence of large quantities of heavy elements. These are in essence pollutants, since they could not have been placed there via normal stellar evolution. Something rich in metals must have landed on their surface sometime after the death of their progenitor star. The popular candidate for pollution is the accretion of asteroids or perhaps even small planets. How exactly this occurs is still an area of active research.

1.4 Supermassive Black Holes

At the heart of most massive galaxies in the Universe lurks a central supermassive black hole (SMBH). These behemoths can weigh in at masses of over a billion suns. How they got there, nobody knows. One of the more interesting proposals involves runaway stellar collisions. Here, stars collect at the very center of the galaxy to create extremely high densities. These stars collide and merge with each other to form one supermassive star, which then collapses to form an SMBH.

Whatever their origins, SMBHs exist. SMBHs with masses below about $10^8 M_\odot$ (where $M_\odot$ is the mass of the Sun) often host central nuclear star clusters. These are tightly packed collections of stars, with densities exceeding a million times the stellar density in the immediate vicinity of the Sun. In other words, if we were to magically drop the Sun and its nearest neighbor, Proxima Centauri, into a nuclear star cluster, well over a hundred stars would fall between them. Such high stellar densities catalyze direct collisions between stars. Grazing collisions can deflect stars onto orbits that can take them close enough to the central SMBH that they are torn apart by its strong tidal forces. This "tidal disruption event" first compresses the star, causing it to rip apart into matter streams. About half of the stellar material eventually forms an accretion disk and accretes onto the SMBH, whereas the other half escapes the SMBH. The accretion onto the SMBH is thought to render the destruction process of the star ultimately visible to distant observers.

Close to the SMBH, where its gravity is strongest, stars orbit on roughly Keplerian orbits. Often, they form disk-like structures not unlike planets orbiting a central host star. And,

as with planets, these strange disks seem to be telling us something about how the stars got there in the first place. Curiously, it seems that stars are able to form in the immediate vicinity of an SMBH. Even more surprising, these star-forming disks share more than a few things in common with protoplanetary disks.

1.5 The Cast of Characters

In order to convey the key issues presented in this book, we will veer away from the narrative path and focus instead on the players. In our case, these are students at the undergraduate level, developing an N-body code for a class-related project. Our story begins in a café. Bob and Alice are enjoying a beverage and a conversation. Alice and Bob are honors students at the local university. Alice is a budding astrophysicist, and Bob is a computer science major.

Alice: Today in class, the professor told us about all these extrasolar planets currently being discovered. It is a zoo. There are tons of planetary systems out there, with a wide range of properties. They have found multiplanet systems orbiting really low-mass stars, about a tenth the mass of our Sun. This means that the host star is much colder than the Sun. So, if you care about the possibility of life existing on these planets, you want to look at planets that orbit much closer to the surface of the star than the Earth orbits relative to our Sun. That way, they might have liquid water on their surface, and water is good for life, at least on Earth.

Bob: So do any of those multiplanet systems actually have planets with liquid water?

Alice: Well, they are still arguing about that. But some people think that they have found a few planets that could have liquid water and are expecting to find many more.

Bob: Whoa. What else did the professor tell you about?

Alice: Well, apparently, there are even multiple star systems that host planets. So, think about what would happen if our Sun had another star orbiting it, far outside the orbits of all its planets.

Bob: What *would* happen?

Alice: I don't know, actually. And I don't think anybody does, really. There are lots of options to consider, and people are just starting to explore these interesting possibilities.

Bob: It would be so cool to answer some of those questions ourselves. Maybe one day….

Alice: Don't count us out just yet! Between the two of us, we already have a lot of the knowledge and skills we would need, and I am sure the professor would be more than happy to answer any questions that come up along the way, and help guide us. I'm not sure how far we will get, but we will never know if we don't at least try.

Bob: Agreed. We should use a computer to do it!

Alice: That's actually a really good idea. Apparently, there is no simple all-encompassing solution for all the interesting astrophysics that can arise in these systems. So, the only way to do it is using computers. I think it would be so exciting to get something like that working. In principle, the discoveries are new enough that we could even answer important questions about planets nobody has thought to ask yet!

Bob: Good Lord, you are ambitious.

Alice: What's wrong with a little ambition?

Bob: Nothing. I just don't think I share your confidence.

Alice: How about I provide the ambition, and you provide the skepticism?

Bob: Ha ha. All right, I'm sold. I have to build a versatile software environment for one of my class projects anyways. The application of the software is up to me and this would be a perfect problem to focus on.

Alice: So, we're doing it? We're going to write a computer program that can evolve planetary systems?

Bob: I'm in.

Alice: Awesome!

Bob pulls out his laptop, his fingers poised over the keyboard. Hesitating, he turns to Alice.

Bob: Okay, so the first thing we need to keep in mind is that we have to choose some method of communicating with the computer. In other words, we need to decide on a programming language. I think we should start with PYTHON, since it was designed to be user-friendly. In other words, it will be the easiest language to teach you, so that you can easily run and adapt the code yourself.

Alice: Great! Wait, are there other options?

Bob: Yes, there are other options. We could use a language like C++ or FORTRAN, but with PYTHON, things like plotting up our results so we can see the data we produce visually will be straightforward.

Alice: Okay, I'm sold. PYTHON it is.

Bob: Then what?

Alice: Well, based on what the professor has said about computer simulations, the first thing that comes to mind is that we will need to define for the computer some sort of three-dimensional grid. We will put our planets or particles or bodies or whatever you call them in this grid, and that will tell the computer where each particle is located, right? We can then give it a fourth dimension for time, and whenever this time parameter changes, the particles move to new locations in the three-dimensional grid. Does that make sense?

Bob: Sort of. It sounds like we are giving to the computer some kind of map and sticking pins in it at the locations of the particles. Then, we basically ask the computer to move the pins, one particle at a time.

Alice: Yeah, kind of. That sounds basically right.

Bob: It sounds super slow, without a computer … and maybe even using a computer, in some cases.

Alice: Why don't I describe to you the general physics we are trying to write a computer program to model, and you can do the actual coding?

Bob: It's a deal. So … what do I do now?

Alice: Well, we want to make a simulation for a planetary system. We will need a host star and some number of planets that are gravitationally bound to and happily orbiting their host star.

Bob: How many planets?

Alice: It doesn't really matter. However many we want. But, if we want to understand how our Solar System would be different if it had a binary star companion, for instance, then that more or less decides for us a target planetary architecture with well-defined initial conditions to study and simulate. Now all we have to do is figure out how to place our modified Solar System on the grid and then hand that over to the computer so it can then move the planets around for us.

Bob: Okay, do we start with all eight planets right away, in addition to the binary star companion? Or do we start small and work our way up?

Alice: That's a good question. I'm not really sure. To be honest, there are a lot of things I'm not so sure about. For example, I know gravity will provide a source of acceleration that will cause the planets to orbit their host star, but I'm a little unclear on exactly how that will work.

Bob: Let's keep it simple at first and start by writing a code for just two bodies. Once we have that working, if we are clever about the exact instructions we give to the computer, it shouldn't be too hard to add in additional bodies as we go along. Sound like a plan?

Alice: It sounds like a *good* plan!

2 Physics Background

Overview. Although Alice and Bob are eager to start coding, they realize that they should first brush up on their physics background, in particular Newtonian mechanics. They ask Prof. Starmover, with whom Alice has interacted considerably during her studies, for a crash course on this topic. Prof. Starmover is happy to provide the relevant background. Some of the more detailed derivations are given in appendices A and C.

Bob: I'm excited to start our first project. So, how does gravity make planets orbit the Sun?

Alice: Uh … Kepler's laws? I seem to recall Prof. Starmover talking a lot about those, and how they decide the general shapes and properties of orbital configurations.

Bob: I think we have gotten ahead of ourselves. Do you think Prof. Starmover would be willing to walk us through the basics? You know her pretty well, right? Clearly, we need to learn a few things before we can even start. What do you think?

Alice: I agree. I think we should get to know more about the physics of planetary systems. Prof. Starmover is an astrophysics professor specialized in planetary dynamics. I'm sure she will be happy to give us some physics background.

2.1 Newton's Laws

The next day, Bob and Alice meet with Prof. Starmover in her office.

Alice: Thank you for meeting with us, Prof. Starmover. Bob and I are very keen to develop a software environment to simulate the dynamics of extrasolar planetary systems. But before we delve right into the nitty-gritty details of coding our own N-body integrator, we would like to know a bit more about the physics background of planetary systems so that we have a better idea of what we are trying to simulate and what physics are important.

Bob: I only remember one thing about gravity from high school: what comes up, must come down. Admittedly, my teacher always used the same pen for the demonstration. I

always found that suspicious but have probably seen too many terrible magic tricks. Anyways, I assume that the rule applies to any object?

Professor: That's true. Gravity acts on all objects. Actually, the fundamental laws of motion that apply to a pen falling to the floor are exactly the same as those that apply to planetary systems. Except in some cases where relativistic effects are important, planetary systems are well described by Newton's laws of motion combined with the Newtonian law of gravity.

Alice: But then why does a pen fall in a straight line, but planets orbit in circles?

Professor: The big difference between the two cases lies with the initial conditions: when it stays close to the surface, a pen falling to the floor can be described to a very good approximation as a body falling in a uniform gravitational field with zero initial velocity. A planet, on the other hand, moves in a nonuniform gravitational field of a star and with nonzero initial velocity (otherwise, it would crash into the star!).

Let us first focus on the simplest case of a body falling freely in the uniform gravitational field of the Earth. Newton's second law of motion states that the acceleration of the body is equal to the gravitational acceleration that points downward to the Earth, that is, if z is the height of the object with respect to some arbitrary point (say, the floor), then the force acting on the object is $F = ma = -mg$, where $g \approx 9.8\,\mathrm{ms}^{-2}$ is the Earth's gravitational acceleration. In other words,

$$\frac{\mathrm{d}^2 z}{\mathrm{d}t^2} = -g. \tag{2.1}$$

Integrating this equation twice with respect to time reveals that the object falls according to

$$z(t) = z_0 + v_{z,0}t - \frac{1}{2}gt^2, \tag{2.2}$$

where z_0 is the initial position (say, the initial height from which the pen is released), and $v_{z,0}$ is the initial speed of the pen (for example, if someone throws a pen up in the air). The motion of the object is described by a simple analytic function, as given by equation (2.2), and does not depend on the object's mass, as famously demonstrated by Galileo's famous Leaning Tower of Pisa experiment, or the *Apollo 15* hammer and feather experiment in 1971 (on the moon, astronaut David Scott dropped a hammer and a feather from the same height, and they reached the ground at the same time).

Bob: You can't fool me. If I throw a plastic bag into the wind, its motion is completely random.

Alice: I think Bob is wondering about aerodynamic drag. Surely that is important for objects falling in Earth's atmosphere?

Professor: Indeed, I am neglecting aerodynamic drag in equation (2.1). Fortunately, we do not have to worry about drag for planets, since they (usually) move in empty space.

There is an exception for young planets that are still embedded in a protoplanetary disk and that interact with the disk and experience drag, somewhat similarly to aerodynamic drag. Protoplanetary disks are made up of gas and dust, and disperse within several million years after forming. This is short compared to the typical lifetime of stars like our Sun; therefore, it is usually safe to ignore drag.

Bob: Phew. That should save us some trouble.

Professor: Forgetting about drag, there is still an important difference between equation (2.1) and the motion of planets. When far away from the surface of the Earth, the Earth's gravitational field is no longer uniform. Instead, we have to use Newton's law of gravitation,

$$F_1 = -\frac{Gm_1m_2}{r_{12}^2}, \tag{2.3}$$

where F_1 is the force on body 1 (an object on the Earth's surface or beyond) due to body 2 (the Earth); m_1 and m_2 are the masses of bodies 1 and 2, respectively; and r_{12} is the distance between the bodies. The minus sign indicates that the force on body 1 is pointing toward body 2. In the case of a falling object on Earth, we can write $r_{12} = R_\oplus + z$, where $R_\oplus$ is the Earth's radius. If the object is close to the surface, then $z \ll R_\oplus$, and $r_{12} \approx R_\oplus$. Equation (2.3) then simply states that

$$F_1 = m_1\frac{\mathrm{d}^2z}{\mathrm{d}t^2} = -\frac{Gm_1m_\oplus}{R_\oplus^2}, \tag{2.4}$$

that is, $\mathrm{d}^2z/\mathrm{d}t^2 = -Gm_\oplus/R_\oplus^2$. If we plug in the numbers, we find that $Gm_\oplus/R_\oplus^2 \simeq 9.8\,\mathrm{ms}^{-2}$, that is, we recover equation (2.1).

Alice: Won't the other body also attract the Earth? In other words, is there also a force due to body 1 on body 2?

Professor: Correct, Newton's third law states that the force on body 2 due to body 1 is simply

$$F_2 = -F_1. \tag{2.5}$$

Therefore, the net force on the system of bodies 1 and 2 is

$$F_1 + F_2 = 0, \tag{2.6}$$

so there is no net force on the system as a whole.

Bob: So, just to get this straight, gravity doesn't care at all about the shape of an object? Not even a little bit? I find that a little hard to believe….

Alice: Again, to translate for Bob, I think he is wondering if you are assuming point masses in equation (2.3)? After all, in reality (and fortunately for us), the Earth is not a point mass.

Professor: That's true, but if we assume that the Earth is a perfect sphere (which is a good approximation for many purposes), then we can apply a famous theorem by Newton, his shell theorem (or, alternatively, Gauss's law), which states that the gravitational force on a body outside of the Earth is the same as the gravitational force on that body due to a point mass with the same mass of the Earth, located at the Earth's center.

Bob: And what about more than two bodies?

Professor: If there are more than two bodies, then we can apply the superposition principle.

Bob: What's that?

Professor: According to the superposition principle, to find the gravitational force due to many bodies on one body, we can vectorially add the forces from each individual pair of that one body with all other bodies. In other words, simply add the forces on one particular body due to all other bodies. In order to write down the equation that describes this, let me first talk for a moment about coordinate systems and vectors.

2.2 Coordinate Systems

Professor: So far, I have been using either the z coordinate or r, the distance between two bodies. Especially when there are more than two bodies, it is convenient to write the equations in terms of vectors. Let us set up a traditional coordinate system (x, y, z) that is inertial. This means that it is moving at a constant or zero velocity.

Alice: Can't we take any coordinate system we like? Why restrict to an inertial one?

Bob: Seriously? We definitely can't do anything we like. Even *I* know there are going to be rules. When are there ever *not* rules? Don't you know anything?

Prof. Starmover laughs.

Professor: If you two can manage to focus, you should work well together.

Bob: You hear that, Alice? We make a good team!

Alice shudders.

Professor: Back to the problem at hand. In truth, we can take any frame we like. For example, we could take a noninertial frame that is fixed to a rapidly rotating body like Jupiter. Jupiter rotates about its axis about once every ten hours. However, this will make the equations more complicated because any acceleration, be it linear acceleration or rotation, introduces additional terms in the equations of motion. Do note that there are certain

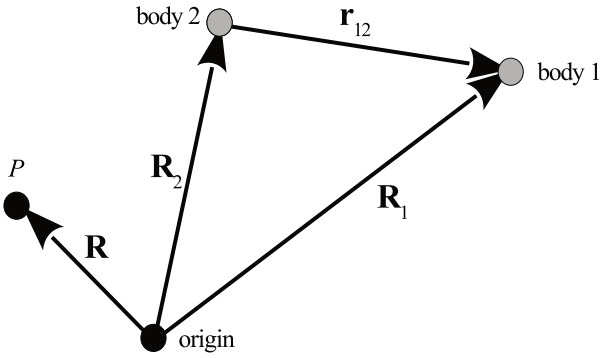


Figure 2.1
Several vectors, $\mathbf{R}$, $\mathbf{R}_1$, and $\mathbf{R}_2$, in a two-dimensional plane. $\mathbf{R}$ is a vector pointing from the origin to an arbitrary point P, whereas $\mathbf{R}_1$ and $\mathbf{R}_2$ point to two bodies labeled "1" and "2," respectively. The difference vector $\mathbf{r}_{12}$ points from the endpoint of $\mathbf{R}_2$ to $\mathbf{R}_1$ where bodies 1 and 2 are located, and its magnitude is the distance between the bodies.

situations in which this can actually be beneficial. For instance, if you are interested in the motion of particles in Jupiter's atmosphere, this transformation in the frame of reference makes the problem much simpler. For our purposes, however, it will be easier to stick to an inertial frame.

Prof. Starmover walks to the chalk board. She picks up a piece of chalk and begins scrawling equations on the board.

Professor: I will indicate the position of any point P in the (x,y,z)-frame with a vector $\mathbf{R}$, which has components along the x-, y-, and z-axes given simply by x, y, and z. The vector $\mathbf{R}$ points from the origin $(0,0,0)$ to the position of the point P; see figure 2.1. Note that I am using boldfaced font to denote vectors; you may be familiar with the arrow notation, $\vec{R}$, but the meaning is exactly the same. A nice aspect about these position vectors is that relative distances are easily expressed as the difference between two position vectors. In particular, the distance between two bodies with position vectors $\mathbf{R}_1$ and $\mathbf{R}_2$ is given by

$$r_{12} = \|\mathbf{R}_1 - \mathbf{R}_2\|, \tag{2.7}$$

where the double vertical lines indicate the norm, or length, of a vector.

Alice: How should I interpret the difference between two vectors?

Professor: Technically, $\mathbf{r}_{12} = \mathbf{R}_1 - \mathbf{R}_2$ is a new vector whose components are given by $(x_1 - x_2, y_1 - y_2, z_1 - z_2)$. But the vector $\mathbf{r}_{12}$ also has a geometric interpretation: take vector $\mathbf{R}_1$, then add the reversed vector $\mathbf{R}_2$: $\mathbf{r}_{12} = \mathbf{R}_1 + (-\mathbf{R}_2)$. I illustrate this in figure 2.1; for simplicity, just in two dimensions. Looking at this figure, we see that the vector $\mathbf{r}_{12}$ points

from $\mathbf{R}_1$ to $\mathbf{R}_2$ and its length, r_{12}, is the distance between the bodies at the endpoints of $\mathbf{R}_1$ and $\mathbf{R}_2$.

Alice: Okay, I think I got this. So in three-dimensional Euclidean space,

$$r_{12} = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2}. \tag{2.8}$$

Professor: That's right. Essentially, equation (2.8) is just an application of Pythagoras' theorem in three dimensions and equation (2.7) is a compact way of writing it.

Let me also introduce two important vector operations between two general vectors. Let $\mathbf{a}$ and $\mathbf{b}$ denote such vectors. The **dot product** (also known as **scalar product**), denoted as $\mathbf{a} \cdot \mathbf{b}$, is a scalar quantity (i.e., just a number, not a vector) given by

$$\mathbf{a} \cdot \mathbf{b} = ||\mathbf{a}|| \, ||\mathbf{b}|| \cos(\alpha), \tag{2.9}$$

where α is the angle between $\mathbf{a}$ and $\mathbf{b}$. A way of looking at it is that $\mathbf{a} \cdot \mathbf{b}$ is the projection of $\mathbf{b}$ on $\mathbf{a}$ (or, equivalently, the projection of $\mathbf{a}$ on $\mathbf{b}$). In Cartesian coordinates, the dot product can be written in the explicit form

$$\mathbf{a} \cdot \mathbf{b} = a_x b_x + a_y b_y + a_z b_z, \tag{2.10}$$

where a_x is the x-component of $\mathbf{a}$ and so on.

Alice: So, if $\mathbf{a} = \mathbf{b}$, then $\mathbf{a} \cdot \mathbf{a} = a_x a_x + a_y a_y + a_z a_z = ||\mathbf{a}||^2$!

Professor: Correct! In words, the projection of a vector along itself gives its own (squared) length. Also, the dot product is maximized when the two vectors are aligned ($\alpha = 0$) and minimized when they are perpendicular ($\alpha = \pi/2$); in the latter case, the dot product is zero.

The second vector operation that I would like to discuss is the **cross product** (also known as the **vector product**), which is in some ways the counterpart of the dot product. The vector product, denoted as $\mathbf{a} \times \mathbf{b}$, is itself a vector quantity (not a scalar quantity like $\mathbf{a} \cdot \mathbf{b}$), with a *magnitude* given by

$$||\mathbf{a} \times \mathbf{b}|| = ||\mathbf{a}|| \, ||\mathbf{b}|| \sin(\alpha). \tag{2.11}$$

Here, α is again the angle between $\mathbf{a}$ and $\mathbf{b}$. The *direction* of $\mathbf{a} \times \mathbf{b}$ is given by the so-called right-hand rule: take your right hand, point your fingers along $\mathbf{a}$, then close your fist by turning your fingers toward $\mathbf{b}$. Your thumb then points into the direction of $\mathbf{a} \times \mathbf{b}$. This means that the cross product $\mathbf{a} \times \mathbf{b}$ is always perpendicular to both $\mathbf{a}$ and $\mathbf{b}$.

Bob fiddles around with his left and right hands, looking a bit confused.

Professor: For example, consider the vectors $\hat{\mathbf{x}}$, $\hat{\mathbf{y}}$, and $\hat{\mathbf{z}}$, which are unit vectors pointing along the x-, y-, and z-directions, respectively. The hats here denote **unit vectors**, that is, a unit vector $\hat{\mathbf{a}}$ is a vector with unit length. By definition, $\hat{\mathbf{a}} = \mathbf{a}/||\mathbf{a}||$, so if we compute the length of $\hat{\mathbf{a}}$, we simply get $||\hat{\mathbf{a}}|| = ||\mathbf{a}||/||\mathbf{a}|| = 1$ (note that $||\mathbf{a}||$ is a scalar quantity, so taking

the length of it does not do anything). Another way of looking at it is that any vector $\mathbf{a}$ can be constructed from its direction (i.e., unit vector $\hat{\mathbf{a}}$) and length (i.e., norm $\|\mathbf{a}\|$), so we have $\mathbf{a} = \hat{\mathbf{a}}\|\mathbf{a}\|$. If we then take, for example, the cross product between $\hat{\mathbf{x}}$ and $\hat{\mathbf{y}}$, we get $\hat{\mathbf{x}} \times \hat{\mathbf{y}} = \hat{\mathbf{z}}$. By the way, this is assuming you have a right-handed coordinate system. If you chose a left-handed coordinate system, we would have $\hat{\mathbf{x}} \times \hat{\mathbf{y}} = -\hat{\mathbf{z}}$. Another example (for a right-handed coordinate system) is $\hat{\mathbf{z}} \times \hat{\mathbf{y}} = -\hat{\mathbf{x}}$.

Bob: Okay, now things start making sense.

Professor: As you can tell from its definition, the cross product is zero when the angle between the two vectors is zero. So, if $\mathbf{a} = \mathbf{b}$, then $\mathbf{a} \times \mathbf{b} = \mathbf{0}$. Also, the magnitude of the cross product is maximized when $\mathbf{a}$ and $\mathbf{b}$ are perpendicular. Note that these properties are opposite to those of the *dot* product and are expressed mathematically by the cosine factor in the definition of the dot product (equation 2.9) versus the sine factor in the definition of the cross product (equation 2.11).

Bob: This is all fine and dandy, but what use do the dot and cross products have?

Professor: They are ubiquitous in mathematics and physics, and there are countless examples in which they are very useful. For example, dot products are very useful in geometry to elegantly describe the equation of a plane, and in physics, the cross product is conveniently used in the equation of the Lorentz force. In fact, dot and cross products will appear all over the place in just a moment!

Now we know more about vectors, let us use them to rewrite Newton's law of gravitation, equation (2.3):

$$\mathbf{F}_1 = -Gm_1m_2\frac{\mathbf{R}_1 - \mathbf{R}_2}{\|\mathbf{R}_1 - \mathbf{R}_2\|^3}. \tag{2.12}$$

Bob: What happened there? Suddenly Newton's law has become a $1/r^3$-force law?

Professor: Looks can be deceiving. I may be dividing by the norm of the separation vector cubed, but I am also multiplying by the separation vector again in the numerator. If we compute the norm of the force, we get

$$\|\mathbf{F}_1\| = Gm_1m_2\frac{\|\mathbf{R}_1 - \mathbf{R}_2\|}{\|\mathbf{R}_1 - \mathbf{R}_2\|^3} = Gm_1m_2\frac{1}{\|\mathbf{R}_1 - \mathbf{R}_2\|^2}, \tag{2.13}$$

so equation (2.12) actually describes a $1/r^2$-force law. The reason for writing equation (2.12) in this form is that it conveniently captures the direction that the force is pointing in: the difference vector $\mathbf{R}_1 - \mathbf{R}_2$ points from body 2 to body 1. To get the actual force on body 1, we need a minus sign, because the force on body 1 is directed in the opposite direction, that is, from body 1 to body 2.

Now let us extend Newton's law of gravitation to an arbitrary number of bodies, denoted by N. For this we use the superposition principle that I alluded to before: the force on a

body i is given by summing the forces from all pairs of bodies, j. In equation language,

$$\mathbf{F}_i = -Gm_i \sum_{\substack{j=1 \\ j\neq i}}^{N} m_j \frac{\mathbf{R}_i - \mathbf{R}_j}{||\mathbf{R}_i - \mathbf{R}_j||^3}. \tag{2.14}$$

Newton's second law in vector form for $\mathbf{R}_i$ reads

$$\mathbf{F}_i = m_i \mathbf{a}_i = m_i \frac{\mathrm{d}^2 \mathbf{R}_i}{\mathrm{d}t^2}, \tag{2.15}$$

where $\mathbf{a}_i$ is the acceleration on body i, so this means that

$$\frac{\mathrm{d}^2 \mathbf{R}_i}{\mathrm{d}t^2} = -G \sum_{\substack{j=1 \\ j\neq i}}^{N} m_j \frac{\mathbf{R}_i - \mathbf{R}_j}{||\mathbf{R}_i - \mathbf{R}_j||^3}. \tag{2.16}$$

This equation forms the basis for any N-body code and, more generally, for most of dynamical astrophysics.

Bob: Wow, that doesn't look too complicated! Can't we solve that using math alone? Or do we need computers to do it for us?

Professor: Well, technically, it can be solved analytically. But it turns out the solution is an infinite series of time, $t^{1/3}$, that converges extremely slowly. For example, to get results useful for astronomical observations in the Solar System, it has been estimated that one would need a total of $10^{8\times10^6}$ terms (much beyond the ability to handle even for modern computers), that is, the solution is not very practical and not much is learned from it. We know how to find a "nice" general analytical solution only for the $N = 2$ case, and for $N = 3$, some relatively simple solutions exist for some special or approximated cases, usually denoted as "restricted" problems. This even works for some cases with $N > 3$. For example, the short-term evolution of the current Solar System is well described by Laplace–Lagrange theory, which uses a number of approximations that allow for "simple" analytic solutions. This theory, although centuries old, is still important for planetary dynamics today, so I will talk about it a bit more later on. But let us first delve into the two-body problem.

2.3 The Two-Body Problem

2.3.1 Reducing the Problem to a One-Body Problem

Bob: But Prof. Starmover, hold on. If the N-body problem is so complicated and useful analytic solutions only exist for specific scenarios and low N, why should we even bother with these special cases and not immediately try to solve equation (2.16) numerically on the computer?

Professor: Of course, in the end, you will be writing and using your own code that numerically solves equation (2.16). But still, a lot can be learned from the $N = 2$ case, especially in the context of planetary systems. To a good approximation, the short-term motion of the Earth around the Sun is a two-body problem, and the other planets and moons don't matter. This is because the Sun is much more massive than the other planets, and therefore the other planets only weakly perturb each other; the second most massive body in the Solar System is Jupiter, which is about 1,000 times less massive than the Sun. On much longer time scales, these perturbations start to add up, and the orbits of the planets do evolve in more complicated ways than the two-body problem alone can account for. But on the time scale of a given orbit, which is by definition a year for the Earth in the Solar System, we can usually ignore the other planets. This reduces it to an $N = 2$ problem.

One of Newton's great accomplishments was to solve equation (2.16) for $N = 2$. Newton used a geometric approach and found that the solutions for bound orbits are ellipses. His methods are hard to follow for a modern audience because nowadays, we are so used to the language of calculus (introduced by Leibniz). As such, I will stick to the calculus approach, since we might as well take advantage of Newton's pioneering efforts in mathematics.

The crux in solving the two-body problem is to reduce it to a one-body problem, in which a single body is moving around a fixed body. Let us first write down the explicit expressions for the two bodies from equation (2.12).

$$\ddot{\mathbf{R}}_1 = -Gm_2 \frac{\mathbf{R}_1 - \mathbf{R}_2}{\|\mathbf{R}_1 - \mathbf{R}_2\|^3}; \tag{2.17a}$$

$$\ddot{\mathbf{R}}_2 = -Gm_1 \frac{\mathbf{R}_2 - \mathbf{R}_1}{\|\mathbf{R}_2 - \mathbf{R}_1\|^3}. \tag{2.17b}$$

For notational convenience, I am using overhead dots to denote derivatives with respect to time (so two dots indicate the second derivative with respect to time). Let's introduce the relative position vector

$$\mathbf{r} \equiv \mathbf{R}_1 - \mathbf{R}_2. \tag{2.18}$$

Of course, we could also have defined $\mathbf{r}$ with a minus sign (i.e., $\mathbf{r} = \mathbf{R}_2 - \mathbf{R}_1$), which would be an equally valid choice and would not change the physics. If we subtract equation (2.17b) from equation (2.17a), we get

$$\ddot{\mathbf{r}} = \ddot{\mathbf{R}}_1 - \ddot{\mathbf{R}}_2 = -G(m_1 + m_2)\frac{\mathbf{r}}{r^3}, \tag{2.19}$$

where $r \equiv \|\mathbf{r}\|$. This is only a single equation for the relative position vector.

Bob: But how can two vector equations suddenly have been reduced to a single vector equation without losing information? In other words, how do we get the two vectors, $\mathbf{R}_1$ and $\mathbf{R}_2$, from $\mathbf{r}$ only?

Professor: To answer this, I also need to introduce the center of mass position vector. It is defined as

$$\mathbf{r}_{\mathrm{CM}} \equiv \frac{m_1\mathbf{R}_1 + m_2\mathbf{R}_2}{m_1 + m_2}. \tag{2.20}$$

From equations (2.17) and (2.20), we easily see that

$$\ddot{\mathbf{r}}_{\mathrm{CM}} = \frac{1}{m_1 + m_2}\left[-Gm_1m_2\frac{\mathbf{r}}{r^3} - Gm_2m_1\frac{-\mathbf{r}}{r^3}\right] = \mathbf{0}, \tag{2.21}$$

that is, the center of mass $\mathbf{r}_{\mathrm{CM}}$ does not accelerate, and therefore it moves with constant or zero velocity. This defines an inertial reference frame. Therefore, $\mathbf{r}_{\mathrm{CM}}$ as a function of time is simply given by

$$\mathbf{r}_{\mathrm{CM}}(t) = \mathbf{r}_{\mathrm{CM},0} + \dot{\mathbf{r}}_{\mathrm{CM},0}\,t, \tag{2.22}$$

where $\mathbf{r}_{\mathrm{CM},0}$ and $\dot{\mathbf{r}}_{\mathrm{CM},0}$ are the initial position and velocity of the center of mass, respectively. With a little bit of algebra, equations (2.18) and (2.20) can be reworked to express $\mathbf{R}_1$ and $\mathbf{R}_2$ in terms of $\mathbf{r}$ (which has a nontrivial equation of motion, equation 2.19) and $\mathbf{r}_{\mathrm{CM}}$ (which has a trivial equation of motion, equation 2.21). The result, which I leave up to you to verify, is

$$\mathbf{R}_1 = \mathbf{r}_{\mathrm{CM}} + \frac{m_2}{m_1 + m_2}\mathbf{r}; \tag{2.23a}$$

$$\mathbf{R}_2 = \mathbf{r}_{\mathrm{CM}} - \frac{m_1}{m_1 + m_2}\mathbf{r}. \tag{2.23b}$$

So we have reduced the two-body problem to a one-body problem. The fictitious body has a mass of $m_1 + m_2$; let us denote the latter mass with $M \equiv m_1 + m_2$ for convenience.

Alice: What about conserved quantities like energy and angular momentum?

Professor: Those quantities are indeed conserved, and they turn out to be very useful for finding the solution to the equation of motion. First, let us write down the energy of the two-body system, which consists of the sum of the kinetic energy and the potential energy, that is,

$$E = \frac{1}{2}m_1\dot{\mathbf{R}}_1^2 + \frac{1}{2}m_2\dot{\mathbf{R}}_2^2 - \frac{Gm_1m_2}{\|\mathbf{R}_1 - \mathbf{R}_2\|}. \tag{2.24}$$

Note that I am using the shorthand notation $\mathbf{v}^2 \equiv \mathbf{v}\cdot\mathbf{v}$ for any vector $\mathbf{v}$; otherwise, the "square" of a vector is ill-defined. In one dimension, this equation reduces to the perhaps more familiar equation

$$E = \frac{1}{2}m_1V_1^2 + \frac{1}{2}m_2V_2^2 - \frac{Gm_1m_2}{|R_1 - R_2|}, \tag{2.25}$$

where $V_i = \dot{R}_i$ is the speed of the ith body. Substituting equations (2.23), we can show with a bit of algebra that equation (2.24) can be written in terms of $\mathbf{r}$ as

$$E = \frac{1}{2}M\dot{\mathbf{r}}_{\rm CM}^2 + \frac{1}{2}\mu\dot{\mathbf{r}}^2 - \mu\frac{GM}{r}, \tag{2.26}$$

where I introduced the "reduced" mass

$$\mu \equiv \frac{m_1 m_2}{M}. \tag{2.27}$$

In an inertial frame of reference, the center of mass $\mathbf{r}_{\rm CM}$ moves with a constant velocity; therefore, $\dot{\mathbf{r}}_{\rm CM}$ is just a constant vector. Since the energy is conserved, we can just as well subtract the term $(1/2)M\dot{\mathbf{r}}_{\rm CM}^2$ from the energy in equation (2.26), leaving again a constant expression. I prefer to divide the energy by the reduced mass μ. We then arrive at the following conserved quantity,

$$E_{\rm sp} \equiv \frac{1}{\mu}\left(E - \frac{1}{2}M\dot{\mathbf{r}}_{\rm CM}^2\right) = \frac{1}{2}\dot{\mathbf{r}}^2 - \frac{GM}{r}. \tag{2.28}$$

Equation (2.28) is known as the energy per unit mass or, more colloquially, simply the specific energy.

Alice: I have to say you are using a *lot* of math, Prof. Starmover.

Bob: You can say that again.

Alice: I would need to check these steps carefully on pen and paper to completely understand it. But am I correct in saying—in less mathematical terms—that the energy of the two-body system is given by the kinetic energy associated with the center of mass, plus the energy associated with the effective one-body orbiting around its center?

Professor: Correct. And while it is important that you be able to follow the detailed derivations, I am omitting them for brevity. A similar result applies to the angular momentum. Let us write it down:

$$\mathbf{L} = m_1\mathbf{R}_1 \times \dot{\mathbf{R}}_1 + m_2\mathbf{R}_2 \times \dot{\mathbf{R}}_2. \tag{2.29}$$

Again, using equations (2.23), we can write this as

$$\mathbf{L} = M\mathbf{r}_{\rm CM} \times \dot{\mathbf{r}}_{\rm CM} + \mu\mathbf{r} \times \dot{\mathbf{r}} \equiv \mathbf{L}_{\rm CM} + \mu\mathbf{k}, \tag{2.30}$$

where I introduced another shorthand notation, $\mathbf{k} \equiv \mathbf{r} \times \dot{\mathbf{r}}$.

Alice: Aha! So the angular momentum can be written as the sum of the angular momentum associated with the center of mass and that associated with the effective one-body.

Professor: That's right.

Bob: I have no idea what just happened.

Alice: Ha ha. Yeah, I'll need to sit down with it later and go through it in more detail if I want to fully understand it. Overall, though, I think it's really clever!

Bob: Newton was a smart man. That's the main thing I've learned so far. But I too will spend more time with it later, in order to better understand it. I can't wait to do some programming. The sooner I understand this stuff, the sooner I can program it.

2.3.2 Solving for the Relative Trajectory

Alice: Now that we have this background knowledge, how can we actually solve for the trajectories of the two bodies or, equivalently, the trajectory traced by $\mathbf{r}$?

Professor: There are multiple ways to solve for the trajectory of $\mathbf{r}$. One "brute-force" approach would be to solve equation (2.19). It is a second-order differential equation, and in principle, you could simply guess the solution or apply more sophisticated mathematical techniques. But let us use a simpler approach based on conservation laws and vector manipulation.

Bob: I'm in.

Professor: First, let me leave it as an exercise for you to check that the energy and angular momentum, equations (2.28) and (2.30), are indeed conserved, by computing the time derivatives and substituting the equation of motion (2.19). You should find that $\dot{E}_{\mathrm{sp}} = 0$ and $\dot{\mathbf{k}} = \mathbf{0}$. The latter already tells about an important aspect of the motion: since $\mathbf{k} = \mathbf{r} \times \dot{\mathbf{r}}$ is conserved, this means that $\mathbf{r}$ and $\dot{\mathbf{r}}$ must lie within the same plane at all times. Next, with a bit more algebra, we can compute the time derivative of the vector $\mathbf{k} \times \dot{\mathbf{r}}$.

Bob: Before you show us the result, Prof. Starmover, how do you know that it is useful to consider this particular vector? I'm not convinced.

Professor: In hindsight, $\mathbf{k} \times \dot{\mathbf{r}}$ is related to another important vector that is conserved in the two-body problem. But to answer your question: it took scientists centuries to discover this, so don't be discouraged if you did not think of it immediately!

To continue, I will leave it as an exercise to you to show that

$$\frac{\mathrm{d}}{\mathrm{d}t}(\mathbf{k} \times \dot{\mathbf{r}}) = \frac{GM}{r^3}\left[\mathbf{r}(\mathbf{r} \cdot \dot{\mathbf{r}}) - \dot{\mathbf{r}}\, r^2\right] = \frac{\mathrm{d}}{\mathrm{d}t}\left(-\frac{GM\mathbf{r}}{r}\right). \tag{2.31}$$

After the second equality, I wrote the expression in terms of the time derivative of yet another vector, $-GM\mathbf{r}/r$. Reordering the terms, this implies that

$$\frac{\mathrm{d}}{\mathrm{d}t}\left(\mathbf{k} \times \dot{\mathbf{r}} + \frac{GM\mathbf{r}}{r}\right) = \mathbf{0}, \tag{2.32}$$

so the vector in the round brackets is a constant vector, say, $\mathbf{C}$. Let me write this constant vector as $\mathbf{C} = -GM\mathbf{e}$, where $\mathbf{e}$ is another constant vector, which is known as the eccentricity or Laplace–Runge–Lenz (LRL) vector.

Alice: Okay, but why introduce the constant $-GM$ in front of $\mathbf{e}$?

Professor: It turns out that the eccentricity vector **e** has a nice geometrical interpretation if we define it this way. For definiteness, let us write down **e** explicitly:

$$\mathbf{e} \equiv \frac{1}{GM}\dot{\mathbf{r}}\times\mathbf{k} - \frac{\mathbf{r}}{r} = \frac{1}{GM}\left[\mathbf{r}\,\dot{\mathbf{r}}^2 - \dot{\mathbf{r}}\,(\mathbf{r}\cdot\dot{\mathbf{r}})\right] - \hat{\mathbf{r}}. \tag{2.33}$$

After the second equality, I used the "BAC-CAB" rule for double vector products, $\mathbf{A}\times(\mathbf{B}\times\mathbf{C}) = \mathbf{B}(\mathbf{A}\cdot\mathbf{C}) - \mathbf{C}(\mathbf{A}\cdot\mathbf{B})$. Also, I am again using the notation $\hat{\mathbf{v}} \equiv \mathbf{v}/v$ to denote unit vectors, that is, vectors with unit length.

We are almost there. Let me pull another rabbit out of my hat and compute

$$\mathbf{r}\cdot\mathbf{e} = \frac{1}{GM}\left[r^2\dot{\mathbf{r}}^2 - (\mathbf{r}\cdot\dot{\mathbf{r}})^2\right] - r = \frac{k^2}{GM} - r, \tag{2.34}$$

where I used that (the constant) k^2 can be written as $k^2 = (\mathbf{r}\times\dot{\mathbf{r}})^2 = r^2\dot{\mathbf{r}}^2 - (\mathbf{r}\cdot\dot{\mathbf{r}})^2$. Remember that **e** is a constant vector. So, as **r** is changing in time, its orientation with respect to **e** will change as well, and we can use **e** as a reference vector to describe the trajectory. Mathematically, we know from the definition of the scalar product that

$$\mathbf{r}\cdot\mathbf{e} = re\cos(\theta), \tag{2.35}$$

where θ is the angle between **r** and **e**; θ is known in the context of celestial mechanics as the "true anomaly." The trajectory is then described by a relation between r and θ, which we can find by equating (2.34) and (2.35), giving

$$r = \frac{k^2}{GM}\frac{1}{1+e\cos(\theta)}. \tag{2.36}$$

Alice: How do we get the vector **r**? That is, the direction of the position vector?

Professor: As we saw before, the motion is confined to a single plane since the angular momentum vector **k** is constant. In equation (2.35), we have already implicitly defined **e** as a basis vector spanning **r**. All we need to do is to define another basis vector—let us call it $\hat{\mathbf{q}}$—that is perpendicular to both **e** and **k**. This is easily achieved by taking the vector cross product of $\hat{\mathbf{k}}$ and **e**, that is,

$$\hat{\mathbf{q}} \equiv \hat{\mathbf{k}}\times\hat{\mathbf{e}}. \tag{2.37}$$

Using the basis vectors $\hat{\mathbf{e}}$ and $\hat{\mathbf{q}}$, which are perpendicular unit vectors that lie in the orbital plane, we can write **r** as

$$\mathbf{r} = r\left[\cos(\theta)\hat{\mathbf{e}} + \sin(\theta)\hat{\mathbf{q}}\right] = \frac{k^2}{GM}\frac{1}{1+e\cos(\theta)}\left[\cos(\theta)\hat{\mathbf{e}} + \sin(\theta)\hat{\mathbf{q}}\right]. \tag{2.38}$$

Alice: Where are the sine terms coming from?

Bob: Where is *any* of this coming from? I have *so* much homework to do tonight….

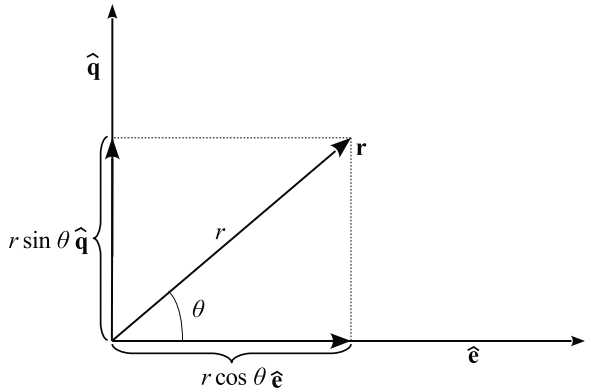


Figure 2.2
The relative position vector $\mathbf{r}$ in the orbital plane spanned by the unit vectors $\hat{\mathbf{e}}$ and $\hat{\mathbf{q}}$. The vector $\mathbf{r}$ can be constructed from its projections along $\hat{\mathbf{e}}$ and $\hat{\mathbf{q}}$.

Professor: Let me draw a picture of the orbital plane in figure 2.2. The horizontal and vertical axes indicate the directions $\hat{\mathbf{e}}$ and $\hat{\mathbf{q}}$, respectively. Also, I have drawn the vector $\mathbf{r}$ in the orbital plane. Since the angle between $\hat{\mathbf{e}}$ and $\mathbf{r}$ is θ, the projected length of $\mathbf{r}$ along $\hat{\mathbf{e}}$ is given by $r\cos(\theta)$. Similarly, the projected length of $\mathbf{r}$ along $\hat{\mathbf{q}}$ is given by $r\sin(\theta)$. If we multiply these projected lengths by $\hat{\mathbf{e}}$ and $\hat{\mathbf{q}}$, respectively, we get two new vectors, $r\cos(\theta)\,\hat{\mathbf{e}}$ and $r\sin(\theta)\,\hat{\mathbf{q}}$, and if we add them vectorially, we get $\mathbf{r}$ back, that is, $\mathbf{r} = r\cos(\theta)\,\hat{\mathbf{e}} + r\sin(\theta)\,\hat{\mathbf{q}}$, which is the same as equation (2.38).

Now that we know the equation for the relative orbit, let us draw some trajectories. We can draw parametric plots of equation (2.38) by setting $x = r(\theta)\cos(\theta)$ and $y = r(\theta)\sin(\theta)$ and plotting x and y as a function of θ in a plane. I have done this in figure 2.3 for $e = 0, e = 0.6$, $e = 1$, and $e = 5$. You will note that the shape of the trajectory depends on e. For $0 \le e < 1$, we get closed ellipses, where $e = 0$ corresponds to a constant r, that is, a circle. These trajectories are also known as Keplerian orbits. For $e = 1$, the trajectory is a parabola, and for $e > 1$, it is a hyperbola. As e increases above 1, the orbit increasingly resembles a straight line.

Alice: What is the scale in your plots? In other words, how is the constant $k^2/(GM)$ related to the physical scale of the orbit?

Professor: In the plots, I simply set $k^2/(GM) = 1$. To relate this to a physical scale, let me focus on the elliptic case ($0 \le e < 1$), which is most relevant for planetary systems. The scale of the orbit is usually expressed in terms of the semimajor axis, a. The semimajor axis is defined such that the long axis of the ellipse is equal to $2a$. For a circular orbit, the diameter of the orbit is twice the radius, and so the semimajor axis is equal to the orbital distance in this case. In the noncircular case, the orbital distance varies as a function of time; here, the semimajor axis is related to the average orbital distance. From equation (2.36), we can

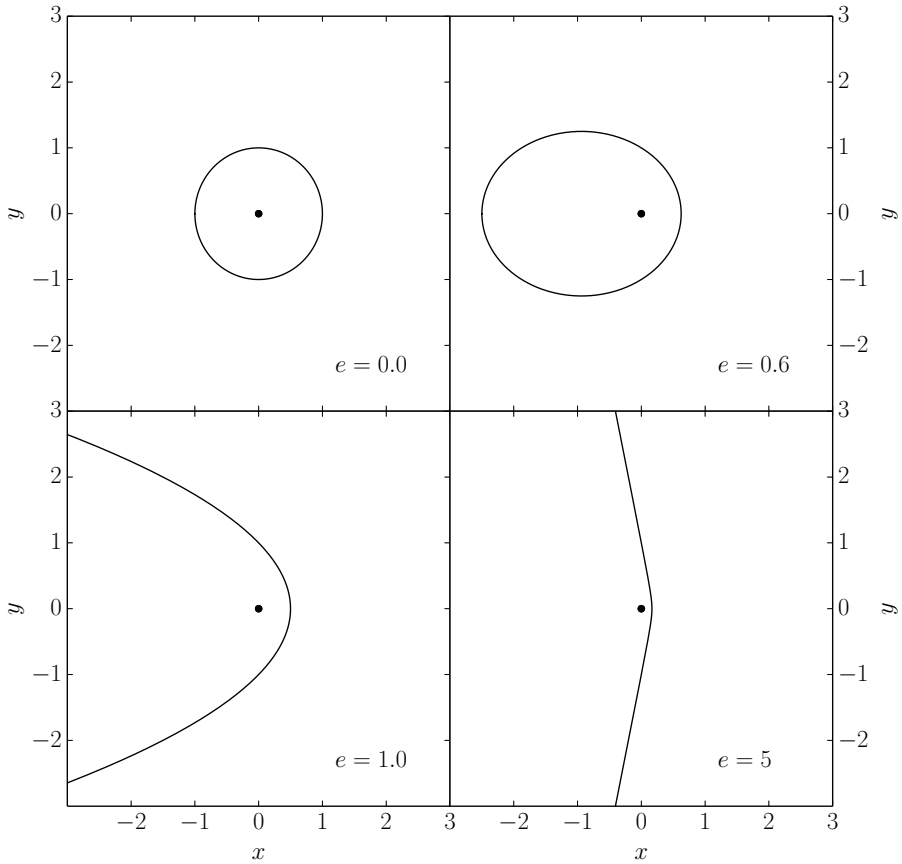


Figure 2.3
Relative orbits in the two-body problem for different values of the eccentricity e.

work out how a is related to $k^2/(GM)$. First, let me remark that the latter is known as the semilatus rectum, $p \equiv k^2/(GM)$. The shortest and longest distances from the origin to our effective one-body are given by setting $\theta = 0$ and $\theta = \pi$ in equation (2.36), respectively. So we get

$$r_{\rm p} = \frac{p}{1+e}; \qquad r_{\rm a} = \frac{p}{1-e}, \tag{2.39}$$

where $r_{\rm p}$, the shortest distance, is known as the periapsis distance, and $r_{\rm a}$, the longest distance, is known as the apoapsis distance. Summing $r_{\rm p}$ and $r_{\rm a}$ gives us

$$r_{\rm p} + r_{\rm a} = \frac{2p}{1-e^2} = 2a, \tag{2.40}$$

where in the last step I used the definition of the semimajor axis. So the semilatus rectum is related to a and e according to $p = a(1-e^2)$, and we can write the equation for r in the canonical form

$$r = \frac{a\left(1-e^2\right)}{1+e\cos(\theta)}. \tag{2.41}$$

As a bonus, we get an expression for the magnitude of $\mathbf{k}$, that is,

$$k = \sqrt{GMp} = \sqrt{GMa(1-e^2)}. \tag{2.42}$$

Bob: I think I'm going to throw up.

Alice: I'm still following you, Prof. Starmover … sort of.

Professor: Don't lose heart. It took the greatest physicists the world has ever known to work out these solutions in detail, and they did not do it overnight.

2.3.3 Getting the Solution as a Function of Time

Alice: Okay, so we now know the trajectory of the effective one-body as a function of the angle θ, but how do we relate this angle to the actual time?

Professor: This is more complicated than you might initially think.

Bob: At this point, you might be surprised, Prof. Starmover.

Professor: Ha ha. It turns out that there is an equation called the Kepler equation that relates an angle related to θ, namely the eccentric anomaly $\mathcal{E}$ (not to be confused with the energy), to the time t. This equation reads

$$\mathcal{E} - e\sin(\mathcal{E}) = n(t-\tau). \tag{2.43}$$

Here, n is the mean motion, given by

$$n \equiv \frac{2\pi}{P} = \sqrt{\frac{GM}{a^3}}, \tag{2.44}$$

where P is the orbital period, which is given by

$$P = 2\pi \sqrt{\frac{a^3}{GM}}. \tag{2.45}$$

Furthermore, τ is the time of periapsis passage, that is, a reference time at which $r = r_p$. Equation (2.45) is known as Kepler's third law: the square of the orbital period scales with the cube of the semimajor axis. The eccentric anomaly $\mathcal{E}$ is related to the true anomaly according to

$$\cos(\theta) = \frac{\cos(\mathcal{E}) - e}{1 - e\cos(\mathcal{E})}; \qquad \sin(\theta) = \sqrt{1-e^2}\,\frac{\sin(\mathcal{E})}{1 - e\cos(\mathcal{E})}; \tag{2.46}$$

$$\cos(\mathcal{E}) = \frac{\cos(\theta) + e}{1 + e\cos(\theta)}; \qquad \sin(\mathcal{E}) = \sqrt{1-e^2}\,\frac{\sin(\theta)}{1 + e\cos(\theta)}. \tag{2.47}$$

So the procedure to compute θ from t is to first compute the right-hand side of equation (2.43) and then solve for $\mathcal{E}$. Note that equation (2.43) cannot be solved analytically for $\mathcal{E}$; therefore, this has to be done numerically. We can then compute θ from equation (2.46), and once we know θ (and e and a), we know the relative orbit r. If we also want the *vector* $\mathbf{r}$, we need to specify the orbital orientation through $\hat{\mathbf{e}}$ and $\hat{\mathbf{q}}$. Note that the orbital orientation can also be specified using orbital elements rather than the orbital vectors $\hat{\mathbf{e}}$ and $\hat{\mathbf{q}}$. I will talk about orbital elements later.

I will not derive these equations here. If you are interested in the derivations, we can come back to this some other time (see appendix A). In fact, I would suggest that you also go to the library and check some classical books like those by Danby (1992), Murray and Dermott (1999), Battin (1999), and Morbidelli (2002).

2.4 The General *N*-Body Problem

Professor: So far, we have looked at the simplest case, $N = 2$. As I mentioned earlier, the $N > 2$ problem is much harder, and generally one needs to solve the equations of motion numerically, that is, to carry out an N-body calculation. Nevertheless, there are some general properties that apply to all N-body systems. Most important, conserved quantities like energy and angular momentum can often tell us something useful about the system. At the very least, conserved quantities can tell us something about the accuracy of numerical integrations.

Generally, there are ten conserved quantities in the N-body problem: two vectors describing the position and velocity of the center of mass (six constants), the total angular momentum vector $\mathbf{L}$ (three constants), and the total energy E. The energy is given by the

sum of the kinetic and potential energies, that is,

$$E = T + V = \frac{1}{2}\sum_{i=1}^{N} m_i \left(\dot{\mathbf{R}}_i \cdot \dot{\mathbf{R}}_i\right) - \frac{1}{2} G \sum_{j=1}^{N} \sum_{\substack{k=1 \\ k \neq j}}^{N} \frac{m_j m_k}{\|\mathbf{R}_j - \mathbf{R}_k\|}. \tag{2.48}$$

The angular momentum vector is given by

$$\mathbf{L} = \sum_{i=1}^{N} m_i \mathbf{R}_i \times \dot{\mathbf{R}}_i, \tag{2.49}$$

and the center of mass position vector is given by

$$\mathbf{r}_{\mathrm{CM}} = \frac{\sum_{i=1}^{N} m_i \mathbf{R}_i}{M}, \tag{2.50}$$

where

$$M = \sum_{i=1}^{N} m_i \tag{2.51}$$

is the total mass. The parameters **a** and **b** are two constant vectors. It can be shown that E and $\mathbf{L}$ are constant and that $\mathbf{r}_{\mathrm{CM}}$ depends linearly on time, with two constant vectors **a** and **b** according to

$$\mathbf{r}_{\mathrm{CM}} = \frac{\mathbf{a}t + \mathbf{b}}{M}. \tag{2.52}$$

I will leave the full derivations of these equations for later (see appendix A.2).

Bob: Oh thank the Lord. I can't take any more.

Alice: I think what Bob means to say, Prof. Starmover, is thank you very much for going over all this detailed physics with us. We will put in the time and read more later.

Professor: I am always happy to help curious and enthusiastic students.

Bob: My enthusiasm has been fluctuating with my understanding, to be completely honest, but I am also very thankful for your time, Prof. Starmover. You sure do know your stuff. And don't worry about me, I'll get the hang of this stuff sooner or later. I tend to understand math better when I am communicating it to a computer, for some reason.

Alice: I'm twice as excited as before we came to meet with you! And Bob is never happy without a keyboard at his fingertips!

Bob: Hey! All right, it's true.

2.5 Multiplanet Systems

The next day, Bob and Alice meet again with Prof. Starmover. Bob and Alice enter the office of Prof. Starmover.

Bob: I did my homework. I think I am actually starting to understand this stuff!

Professor: Very glad to hear it!

Bob: We now know a lot about the two-body problem and a bit about the N-body problem. I can't wait to start coding it up and see if we can reproduce the analytical results that you have showed us, Prof. Starmover.

Alice: I agree that the two-body problem would be a good starting place for our integrator. But I would first like to know a bit more about the more general N-body problem, in particular multiplanet systems. Since the two-body problem was already solved by Newton in the seventeenth century, I am sure that many clever scientists have looked at the $N > 2$ problem, even though useful general analytical solutions do not exist as you have pointed out, Prof. Starmover.

Professor: That is true. Over the centuries, many great scientists such as Laplace, Lagrange, Euler, Haret, Poincaré, and Sundman have made much progress with the $N > 2$ problem. Understandably, most attention has been given to the $N = 3$ problem, but scientists have also considered the case of multiplanet systems with many planets (i.e., more than two). In the latter case, there is a central massive body orbited by a number of much less massive bodies. This applies, for example, to the Solar System.

Alice: Why don't you tell us about multiplanet systems, since it relates to our project?

Professor: The French scientists Laplace and Lagrange made great strides in the theory of multiplanet systems in the eighteenth century. Their theory is known as Laplace–Lagrange theory, and it has been used with great success for the Solar System. The derivation of their theory is rather complicated and I will not give a detailed description here. The essence of the theory is that they express the Hamiltonian of the system in terms of the orbital elements and carry out an expansion in terms of inclinations and eccentricities.

2.5.1 Hamiltonian Mechanics

Bob: Hold on, Prof. Starmover. What is a Hamiltonian, and how does it relate to what we have talked about so far?

Professor: Let me not digress by discussing Hamiltonian mechanics in detail.

Bob: Would we be here all day?

Professor: We would be here all year. Suffice it to say that classical mechanics, as we have discussed until now, can be recast in terms of Hamilton's equations. In Hamiltonian mechanics, one describes the state of the system in terms of "canonical" coordinates (q_i, p_i). In our context, you can interpret q_i as the component of a position vector of a body, whereas p_i is the associated momentum. Furthermore, the Hamiltonian $\mathcal{H}$ is just the total energy of the system, that is, the sum of the kinetic and potential energies. Hamilton's equations state that the time evolution of a dynamical system is given by derivatives of the Hamiltonian:

$$\frac{\mathrm{d}p_i}{\mathrm{d}t} = -\frac{\partial \mathcal{H}}{\partial q_i}; \qquad \frac{\mathrm{d}q_i}{\mathrm{d}t} = \frac{\partial \mathcal{H}}{\partial p_i}. \tag{2.53}$$

To make this a bit more concrete, consider a single particle with position q in one dimension subject to a potential $V(q)$. Its momentum is $p = mv = m\dot{q}$. The equation of motion according to classical mechanics is

$$F = \dot{p} = -\frac{\partial V(q)}{\partial q}. \tag{2.54}$$

Now let us write the Hamiltonian in terms of q and p. The kinetic energy is $\frac{1}{2}mv^2 = \frac{1}{2}m\dot{q}^2 = p^2/(2m)$, so

$$\mathcal{H} = T + V = \frac{p^2}{2m} + V(q). \tag{2.55}$$

Substituting Hamilton's equations gives

$$\dot{p} = -\frac{\partial V(q)}{\partial q}; \qquad \dot{q} = \frac{p}{m}. \tag{2.56}$$

The first equation is just the equation of motion, and the second defines the relation between momentum and velocity. So classical mechanics and Hamilton's equations are equivalent.

Alice: Okay, but Hamiltonian mechanics seems more abstract than classical mechanics. If it is equivalent, then why bother with it at all?

Professor: Hamiltonian mechanics has advantages in complex systems in which the "standard" Cartesian coordinates do not work well. In particular, it has been used extensively in celestial mechanics. Getting back to Laplace and Lagrange: they expressed $\mathcal{H}$ in terms of the orbital elements and subsequently expanded it in terms of the "coordinates" for the eccentricities and inclinations. This implicitly assumes that the eccentricities and inclinations are small, which is a good approximation for the Solar System. Subsequently, they retained in their expansion only "secular terms," that is, terms that do not depend on the orbital phases and that describe the long-term evolution.

Bob: You have said the words "orbital elements" a few times now. What are those?

Professor: Fair enough. Before continuing, I think it might be useful to briefly talk about orbital elements.

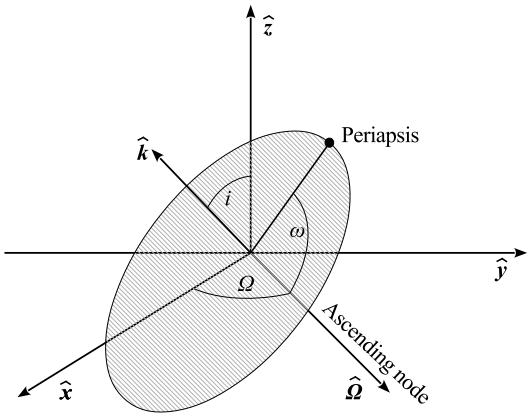


Figure 2.4
Illustration of the orbital elements in the (x, y, z)-plane.

2.5.2 Orbital Elements

Professor: The appealing aspect of Hamiltonian mechanics is that the equations of motion follow very easily once the coordinates have been chosen and the Hamiltonian has been expressed in terms of these coordinates. In the case of multiplanet systems and the Solar System in particular, the Cartesian coordinates (x, y, z) and (v_x, v_y, v_z) that you are familiar with are not the most convenient coordinates to use. This is because on short time scales, the planets are orbiting a central object on nearly Keplerian orbits. Purely Keplerian orbits are described by five constant orbital elements and one time-varying element, that is, the orbital phase (the true anomaly θ that we discussed before). In contrast, for purely Keplerian orbits, the Cartesian coordinates are all time varying, even for circular orbits. As a rule of thumb, the more "constant" the coordinates are, the simpler the equations of motion. In summary, five of the orbital elements define the orientation and shape of the orbit, and the true anomaly gives the angular position along such orbit.

Bob: So what are the orbital elements precisely, and how are they related to the more familiar Cartesian coordinates?

Professor: The six orbital elements are the semimajor axis a, the eccentricity e, the true anomaly θ, the inclination i, the argument of pericenter ω, and the longitude of the ascending node Ω. You are already familiar with the semimajor axis, the eccentricity, and the true anomaly θ. To explain the other elements, let me draw a picture. In figure 2.4, I have drawn a Keplerian orbit and a Cartesian reference frame. When describing orbital elements, one has to specify the "reference plane" and the "reference direction." Traditionally, the reference plane is taken to be the (x, y)-plane, and the reference direction is taken to be the direction of increasing x (i.e., $\hat{\mathbf{x}}$). The orbital plane intersects with the reference plane at a line, which is known as the nodal line. Along this line, the direction corresponding to the body passing

from below to above the reference plane is the ascending node. Then, we define the angle measured in the reference plane between the reference direction and the ascending node as Ω, the longitude of the ascending node. The vertical tilt of the orbital plane with respect to the reference frame is the inclination i. Zero inclination means that the orbital and reference planes coincide. The latter also occurs if $i = \pi$. If $i < \pi/2$, the orbit is also referred to as being prograde, whereas for $i > \pi$, it is retrograde. In the latter case, the body is moving in the opposite direction compared to prograde orbits. Last, the argument of pericenter ω is the angle measured in the orbital plane between the line of the ascending nodes and the periapsis direction of the orbit, that is, it defines the orientation of the orbit within the orbital plane.

Alice: How are the Cartesian coordinates and orbital elements related?

Professor: There are equations for computing the orbital elements from the Cartesian position and velocities and vice versa. I will not give their derivation, but because you will be needing them frequently later on, I give the equations below.

2.5.2.1 From orbital elements to Cartesian coordinates

The eccentricity and q-vectors are given in terms of the orbital elements by

$$\begin{aligned}\hat{\mathbf{e}} = {} & [\cos(\Omega)\cos(\omega) - \sin(\Omega)\sin(\omega)\cos(i)]\,\hat{\mathbf{x}} + [\sin(\Omega)\cos(\omega) + \cos(\Omega)\sin(\omega)\cos(i)]\,\hat{\mathbf{y}} \\ & + \sin(\omega)\sin(i)\,\hat{\mathbf{z}}; \end{aligned} \tag{2.57a}$$

$$\begin{aligned}\hat{\mathbf{q}} = {} & [-\cos(\Omega)\sin(\omega) - \sin(\Omega)\cos(\omega)\cos(i)]\,\hat{\mathbf{x}} + [-\sin(\Omega)\sin(\omega) + \cos(\Omega)\cos(\omega)\cos(i)]\,\hat{\mathbf{y}} \\ & + \cos(\omega)\sin(i)\,\hat{\mathbf{z}}. \end{aligned} \tag{2.57b}$$

In terms of these vectors, the relative position and velocity vectors are given by

$$\mathbf{r} = \frac{a\left(1-e^2\right)}{1+e\cos(\theta)}\left[\cos(\theta)\,\hat{\mathbf{e}} + \sin(\theta)\,\hat{\mathbf{q}}\right]; \tag{2.58a}$$

$$\mathbf{v} = \sqrt{\frac{GM}{a(1-e^2)}}\left[-\sin(\theta)\,\hat{\mathbf{e}} + (e+\cos(\theta))\,\hat{\mathbf{q}}\right]. \tag{2.58b}$$

Also, there is a useful relation for the mutual inclination between two orbits in terms of the orbital elements that can be derived from equations (2.57). From the latter equations, one can show that the direction of the orbital angular momentum vector is given by

$$\hat{\mathbf{k}} = \hat{\mathbf{e}} \times \hat{\mathbf{q}} = \sin(\Omega)\sin(i)\,\hat{\mathbf{x}} - \cos(\Omega)\sin(i)\,\hat{\mathbf{y}} + \cos(i)\,\hat{\mathbf{z}}. \tag{2.59}$$

Note that $\hat{\mathbf{k}}$ does not depend on ω, as should it be: the in-plane orientation angle of the line of apsides does not affect the orbital angular momentum vector. By dotting the unit angular momentum vectors of two orbits (indicated with "1" and "2") with each other, we get the

mutual inclination, that is,

$$\cos(i_{12}) = \hat{\mathbf{k}}_1 \cdot \hat{\mathbf{k}}_2 = \cos(i_1)\cos(i_2) + \sin(i_1)\sin(i_2)\cos(\Omega_1 - \Omega_2). \tag{2.60}$$

2.5.2.2 From Cartesian coordinates to orbital elements

The specific energy is given by

$$E_{\text{sp}} = \frac{1}{2}\mathbf{v}^2 - \frac{GM}{r}, \tag{2.61}$$

from which the semimajor axis follows according to

$$a = -\frac{GM}{2E_{\text{sp}}}. \tag{2.62}$$

The eccentricity can be determined from the magnitude of the LRL vector:

$$\mathbf{e} = \frac{1}{GM}\mathbf{v} \times (\mathbf{r} \times \mathbf{v}) - \hat{\mathbf{r}} = \frac{1}{GM}\left[\mathbf{r}\,\mathbf{v}^2 - \mathbf{v}\,(\mathbf{r} \cdot \mathbf{v})\right]. \tag{2.63}$$

The inclination is the angle between the orbital angular momentum vector and a reference direction, which we take to be the z-axis. Therefore,

$$\cos(i) = \hat{\mathbf{z}} \cdot \hat{\mathbf{k}}, \tag{2.64}$$

where

$$\hat{\mathbf{k}} = \frac{\mathbf{r} \times \mathbf{v}}{\|\mathbf{r} \times \mathbf{v}\|}. \tag{2.65}$$

The longitude of the ascending nodes Ω follows from the ascending node vector, $\hat{\mathbf{\Omega}}$:

$$\hat{\mathbf{\Omega}} = \hat{\mathbf{z}} \times \hat{\mathbf{k}}, \tag{2.66}$$

such that

$$\cos(\Omega) = \hat{\mathbf{x}} \cdot \hat{\mathbf{\Omega}}; \qquad \sin(\Omega) = \hat{\mathbf{y}} \cdot \hat{\mathbf{\Omega}}. \tag{2.67}$$

Also, a vector perpendicular to $\hat{\mathbf{e}}$ and $\hat{\mathbf{k}}$ is

$$\hat{\mathbf{q}} = \hat{\mathbf{k}} \times \hat{\mathbf{e}}. \tag{2.68}$$

And we get for the argument of pericenter, ω,

$$\cos(\omega) = \hat{\mathbf{e}} \cdot \hat{\mathbf{\Omega}}; \qquad \sin(\omega) = -\hat{\mathbf{q}} \cdot \hat{\mathbf{\Omega}}. \tag{2.69}$$

Finally, the true anomaly is determined by

$$\cos(\theta) = \hat{\mathbf{e}} \cdot \hat{\mathbf{r}}; \qquad \sin(\theta) = \hat{\mathbf{q}} \cdot \hat{\mathbf{r}}. \tag{2.70}$$

Alice: Why are you giving expressions for both the sines and cosines of the angles?

Professor: Except for the inclination i, the orbital angles have a range of 2π radians, or 360 degrees. In order to specify them, a single trigonometric projection is therefore not enough; we need both projections of the angles on the unit circle on the x- and y-axes, that is, the sines and cosines. In a computer code, you would get the actual orbital angles by using the arctangent function (preferably the ATAN2 function) with the following identity: $\alpha = \arctan[\sin(\alpha)/\cos(\alpha)]$ for some angle α.

Bob: Talking about computer code, will these relations hold for arbitrary orbits?

Professor: There are indeed a number of singular cases that can be tricky to deal with numerically. If $e = 1$, the orbit is a parabola, and the orbital energy E_{sp} is zero. Equation (2.62) then implies $a \to \infty$.

Alice: If $e = 1$ and $a \to \infty$, how do we get the angular momentum of the orbit? It seems ill-advised to use equation (2.42).

Professor: A parabolic orbit is characterized by the periapsis distance, r_{p}. At periapsis, $\mathbf{r}$ and $\mathbf{v}$ are perpendicular, so we simply have $k = r_{\text{p}} v_{\text{p}}$, where v_{p} is the speed at periapsis. We can get a relation between r_{p} and v_{p} by using equation (2.61) together with the fact that E_{sp} is zero, that is,

$$v_{\text{p}} = \sqrt{\frac{2GM}{r_{\text{p}}}}. \tag{2.71}$$

From this, we get

$$k = \sqrt{2GMr_{\text{p}}}. \tag{2.72}$$

For hyperbolic orbits ($E_{\text{sp}} > 0$), keep in mind that equation (2.62) implies that the semi-major axis is negative.

There are also some complications for bound orbits ($0 \leq e < 1$), in particular if $e = 0$ (perfectly circular orbit). In the latter case, there is no well-defined closest point in the orbit (all points are equally distant to the focus), and so $\hat{\mathbf{e}}$ and the argument of periapsis, ω, are not defined. One could set $\hat{\mathbf{e}}$ to be a fixed vector in this case (e.g., $\hat{\mathbf{e}} = \hat{\mathbf{x}}$) and keep in mind that ω is not well defined. Another problem arises if $i = 0$ (equatorial orbits), or $i = \pi$ (retrograde orbits). In this case, the nodal line (i.e., $\hat{\mathbf{\Omega}}$) is not defined. Again, we could, for example, set $\hat{\mathbf{\Omega}} = \hat{\mathbf{x}}$, and remember that Ω is not well defined.

2.5.3 Laplace–Lagrange Theory

Professor: Let us get back to Laplace–Lagrange theory. The essence of the theory is to express the Hamiltonian of a multiplanet system in terms of the orbital elements and to carry out an expansion in terms of inclinations and eccentricities. In other words, the theory applies only if the eccentricities and inclinations are small, which is a good approximation

for the Solar System, although not necessarily for all exoplanetary systems. Rather than giving a full derivation, which is complicated, I will just state the result for the evolution of the eccentricities and the inclinations as a function of time. First, let me introduce new variables h_j, k_j, p_j, and q_j for the jth planet according to

$$h_j = e_j \sin(\bar{\omega}_j); \qquad k_j = e_j \cos(\bar{\omega}_j); \qquad p_j = I_j \sin(\Omega_j); \qquad q_j = I_j \cos(\Omega_j). \tag{2.73}$$

Here, $\bar{\omega}_j$ is the longitude of periapsis, given by the sum of the argument of pericenter and the longitude of the ascending node, that is, $\bar{\omega}_j \equiv \omega_j + \Omega_j$. Also, let me introduce the quantities g_i and f_i, which are known as the secular eccentricity and inclination frequencies, respectively. They are defined as the eigenvalues of the matrices A_{jk} and B_{jk}, respectively, where A_{jk} and B_{jk} are given explicitly in terms of the masses of the central star, $M_\star$, and of the planets, m_k, and the semimajor axes of the planets, a_k:

$$A_{jj} = \frac{n_j}{4} \sum_{\substack{k=1 \\ k \neq j}}^{N} \frac{m_k}{M_\star + m_j} \alpha_{jk} \bar{\alpha}_{jk} b^{(1)}_{3/2}(\alpha_{jk}); \quad A_{jk} = -\frac{n_j}{4} \frac{m_k}{M_\star + m_j} \alpha_{jk} \bar{\alpha}_{jk} b^{(2)}_{3/2}(\alpha_{jk}); \tag{2.74}$$

$$B_{jj} = \frac{n_j}{4} \sum_{\substack{k=1 \\ k \neq j}}^{N} \frac{m_k}{M_\star + m_j} \alpha_{jk} \bar{\alpha}_{jk} b^{(1)}_{3/2}(\alpha_{jk}); \quad B_{jk} = \frac{n_j}{4} \frac{m_k}{M_\star + m_j} \alpha_{jk} \bar{\alpha}_{jk} b^{(1)}_{3/2}(\alpha_{jk}). \tag{2.75}$$

Here, $n_j \equiv \sqrt{GM_\star / a_j^3}$ is the mean motion of orbit j (neglecting the planetary masses relative to the stellar mass), and α_{jk} and $\bar{\alpha}_{jk}$ are semimajor axis ratios:

$$\alpha_{jk} = \begin{cases} a_k/a_j, & a_j > a_k; \\ a_j/a_k, & a_j < a_k; \end{cases} \tag{2.76}$$

$$\bar{\alpha}_{jk} = \begin{cases} 1, & a_j > a_k; \\ a_j/a_k, & a_j < a_k. \end{cases} \tag{2.77}$$

Finally, $b^{(1)}_{3/2}(\alpha)$ and $b^{(2)}_{3/2}(\alpha)$ are functions of the semimajor axis ratios:

$$b^{(1)}_{3/2}(\alpha) = \frac{1}{\pi} \int_0^{2\pi} \frac{\cos(\psi)\, \mathrm{d}\psi}{(1 - 2\alpha\cos(\psi) + \alpha^2)^{3/2}}; \tag{2.78}$$

$$b^{(2)}_{3/2}(\alpha) = \frac{1}{\pi} \int_0^{2\pi} \frac{\cos(2\psi)\, \mathrm{d}\psi}{(1 - 2\alpha\cos(2\psi) + \alpha^2)^{3/2}}. \tag{2.79}$$

By the way, don't worry if you are daunted by the concept of matrices and eigenvalues. I can briefly explain these later (see appendix C).

Given all of these definitions, the explicit time evolution of h_j, k_j, p_j, and q_j, and therefore of the eccentricities, inclinations, arguments of pericenter, and the longitudes of the

ascending nodes, is given by

$$h_j = \sum_{i=1}^{N} e_{ji} \sin(g_i t + \beta_i); \quad k_j = \sum_{i=1}^{N} e_{ji} \cos(g_i t + \beta_i); \tag{2.80}$$

$$p_j = \sum_{i=1}^{N} I_{ji} \sin(f_i t + \gamma_i); \quad q_j = \sum_{i=1}^{N} I_{ji} \cos(f_i t + \gamma_i). \tag{2.81}$$

Here, e_{ji}, β_i, I_{jk}, and γ_i are constants that are determined by the initial conditions. More specifically, they can be computed once the initial e_j, i_j, ω_j, and Ω_j values are specified.

Alice and Bob stare at Prof. Starmover with glazed eyes.

Alice: This is all quite … abstract. Could you give us an example?

Professor: Certainly! In figure 2.5, I show how the eccentricities and the inclinations of the planets in the Solar System (including the dwarf planet Pluto) evolve according to the Laplace–Lagrange solution outlined above. I am showing the evolution from the current orbital elements until a few Myr in the future. You will note that the eccentricity and inclination of each planet oscillate with multiple frequencies. These frequencies are described by the quantities g_i and f_i that were defined above; in particular, g_i is associated with oscillations in the eccentricity, whereas f_i is associated with oscillations in the inclination.

2.5.4 Nonlinear Evolution

Bob: Let me get this straight. Laplace–Lagrange theory gives us the evolution without having to actually solve Newton's equations of motion?

Professor: Yes, but only to a certain extent. Laplace–Lagrange theory works well as long as the eccentricities and inclinations remain small. This is currently the case for the Solar System, but it turns out that on much longer time scales of order billions of years, the evolution becomes "nonlinear." Nonlinear evolution can occur when secular frequencies overlap, leading to potentially chaotic evolution, which can have dramatic implications. In recent long-term N-body integrations, it has been found that in the case of the Solar System, there is a possibility on the order of a few percent that the orbit of Mercury will become unstable 5 Gyr from now. Mercury will then either collide with the Sun or be ejected from the Solar System. See, for example, Laskar and Gastineau (2009), and Lithwick and Wu (2011).

So, although the Solar System may seem stable according to Laplace–Lagrange theory, reality is more complicated. This shows the necessity for doing N-body integrations. Although they generally cannot give us the insight of more approximate but analytic approaches like Laplace–Lagrange theory, N-body integrations give us the "correct" answer. This assumes, of course, that the integrations themselves are carried out correctly.

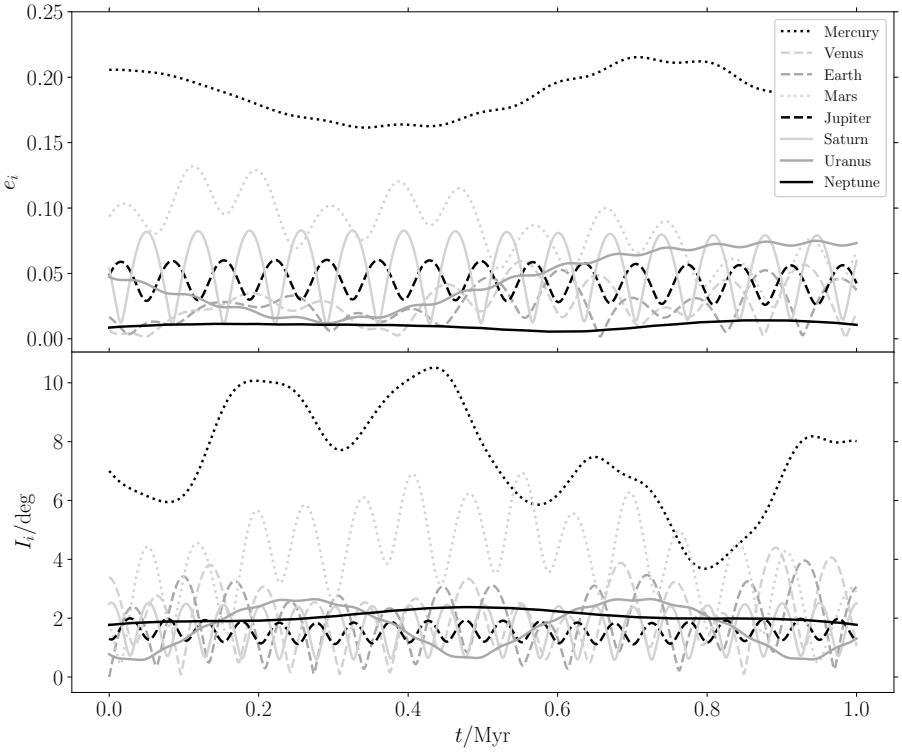


Figure 2.5
The evolution of the eccentricities and inclinations of the planets in the Solar System according to Laplace–Lagrange theory.

Bob: On that note, let us start writing our own N-body integrator!

Alice: Indeed, let's get started with our own code!

Bob takes out his laptop, his fingers poised over the keyboard in anticipation. Hesitating, Bob turns to Alice, a confused look on his face.

Bob: Uh … now what?

Alice: Hmmm…. We might need to ask a few more questions first.

Bob: I *knew* it was too good to be true that the time for coding is upon us….

3 First Two-Body Code

Overview. Having received a firm theoretical background from Prof. Starmover, Alice and Bob are keen to implement their own N-body integrator. They decide that now is the time to discuss practical issues and get their hands dirty! They first choose a programming language and then write their first two-body integrator, with some success.

3.1 Choosing a Programming Language

Bob: Finally, my time has arrived. Stand back! I need room to work my programming magic!

Alice: Sure thing, Bob. Whatever you say. Okay, here we go, getting ready to program our own integrator! I'm excited! We are using PYTHON, right?

Bob: Yeah, that's the one I prefer. But there are many other options! The most popular programming language nowadays seems to be JAVA, followed by C/C++, and then PYTHON[1]. But as I understand it, many physicists are still using FORTRAN.

Alice: Why is that?

Bob: FORTRAN stands for "Formula Translation." Just as its name suggests, it is a "traditional" programming language used to perform calculations related to various numerical problems. But in recent years, computers have become more and more powerful. That is great news for us in terms of being able to solve more complicated and sophisticated problems. But it's also bad news for us, in the sense that numerical codes are becoming increasingly complicated as a result of this. More and more scientists now prefer to use object-oriented programming languages to structure their codes in a modular way. This makes the subsequent maintenance much easier.

Alice: Okay, but what about C/C++, JAVA, and PYTHON[2]?

---

[1] https://www.tiobe.com/tiobe-index/

[2] https://www.python.org

Bob: I can understand the popularity of JAVA and C/C++. Many websites are programmed in JAVA. The ANDROID operating system for our mobile phones is also powered by JAVA. Of similar widespread use, C/C++ is a language for operating system development. Recently, a new language named JULIA[3] has emerged, which is mainly optimized for high-performance computing. However, in astronomy/astrophysics and data science, PYTHON is a preferred language. Many scientific packages are available in PYTHON.

Alice: What are the advantages of PYTHON?

Bob: PYTHON is a very user-friendly programming language. It is highly readable. Writing a "Hello World" in PYTHON is just one line:

```
print('Hello␣World')
```

Alice: That's simple enough! But where should we run this line?

Bob: PYTHON is an interpreted language, which means that compilation is not needed. Instead, it provides an interactive shell, allowing us to execute the commands line-by-line[4]:

```
Python 3.7.5 (default, Oct 25 2019, 15:51:11)
[GCC 4.2.1 Compatible Clang 4.0.1 (tags/RELEASE_401/final)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> a = 1
>>> b = 2
>>> print('hello', a+b)
hello 3
```

Alice: This is pretty cool! But wait, I learned from my C/C++ course that one needs to declare variables before using them. Here, you just assign `a=1` and `b=2` directly. What are their data types?

Bob: PYTHON is a dynamic language. When you assign a value to a variable (i.e., `a=1`), the interpreter detects that `a` has never been declared previously and that `1` is an integer. So it allocates a memory space to store an integer variable. It is the same for `b` and all other assignments.

Alice: This looks very convenient, but I am not sure if I feel entirely comfortable with it. After attending the C/C++ course, the concept of variable declaration is deeply rooted in my mind.

Bob: It is the same basic structure for JAVA and FORTRAN as well.

Alice: Okay, I'll take the bait. Isn't it dangerous to use a variable directly?

---

[3] https://julialang.org

[4] The PYTHON code snippets in this book are compatible with both PYTHON 2 and PYTHON 3. However, we encourage the readers to run, test, and develop their codes with PYTHON 3. For the differences between the two major versions of PYTHON, see https://wiki.python.org/moin/Python2orPython3.

Bob: Great point. This is actually a controversial topic. On one hand, it is more convenient to use a variable whenever you need it, which is a bonus for many programmers. Besides, the PYTHON interpreter is smart enough to detect data-type inconsistency. For example, the following code will raise an exception:

```
>>> a=1
>>> b='a␣string'
>>> print(a+b)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

On the other hand, this same advantage can become a pitfall when the programmer makes a typo. Consider the following example:

```
>>> my_variable = 10
>>> i = my_variable ** 2
>>> if i < 50:
...     my_variable = my_variable + 1
... else:
...     my_varaible = my_variable - 1
```

As you can see, the programmer makes a typo on the last line, which causes PYTHON to create a new variable. This is certainly a serious problem, since it is not what the programmer intended and no error or exception will be flagged.

Alice: I get it, but why then do we want to use PYTHON and not one of the other languages?

Bob: Nothing is perfect, including those other languages. I think the convenience of PYTHON is more important in the long run, for most common applications. As long as we test our code carefully, errors like this are detectable.

Alice: But what if I have 1,000 lines of code? Must I type them line-by-line in the interactive console?

Bob: Does that sound to you like something *I* would do?

Alice: Frankly, yes.

Bob: Sigh. Fair enough, I suppose I would. But no, we don't have to type it all in line-by-line. We will put our code into a PYTHON script file and execute it like this:

```
$ python my_code.py
```

Alice: Great! All right, I'm starting to come around about using PYTHON. I notice that you used indentations to delimit code blocks. So PYTHON doesn't use curly brackets? And there is no need to put a semicolon at the end of each statement?

Bob: You got it! This is why PYTHON codes are more readable to humans. Let's take a look at the following snippet of code:

```
import math

def sine(a):
    return math.sin(a)

for i in range(10):
    x = i
    y = sine(i)
    print(x, y)
```

It is straightforward to read, isn't it? PYTHON is a modular language. In this particular example, we initialize an `iterator` from 0 to 9, and calculate the corresponding sin() values using the method provided by the `math` module.

Alice: When you import the `math` module, you just reminded me of the MATLAB language.[5] I very much like the feature that variables are vectorized. Must we write `for` loops in PYTHON manually?

Bob: Actually, PYTHON is vectorized as well! The above example can be rewritten using `numpy`,[6] a package dedicated for numerical calculations.

```
import numpy as np
x = np.arange(0, 10)
y = np.sin(x)
```

This shortens the code to only three lines. By the way, PYTHON allows us to rename the imported names. On the first line, I just rename the package name `numpy` as `np`, because this saves me time typing the code.

Alice: Wait … I cannot reproduce your result on my laptop. When I try to import `numpy`, I encounter this error:

```
>>> import numpy as np
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ModuleNotFoundError: No module named 'numpy'
```

Bob: Ah! It is because you haven't installed `numpy` properly. Installing a PYTHON package is usually quite easy. You could just use PIP, the package management system of PYTHON to finish the installation of the package within one line in the terminal:

```
pip install numpy
```

[5] https://www.mathworks.com/products/matlab.html

[6] http://www.numpy.org

Alice: That's it? In my C/C++ class, I remember that I needed to compile the source code with the MAKEFILE. What if the package that you want to install depends on other packages that haven't been installed on the system?

Bob: This is an excellent question! PIP can automatically resolve the dependency. For example, let's say that we would like to install the PYTHON plotting library `matplotlib`, which depends on `numpy`. If we simply use one line of code `pip install matplotlib`, the PIP system will automatically calculate its dependencies, fetch the appropriate version and install them, and eventually install `matplotlib` at the end!

Alice: I am falling in love with PYTHON!

Bob: To make you even more excited.... Since you mentioned MATLAB. One very cool feature of MATLAB is that it comes with amazing visualization functionalities. I just mentioned `matplotlib`: in PYTHON, we could also easily make 2D and 3D plots using the `matplotlib` package. Let's look at a slightly more elaborate example:

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(-10, 10, 1000)
y = np.sin(x) + np.sin(2 * x)

plt.plot(x, y)
plt.xlabel('$x$')
plt.ylabel('$y$')
plt.savefig('test_plot.pdf')
plt.show()
```

In this example, we use the `linspace()` method from `numpy` to generate a linear vector, ranging from -10 to 10, with 1,000 steps. Then we calculate two sinusoidal components and add them together with the operator `+`. No loop is ever needed, it just works! At the end, we customize the labels on the x-axis and y-axis using LaTeXsyntax, and the plot is saved as a PDF figure, which is a publication-quality vector image as shown in figure 3.1. Making a plot is really easy, right?

Alice: You're kidding! That rocks! It would have taken a lot more lines to write a similar code in C/C++. Okay, now I am fully convinced! Let's go for PYTHON!

3.2 Forward Euler Integrator

Alice: So we are ready to take the very first step! My impression is that there are many different types of algorithms for numerical simulations. Which one should we use?

Bob: Great question! I've been reading up on this and have some thoughts. Different integrators are useful for different types of gravitational systems. For example, the fourth-order Hermite integrator (Makino and Aarseth, 1992) using the "predictor-corrector"

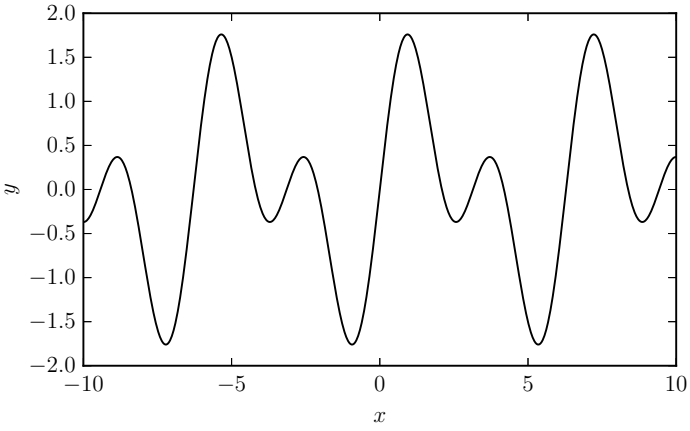


Figure 3.1
Plot generated by the PYTHON script representing $f(x) = \sin(x) + \sin(2x)$.

scheme is optimal for star cluster dynamics. On smaller scales, symplectic integrators such as the Wisdom–Holman integrator are often used for investigating secular evolution of planetary systems, since the numerical errors do not compound over time. Sometimes people also use the Bulirsch–Stoer integrator or IAS15 for better handling of close encounters.

Alice: Wait. You just brought up tons of new terms. What is what?

Bob: Let's go through a few integrators one by one and see which is more suitable to our problem. Let's start with the simplest possible integrator. So I learned from the reading I did that the simplest integrator is the forward Euler integration scheme. Every step is advanced by assuming that the object drifts at a uniform velocity during a small time interval Δt. Like this:

$$\mathbf{x} = \mathbf{x}_0 + \mathbf{v}_0 \Delta t \tag{3.1a}$$

$$\mathbf{v} = \mathbf{v}_0 + \mathbf{a}_0 \Delta t, \tag{3.1b}$$

where $\mathbf{v}_0$ is the velocity of the particle at time t_0, $\mathbf{x}$ is the expected position at $t = t_0 + \Delta t$, and $\mathbf{a}_0$ is the gravitational acceleration at time t_0.

Alice: I see why this is the simplest algorithm. These are just the basic kinematic equations of linear motion in physics. Let's code it up and see what will happen! Let's assume the simplest possible case, where there are only two particles in the system. This two-body system has a well-known analytical solution, so we can validate the results of our integrator using our theoretical understanding of the actual physics!

Let's say we want to simulate the Sun–Earth system. The semimajor axis of the Earth is $a = 1$ AU. For simplicity, let's just assume that the Earth orbits the Sun on a circular orbit, and so the distance between the Earth and the Sun is a constant $r = 1$ AU. The mass of the

Sun and that of the Earth are M and m, respectively. Let G be the universal gravitational constant and $\mathbf{r}$ be the position vector of the Earth relative to the Sun. So the gravitational acceleration between the Sun and the Earth is

$$\mathbf{a} = -G\frac{M\mathbf{r}}{r^3}, \tag{3.2}$$

where the minus sign in the beginning of the equation indicates that the gravitational acceleration is pointing towards the opposite direction of $\mathbf{r}$ (i.e., the origin). The centrifugal force per unit mass of the Earth is

$$\mathbf{a}_c = \frac{\mathbf{v}^2}{r}, \tag{3.3}$$

where $\mathbf{v}$ is the linear velocity of the Earth. In order to have the Earth moving on a circular orbit, $|\mathbf{a}| = \mathbf{a}_c$ must hold everywhere along the orbit. Equating equation (3.2) and equation (3.3), we have $\mathbf{v} = \sqrt{GM/\mathbf{r}}$. This is often called the *circular velocity*.

Bob: Let's plug in some of these numbers, so that we have the initial conditions ready. If $M = 1$, then $m = 3\times10^{-6}$ (the mass ratio between the Sun and the Earth is 3×10^{-6}). Since it is just a toy model, let's assume that $G = 1$, in which case we have $v = 1$. It is convenient to assume that the Sun is at the origin of a Cartesian coordinate system with position vector $\mathbf{X} = [0,0,0]$ and zero velocity, $\mathbf{V} = [0,0,0]$, then we have $\mathbf{x} = [1,0,0]$, $\mathbf{v} = [0,1,0]$. Now we have our initial conditions, written in PYTHON as:

```
M = 1.0
m = 3.0e-6

X = [0., 0., 0.]
V = [0., 0., 0.]
x = [1., 0., 0.]
v = [0., 1., 0.]
```

Alice: Nice numbers, Bob.

Bob: Thank you, thank you. I typed them myself.

Alice: But it is a three-dimensional problem. Since PYTHON is vectorized, is it possible to avoid manually calculating the three components of each vector by using `numpy`?

Bob: Wow, I'm impressed. That's a great idea! In this case, I will have to wrap the lists of coordinates with `numpy.array()`:

```
import numpy as np
M = 1.0
m = 3.0e-6
G = 1.0

X = np.array([0., 0., 0.])  # the position vector of the Sun
x = np.array([1., 0., 0.])  # the position vector of the Earth
V = np.array([0., 0., 0.])  # the velocity vector of the Sun
v = np.array([0., 1., 0.])  # the velocity vector of the Earth
```

The acceleration can be calculated with

```
a = -G * M * (x - X) / np.linalg.norm(x - X) ** 3
```

Alice: I think the acceleration **a** should be placed in a loop, so that whenever the position is updated, it will be updated accordingly.

Bob: You are right. We need a loop to do the actual integration! In order to construct the loop, we will need to define when it ends and to choose a time step. Let's try this:

```
dt = 0.1
t_end = 10
```

And then we put the calculation for **a** in the loop.

```
for t in np.linspace(0, t_end, t_end/dt):
    a = -G * M * (x - X) / np.linalg.norm(x - X) ** 3
```

Alice: The `for` loop looks elegant! So far, we have the velocity vector **v** and acceleration vector **a**, so the **x** and **v** vectors after `dt` are

```
x = x + v * dt
v = v + a * dt
```

Bob: Yes, let me put these two lines into the loop too. I think it would be nice to make a plot of the $(x-y)$ space to test our integrator. We need to save our calculation at each time step. We can use PYTHON's `list` data structure to store the vectors. I will need to declare three lists for x, y, and z, respectively:

```
x_vec = []
y_vec = []
z_vec = []
```

Alice: Oh wow, it is so easy to define vectors in PYTHON! When I was programming in C/C++, I had to use the `malloc()` function (C) or the `new` keyword to allocate memory. And of course I need to use `free()` or `delete` to clean up the memory at the end. Is the memory management in PYTHON automatic?

Bob: Yes! PYTHON has its own memory management system. When it detects that there is no more reference to a variable or an object, it will free the memory occupied by the object. This mechanism is called "garbage collection." JAVA also implements a similar mechanism.

So, let's put everything together, and we have the full code for our first integrator!

```
import numpy as np
import matplotlib.pyplot as plt

M = 1.0
m = 3.0e-6
```

```
G = 1.0

X = np.array([0, 0, 0])  # the position vector of the Sun
x = np.array([1, 0, 0])  # the position vector of the Earth
V = np.array([0, 0, 0])  # the velocity vector of the Sun
v = np.array([0, 1, 0])  # the velocity vector of the Earth

dt = 0.001
t_end = 100
x_vec = []  # x of the Earth
y_vec = []  # y of the Earth
X_vec = []  # x of the Sun
Y_vec = []  # y of the Sun

for t in np.linspace(0, t_end, int(t_end/dt)):
    # acc Sun --> Earth
    a = -G * M * (x - X) / np.linalg.norm(x - X) ** 3
    # acc Earth --> Sun
    A = -G * m * (X - x) / np.linalg.norm(X - x) ** 3

    # advance the positions and velocities
    x = x + v * dt
    X = X + V * dt
    v = v + a * dt
    V = V + A * dt

    # store the data for plotting
    x_vec.append(x[0])
    y_vec.append(x[1])
    X_vec.append(X[0])
    Y_vec.append(X[1])

# make the plot
plt.plot(x_vec, y_vec)
plt.plot(X_vec, Y_vec)
plt.xlabel('$x$')
plt.ylabel('$y$')
plt.show()
```

Alice: I'm excited to test it. I see that you set `dt = 0.1`.

5 minutes later…

Alice: Ouch, that doesn't look very good (see figure 3.2)!

Bob: Looks like the Earth is escaping from the Sun! That doesn't make sense. Let's decrease the time step by a factor of 10. Now with $\Delta t = 0.01$, it looks like this (see figure 3.3).

Alice: Still doesn't look right. Let's try $\Delta t = 0.001$. Take a look at figure 3.4a.

Bob: Okay, it is getting better. Let's go even further with $\Delta t = 0.0001$ (figure 3.4b).

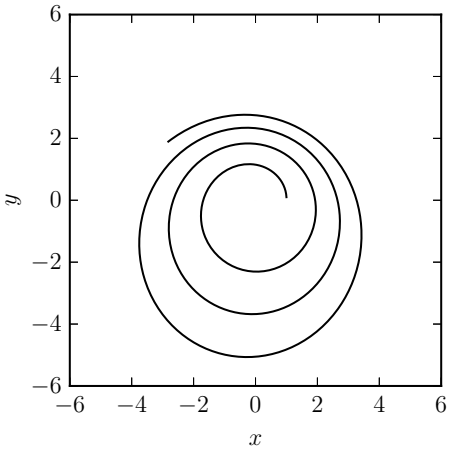


Figure 3.2
First attempt of the forward Euler integrator, with $\Delta t = 0.1$.

Alice: Getting better and better. But, the code is becoming so slow! It takes more than ten seconds to integrate just a few orbits. The computational time would be unimaginable if we would need to integrate a system for millions of years, which is necessary for studying the secular evolution of planetary systems.

In our current configuration, the Sun is much heavier than the Earth. Out of curiosity, let's try another system. As shown in figure 3.5a, let's assume a binary system with two equal-mass particles. They orbit their common center of mass on circular orbits, both with semimajor axes $a = 1$ AU. I would expect that their orbits overlap each other, just as shown in the figure. In this configuration, the two bodies have a constant separation of $r = 2$ AU. According to Newton's law of universal gravitation, the acceleration will be reduced to one fourth of its previous value, so in order to have them orbiting on the same circular orbit, we should reduce the magnitude of their velocities by half.

Bob: I appreciate the fast mental calculation! So, we just need to modify the code slightly by changing the initial conditions:

```
M = 1.0
m = 1.0

X = np.array([-1, 0, 0])  # the position vector of P1
x = np.array([1, 0, 0])  # the position vector of P2
V = np.array([0, -0.5, 0])  # the velocity vector of P1
v = np.array([0, 0.5, 0])  # the velocity vector of P2
```

And let's run the code again. We'll use `dt=0.0001` and `t_end=100`. The result is in figure 3.5b.

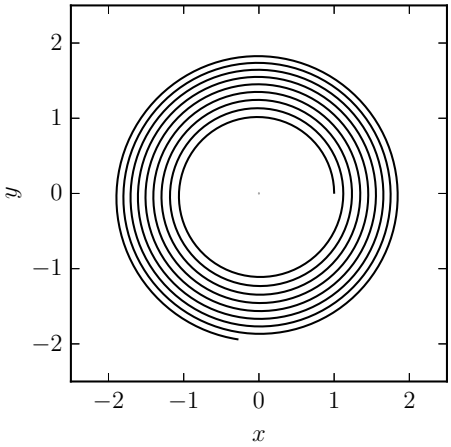


Figure 3.3
Second attempt of the forward Euler integrator, with $\Delta t = 0.01$.

Alice: Not bad! At least it is consistent with our theoretical understanding. But it is so slow! Do you know how to measure the execution time?

Bob: Yeah, it's slower than molasses. We can use the UNIX command `time` to measure the execution time. We just place it at the beginning of our command, like this:

```
$ time python forward_euler.py
python forward_euler.py  25.83s user 0.32s system 99% cpu 26.294 total
```

In conclusion: approximately twenty-six seconds of execution time is needed for just a few orbits.[7]

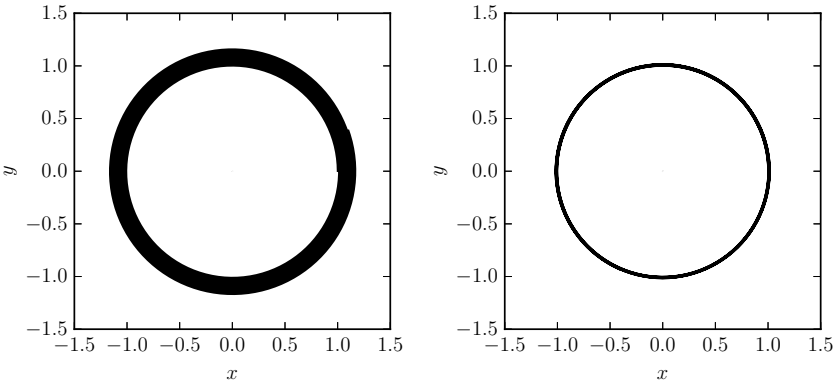
Alice: All right, let's recap all this for a second. We are forced to use tiny integration time steps. This is because, given some point along the orbit, our current integrator advances the particles tangentially for some predefined time step. In principle, if our time step is infinitely small, the integration should be strictly correct. In practice, though, we chose finite time steps, and so the error is building up with every single step.

Bob: Kind of scary, if you think about it.

Alice: After some integration time, your error has compounded to the point that the particles have significantly diverged from their correct trajectories.

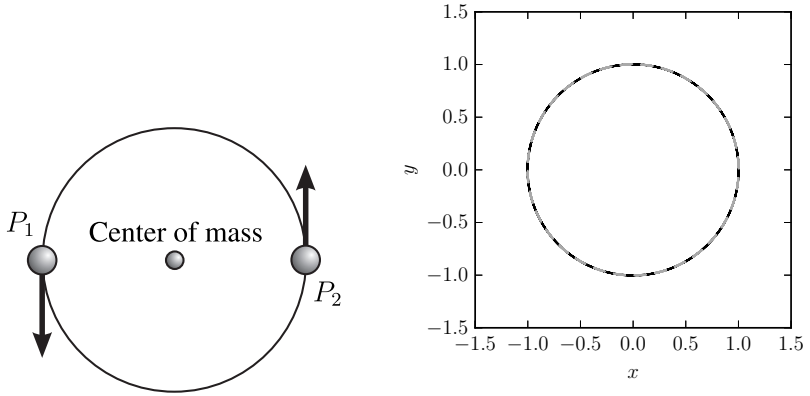
Bob: I suppose…. I think we should move on to a better integrator.

[7] The actual execution time may differ on different computers.



(a) Third attempt, with $\Delta t = 0.001$ **(b)** Fourth attempt, with $\Delta t = 0.0001$

Figure 3.4
Third and fourth attempts of the forward Euler integrator.



(a) Schematic representation **(b)** Integration with the forward Euler scheme ($\Delta t = 0.0001$)

Figure 3.5
Two equal-mass particles (P_1 and P_2) orbiting their common center of mass.

Alice: Okay, in the meanwhile, we need to work out a way to quantitatively measure the accuracy of the integrator, because the difference between `dt=0.00001` and `dt=0.000001` will be difficult to tell using just our eyes.

Bob: Let's talk to Prof. Starmover tomorrow to get some more advice. Right now, I'm tired of programming. All that performing magic has left me famished. Are you hungry?

Alice: I could eat. Let's go get food.

4 Accuracy and Performance of the Integration

Overview. Alice and Bob realize that numerical integration schemes provide only an approximation of the actual solution to a system of differential equations. Therefore, it is important to assess the accuracy of the approximation and to understand where numerical errors come from. Conserved quantities like the energy and angular momentum of the system are useful indicators for quantifying integration errors. Alice and Bob study concepts of truncation and round-off error, as well as learn how to use conservation laws to determine how accurate the integration is. In addition, they start applying some basic techniques to optimize the performance of the code.

Alice and Bob reconvene after grabbing food to discuss their progress so far.

Alice: It's exciting to see our first results! But I'm a little disappointed at how poor the accuracy of our integrator is. If we had this much trouble simulating such simple toy problems, we definitely aren't going to get the right answer when integrating more complicated systems, like extrasolar planets….

Bob: Yeah, you're right. Ultimately, we want to turn our first coding attempts into a proper, reliable simulator capable of integrating more complicated systems. It might be worthwhile to spend some time thinking about how to organize the program. In the long run, this should help us to isolate potential problems in the code, if they happen to arise.

Alice: The main part is, of course, propagating the orbit. They don't call it a "propagator" for nothing!

Bob: Fair enough. Once we get the equations of motion right, the rest should follow.

Alice: Yes, with that in place, simulating different planetary systems will just be a matter of changing the initial conditions and the masses of the bodies.

Bob: Right. The initial conditions. Now that you mention it, another important block in the code will be the initialization step. This should come before the integration.

Alice: And after integrating the orbit, we would like to plot the trajectories of the particles. That's the best part!

4.1 Structuring the Code

Alice: Let me write these ideas down. So, we want the final structure of the program to go a little something like this:

1. Load the initial conditions of the two bodies, $\mathbf{R}_{i0}$ and $\mathbf{V}_{i0}$, together with their masses, m_1 and m_2; the final integration time, t_f; and the time step, Δt.
2. Integrate the orbit up to t_f.
3. Plot the orbits.

Bob: You crushed it. That's exactly how I would write the program.

Alice: Great!

Bob: Okay, the first step we have to focus on is improving the integrator. Once we have that working, we can start to optimize the code even if it isn't efficient or tidy. I remember from previous assignments that the preferred method of writing a more complicated code is to first start with something simple, make sure that it works, and then add more advanced tricks as you go. This is much better than starting a complicated code from scratch. If you can get the simple version working right from the beginning, then it's easier to find mistakes when adding new elements to the code.

Alice: Okay, that's a good suggestion. With that in place, we can test out different integration schemes. I suggest that we implement various methods one at a time and check how well each one performs as we go. The forward Euler method was pretty easy to implement, although the results weren't very accurate.

Bob: If we are going to implement different integrators, it might be worth turning the code into something more modular. Then, we will be able to choose which integrator we want in the main part of the code, without the need for making major changes to the program.

Alice: Okay, we can decide how to swap out the integrator when we implement the propagator.

Bob: One thing I remember is that conventional integrators deal with first-order systems of the form

$$\frac{\mathrm{d}\mathbf{x}}{\mathrm{d}t} = \mathbf{f}(t, \mathbf{x}). \tag{4.1}$$

The vector $\mathbf{x}(t)$ stores the solution to the problem. In our case, I think the solution is decided by $\mathbf{R}_1(t)$ and $\mathbf{R}_2(t)$. Is that right?

Alice: Almost, you forgot about the velocities $\mathbf{V}_1(t)$ and $\mathbf{V}_2(t)$! We will need them too.

Bob: Damnit! Yeah, that makes sense.

Alice: Our solution is actually decided by $\mathbf{x}(t) = [\mathbf{R}_1(t), \mathbf{R}_2(t), \mathbf{V}_1(t), \mathbf{V}_2(t)]$.

Bob: So, the solution has dimension 12 for the two-body problem?

Alice: Yep.

Bob: Okay, so the new functional form of the solution is evaluated at first order, but we increased its dimension.

Alice: I guess it is the price to pay. Anyway, the two-body problem will take the general form

$$\frac{\mathrm{d}\mathbf{R}_1}{\mathrm{d}t} = \mathbf{V}_1; \tag{4.2a}$$

$$\frac{\mathrm{d}\mathbf{R}_2}{\mathrm{d}t} = \mathbf{V}_2; \tag{4.2b}$$

$$\frac{\mathrm{d}\mathbf{V}_1}{\mathrm{d}t} = -Gm_2\frac{\mathbf{R}_1-\mathbf{R}_2}{\|\mathbf{R}_1-\mathbf{R}_2\|^3}; \tag{4.2c}$$

$$\frac{\mathrm{d}\mathbf{V}_2}{\mathrm{d}t} = -Gm_1\frac{\mathbf{R}_2-\mathbf{R}_1}{\|\mathbf{R}_2-\mathbf{R}_1\|^3}. \tag{4.2d}$$

Bob: Isn't the denominator in the third and fourth equations the same? Since we are taking the norm of the vector, the order of the subtraction doesn't matter, right?

Alice: Yes, that's right. We can call it r^3, with $r = \|\mathbf{R}_1 - \mathbf{R}_2\|$. The right-hand side of the differential equations will then read

$$\mathbf{f} = \begin{bmatrix} \mathbf{V}_1 \\ \mathbf{V}_2 \\ -Gm_2\dfrac{\mathbf{R}_1-\mathbf{R}_2}{r^3} \\ -Gm_1\dfrac{\mathbf{R}_2-\mathbf{R}_1}{r^3} \end{bmatrix}. \tag{4.3}$$

Bob: Now we have a first-order system of equations that we can integrate with any conventional integrator. That's pretty cool. Wait … I have an idea.

Alice: Don't hurt yourself.

Bob: We should create a function in PYTHON to evaluate $\mathbf{f}(t,\mathbf{x})$. We will also need the gravitational constant G and the masses of the bodies.

Alice: That's right. The interface will be something like this, right?

```
def ode_two_body_first_order( x, t, G, masses ):
```

Bob: Yeah…. The only thing that bugs me is how we are passing the auxiliary parameters (`G` and `masses`) to the function. What if we need more parameters later on? Ending up with a long list of input arguments doesn't look very convenient. Besides, mistakes will be more likely. I think this is a good place to use a PYTHON *dictionary*, which is essentially a container where we can store different variables as entries of the dictionary. To define a

dictionary, we will define its entries as pairs of the form `'argument': value`. Let's name our parameters dictionary "`params`" and add our two parameters to it:

```
params = {'G': G, 'masses': masses}
```

Alice: That's cool! If we needed more parameters, we would just add more entries to the dictionary without having to modify the call to the function. How can we access the entries in the dictionary?

Bob: It's pretty easy. To get the value of G, for example, we just have to call `params['G']`. With this, our differential equations in PYTHON will read:

Snippet 4.1
Function `ode_two_body_first_order`: Evaluate the first-order two-body acceleration.

```
def ode_two_body_first_order( x, t, params ):
    # Retrieve the parameters:
    G = params['G']
    masses = params['masses']
    # Initialize vector:
    dxdt = x * 0
    # Retrieve state vector:
    R1 = x[0:3]
    R2 = x[3:6]
    r3 = np.linalg.norm(R2 - R1)**3
    # Differential equations:
    dxdt[0:3] = x[6:9]  # dR1/dt = V1
    dxdt[3:6] = x[9:12] # dR2/dt = V2
    dxdt[6:9] = -G * masses[0] * (R1 - R2) / r3 # dV1dt
    dxdt[9:12]= -G * masses[1] * (R2 - R1) / r3 # dV2dt
    # Output:
    return dxdt
```

Retrieving the parameters at the beginning is optional, but I'm doing it to simplify the code within the function.

Alice: The function doesn't use the time, does it?

Bob: No, I included it just to keep the same general structure for the problem. Plus, we might need it later.

Alice: It looks like the command `dxdt = x * 0` isn't really doing anything….

Bob: Looks pretty dumb, right?

Alice: Uh … yeah. This feels like a trick question….

Bob: You actually need it to initialize the vector defining the right-hand side of the equations. Without that, you can't assign values to its components.

Alice: Okay, let's keep it like that. We need to join the initial position and velocity vectors to initialize the state vector.

Bob: Yep, that is call concatenating. Like this

```
x = np.concatenate((R1, R2, V1, V2))
```

The loop then simplifies to

```
 # Store solution:
sol_state = [x]
for t in sol_time[1:]:
    # Evaluate right-hand side:
    dxdt = ode_two_body_first_order(x, t, params)
    # Advance step:
    x = x + dxdt * dt
    # Store solution:
    sol_state = np.concatenate((sol_state, [x]))
```

Alice: It looks more compact now.

Bob: In fact, we could turn this into a function that propagates the system from t_0 to t_f. The inputs will be the initial state vector, the times, the gravitational constant G, and the masses (stored in the dictionary `params`).

Snippet 4.2
Function `integrate_euler`: Forward Euler integrator.

```
def integrate_euler( x0, t0, tf, dt, params ):
    # Number of points in the dense output:
    npts = int(np.floor((tf - t0) / dt)) + 1
    # Initial state:
    x = x0
    # Vector of times:
    sol_time = np.linspace(t0, t0 + dt * (npts - 1), npts)
    # Store solution:
    sol_state = [x]
    # Main loop:
    for t in sol_time[1:]:
        # Right-hand side of the ode:
        dxdt = ode_two_body_first_order(x, t, params)
        # Advance step:
        x = x + dxdt * dt
        # Store solution:
        sol_state = np.concatenate((sol_state, [x]))
    # Output:
    return sol_time, sol_state
```

This function will return the state of each particle (position and velocity) at the time steps specified by `sol_time`. We have reduced the code to

```
# The initial conditions:
R1 = np.array([ 1.0, 0.0, 0.0])
R2 = np.array([-1.0, 0.0, 0.0])
V1 = np.array([0.0,  0.5, 0.0])
V2 = np.array([0.0, -0.5, 0.0])
x = np.concatenate((R1, R2, V1, V2))
```

```
# Mass of the bodies:
masses = [1.0, 1.0]
# Gravitational parameter:
G = 1.0
# Initial and final time:
t0 = 0.0
tf = 100.0
# Time step:
dt = 0.0001

# Parameters:
params = {'G': G, 'masses': masses}

# Integrate orbit:
sol_time, sol_state = integrate_euler(x, t0, tf, dt, params)

# Plot the orbit!
```

Alice: This makes it easier to see the structure of the code. The first part defines the initial conditions. The second part integrates the orbit. The final and *best* step is plotting the trajectory.

Bob: Yeah. You got it. And until we agree on the best way to initialize more general problems, we can create a function in which we define the initial conditions and the physical parameters that we need. Something like this:

Snippet 4.3
Function `initialize_problem`: Define initial conditions for the integration script.

```
def initialize_problem():
    # Initial conditions:
    R1 = np.array([ 1.0, 0.0, 0.0])
    R2 = np.array([-1.0, 0.0, 0.0])
    V1 = np.array([0.0,  0.5, 0.0])
    V2 = np.array([0.0, -0.5, 0.0])
    x = np.concatenate((R1, R2, V1, V2))
    # Mass of the bodies:
    masses = [1.0, 1.0]
    # Gravitational parameter:
    G = 1.0
    # Initial and final time:
    t0 = 0.0
    tf = 100.0
    # Time step:
    dt = 0.0001
    # Parameters:
    params = {'G': G, 'masses': masses}
    # Output:
    return x, t0, tf, dt, params
```

Alice: Okay, I see. Later, we can replace the contents of this function if we need to. That makes sense. Dare I say? That's clever! Since we are encapsulating the code into functions, why don't we also create a function to do the plotting while we're at it?

Bob: You are starting to sound like a plotting enthusiast. It will be nice to reduce the problem to calling just three functions, though. All right, let's create the function:

Snippet 4.4
Function `plot_trajectory`: Plot the two-body orbits.

```
def plot_trajectory( sol_state ):
    fig = plt.figure()
    ax = fig.add_subplot(111, aspect='equal')
    ax.plot(sol_state[:,0], sol_state[:,1],"b-")
    ax.plot(sol_state[:,0 + 3], sol_state[:,1 + 3],"g-")
    plt.show()
```

Alice: The entire propagator is just three lines of code!

Bob: Wait, you need to import two PYTHON modules! We are using `numpy` for math and `matplotlib` for plotting. This is our final program:

```
import numpy as np
import matplotlib.pyplot as plt

# The initial conditions:
x, t0, tf, dt, params = initialize_problem()

# Integrate orbits
sol_time, sol_state = integrate_euler(x, t0, tf, dt, params)

# Plot trajectories
plot_trajectory(sol_state)
```

4.2 Speeding Up the Code: Allocating Arrays

Alice: Great. Our propagator now decomposes into three very clear steps. In fact, we can complicate the problem and the model as much as we want, but we should still keep this structure. Even when we advance to the actual N-body problem. The contents of each function will change, of course, but I don't see the flow of the program changing much. Since we know how the state of a given particle evolves over time, we can use what we learned about conservation laws to check our propagator.

Bob: Right, I almost forgot about those. I was really focused on the code itself. In fact, can we take a closer look at the performance before we start looking into the conservation laws? I think I can make the code faster.

Alice: What do you have in mind?

Bob: Well, let me try one thing. When we store the solution, we are appending the state vector at each step. One thing that I remember from my programming classes is that how arrays are accessed and stored can have a big impact on the speed of the code. Preallocating the arrays might be faster than appending them.

Alice: *Preallocating*? That sounds fancy…. What is that?

Bob: When you create an array, you need to store the components in the computer's memory. Think of it as having one "cell" for each element that you are going to store in the array. If we keep appending elements to the array, the computer will need to find new cells for them at every step. Conversely, if we instead allocate memory for the array right at the beginning, then we are telling the computer to reserve space for the elements of the array. Then, we just have to fill those spaces with the new elements at each time step. The computer will know exactly where to put them. That will be faster than having the computer looking for the right place to store new elements.

Alice: Okay, so what does all this involve in terms of editing the code?

Bob: I just have to change the function `integrate_euler`, and we can try it out. The interface and first few lines will be the same:

```
def integrate_euler( x0, t0, tf, dt, params ):
    # Number of points in the dense output:
    npts = int(np.floor((tf - t0) / dt)) + 1
    # Initial state:
    x = x0
    # Vector of times:
    sol_time  = np.linspace(t0, t0 + dt * (npts - 1), npts)
```

Then, we can allocate the memory that the array containing the solution will need.

Alice: Do we know the size of the array?

Bob: Yes, since we are storing the state vector $\mathbf{x}$ in consecutive rows, we will need the same number of columns as there are elements in $\mathbf{x}$. This will ensure that the solution fits the array. We can compute the number of elements using `len(x0)`. Then, we need `npts` rows:

```
    sol_state = np.zeros((npts,len(x0)))
```

We can now store the initial state

```
    sol_state[0,:] = x
```

We will need a counter to perform the advancing, `count`. This points to the right row of the array where we want to store the state vectors. PYTHON provides a very convenient function for doing this: `enumerate`.

```
# Launch integration:
for count, t in enumerate(sol_time[1:], 1):
    # Evaluate right-hand side:
    dxdt = ode_two_body_first_order(x, t, params)
    # Advance step:
    x = x + dxdt * dt
    # Store solution:
    sol_state[count,:] = x
# Output
return sol_time, sol_state
```

Alice: You chose to store the solution as rows instead of columns. Is there a reason for that? Could we use columns instead of rows? Vectors are typically represented as columns, and it's easier for my brain to visualize things if presented in the same form as the math is usually written.

Bob: Sure. In principle, we could store them as columns instead. But each programming language has a preferred way of storing data in arrays. It is called row- and column-major order. Keep in mind that an array is nothing more than a series of numbers that are ultimately stored as a sequence. For example, C/C++ and MATHEMATICA are row-major because they store one row after the other. Consequently, it is more efficient to store and access the data in each row first. FORTRAN and MATLAB, on the other hand, are column-major, because they store the first column and then the second, the third, and so on. It's a matter of how each programming language goes through the elements in an array.

Alice: So, in this case, it means that PYTHON uses … row-major order?

Bob: Correct. Well, PYTHON itself does not support arrays per se; that's why we use `numpy`. So it is `numpy` that uses row-major order. In the code, we are slicing the array by rows using the counter `count`, and we are making each slice equal to `x`. This is made apparent by looking at the dimensions of `sol_state`, which has `npts` rows and `len(x0)` columns.

Alice: Okay, I see it now. As exciting as I find the topic of memory access, let's move on. How do we evaluate the speed or performance of the code?

Bob: It is called *profiling*; most programming languages provide standard tools for evaluating the performance of the code. They are especially useful for finding bottlenecks in the code, so you can easily speed up the most critical parts of the code. I checked the PYTHON documentation. This led me to the `timeit` module, which contains the function `default_timer`. If we replace the command

```
start = timeit.default_timer()
```

right at the beginning of the code and

```
stop = timeit.default_timer()
```

at the end, the command

```
print("Runtime:", stop - start, "s")
```

will tell us the runtime, in seconds.

Alice: Wait…. Will we need to do this in both effective versions of the code, the one that appends and the one that allocates?

Bob: Yes. Let's start with the first version of the code, in which we are not allocating the arrays. We will set the initial conditions to $\mathbf{R}_1 = [1,0,0]$, $\mathbf{R}_2 = [-1,0,0]$, $\mathbf{V}_1 = [0,1/2,0]$, and $\mathbf{V}_1 = [0,-1/2,0]$, $G = 1$, $m_1 = m_2 = 1$ with the final time set to $t_f = 15$. Using a time step $\Delta t = 10^{-4}$, the runtime for this example case is

```
Runtime: 232.162275076 s
```

If we run the new version of the code, allocating arrays, we get instead

```
Runtime: 4.08495998383 s
```

Alice: Wow, that is a *big* improvement!

Bob: More than fifty times faster. Read it and weep.

Alice: We have a winner. We should definitely allocate arrays before using them. Let me run the code again, just to be sure. For the first code I get

```
Runtime: 232.455005884 s
```

and for the improved version I get

```
Runtime: 4.12986111641 s
```

The times have changed! How can the runtime change if the code is exactly the same?

Bob: There are a few reasons for this. First of all, the computer is running many other programs in the background. Each time we run the code, the computer might have different programs running. This makes the calculation faster or slower, depending on what is going on behind the curtains and the available resources. Second, if we were using a different computer, then the times would likely be much more different.

Alice: Hmmm, this reminds me of taking measurements in the lab. The reality of collecting experimental data is that you suffer from different sources of error. The way we deal with this in practice is to perform several measurements and then compute an average value. This provides a sort of sanity check that the result you get is really representative of the typical case. Let's try it. If we run the code ten times and compute the average value, we get:

```
Runtime: 232.339545817 s
Runtime: 4.3131291151 s
```

Bob: I like it. We can use this technique later to test the next versions of the propagator.

Alice: Are you happy with how the code looks for now?

Bob: Yep!

4.3 Checking the Conservation Laws

Alice: The first thing we should check is energy conservation.

Bob: I knew that one!

Alice: I'm getting more and more used to writing functions for doing specific things. Let me write a function to calculate the energy.

Bob: It's all yours.

Alice: First things first. We will need to compute the total energy

$$E = \frac{m_1}{2}(\mathbf{V}_1 \cdot \mathbf{V}_1) + \frac{m_2}{2}(\mathbf{V}_2 \cdot \mathbf{V}_2) - G\frac{m_1 m_2}{r}, \qquad \text{with} \qquad r = \|\mathbf{R}_1 - \mathbf{R}_2\|, \tag{4.4}$$

at each time step, and store the value of the energy in an array.

Bob: Don't forget to allocate the array!

Alice: I know, I know. In order to initialize the array, I'll just set it equal to one of the columns of the array containing the solution. This way we will get the right dimension. Is that okay?

Bob: Sounds good.

Alice: Okay, so, the function will be something like

Snippet 4.5
Function `compute_energy`: Compute the two-body energy.

```
def compute_energy( sol_state, params ):
    # Retrieve the parameters:
    G = params['G']
    masses = params['masses']
    # Allocate:
    energy = sol_state[:,0] * 0
    # Run loop:
    for count, x in enumerate(sol_state):
        # Compute energy:
        energy[count] =  0.5 * masses[0] * np.linalg.norm(x[6:9])**2 \
                       + 0.5 * masses[1] * np.linalg.norm(x[9:12])**2 \
                       - G * masses[0] * masses[1] / np.linalg.norm(x[0:3]
                            - x[3:6])
    return energy
```

Now we just call the function,

```
energy = compute_energy(sol_state, params)
```

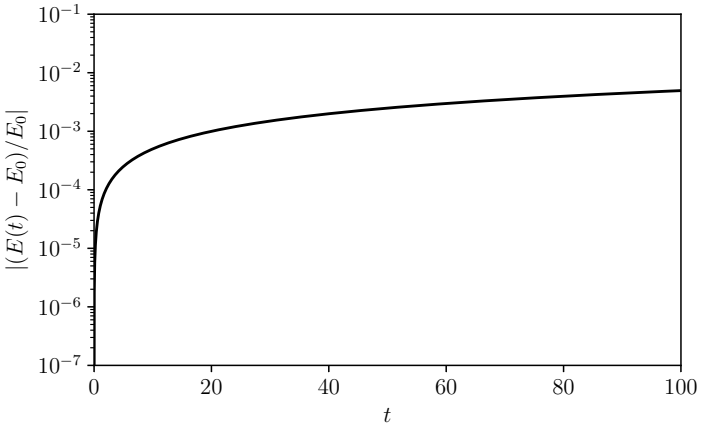


Figure 4.1
Time evolution of the total energy in the two-body problem.

Bob: Let's take a closer look, and plot how the energy evolves in time. Or better yet, how much the energy changes with respect to its initial value. This way we can see whether or not the changes are relevant. Let's define the error as

$$\varepsilon(t) = \left| \frac{E(t) - E_0}{E_0} \right|. \tag{4.5}$$

The figure will be created by:

```
plt.figure()
plt.semilogy(sol_time, np.abs((energy - energy[0]) / energy[0]))
plt.xlabel('Time')
plt.ylabel('|(E(t)-E0)/E0|')
plt.show()
```

Figure 4.1 shows the change in the total energy if we integrate the orbit up to a final time $t_f = 100$, with $\Delta t = 10^{-4}$.

Alice: Hmmmm, that does not look constant to me. And the difference seems to grow in time. What gives?

Bob: Maybe the time span was too long?

Alice: Well, the orbit is circular, so we can easily find its period:

$$P = 2\pi \sqrt{\frac{r^3}{G(m_1 + m_2)}} = 4\pi \approx 12.6. \tag{4.6}$$

For our total integration time, this means that we are propagating for approximately eight revolutions. Ugh. That is not a long propagation!

Bob: This doesn't look good….

Alice: What about the angular momentum? Is it conserved?

Bob: How do we calculate it?

Alice: The angular momentum vector can be written explicitly like this:

$$\mathbf{L} = m_1\mathbf{R}_1 \times \mathbf{V}_1 + m_2\mathbf{R}_2 \times \mathbf{V}_2 = \begin{bmatrix} m_1(y_1\dot{z}_1 - z_1\dot{y}_1) + m_2(y_2\dot{z}_2 - z_2\dot{y}_2) \\ m_1(z_1\dot{x}_1 - x_1\dot{z}_1) + m_2(z_2\dot{x}_2 - x_2\dot{z}_2) \\ m_1(x_1\dot{y}_1 - y_1\dot{x}_1) + m_2(x_2\dot{y}_2 - y_2\dot{x}_2) \end{bmatrix}. \tag{4.7}$$

We can check how much its magnitude changes using a function similar to the one we wrote for the energy.

Bob: Wait a minute…. Instead of writing the angular momentum in terms of the coordinates of the position and velocity vectors, why don't we check if `numpy` supports the cross product of two vectors?

Alice: Sounds good to me.

Bob: Let me see…. Yes, `np.cross` will do the work! We'll need to take the norm of the angular momentum vector. I checked the documentation and we can use `np.linalg.norm` for that.

Alice: Outstanding! This will simplify the function. With this, we just need to write the code in snippet 4.6.

Snippet 4.6
Function `compute_angular_momentum`: Compute the magnitude of the two-body angular momentum.

```
def compute_angular_momentum( sol_state, params ):
    # Retrieve the parameters:
    masses = params['masses']
    # Initialize:
    angmomentum = sol_state[:,0] * 0
    # Run loop:
    for count, x in enumerate(sol_state):
        # Compute energy:
        angmomentum[count] = np.linalg.norm(masses[0] * np.cross(x[0:3], x
            [6:9]) + masses[1] * np.cross(x[3:6], x[9:12]))
    # Output
    return angmomentum
```

If we propagate the same orbit and monitor the change in the total angular momentum, we obtain figure 4.2.

Bob: Uh oh…. It's not constant either….

Alice: Something must be going wrong.

Bob: Yeah, the integration doesn't look right.

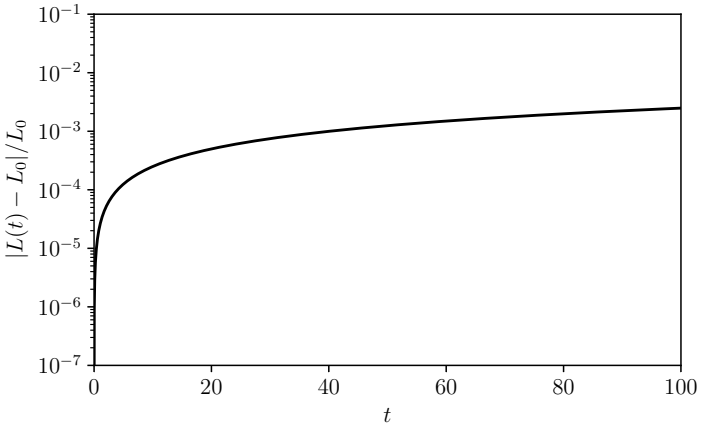


Figure 4.2
Time evolution of the total angular momentum in the two-body problem.

Alice: Maybe it's just an issue with the time step?

Bob: Oh yeah, remember all the problems we had before? We can repeat the experiments with two different time steps. For example, ten times larger and ten times smaller, $\Delta t = 10^{-3}$ and $\Delta t = 10^{-5}$.

Alice: Intuitively, I expect the angular momentum conservation to behave better using a smaller time step.

Bob: Yes, if decreasing the time step doesn't improve the results, then something must be wrong in the code.

Alice: Okay, I plotted the results in figure 4.3. As expected, the conservation of the angular momentum is directly related to the time step. Smaller time steps yield smaller changes in $\|\mathbf{L}\|$.

Bob: That's a relief. This means that the problem was related to our choice for the step size, in conjunction with the integration scheme we are using.

Alice: Let's do one final check for the orbit of body 1 with respect to body 2. We learned that the eccentricity should be constant too. In fact, the orbit in our example is circular, so the eccentricity should be zero at all times.

Bob: I doubt it, frankly.

Alice: Well, we will see. The eccentricity vector of the relative orbit of body 1 with respect to body 2 reads

$$\mathbf{e}_1 = \frac{\mathbf{v}_1 \times (\mathbf{r}_1 \times \mathbf{v}_1)}{GM} - \frac{\mathbf{r}_1}{r}, \tag{4.8}$$

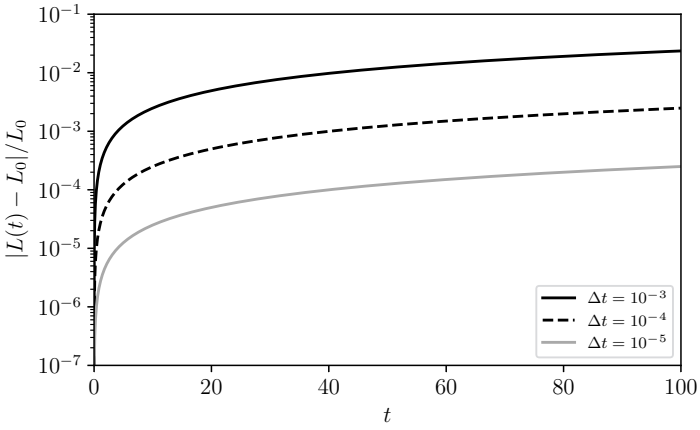


Figure 4.3
Conservation of the total angular momentum using different time steps.

with $\mathbf{r}_1 = \mathbf{R}_1 - \mathbf{R}_2$, $\mathbf{v}_1 = \mathbf{V}_1 - \mathbf{V}_2$, and $M = m_1 + m_2$. We should compute this vector at each step and store its magnitude. The following function will do it:

Snippet 4.7
Function `compute_eccentricity`: Compute the eccentricity of body 1 with respect to body 2 in the two-body problem.

```
def compute_eccentricity( sol_state, params ):
    # Retrieve the parameters:
    G = params['G']
    masses = params['masses']
    eccentricity = sol_state[:,0] * 0
    # Total mass:
    M = masses[0] + masses[1]
    for count, x in enumerate(sol_state):
        # Compute eccentricity:
        r1 = x[0:3] - x[3:6]
        v1 = x[6:9] - x[9:12]
        vecc = np.cross(v1, np.cross(r1, v1)) / (G * M) - r1 / np.linalg.
            norm(r1)
        eccentricity[count] = np.linalg.norm(vecc)
    # Output:
    return eccentricity
```

Figure 4.4 shows the evolution of the eccentricity using the three time steps that we used before.

Bob: I am not surprised to see that the eccentricity is not zero, but now we see an interesting pattern. It seems to be oscillating.

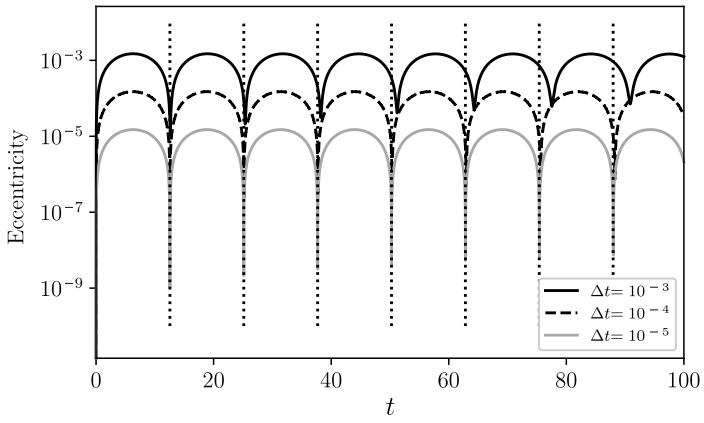


Figure 4.4
Time evolution of the eccentricity compared to the orbital period. Note the clearly periodic oscillations.

Alice: Wait a minute…. We are propagating approximately eight revolutions, and we have eight oscillations. Let me plot a vertical line after each complete revolution. The separation between these lines should be equal to 4π:

```
for k in range(1,9):
    plt.plot([4 * np.pi * k, 4 * np.pi * k], [1e-10, 1e-2], "k:")
```

Bob: The lines capture the oscillation period perfectly! This means that the error in the eccentricity is at its maximum when the phase of the binary orbit is exactly opposing the initial point of integration. Then, the eccentricity error reapproaches zero when the integration reaches the initial starting point again. In fact, for the larger step size, the period of the orbit starts to drift too. Look at how the last orbits are starting to become shifted relative to the first few orbits.

Alice: It is nice to connect numerical errors to physical properties! But we really need to understand this business about the errors in more detail. I think it is time to ask Prof. Starmover about other integration schemes.

Bob: Yeah, I agree. Let's see if we can find her.

5 Fixed Step-Size Integration

Overview. Our protagonists have come to the conclusion that the forward Euler integration scheme is not accurate enough for integrating complicated dynamical systems. Prof. Starmover provides them with an introduction to more advanced fixed-step integration algorithms, including the Runge–Kutta, Adams–Bashforth, and leapfrog methods. She pays special attention to explaining the core differences between single- and multistep methods, as well as between explicit and implicit methods. She also presents symplectic integrators using an example and featuring its main benefits. Alice and Bob compare the performance of these schemes monitoring their accuracy and speed.

Bob and Alice head over to Prof. Starmover's office to see if she is available. In luck, Prof. Starmover is sitting at her desk, reading quietly.

5.1 Truncation and Round-Off Errors

Alice: Good morning, Prof. Starmover.

Bob: How are you, Prof. Starmover?

Professor: Hello, Alice, Bob. How is the propagator going?

Alice: We've propagated some simple test cases and checked the energy conservation, the angular momentum conservation, and the time evolution of the orbital eccentricity. To our surprise, we found that none of them are constant in practice. They grow over time!

Bob: And their evolution depends on the time step that we use.

Alice: Right! That reminds me: I have a question about the step size. Would an infinitely small time step solve all of our issues? Could we set $\Delta t = 10^{-30}$ and get negligible errors?

Professor: That is a fair question. The short answer is no. In general, you will not be able to get rid of errors by making the step size infinitely small. The long answer has two parts but explains the general idea. First, there is the mechanism via which computers store numbers. This is an important component. Second, there is the issue of the time step and how its size affects the accuracy of the integration.

Let's start with the first part, namely, how computers store numbers. Computers work with what is called floating-point arithmetic. In order to store a number, the computer reserves a number of bytes in the RAM memory to store a certain number of significant digits, the sign, and an exponent with a given base. Think of this as using scientific notation but in binary format. The key concept here is that the number of significant digits that the computer stores is limited. Hence, the floating-point representation is an approximation of the actual number. For example, if you try to store the number π with twenty decimal figures in PYTHON,

```
>>> pi = 3.14159265358979323846
```

it turns out that the computer will not store all the provided information. Instead, the number that the computer stores is

```
>>> pi = 3.141592653589793
```

Bob: We lost five significant digits!

Professor: Don't panic. But that's right. PYTHON, like MATLAB and most programming languages, stores numbers in double-precision floating-point format. This standard uses sixty-four bits to store a number, which yields between fifteen and seventeen significant digits. Programming languages in which variables need to be declared typically support both single and double precision. Single-precision numbers contain roughly eight significant digits.

The fact that computers store approximations of the actual numbers leads to *round-off* errors. This is because the precision is reduced, and hence some information is lost, when reducing the number of significant digits. For example, let's say that we define a second variable, changing the last few digits of π:

```
>>> pi2 = 3.14159265358979377777
```

This variable is clearly different from the actual value `pi`, right? If we subtract these two numbers, what do you think we should get?

Alice: The difference is 0.00000000000000053931, or 5.3931×10^{-16}.

Professor: Let's check

```
>>> pi2 - pi
4.440892098500626e-16
```

Bob: Whoa. That's completely wrong!

Alice: Bob is right! I'm sure my solution is correct!

Professor: You are indeed correct. Again, don't panic! This is precisely the kind of error I was talking about, due to round-off. The computer only stores an approximation of the two numbers, with a limited precision. If the numbers are too similar, the computer might

not be able to notice any difference. There is a limit on how similar two numbers can be for the computer to be able to distinguish them. This is given by the *machine zero* ϵ, which defines the smallest representable positive number such that

$$1.0 + \epsilon \neq 1.0. \tag{5.1}$$

For example, in `numpy`, it is

```
>>> np.finfo(float).eps
2.2204460492503131e-16
```

Returning to your question, Alice, what do you think now about using 10^{-30} as your chosen time step?

Alice: It will be much smaller than the machine zero, so the computer will not understand. It won't be able to advance from t to $t + \Delta t$, for starters.

Professor: Correct! In fact, all you will get is numerical noise, because PYTHON does not have enough precision to calculate such small differences. Not to mention, using a very small step size will require a lot of operations to reach the final time, and the round-off error will accumulate or compound.

Alice: So smaller time steps do not reduce the round-off error?

Professor: On the contrary, they will potentially increase the error due to round-off and the accumulation problem. The more operations you perform, the more important is the fact that you are losing precision when you store each number.

The second part of the answer to your question has to do with the second source of error in numerical integrations, the *truncation error*. Keep in mind that integration schemes used for solving differential equations are approximations of the solution. For example, let's recover the first-order system that you wrote in equation (4.1). The solution at time t is $\mathbf{x}(t)$. If we advance the time to $t + \Delta t$, the solution will be $\mathbf{x}(t + \Delta t)$. If we assume $\Delta t \ll t$, we can write the solution in terms of the series expansion

$$\mathbf{x}(t + \Delta t) \approx \mathbf{x}(t) + \frac{\mathrm{d}\mathbf{x}}{\mathrm{d}t}\Delta t + \frac{1}{2}\frac{\mathrm{d}^2\mathbf{x}}{\mathrm{d}t^2}\Delta t^2 + \ldots \tag{5.2}$$

If we retain only the first-order term, we simplify the solution to

$$\mathbf{x}(t + \Delta t) \approx \mathbf{x}(t) + \frac{\mathrm{d}\mathbf{x}}{\mathrm{d}t}\Delta t. \tag{5.3}$$

Bob: And that is called the forward Euler method of integration, right?

Professor: Yes. By truncating the series to first order, we simplify the algorithm, but we are losing accuracy. This is precisely the truncation error: the error in the solution due to neglecting high-order terms from the expansion. The Euler scheme is a first-order method, which means that the error is of the order $\mathcal{O}(\Delta t^2)$. In other words, if we subtract equation (5.3) from equation (5.2), we find that the truncation error ε_t for Euler's method

is

$$\varepsilon_t = \frac{1}{2}\frac{\mathrm{d}^2\mathbf{x}}{\mathrm{d}t^2}\Delta t^2 + \ldots \tag{5.4}$$

The dominant term is of order Δt^2.

Alice: In this case, reducing the step size, Δt, does reduce the truncation error, doesn't it?

Professor: Yes, smaller step sizes yield smaller truncation errors. But larger round-off errors.

Bob: A first-order solution seems too simple. I guess we will need to retain more terms in equation (5.2). Can we do that?

Professor: That gives you the *order* of your integration method. If an integration method has order p, then the truncation error will be $\mathcal{O}(\Delta t^{p+1})$. The Euler scheme has order one, and therefore the truncation error is of order $\mathcal{O}(\Delta t^2)$. The accumulation of the truncation error as the integration advances yields the global truncation error of the method. High-order methods have smaller truncation errors but will need to perform more operations. This will make your program slower, and round-off errors will accumulate more rapidly.

Alice: So what is the best solution?

Professor: The best solution depends on the exact problem you want to solve. There are many options. You can start with a Runge–Kutta and an Adams–Bashforth scheme to become familiar with more advanced conventional integrators. Then, we can take a look at the leapfrog integrator. It is fairly simple, and it will be very useful for your project.

5.2 Runge–Kutta Methods

5.2.1 Explicit Methods

Alice: What are the differences between the various methods of integration?

Professor: The idea is always the same: to approximate the solution at every time step. But there are many ways to do it. Let's start with the Runge–Kutta methods.

Bob: Is there more than one Runge–Kutta method?

Professor: Oh yes, there are entire families of Runge–Kutta methods. Actually, the forward Euler method that you have been working with belongs to this family. I recommend that you take a look at the first book of the series by Hairer et al. (1991), in particular part II. The Euler method works as follows. First, we update the state using a linear approach:

$$\mathbf{x}(t+\Delta t) = \mathbf{x}(t) + \mathbf{f}(t,\mathbf{x})\Delta t. \tag{5.5}$$

If we call the current time t_n, and $\mathbf{x}(t_n) \equiv \mathbf{x}_n$ is the current solution corresponding to that time, we can write the next step as $\mathbf{x}(t_n + \Delta t) \equiv \mathbf{x}_{n+1}$. The previous equation takes the form

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \mathbf{f}(t_n,\mathbf{x}_n)\Delta t. \tag{5.6}$$

The value of $\mathbf{f}(t, \mathbf{x})$ is computed at the beginning of the interval Δt, and it does not change until we reach $t + \Delta t$. This is technically an incorrect assumption because $\mathbf{f}(t, \mathbf{x})$ does change in time. Hence, it is only an approximation of the true solution. Intuitively, if you had to choose only one point to represent an interval, which point would you choose?

Alice: I guess I would choose the center of the interval, the midpoint.

Professor: Exactly. Instead of using the value at the beginning of the interval, we can use its value at the midpoint of the interval:

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \mathbf{f}\left(t_n + \frac{\Delta t}{2}, \mathbf{x}\left(t_n + \frac{\Delta t}{2}\right)\right)\Delta t. \tag{5.7}$$

The solution at the midpoint can be approximated as

$$\mathbf{x}\left(t_n + \frac{\Delta t}{2}\right) \approx \mathbf{x}_n + \mathbf{f}(t_n, \mathbf{x}_n)\frac{\Delta t}{2} = \mathbf{x}_n + k_1\frac{\Delta t}{2}. \tag{5.8}$$

Under this notation, equation (5.7) transforms into

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \mathbf{f}(t_n + \Delta t/2, \mathbf{x}_n + k_1\Delta t/2)\Delta t = \mathbf{x}_n + k_2\Delta t. \tag{5.9}$$

This simple methodology defines a Runge–Kutta method of order two:

$$k_1 = \mathbf{f}(t_n, \mathbf{x}_n); \tag{5.10a}$$

$$k_2 = \mathbf{f}(t_n + \Delta t/2, \mathbf{x}_n + k_1\Delta t/2); \tag{5.10b}$$

$$\mathbf{x}_{n+1} = \mathbf{x}_n + k_2\Delta t + \mathcal{O}(\Delta t^3). \tag{5.10c}$$

The cornerstone of the method is the use of a better approximation for $\mathbf{f}(t, \mathbf{x})$ along the interval Δt. Instead of assuming it to be constant and equal to its initial value, we sample the function at the center of the interval.

This technique can be generalized in order to improve its accuracy. For example, the fourth-order Runge–Kutta method (RK4), often referred to as the classical Runge–Kutta method, discretizes the interval Δt in four stages. This yields:

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \frac{\Delta t}{6}(k_1 + 2k_2 + 2k_3 + k_4), \tag{5.11}$$

with

$$k_1 = \mathbf{f}(t_n, \mathbf{x}_n); \tag{5.12a}$$

$$k_2 = \mathbf{f}(t_n + \Delta t/2, \mathbf{x}_n + k_1\Delta t/2); \tag{5.12b}$$

$$k_3 = \mathbf{f}(t_n + \Delta t/2, \mathbf{x}_n + k_2\Delta t/2); \tag{5.12c}$$

$$k_4 = \mathbf{f}(t_n + \Delta t, \mathbf{x}_n + k_3\Delta t). \tag{5.12d}$$

The local truncation error of this method is $\mathcal{O}(\Delta t^5)$. I think you should implement this method in your propagator, as an example of a higher-order integrator.

Table 5.1
Butcher tableau for explicit Runge–Kutta methods.

0					
c_2	a_{21}				
c_3	a_{31}	a_{32}			
$\vdots$	$\vdots$		$\ddots$		
c_s	a_{s1}	a_{s2}	$\dots$	$a_{s,s-1}$	
	b_1	b_2	$\dots$	b_{s-1}	b_s

Bob: How do you build other Runge–Kutta methods?

Alice: In general, a Runge–Kutta method with s stages provides a solution of the form

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \Delta t \sum_{i=1}^{s} b_i k_i, \tag{5.13}$$

where

$$k_1 = \mathbf{f}(t_n, \mathbf{x}_n); \tag{5.14a}$$

$$k_2 = \mathbf{f}(t_n + c_2\Delta t, \mathbf{x}_n + a_{21}k_1\Delta t); \tag{5.14b}$$

$$k_3 = \mathbf{f}(t_n + c_3\Delta t, \mathbf{x}_n + (a_{31}k_1 + a_{32}k_2)\Delta t); \tag{5.14c}$$

$$\dots$$

$$k_s = \mathbf{f}(t_n + c_s\Delta t, \mathbf{x}_n + \Delta t \sum_{j=1}^{s-1} a_{sj}k_j). \tag{5.14d}$$

Different values of the Runge–Kutta matrix a_{ij}, the weights b_i, and the nodes c_i yield different Runge–Kutta methods. The coefficients are usually given in a Butcher tableau (Butcher, 1964), like the one in table 5.1.

Alice: Does this mean that I can now create my own Runge–Kutta method by simply choosing the values of the coefficients at will?

Professor: I'm afraid it doesn't work that way. If you want your s-stage Runge–Kutta method to have order p (remember that the order is related to the local truncation error), the coefficients need to satisfy certain conditions. Not to mention the stability and convergence of the method.

Alice: Too bad … let's focus on RK4 then. The first part of the function that carries out the integration should be just like the one we have for Euler's method:

Snippet 5.1
Function `integrate_runge_kutta`: Runge–Kutta integrator.

```
def integrate_runge_kutta( x0, t0, tf, dt, params ):
    # Allocate dense output:
    npts = int(np.floor((tf - t0) / dt)) + 1
    # Initial state:
    x = x0
    # Vector of times:
    sol_time = np.linspace(t0, t0 + dt * (npts - 1), npts)
    # Allocate and store initial steps:
    sol_state = np.zeros((npts, len(x0)))
    sol_state[0,:] = x
```

Bob: Yes, we should keep the same interface and allocate the solution.

Professor: For the actual integration, you just need to compute the coefficients k_i and then evaluate the solution.

Bob: We can do it easily thanks to our function `ode_two_body_first_order`:

```
# Launch integration:
for count, t in enumerate(sol_time[1:], 1):
    # Evaluate coefficients:
    k1 = ode_two_body_first_order(x, t, params)
    k2 = ode_two_body_first_order(x + 0.5 * dt * k1, t + 0.5 * dt, params)
    k3 = ode_two_body_first_order(x + 0.5 * dt * k2, t + 0.5 * dt, params)
    k4 = ode_two_body_first_order(x + dt * k3, t + dt, params)
    # Advance the state:
    x = x + dt * (k1 + 2 * k2 + 2 * k3 + k4) / 6.0
    # Store step:
    sol_state[count,:] = x
# Output:
return sol_time, sol_state
```

Alice: Great! We have a new integrator to try.

5.2.2 Implicit Methods

Professor: The Runge–Kutta methods that I described are called *explicit*, because the coefficients k_i can be obtained explicitly and sequentially from the values of the previous coefficients. The *implicit* methods, on the other hand, use the coefficients

$$k_s = \mathbf{f}(t_n + c_s \Delta t, \mathbf{x}_n + \Delta t \sum_{j=1}^{s} a_{sj} k_j). \tag{5.15}$$

The sum extends up to $j = s$, meaning that k_s appears both inside the function on the right-hand side and on the left-hand side. This construction defines a system of algebraic equations that needs to be solved for the coefficients k_i. The solution is typically obtained with predictor-corrector algorithms.

Bob: That sounds complicated. Why would anyone use an implicit method instead of an explicit one?

Professor: Implicit methods are more stable than explicit methods. They tend to converge to the actual solution in more cases.

Alice: What do you mean by "converge"?

Professor: I mean that the method is actually able to successfully represent the solution to the differential equations. Let's look at a counterexample to illustrate the basic idea of convergence. In this case, we will see that the method fails to converge. Consider the simple differential equation,

$$\frac{\mathrm{d}x}{\mathrm{d}t} = -20x = f(t,x). \tag{5.16}$$

The solution is

$$x(t) = x_0 \mathrm{e}^{-20t}. \tag{5.17}$$

It goes to zero as time advances, that is, $\lim_{t\to\infty} x(t) = 0$. Let's see what happens if we integrate this equation with the forward Euler method starting from $x_0 = 1$ and with step size $\Delta t = 0.1$.

Bob: Can I do it?

Professor: Of course. Go ahead!

Bob: Okay. If we take one step, we obtain

$$x_1 = x_0 + f(t,x_0)\Delta t = 1 - 2 = -1. \tag{5.18}$$

The second step takes us to

$$x_2 = x_1 + f(t,x_1)\Delta t = -1 + 2 = 1. \tag{5.19}$$

Well, this is strange … let's keep going:

$$x_3 = x_2 + f(t,x_2)\Delta t = 1 - 2 = -1. \tag{5.20}$$

Seriously?

$$x_4 = x_3 + f(t,x_3)\Delta t = -1 + 2 = 1. \tag{5.21}$$

I give up. The solution is oscillating, and it will never converge to zero.

Professor: That's right! This is an example of a problem where the method fails to converge. Implicit methods have a better convergence rate and are particularly well suited for solving partial differential equations. But you have to solve a system of algebraic equations. Don't worry too much about implicit methods right now; let's stick to the RK4 method.

5.3 Multistep Methods

Professor: Runge–Kutta methods take the information from the previous step and then advance to the next one sequentially. That's why they are called single-step methods. One step at a time.

Alice: Is there any other way to do it?

Professor: Yes, indeed. There are also multistep methods. They approximate the solution at each step using information from more than one previous step.

Let's construct a two-step method to see how multistep methods work. When deriving Euler's method, we started from the general solution

$$\mathbf{x}(t+\Delta t) = \mathbf{x}(t) + \int_t^{t+\Delta t} \mathbf{f}(\tau, \mathbf{x})\, \mathrm{d}\tau \tag{5.22}$$

and, assuming $\mathbf{f}(t, \mathbf{x})$ remains constant between steps, we obtained the approximation

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \mathbf{f}(t_n, \mathbf{x}_n)\Delta t. \tag{5.23}$$

Instead of assuming that $\mathbf{f}(t, \mathbf{x})$ stays constant, we will approximate the function using an interpolating polynomial. This is the key concept of multistep methods: the construction of an interpolating function using information from several previous steps.

In order to generate a two-step method to approximate $\mathbf{x}_{n+1}$, we need the values on the right-hand side $\mathbf{f}$ at two points: t_n and t_{n-1}. We seek a representation of $\mathbf{f}(t, \mathbf{x})$ in the form of a polynomial $\mathbf{P}(t)$. The polynomial can be obtained from the standard interpolation formula:

$$\mathbf{P}(t) = \frac{t - t_{n-1}}{t_n - t_{n-1}}\mathbf{f}(t_n, \mathbf{x}_n) + \frac{t - t_n}{t_{n-1} - t_n}\mathbf{f}(t_{n-1}, \mathbf{x}_{n-1}) = \mathbf{a}_1 + \mathbf{a}_2 t, \tag{5.24}$$

with

$$\mathbf{a}_1 = \frac{t_n \mathbf{f}(t_{n-1}, \mathbf{x}_{n-1}) - t_{n-1}\mathbf{f}(t_n, \mathbf{x}_n)}{t_n - t_{n-1}}, \qquad \mathbf{a}_2 = \frac{\mathbf{f}(t_n, \mathbf{x}_n) - \mathbf{f}(t_{n-1}, \mathbf{x}_{n-1})}{t_n - t_{n-1}}. \tag{5.25}$$

We can now replace $\mathbf{f}(t, \mathbf{x})$ in equation (5.22) with the polynomial approximation in order to compute the time integral,

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \int_{t_n}^{t_{n+1}} \mathbf{P}(\tau)\, \mathrm{d}\tau = \mathbf{x}_n + \mathbf{a}_1(t_{n+1} - t_n) + \frac{1}{2}\mathbf{a}_2(t_{n+1}^2 - t_n^2). \tag{5.26}$$

Alice: We are still assuming that the time step is constant, right? So isn't it true that $t_{n+1} - t_n = t_n - t_{n-1} = \Delta t$?

Professor: Correct, I was keeping things a bit more general. But if we now introduce that simplification, we arrive at

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \mathbf{a}_1 \Delta t + \frac{1}{2}\mathbf{a}_2 \Delta t(2t_n + \Delta t) = \mathbf{x}_n + \frac{\Delta t}{2}[3\mathbf{f}(t_n, \mathbf{x}_n) - \mathbf{f}(t_{n-1}, \mathbf{x}_{n-1})]. \tag{5.27}$$

This solution is the two-step Adams–Bashforth method, or AB2.

Bob: It seems pretty simple to implement! We should try it.

Alice: I like it too. But wait, the t_{n-1} term is confusing…. What happens with the first step? We would get

$$\mathbf{x}_1 = \mathbf{x}_0 + \frac{\Delta t}{2}[3\mathbf{f}(t_0, \mathbf{x}_0) - \mathbf{f}(t_{-1}, \mathbf{x}_{-1})], \tag{5.28}$$

and I don't know if negative indices make sense.

Professor: You have to be careful when initializing multistep methods; you need not only the initial conditions $\mathbf{x}_0$ but also the initial values for all the steps that you will use. You don't know them a priori, so they need to be approximated somehow. The typical approach is to use a single-step method to generate the $\mathbf{x}_i$ required by the multistep method. I recommend that you approximate them using a simple forward Euler scheme and then launch the Adams–Bashforth. But you could also use the RK4 method to initialize the first steps.

Bob: Great, now I'm curious. Are there other multistep methods?

Professor: Of course! As with Runge–Kutta methods, for example, you will find many multistep integrators. It is a matter of changing the interpolating polynomial and accounting for more steps. And there are also implicit multistep methods. But AB2 is a good starting point to become familiar with multistep methods.

Alice: Okay, it's time to implement the Adams–Bashforth method. Again, let's keep the same interface for the function and the initialization steps.

Snippet 5.2
Function `integrate_adams_bashforth`: Adams–Bashforth integrator.

```
def integrate_adams_bashforth( x0, t0, tf, dt, params ):
    # Allocate dense output:
    npts = int(np.floor((tf - t0) / dt)) + 1
    # Initial state:
    x = x0
    # Vector of times:
    sol_time = np.linspace(t0, t0 + dt * (npts - 1), npts)
    # Allocate and store initial steps:
    sol_state = np.zeros((npts, 12))
    sol_state[0,:] = x
```

Now we can write the loop….

Bob: Wait! We still need to initialize $\mathbf{x}_1$, because this is a two-step method.

Alice: Thanks. I can't believe I forgot about it when I was the one concerned about the initialization. In order to take one step using the Euler method, we need to evaluate the right-hand side at t_0:

```
# Compute second step:
dxdt0 = ode_two_body_first_order(x, t0, params)
```

and then propagate to t_1:

```
x = x + dxdt0 * dt
sol_state[1,:] = x
```

Professor: Yes, now you have both steps needed for the initialization.

Bob: Inside the loop, we will need to evaluate the force function twice, because the method requires $\mathbf{f}(t_n, \mathbf{x}_n)$ and $\mathbf{f}(t_{n-1}, \mathbf{x}_{n-1})$.

Professor: Hold on, not necessarily. Note that when you advance one step, from t_{n-1} to t_n, $\mathbf{f}(t_n, \mathbf{x}_n)$ will become $\mathbf{f}(t_{n-1}, \mathbf{x}_{n-1})$, so you don't need to recompute it. You can just reuse it.

Bob: That should make things faster. The first part of the code will then be:

```
# Launch integration:
for count, t in enumerate(sol_time[1:], 1):
    # Evaluate right-hand side:
    dxdt = ode_two_body_first_order(x, t, params)
    # Advance step:
    x = x + 0.5 * dt * (3 * dxdt - dxdt0)
    # Store solution:
    sol_state[count,:] = x
```

Finally, we need to update the solution and to advance our counter.

```
    # Update:
    dxdt0 = dxdt
    x0 = x
# Output:
return sol_time, sol_state
```

And voilà! We have another integration scheme in our propagator!

5.4 Leapfrog Integrator

Professor: Before you start comparing the performance of your new integrators, there is a particularly useful integrator I would like to show you: the leapfrog method.

Alice: Who chose the name?

Professor: Actually, the name is very well chosen and is motivated by how the integrator works. Depending on the field, it is also called the Störmer method or the Verlet method.

One important thing to keep in mind is that the leapfrog method applies to systems of second order, for which the accelerations do not depend on time. We should be able to write the problem as follows:

$$\frac{\mathrm{d}^2\mathbf{y}}{\mathrm{d}t^2} = \mathbf{F}(\mathbf{y}). \tag{5.29}$$

Alice: Well, let's see … if we write $\mathbf{y}(t) = [\mathbf{R}_1(t), \mathbf{R}_2(t)]$, the differential equations of the two-body problem take the form

$$\frac{\mathrm{d}^2\mathbf{y}}{\mathrm{d}t^2} = \begin{bmatrix} -Gm_2\dfrac{\mathbf{R}_1 - \mathbf{R}_2}{r^3} \\ -Gm_1\dfrac{\mathbf{R}_2 - \mathbf{R}_1}{r^3} \end{bmatrix}. \tag{5.30}$$

The time does not appear explicitly, so we should be able to integrate our problem with this new method, right?

Professor: That's right, Alice. In fact, you should implement this method too. It is a very elegant yet simple improvement to Euler's method.

Bob: Wait, I lost you for a second. Why have you changed the notation from $\mathbf{x}(t)$ to $\mathbf{y}(t)$?

Alice: I think it's just to avoid confusion; vector $\mathbf{x}(t)$ contains both the positions and velocities of the bodies, whereas $\mathbf{y}(t) = [\mathbf{R}_1(t), \mathbf{R}_2(t)]$ contains only the position vectors. Right?

Professor: That's right. Euler's method propagates the position vectors as follows:

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \dot{\mathbf{y}}_n \Delta t. \tag{5.31}$$

In this case, we are assuming that the velocities $\dot{\mathbf{y}}$ are constant over the time interval extending from t_n to t_{n+1}, and we are taking their values at the beginning of the interval. With the leapfrog method, as with the Runge–Kutta methods, we use the value of the velocity at the midpoint $t_n + \Delta t/2$ instead,

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \dot{\mathbf{y}}_{n+1/2} \Delta t. \tag{5.32}$$

The velocities are propagated in a similar way, according to

$$\dot{\mathbf{y}}_{n+1/2} = \dot{\mathbf{y}}_{n-1/2} + \mathbf{F}(\mathbf{y}_n)\Delta t. \tag{5.33}$$

Bob: I understand what you are doing, but this is a bit messy. The positions and velocities are given at different times….

Professor: There's still some work to be done to make the method practical. First, we need to compute $\dot{\mathbf{y}}_{1/2}$ from $\dot{\mathbf{y}}_0$. A simple Euler step will suffice,

$$\dot{\mathbf{y}}_{1/2} = \dot{\mathbf{y}}_0 + \mathbf{F}(\mathbf{y}_0)\frac{\Delta t}{2}. \tag{5.34}$$

This suggests decomposing the time interval $[t_n, t_{n+1}]$ into two equal intervals, or

$$\dot{\mathbf{y}}_{n+1/2} = \dot{\mathbf{y}}_n + \mathbf{F}(\mathbf{y}_n)\frac{\Delta t}{2}; \tag{5.35a}$$

$$\dot{\mathbf{y}}_{n+1} = \dot{\mathbf{y}}_{n+1/2} + \mathbf{F}(\mathbf{y}_{n+1})\frac{\Delta t}{2}. \tag{5.35b}$$

Introducing the former expression into equation (5.32) yields the synchronized form of the leapfrog integrator, often called velocity Verlet:

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \dot{\mathbf{y}}_n \Delta t + \mathbf{F}(\mathbf{y}_n)\frac{\Delta t^2}{2}; \tag{5.36a}$$

$$\dot{\mathbf{y}}_{n+1} = \dot{\mathbf{y}}_n + [\mathbf{F}(\mathbf{y}_n) + \mathbf{F}(\mathbf{y}_{n+1})]\frac{\Delta t}{2}. \tag{5.36b}$$

Are you feeling any better about this method, Bob?

Bob: Sure, now it seems to me like we can actually implement it. Do you think we can still use our function `ode_two_body_first_order`, Alice? We changed the notation quite a bit.

Alice: Let's see. In the end, vector $\mathbf{x}(t)$ decomposes into $\mathbf{y}(t)$ and $\dot{\mathbf{y}}(t)$. So the derivative of $\mathbf{x}(t)$, $\mathbf{f}(t, \mathbf{x})$, which is the output of our function, decomposes into $\dot{\mathbf{y}}(t)$ and $\mathbf{F}(\mathbf{y})$. So yes, `ode_two_body_first_order` will provide all the information we need.

Bob: That's great. We don't need new functions! Let's start with the interface and the initialization:

Snippet 5.3
Function `integrate_leapfrog`: Leapfrog integrator.

```
def integrate_leapfrog( x0, t0, tf, dt, params ):
    # Allocate dense output:
    npts = int(np.floor((tf - t0) / dt)) + 1
    # Initial state:
    x = x0
    # Vector of times:
    sol_time = np.linspace(t0, t0 + dt * (npts - 1), npts)
    # Allocate and store initial steps:
    sol_state = np.zeros((npts, len(x0)))
    sol_state[0,:] = x
```

Hmmm, like in the case of the Adams–Bashforth method, we need to evaluate the right-hand side at two different points, t_n and t_{n+1}.

Alice: We can evaluate $\mathbf{f}(t_0, \mathbf{x}_0)$ before starting the loop and then update the value of the function after advancing each step. Just like we did for the multistep method!

Bob: Outstanding! Let's do that.

```
    # Initial acceleration
    dxdt0 = ode_two_body_first_order(x, 0.0, params)
```

Now the actual integration will be carried out by:

```
# Launch integration:
for count, t in enumerate(sol_time[1:], 1):
    # Propagate position vectors:
    x[0:3] = x[0:3] + x[6:9] * dt + 0.5 * dxdt0[6:9] * dt**2
    x[3:6] = x[3:6] + x[9:12] * dt+ 0.5 * dxdt0[9:12] * dt**2
```

```
    # Evaluate accelerations at current point:
    dxdt = ode_two_body_first_order( x, t, params )
    x[6:9] = x[6:9] + 0.5 * (dxdt0[6:9] + dxdt[6:9]) * dt
    x[9:12]= x[9:12]+ 0.5 * (dxdt0[9:12] + dxdt[9:12]) * dt
    # Store solution:
    sol_state[count,:] = x
    # Update:
    dxdt0 = dxdt
return sol_time, sol_state
```

Professor: I see that you are using slices of vectors a lot, instead of defining new variables. That's a good practice, because it saves memory and is more efficient. The only problem is that it makes the code a bit harder to follow.

Alice: Yeah, I had to read it a couple of times to fully understand it. Let's see if I got it right. The position vectors of the two particles are `x[0:3]` and `x[3:6]`, their velocities are `x[6:9]` and `x[9:12]`, and the accelerations are `dxdt[6:9]` and `dxdt[9:12]`. Oh, and the acceleration provided by `dxdt` is $\mathbf{f}(t_{n+1}, \mathbf{x}_{n+1})$, whereas `dxdt0` defines $\mathbf{f}(t_n, \mathbf{x}_n)$. Am I right?

Bob: Yes, that's exactly it. I'm becoming more and more comfortable with vector notation.

5.5 Symplectic Integrators

Professor: Let me tell you why I wanted to explain the leapfrog method to you. The reason is that it is a *symplectic* method. This property is relevant when considering Hamiltonian systems, those that can be described using Hamilton's equations:

$$\frac{\mathrm{d}q_i}{\mathrm{d}t} = \frac{\partial \mathcal{H}}{\partial p_i}, \qquad \frac{\mathrm{d}p_i}{\mathrm{d}t} = -\frac{\partial \mathcal{H}}{\partial q_i}. \tag{5.37}$$

If we combine the position $\mathbf{q}$ and momenta $\mathbf{p}$ coordinates into $\mathbf{x} = (\mathbf{q}, \mathbf{p})$, the Hamilton equations admit an alternative representation,

$$\frac{\mathrm{d}\mathbf{x}}{\mathrm{d}t} = \mathbf{J}\nabla_{\mathbf{x}}\mathcal{H}, \tag{5.38}$$

in terms of the symplectic matrix

$$\mathbf{J} = \begin{bmatrix} 0 & \mathbf{I} \\ -\mathbf{I} & 0 \end{bmatrix}. \tag{5.39}$$

Recall that $\nabla_{\mathbf{x}}\mathcal{H}$ denotes the gradient of the Hamiltonian with respect to the components of $\mathbf{x}$, that is,

$$\nabla_{\mathbf{x}}\mathcal{H} = \left(\frac{\partial \mathcal{H}}{\partial q_i}, \frac{\partial \mathcal{H}}{\partial p_i}\right). \tag{5.40}$$

Bob: When hearing the word "matrix," I tend to think of a certain famous movie from 1999.

Alice: I don't think Prof. Starmover is talking about that….

Professor: Indeed not. But if you want to take the red pill and find out more about matrices in the mathematical context, I suggest you return another time (see appendix C). For now, suffice it to say that matrices are objects that operate on vectors and result in new vectors. So, $\mathbf{J}\nabla_{\mathbf{x}}\mathcal{H}$ is just another vector. The symplectic matrix works like an identity matrix, up to a minus sign, and if you work out the matrix multiplication $\mathbf{J}\nabla_{\mathbf{x}}\mathcal{H}$, you recover exactly Hamilton's equations, equation (5.37).

Getting back to Hamiltonian systems, they have very interesting properties. A good starting point if you want to dig deeper into Hamiltonian mechanics is the classical book by H. Goldstein, for example, chapter 8 of its third edition (Goldstein et al., 2001). To discuss these systems more effectively, let me introduce a slightly abstract concept: the flow of the system ϕ_t. The flow can be regarded as the mapping

$$(q_i, p_i) = \phi_t(q_0, p_0), \tag{5.41}$$

which represents the solution to Hamilton's equations in phase space. It simply tells us how the position and momentum coordinates evolve in time. We say that the flow is symplectic because it preserves the Hamiltonian nature of the problem.

Bob: What do you mean by "phase space"?

Professor: It is the space where the solution (q_i, p_i) to the d-dimensional Hamiltonian system resides. The dimension of the phase space is $2d$. For example, in our N-body problem, it is the space that contains all possible positions and velocities of our N bodies, and it will be of dimension $6N$.

Bob: And what do you mean by the term "flow"?

Professor: You can think of it as the time evolution of the particles' trajectories. For example, a simple description of the dynamical flow will be a linear mapping like

$$\mathbf{x}_{n+1} = \mathbf{A}\,\mathbf{x}_n, \tag{5.42}$$

where $\mathbf{A}$ is just a $6N \times 6N$ matrix. It describes how the solution advances from t_n to t_{n+1}.

Alice: Okay, but what does being symplectic *actually* mean? I mean, what is the utility of this approach?

Professor: The transformation from $\mathbf{x}_n$ to $\mathbf{x}_{n+1}$ in equation (5.42) will be symplectic if it abides by Hamilton's equations of motion (5.37). It is actually not that hard to find the corresponding mathematical conditions.

First, if Hamilton's equations hold, we should be able to write

$$\frac{\mathrm{d}\mathbf{x}_{n+1}}{\mathrm{d}t} = \mathbf{J}\nabla_{\mathbf{x}_{n+1}}\mathcal{H} \tag{5.43}$$

as well as

$$\frac{\mathrm{d}\mathbf{x}_n}{\mathrm{d}t} = \mathbf{J}\nabla_{\mathbf{x}_n}\mathcal{H}. \tag{5.44}$$

If we now differentiate equation (5.42) with respect to $\mathbf{x}_n$, we'll obtain

$$\frac{\partial \mathbf{x}_{n+1}}{\partial \mathbf{x}_n} = \mathbf{A}. \tag{5.45}$$

In two dimensions, where $\mathbf{x}_n = (q_n, p_n)$ and $\mathbf{x}_{n+1} = (q_{n+1}, p_{n+1})$, it will be

$$\mathbf{A} = \begin{bmatrix} \dfrac{\partial q_{n+1}}{\partial q_n} & \dfrac{\partial q_{n+1}}{\partial p_n} \\ \dfrac{\partial p_{n+1}}{\partial q_n} & \dfrac{\partial p_{n+1}}{\partial p_n} \end{bmatrix}. \tag{5.46}$$

Equation (5.45) allows us to relate the time derivatives of $\mathbf{x}_{n+1}$ and $\mathbf{x}_n$ like

$$\frac{\mathrm{d}\mathbf{x}_{n+1}}{\mathrm{d}t} = \frac{\partial \mathbf{x}_{n+1}}{\partial \mathbf{x}_n}\frac{\mathrm{d}\mathbf{x}_n}{\mathrm{d}t} = \mathbf{A}\frac{\mathrm{d}\mathbf{x}_n}{\mathrm{d}t}. \tag{5.47}$$

Similarly, the gradients of the Hamiltonian relate through

$$\nabla_{\mathbf{x}_n}\mathcal{H} = \left[\frac{\partial \mathbf{x}_{n+1}}{\partial \mathbf{x}_n}\right]^\top \nabla_{\mathbf{x}_{n+1}}\mathcal{H} = \mathbf{A}^\top \nabla_{\mathbf{x}_{n+1}}\mathcal{H}. \tag{5.48}$$

Professor: We need the transpose of $\mathbf{A}$ because of how the partial derivatives are organized. Recovering our two-dimensional example, and if you recall how the chain rule works,

$$\frac{\mathrm{d}f}{\mathrm{d}x} = \frac{\mathrm{d}f}{\mathrm{d}u}\frac{\mathrm{d}u}{\mathrm{d}x}, \tag{5.49}$$

we'll get

$$\begin{aligned} \nabla_{\mathbf{x}_n}\mathcal{H} &= \begin{bmatrix} \dfrac{\partial \mathcal{H}}{\partial q_n} \\ \dfrac{\partial \mathcal{H}}{\partial p_n} \end{bmatrix} = \begin{bmatrix} \dfrac{\partial \mathcal{H}}{\partial q_{n+1}}\dfrac{\partial q_{n+1}}{\partial q_n} + \dfrac{\partial \mathcal{H}}{\partial p_{n+1}}\dfrac{\partial p_{n+1}}{\partial q_n} \\ \dfrac{\partial \mathcal{H}}{\partial q_{n+1}}\dfrac{\partial q_{n+1}}{\partial p_n} + \dfrac{\partial \mathcal{H}}{\partial p_{n+1}}\dfrac{\partial p_{n+1}}{\partial p_n} \end{bmatrix} = \begin{bmatrix} \dfrac{\partial q_{n+1}}{\partial q_n} & \dfrac{\partial p_{n+1}}{\partial q_n} \\ \dfrac{\partial q_{n+1}}{\partial p_n} & \dfrac{\partial p_{n+1}}{\partial p_n} \end{bmatrix}\begin{bmatrix} \dfrac{\partial \mathcal{H}}{\partial q_{n+1}} \\ \dfrac{\partial \mathcal{H}}{\partial p_{n+1}} \end{bmatrix} \\ &= \mathbf{A}^\top \nabla_{\mathbf{x}_{n+1}}\mathcal{H}. \end{aligned} \tag{5.50}$$

Alice: Oooh, I see it now. Wasn't that hard!

Professor: Glad you see it now. Substituting equations (5.47) and (5.48) into equation (5.43) yields

$$\frac{\mathrm{d}\mathbf{x}_{n+1}}{\mathrm{d}t} = \mathbf{A}\mathbf{J}\mathbf{A}^\top \nabla_{\mathbf{x}_{n+1}}\mathcal{H}. \tag{5.51}$$

Bob: How can this equation be right? We already agreed that the derivative of $\mathbf{x}_{n+1}$ is given in equation (5.43)!

Professor: Yes, so what does this tell us? What would happen if you equate those two expressions? You are completely right; they should be equivalent.

Bob: Equating them gives

$$\mathbf{A}\mathbf{J}\mathbf{A}^{\top} = \mathbf{J}. \tag{5.52}$$

Professor: Correct! Equation (5.52) tells us the conditions that matrix $\mathbf{A}$ must satisfy for the Hamilton equations to hold, that is, for the map (5.42) to be symplectic. In other words, if we use an integrator that advances the solution using a matrix that satisfies equation (5.52), then the integrator is symplectic. Conversely, if the mapping does not satisfy equation (5.52), then Hamilton's equations will no longer hold.

We can now check the symplecticness of the leapfrog scheme for a simple case. Consider the Hamiltonian of a one-dimensional harmonic oscillator

$$\mathcal{H} = \frac{1}{2}(p^2 + k^2 q^2). \tag{5.53}$$

The Hamilton equations reduce to

$$\dot{q} = p, \qquad \dot{p} = -k^2 q = F(q). \tag{5.54}$$

If we recover equation (5.36a) and specialize to the problem at hand, we obtain

$$q_{n+1} = q_n + p_n \Delta t + F(q_n)\frac{\Delta t^2}{2} = p_n \Delta t + \left(1 - \frac{1}{2}k^2 \Delta t^2\right) q_n. \tag{5.55}$$

Similarly, from equation (5.36b) and considering the previous result, it follows that

$$p_{n+1} = \left(1 - \frac{1}{2}k^2 \Delta t^2\right) p_n + k^2 \Delta t \left(\frac{1}{4}k^2 \Delta t^2 - 1\right) q_n. \tag{5.56}$$

These equations define the linear mapping

$$\begin{bmatrix} q_{n+1} \\ p_{n+1} \end{bmatrix} = \mathbf{A} \begin{bmatrix} q_n \\ p_n \end{bmatrix} = \begin{bmatrix} a & b \\ c & a \end{bmatrix} \begin{bmatrix} q_n \\ p_n \end{bmatrix}, \tag{5.57}$$

where

$$a = 1 - \frac{1}{2}k^2 \Delta t^2, \qquad b = \Delta t, \qquad c = k^2 \Delta t \left(\frac{1}{4}k^2 \Delta t^2 - 1\right). \tag{5.58}$$

The condition given in equation (5.52) reduces to

$$a^2 - bc = 1. \tag{5.59}$$

You can now check that the values of a, b, and c satisfy this condition.

Alice: Let's see….

$$\left(1 - \frac{1}{2}k^2 \Delta t^2\right)^2 - \Delta t \left[k^2 \Delta t \left(\frac{1}{4}k^2 \Delta t^2 - 1\right)\right] = 1 - k^2 \Delta t^2 + \frac{1}{4}k^4 \Delta t^4 - \frac{1}{4}k^4 \Delta t^4 + k^2 \Delta t^2 = 1. \tag{5.60}$$

Yes, they do! Did we just prove that the leapfrog method is symplectic?

Professor: Exactly! And when you check the energy conservation again using this algorithm, you will notice something interesting. Although the energy (or equivalently the Hamiltonian) is not perfectly conserved due to the approximation that we are using, the fact that the Hamilton equations hold on every step prevents the error from growing in time. We say that there is no secular growth in the error, which means that it remains bounded.

Alice: Aha! A solution to the previous problem we were encountering before coming to talk to you!

Professor: Well, I'm glad this helped. But I should be going now.

Bob: Thank you very much for your time, Prof. Starmover. We have many new tricks to explore for the time being.

Alice: Thank you, and see you next week, Prof. Starmover.

Professor: Good luck with the propagator! I look forward to hearing about your progress.

5.6 Numerical Performance

Alice and Bob return to their workspace at school.

Bob: We now have four integrators available! And, thanks to using the same interface, we can easily change between them:

```
sol_time, sol_state1 = integrate_euler(x, t0, tf, dt, params)
sol_time, sol_state2 = integrate_runge_kutta(x, t0, tf, dt, params)
sol_time, sol_state3 = integrate_adams_bashforth(x, t0, tf, dt, params)
sol_time, sol_state4 = integrate_leapfrog(x, t0, tf, dt, params)
```

Alice: Why don't we check how well the energy is conserved when using different integrators? It should be very easy now that we have the required blocks for each integrator. I want to do a longer integration, say $t_f = 500$. That's about $500/4\pi \approx 40$ orbits. I will take a step size of 0.01 for the integrations. Okay, I just plotted everything together in figure 5.1.

Bob: There are some very interesting tendencies…. Look at how small the error for the Runge–Kutta method is! I am sure it has to do with its higher order.

Alice: That's right, I guess, and the time step was not very small. The errors at the end of the integration are of the order of 10^{-11}. Since the integrator is of order 4, the truncation error is $\mathcal{O}(\Delta t^5)$. Our time step is $\Delta t = 10^{-2}$, so the local truncation error should be of the order of 10^{-10}. It is close!

Bob: Prof. Starmover said that we will not be able to get any better than machine zero precision, but I cannot remember what that is.

Alice: Around 10^{-16}, she said.

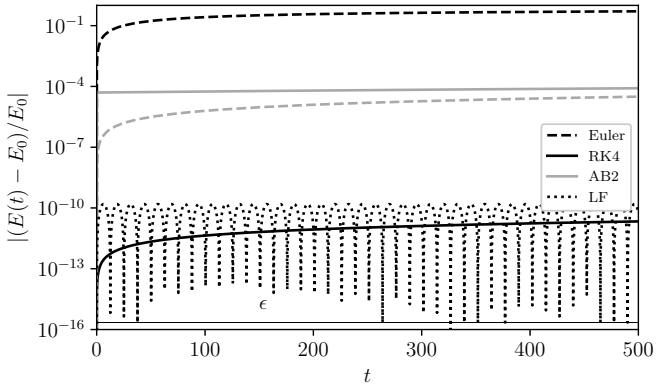


Figure 5.1
Error in energy conservation for different integrators ($\Delta t = 0.01$ and $t_f = 500$). The dash-dot line at the bottom represents the machine zero, ϵ.

Bob: Well, we are not too far off then! Especially if we compare the results with Euler's method. That method seems *terrible*….

Alice: There is something funny going on with the Adams–Bashforth method. The error in the energy seems to be constant. Am I doing something wrong? As always, I am plotting $|(E - E_0)/E_0|$.

Bob: It seems that the energy suddenly changed after the first point.

Alice: Wait, aren't we using a one-step Euler to initialize our multistep method? Maybe that is introducing a pretty large error. Let me check by computing the error as the deviation with respect to the second point, instead of the first: $|(E - E_1)/E_1|$. I plotted the new error measurement with a dashed line.

Bob: That looks much better. The error is growing like with the Runge–Kutta or Euler method. This means that the error of the initialization step is much larger than the error of the actual multistep method. We'll be better off using RK4 to initialize the problem.

Alice: Let's see how leapfrog behaves. I am curious after all we learned about symplectic integrators: the error in the energy should not grow steadily in time.

Bob: That's exactly what we see in the plot: although the energy is not perfectly conserved, the error is bounded with a maximum value around 10^{-10}. The Runge–Kutta method seems more accurate, though.

Alice: It is of higher order and that should reduce the truncation errors. But it grows in time, so eventually it will become larger. Something to take into account, if we have to run long simulations.

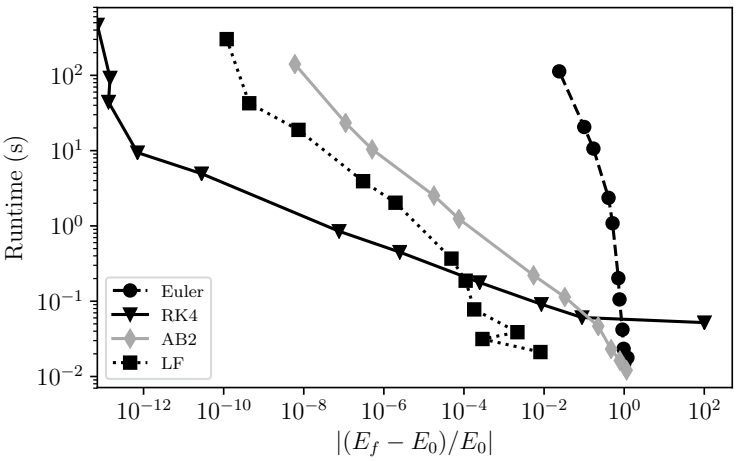


Figure 5.2
Performance of different integration schemes. E_f denotes the value of the energy at the end of the integration interval.

Bob: This test gives us information about the accuracy of the solution, but what about the runtime? We haven't looked into that yet. After all, being the fastest integrator isn't necessarily a good thing if the method isn't very accurate. Accuracy and runtime should be analyzed at the same time.

Alice: Agreed. And there is still the issue of the step size, which always concerns me. Does a method work better with decreasing step size? If so, how far should we go?

Bob: I have no idea, to be honest.

Alice: Well, we can do the following test: let's run each method using different time steps and measure the error in the energy conservation. That way, we can see how small the step size needs to be.

Bob: Sounds good. Why don't we measure the runtime too?

Alice: Good idea! In fact, we could plot them together. The plot will tell us how much faster or slower one method is with respect to the others, for the same level of accuracy.

Bob: The last thing you said is the key. Let's make the plot.

Alice: Okay, you can see the results in figure 5.2. I have to say, I wasn't expecting the differences to be this large! Look at that, RK4 is more than twelve orders of magnitude more accurate than Euler if you spend ten seconds integrating the orbit.

Bob: You can also look at it from another point of view: if we want the error to be less than, for example, 10^{-8}, RK4 will be about fifteen times faster than the leapfrog method, which is approximately ten times faster than AB2.

Alice: *And* you can't even reach that accuracy with Euler.

Bob: Right, it clearly stays behind the rest. Note that for large step sizes, the leapfrog method performs better than RK4.

Alice: This was a fair bit of work, but I'm starting to appreciate that choosing the right integrator might make a huge difference. Let's not forget that!

Bob: Well … I have to say that if you don't choose the right step size, then the results can be as bad as the integration using the Euler method. At least that is true with lower-order methods.

Alice: I guess you are right. We are again facing the problem of how to choose the step size! The main problem, as I see it, is that reducing the time step makes the integrator much slower, and we might want to be fast in some cases. Finding the right step size is really tricky!

Bob: I'm sure someone must have already looked into this…. We should ask Prof. Starmover about this in our next meeting.

6 Variable Step-Size Integration

Overview. Alice and Bob are faced with the problem that they generally do not know what integration time step to take. Choosing the optimum time step for integrating a specific orbit configuration is a complicated task. Furthermore, using a constant step size presents several limitations. First, as the system evolves over time, smaller time steps might be required to adequately model rapid events like close approaches between bodies. Second, when the dynamics are smooth and the system evolves slowly, larger time steps should be used to speed up the propagation. To overcome these limitations, variable step-size integration methods change the step size automatically during the integration, finding the largest time step that still meets a prescribed accuracy level. With the help of Prof. Starmover, Alice and Bob implement into their code the embedded Runge–Kutta methods and typical step-size control algorithms. They review their code at the end of this chapter.

It is Monday afternoon and Bob is running late for their meeting with Prof. Starmover. Alice decides she has waited long enough and knocks on Prof. Starmover's door.

Professor: Come in!

Alice: Good afternoon, Prof. Starmover.

Professor: Hello, Alice. Is Bob coming?

Alice: He was supposed to, but I don't know where he is….

Professor: No problem, we can start talking about your results while we wait for him.

Alice: The methods we implemented were way more accurate and efficient than Euler's method! But we still have the problem of not knowing what step size to choose!

Professor: That's a very good point. So far, we've only considered fixed-step methods in which Δt is defined by the user. But you can also use algorithms with an *adaptive step size*. When using these methods, instead of defining the step size, the user defines a certain tolerance. The algorithm finds the right size for Δt at every step in order to make sure that the error is smaller than the chosen tolerance. You can get significant speedups too because the integrator uses larger step sizes when the problem is very smooth. They are….

A rapid knock at the door, followed by Bob rushing into the room.

Bob: Sorry I'm late! I got caught up in my last class. What did I miss?

Alice: Prof. Starmover was just explaining how adaptive methods work. They choose the step size automatically!

Bob: Cool!

6.1 Estimating the Local Truncation Error

Professor: Indeed. I'll explain how variable step-size Runge–Kutta methods work, which are also known as *embedded* methods. The algorithm propagates two Runge–Kutta methods, providing the solutions

$$\mathbf{x}_{n+1} = \mathbf{x}_n + (b_1 k_1 + \ldots + b_s k_s)\Delta t, \tag{6.1a}$$

$$\tilde{\mathbf{x}}_{n+1} = \mathbf{x}_n + (\tilde{b}_1 k_1 + \ldots + \tilde{b}_s k_s)\Delta t, \tag{6.1b}$$

in which the coefficients k_i take the same values in both propagations. If the first method is of order p, then the coefficients $\tilde{b}_i$ are defined in such a way that the second method is of order $\tilde{p} = p - 1$ (or $\tilde{p} = p + 1$). The solution is given by the first method, and the second method is only used to estimate the local truncation error. If $\mathbf{x}$ has dimension N, then the error on the ith component at t_{n+1} is estimated as

$$\varepsilon_{i,n+1} = |x_{i,n+1} - \tilde{x}_{i,n+1}|, \qquad i = 1, \ldots, N. \tag{6.2}$$

The algorithm then chooses the largest step size that still keeps the error below the specified tolerance. You might have seen the acronym RKF4(5); this is a variable step-size Runge–Kutta method of order 4, which uses a method of order 5 to estimate the local truncation error, $p(\tilde{p})$. This is called the Runge–Kutta–Fehlberg method. Other examples are the Cash–Karp method and the Dormand–Prince method.

Alice: How much harder to implement are variable step-size integrators compared to ones with a fixed step size?

Professor: The idea behind the integrators themselves is essentialy the same. The main difference is that the variable step-size integrators incorporate step-size control routines. Do you remember how we constructed the fixed-step Runge–Kutta methods? How did we define the coefficients?

Alice: Sure, we needed the Butcher tableaus, like in table 5.1.

Professor: That's right. For starters, you can take a look at chapter II.5 of the book by Hairer et al. (1991).

Bob: Wow, that's a lot of tables with coefficients!

Alice: You aren't kidding….

Bob: But Runge–Kutta methods are actually pretty cool; by changing the coefficients, you get different methods. Actually … we could make our implementation of the integrator independent of the exact coefficients! Imagine that? This way, we could use different integration methods by simply selecting different sets of coefficients.

Professor: That's a very good idea, Bob.

Bob: The main part of a Runge–Kutta method is computing the different stages, following equation (5.14), right? In fact, we can stick to the last equation, the one that generalizes the value of k_s:

$$k_s = \mathbf{f}(t_n + c_s\Delta t, \mathbf{x}_n + \Delta t \sum_{j=1}^{s-1} a_{sj}k_j). \tag{6.3}$$

Given the Runge–Kutta matrix a_{ij}, the weights b_i, and the nodes c_i, we should be able to compute k for all stages. Let me start the main loop and code this formula. I want to store the values of k in an array. In addition, we will need two extra loops, one to compute all stages (from 1 to s) and another one to compute the sum (from $j = 1$ to $s - 1$).

```
integrate = True
while integrate:
    # Evaluate stages: first, compute k1:
    ks[0,:] = ode_two_body_first_order(x0, t, params)
    # Loop through all stages to compute k's:
    for stage in range(1, stages):
        # Loop to compute the sum over the previous stages:
        auxvec = x0 * 0
        for j in range(0, stage):
            auxvec += a[stage - 1, j] * ks[j,:]
        ks[stage,:] = ode_two_body_first_order(x0 + auxvec * dt, t + c[
            stage - 1] * dt, params)
```

Alice: What do you mean with `while integrate:`?

Bob: I don't know when to stop the integration because the time step is not fixed. This means that we cannot know a priori how many steps we will be taking. So I just left it in this generic form, waiting for the moment when we figure out how to stop the integration.

Professor: It's actually easier than you might think.

Bob: Hmmm…. We know that t will advance up to t_f. So the stopping condition will be $t = t_f$. But we are defining the time at the next step as $t_{n+1} = t_n + \Delta t_n$. If we are at the second-to-last step, t_n is still less than t_f. But $t_n + \Delta t_n$ might be larger than t_f! So we'll have $t_{n+1} > t_f$.

Alice: Yeah, you're right … but wait! What if we force the step size to satisfy $t_n + \Delta t_n = t_f$? Let's see: if we discover that $t_n + \Delta t_n$ will take us beyond the final integration time, we can adjust the step size to $\Delta t = t_f - t_n$. That's it! And since $t_f < t_n + \Delta t_n$, we know that the adjusted time step will be smaller than the required one so we are not affecting the accuracy.

Bob: Good idea! We can then solve the overshooting problem and accordingly adjust the time step as follows:

```
# Detect last step and correct overshooting:
if ( t + dt > tf ):
    dt = tf - t
```

We'll carry out this check right after we estimate the value of Δt for the next step.

Professor: Good job!

Alice: Okay, give me a second to see if I got the rest of what Bob did. We compute k_1 like `ks[0,:]` because of PYTHON's indexing format. Then we define the loop to compute stages k_2 to k_s. You are calling s "`stages`," right?

Bob: Yes, I thought it might be easier to follow.

Alice: Then you are computing the sum, from 1 to $s-1$, of the coefficients of the Runge–Kutta matrix a_{sj}, multiplied by k_j.

Bob: Finally, I use the result to compute the next k.

Alice: And the code is both allocating the auxiliary vector `auxvec` and using row-major order. Cool! We can now propagate the state of the system using equation (5.13):

```
# Advance the state:
auxvec = x0 * 0
for j in range(0, stages):
    auxvec += ks[j,:] * b[j]
x = x0 + dt * auxvec
```

Bob: Is that it? We have all the pieces for our variable step-size integrator!

Alice: Not so fast … we simply wrote a Runge–Kutta integrator generalizing the coefficients. But the time step is still fixed.

Bob: Oops, you're right.

Professor: This is when equation (6.1) becomes handy.

Alice: Okay, so the idea is to propagate a second solution,

$$\tilde{\mathbf{x}}_{n+1} = \mathbf{x}_n + (\tilde{b}_1 k_1 + \ldots + \tilde{b}_s k_s)\Delta t, \tag{6.4}$$

with a different set of coefficients, and then compare it to $\mathbf{x}_{n+1}$:

$$\Delta\mathbf{x}_{n+1} = \mathbf{x}_{n+1} - \tilde{\mathbf{x}}_{n+1} = [(b_1 - \tilde{b}_1)k_1 + \ldots + (b_s - \tilde{b}_s)k_s]\Delta t. \tag{6.5}$$

Bob: Well, we already have $\mathbf{x}_{n+1}$ and k_j. Moreover, the error estimation depends only on the difference between the weights b_j and $\tilde{b}_j$. We can compute this difference for the error estimation outside the main loop,

```
dest = b - bt
```

and then the error estimation will be

```
# Compute error estimate:
est = x0 * 0
for j in range(0, stages):
    est += ks[j,:] * dest[j]
est *= dt
```

Alice: You can reuse my loop to compute the state and the error estimation all at once:

```
# Advance the state and compute error estimate:
auxvec = x0 * 0
est = x0 * 0
for j in range(0, stages):
    auxvec += ks[j,:] * b[j]
    est += ks[j,:] * dest[j]
x = x0 + dt * auxvec
est *= dt
```

But I have no idea what to do now, with this error estimation….

6.2 Step-Size Control

Professor: What you have is pretty good already! The next step is to use the variable `est` to estimate the error. Ideally, the difference between the two propagations, $\mathbf{x}_{n+1}$ and $\tilde{\mathbf{x}}_{n+1}$, should be zero. But, in practice, we can only hope that the difference will be smaller than a certain absolute tolerance ε_{abs},

$$|x_{i,n+1} - \tilde{x}_{i,n+1}| \leq \varepsilon_{\text{abs},i}. \tag{6.6}$$

If we set the absolute tolerance to be something small, like 10^{-12}, then the error should be of that order. We can refine this a bit more and introduce a measure of the relative error too; for that, let me rewrite the previous equation as follows:

$$|x_{i,n+1} - \tilde{x}_{i,n+1}| \leq s_i \tag{6.7}$$

and define s_i as

$$s_i = \varepsilon_{\text{abs},i} + \varepsilon_{\text{rel},i}|x_{i,n}|. \tag{6.8}$$

This allowed me to introduce a relative tolerance $\varepsilon_{\text{rel},i}$, which is multiplied by the corresponding component of the state vector. To be more conservative, we can consider both the current and the next values of the state vector and take the largest value of the two:

$$s_i = \varepsilon_{\text{abs},i} + \varepsilon_{\text{rel},i} \max(|x_{i,n}|, |x_{i,n+1}|). \tag{6.9}$$

Now that we have the components s_i of the vector $\mathbf{s}$, we typically define the error e at time t_{n+1} as

$$e_{n+1} = d(\mathbf{x}_{n+1} - \tilde{\mathbf{x}}_{n+1}, \mathbf{s}). \tag{6.10}$$

Bob: Wait, what is that?

Professor: The function d is simply a special way to define a norm (Hairer et al., 1991, p. 168), accounting for the tolerances:

$$d(\mathbf{v},\mathbf{s}) = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\frac{v_i}{s_i}\right)^2}. \tag{6.11}$$

In PYTHON , it will be:

```
def rk_norm( v, sc ):
    n = len(v)
    norm = np.sqrt( sum((v / sc)**2) / n )

    return norm
```

Do you think you can code the error estimation in PYTHON yourselves?

Alice: Sure, it will be something like…

```
# Estimate error:
for i in range(0, dim):
    sc[i] = atol[i] + max([abs(x0[i]), abs(x[i])]) * rtol[i]
err = rk_norm(est, sc)
```

Professor: Good job. Once you have an estimation of the error, the next step is to evaluate it. Is it small enough? Is it too large? Thanks to how the error was estimated, if e_{n+1} is less than 1, this means that the components of the error estimation vector are smaller than the tolerance parameter s_i, satisfying equation (6.7). Thus, we will *accept* the step if $e_{n+1} \leq 1$ and *reject* it otherwise.

Bob: Let me try to code this up: if the error estimated by Alice's code is less than or equal to 1, then we will advance one step, like we did in the fixed-step case. That is,

```
# Accept the step:
if ( err <= 1 ):
    # Advance the time:
    t = t + dt
    # Store step:
    sol_state[count,:] = x
    sol_time[count] = t
    # Update state:
    x0 = x
```

On the other hand, if the step is rejected we have to…. Wait, what should we do if the step is rejected?

Professor: Let me explain how to choose the time step for the next step, Δt_{n+1}. This will answer your question. A common practice, coming from many years of accumulated

experience and research (see Hairer et al., 1991, chap. II.4), is to take

$$\Delta t_{n+1} = f \Delta t_n e_{n+1}^{-1/(1+q)}, \tag{6.12}$$

where Δt_n is the step size that we used in the current step, and f is a constant factor depending on the order of the integrator,

$$f = 0.38^{1/(1+q)}. \tag{6.13}$$

If the integrator is of order p and the estimation is of order $\tilde{p}$, then $q = \min(p, \tilde{p})$.

Alice: This seems easy to implement. Like this:

```
dt = dt * fac / err**(1.0 / (1.0 + q))
```

Bob: What about the coefficients? We have to initialize them. We can do it outside the main loop, at the very beginning:

```
q = min([p, pt])
fac = 0.38**(1.0 / (1.0 + q))
```

Professor: Yes, but every time you work with automatic control algorithms like this one, it is wise to check that the values are reasonable. In this case, we would like to check several things. First, we need to ensure that the new time step is not much larger or smaller than the previous one,

$$f_{\min}|\Delta t_n| \le |\Delta t_{n+1}| \le f_{\max}|\Delta t_n|. \tag{6.14}$$

If the new time step is larger than the upper bound, we will set its value to $f_{\max}|\Delta t_n|$. On the other hand, if it is less than the lower bound, we will set it equal to $f_{\min}|\Delta t_n|$. Useful values for $f_{\min}$ and $f_{\max}$ are 0.33 and 6, respectively.

One final check should ensure that the step size is not too small compared to the actual time interval. For example,

$$\Delta t_{n+1} \ge 10^{-12}(t_f - t_0). \tag{6.15}$$

If you are integrating a system over 1 Gyr, this constraint forces the time step to be no less than 0.37 days, which is already very small.

Alice: Then, this more sophisticated step-size control algorithm will be

```
dtnew = dt * fac / err**(1.0 / (1.0 + q))
# Check whether the new stepsize is adequately bounded:
if ( abs(dtnew) > facmax * abs(dt) ):
    dt = facmax * dt
elif ( abs(dtnew) < facmin * abs(dt) ):
    dt = facmin * dt
elif ( dtnew / (tf - t0) < 1e-12 ):
    dt = (tf - t0) * 1e-12
else:
    dt = dtnew
```

Oh, and we need to adjust the time step for the last step. Just like we talked about:

```
# Detect last step and correct overshooting:
if ( t + dt > tf ):
    dt = tf - t
```

Bob: The coefficients are as follows:

```
facmax = 6.0
facmin = 0.33
```

Professor: You now have a step-size control algorithm! If the step is rejected, a new step size will be estimated, which is smaller than the previous one to reduce the error, and the step will be repeated until the specified tolerance is met.

6.3 Initial Step Size

Professor: The only remaining issue is what step size you should choose to start the integration.

Bob: I didn't think about that….

Professor: It's more or less similar to the error estimation that we already discussed. First, we'll compute the tolerance-like factor s_i, as follows:

$$s_i = \varepsilon_{\text{abs},i} + \varepsilon_{\text{rel},i}|x_{i,0}|. \tag{6.16}$$

The only difference here is that we're now using the components of the initial state vector, $\mathbf{x}_0$. Second, we evaluate the right-hand side of our system of ordinary differential equations (ODEs), that is, the derivatives of $\mathbf{x}$:

$$\mathbf{f}_0 = \mathbf{f}(t_0, \mathbf{x}_0). \tag{6.17}$$

The values of $\mathbf{f}_0$ and $\mathbf{x}_0$ define the two quantities

$$d_0 = d(\mathbf{x}_0, \mathbf{s}), \qquad \text{and} \qquad d_1 = d(\mathbf{f}_0, \mathbf{s}). \tag{6.18}$$

We can take

$$\Delta t \equiv \Delta t_0 = \frac{d_0}{d_1} \tag{6.19}$$

as a good initial step size, which corresponds to the ratio between the initial state and its time derivative. To bound its value, it is convenient to force $\Delta t_0 = 10^{-4}$ if either $d_0 < 10^{-5}$ or $d_1 < 10^{-5}$.

As with estimating the step size for the next step, we can refine the estimation by first taking one forward step with the Euler integration scheme. For that, let's write

$$\mathbf{x}_1 = \mathbf{x}_0 + \frac{\Delta t_0}{100}\mathbf{f}_0, \tag{6.20}$$

and let's evaluate the derivatives at t_1:

$$\mathbf{f}_1 = \mathbf{f}(t_0 + \Delta t_0/100, \mathbf{x}_0), \tag{6.21}$$

which leads to the parameter

$$d_2 = \frac{1}{\Delta t_0} d(\mathbf{f}_0 - \mathbf{f}_1, \mathbf{s}). \tag{6.22}$$

Using d_2 as an estimate provides information about how much the derivatives have changed. Experience once again dictates that we use the following logic to choose the step size: if both d_1 and d_2 are smaller than 10^{-15}, we set Δt_1 to $\max(10^{-6}, 10^{-3}\Delta t_0)$; otherwise, we set Δt_1 equal to

$$\Delta t_1 = \left[\frac{0.01}{\max(d_1, d_2)}\right]^{\frac{1}{1+p}}. \tag{6.23}$$

Finally, the initial time step is

$$\Delta t = \min(100\Delta t_0, \Delta t_1). \tag{6.24}$$

Alice: Wow, this is really starting to look like an art. I mean, setting all those parameters seems like it requires some serious skill, not to mention understanding the logic behind those choices.

Professor: Yes, that's true. But the good thing is that this selection process has been extensively tested, and you can implement it directly.

Bob: The first part seems easy:

Snippet 6.1
Estimating the initial step for variable step-size integrators.

```
# Compute scaling:
sc = atol + abs(x0) * rtol

# Evaluate function:
f0 = ode_two_body_first_order(x0, t0, params)
d0 = rk_norm(x0, sc)
d1 = rk_norm(f0, sc)

# Enforce bounds:
if ( (d0 < 1e-5) or (d1 < 1e-5)):
    dt0 = 1e-6
else:
    dt0 = 0.01 * (d0 / d1)

# Perform one Euler step:
x1 = x0 + dt0 * f0

# Evaluate derivatives:
f1 = ode_two_body_first_order(x1, t0 + dt0, params)
d2 = rk_norm(f1 - f0, sc) / dt0
if ( max(d1, d2) <= 1e-15 ):
```

```
    dt1 = max([1e-6, dt0 * 1e-3])
else:
    dt1 = (0.01 / max([d1, d2]))**(1.0 / (p + 1))
dt = min([100 * dt0, dt1])
```

For the second part, we needed the second estimator using a one-step Euler.

Alice: Phew, this is a lot! I will need to review this code again later, on my own.

Professor: Nicely done, you two. You now have all the pieces in place to complete your variable step-size propagator. I think you should spend some time now testing your code.

Bob: Thanks for all the help, Prof. Starmover!

Alice: Thanks, see you next week!

Alice and Bob leave Prof. Starmover's office and head to the study room.

6.4 Implementing Flexibility

Bob: The step-size control algorithm was actually more complicated than I thought. But I think we have it now. Let's create our brand-new variable step-size integration function. We'll need the same inputs that we had in the fixed-step case.

Alice: Right. Plus the coefficients for the Runge–Kutta method: the coefficients of the Runge–Kutta matrix a_{ij}, the weights b_j, the weights of the estimation stage $\tilde{b}_j$, and the nodes c_j.

Alice: We should initialize all the required parameters too: the dimension of the system, the number of stages (given by the number of weights, b_j), and all the factors used to control the step size….

Bob: Oh, I just realized that we also need the order p of the main method….

Alice: …and the order $\tilde{p}$ of the method used for the estimation step, in order to compute q. Actually, we could implement a function with all the required coefficients for a given order and call it at the very beginning to retrieve these coefficients. I'd name the function `butcher_tableaus_rk`. Let's leave that function for later (snippet 6.4) and proceed with the rest of the code:

Snippet 6.2
Function `integrate_rk_embedded`: Integrate using variable step-size RK integrator.

```
def integrate_rk_embedded( x0, t0, tf, atol, rtol, order, params ):

    # Retrieve coefficients:
    a, b, bt, c, p, pt = butcher_tableaus_rk(order)
```

Bob: We should allocate the array containing the states. But we have a problem: we don't know the dimension a priori. How can we allocate it?

Alice: Oh, that's right. Good catch!

Bob: One thing that we can do is to start with a large array. Then, if we reach the limit (the memory buffer) of the array, we can extend it.

Alice: Okay, sounds like a good option to me. Let's do that:

```
# Allocate solution:
npts =      100000
sol_state = np.zeros((npts, dim))
sol_time =  np.zeros((npts))

# Initial values for the time and state vectors:
sol_time[0] = t0
sol_state[0,:] = x0
```

Bob: Finally, we can start storing the solution inside the main loop:

Snippet 6.3
Main integration loop for adaptive Runge–Kutta methods.

```
# Launch integration:
count = 1  # Initialize counter
t = t0
integrate = True
while integrate:

    # Evaluate stages, advance state, and estimate error
    ....

    # Accept step:
    if ( err <= 1 ):
        t  = t + dt

        # Store step:
        sol_state[count,:] = x
        sol_time[count] = t

        # Update:
        count += 1
        x0 = x

        # Check buffer size and extend if needed:
        if ( count == sol_state.shape[0] ):
            sol_state = np.concatenate((sol_state, sol_state[0: npts, :]))
            sol_time  = np.concatenate((sol_time, sol_time[0: npts]))

        # Stop integration when tf is reached:
        if ( t >= tf ):
            integrate = False

    # Estimate new step size, correct overshooting
    ....
```

Alice: Thanks to having the counter, we can return only the part of the array that we actually need:

```
return sol_time[0:count], sol_state[0:count,:]
```

Bob: Okay, so the integration routine is now done (see snippet 6.5 on page 102). The last step will be to implement the coefficients corresponding to the different methods. Why don't we split the work? I can work on the Runge–Kutta–Fehlberg methods 4(5) and 7(8).

Alice: Sure, I'll take … the Dormand–Prince 5(4) and Verner's method 6(5).

Bob: Great! We now have all the coefficients that we'll need. Let's put them together into snippet 6.4 in a way that makes it is easy to switch between methods.

6.5 Numerical Performance

Alice: Our integrators are now ready! It's time to see how well they perform. We can run the same performance test that we used to generate figure 5.2. The only difference is that, instead of changing the size of the time step, we'll have to change the tolerance.

Bob: Okay, let's recover the problem we were previously using as a test, the one in which the particles describe circular orbits. What if we change it a little bit to have eccentric orbits instead?

Alice: That should be easy. To do that, we just change the initial value of the velocity, and that's it. Say,

$$\begin{aligned} \mathbf{R}_1 &= [-1,0,0], \quad \mathbf{V}_1 = [0,-0.48,0], \\ \mathbf{R}_2 &= [+1,0,0], \quad \mathbf{V}_2 = [0,+0.48,0], \end{aligned} \tag{6.25}$$

with $G = 1$, and $m_1 = m_2 = 1$.

Bob: Do we need the position and velocity vectors to be symmetric?

Alice: I only did that to ensure that the center of mass coincides with the origin and that it doesn't move. The position of the center of mass will then be

$$\mathbf{R}_c = \frac{m_1\mathbf{R}_1 + m_2\mathbf{R}_2}{m_1 + m_2} = \frac{1}{2}(\mathbf{R}_1 + \mathbf{R}_2) = [0,0,0]. \tag{6.26}$$

The velocity of the center of mass comes from the definition of the total momentum of the system,

$$\mathbf{V}_c = \frac{m_1\mathbf{V}_1 + m_2\mathbf{V}_2}{m_1 + m_2} = \frac{1}{2}(\mathbf{V}_1 + \mathbf{V}_2) = [0,0,0]. \tag{6.27}$$

See? For equal masses, if we set $\mathbf{R}_1 = -\mathbf{R}_2$ and $\mathbf{V}_1 = -\mathbf{V}_2$, the center of mass will be at the origin, and it will stay there.

Bob: I see now … I didn't think about that for more general cases.

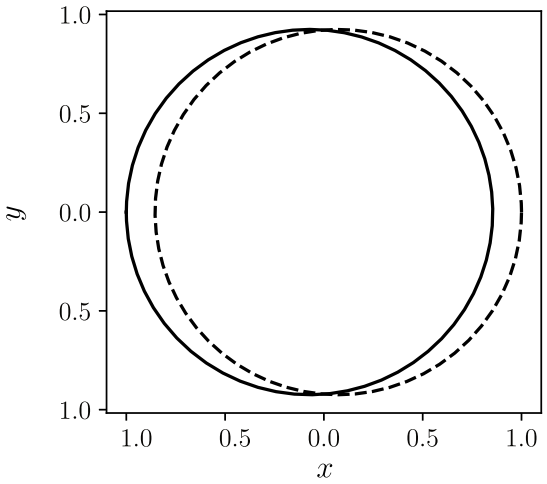


Figure 6.1
An example of the trajectories followed by two gravitationally bound equal-mass particles in an eccentric orbit.

Alice: The relative velocity is $v = \|\mathbf{V}_2 - \mathbf{V}_1\| = 0.96$, and the relative distance is $r = \|\mathbf{R}_2 - \mathbf{R}_1\| = 2$. If we recover equation (2.28), we can calculate the semimajor axis of the relative ellipse, a:

$$-\frac{GM}{2a} = \frac{v^2}{2} - \frac{GM}{r} = -0.5392 \implies a = 1.8546. \tag{6.28}$$

Thus, the period is

$$P = 2\pi\sqrt{\frac{a^3}{GM}} = 11.2212. \tag{6.29}$$

Bob: Let me propagate the orbits for one complete revolution to see how they look….

Alice: Good idea.

Bob: Okay, I plotted the orbits in figure 6.1.

Alice: Now that we know the period, we can decide the integration time for our performance tests.

Bob: Right. We could set $t_f = 200$, which corresponds to approximately twenty revolutions.

Alice: Sounds good to me. To compare the performance of the integrators, we'll follow these steps:

1. Set an absolute and relative tolerance (we can assume $\varepsilon_{\text{abs}} = \varepsilon_{\text{rel}}$ for simplicity).

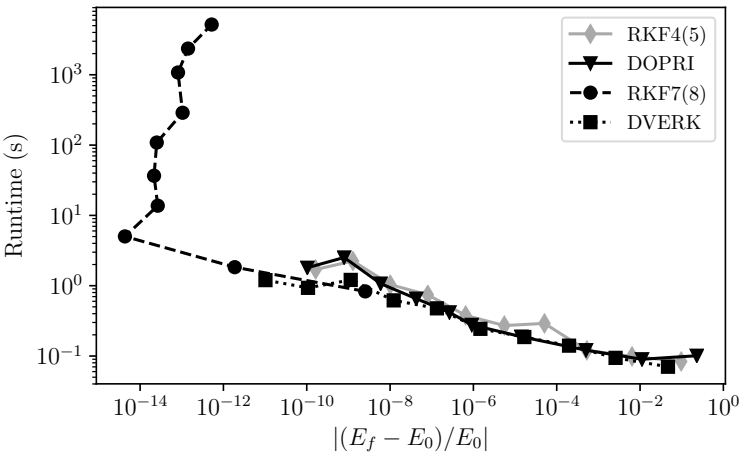


Figure 6.2
A comparison of the performances of different variable step-size integrators.

2. Propagate the system from $t_0 = 0$ to $t_f = 200$. We will repeat the propagation with each integrator.
3. Measure the runtime and compute the change in the energy for each of the propagations.
4. Reduce the tolerance; repeat steps 2 and 3.

I guess the only question left is how to pick the values of the tolerance.

Bob: Well, we can get an idea by looking at figure 5.1. Ignoring Euler's method, it looks like the errors range from 10^{-4} to 10^{-12}. Prof. Starmover said that the tolerance will more or less bound the value of the error, so why don't we take values for the tolerance parameter lying somewhere in that interval?

Alice: Good idea, Bob! Let's define our set of tolerance parameters like this:

```
vec_tols = 10 ** (-np.linspace(4, 12, num=10))
```

Bob: I just plotted the results in figure 6.2. The points that are furthest to the right correspond to the largest tolerances and are less accurate. The first thing I notice is that the RKF7(8) method seems to be much more accurate! Look at that, it almost reaches machine precision, with an error of 10^{-14}. I guess it is because it has the highest order?

Alice: I agree. But something funny happens after it reaches an error of 10^{-14}: decreasing the tolerance actually increases the error! And the runtime goes up significantly. I wonder … maybe that's what Prof. Starmover meant when she said "*small step sizes yield small truncation errors. But larger round-off errors.*"

Bob: I think you're right! It looks like the integrator is performing so many operations that the round-off error is introducing a lot of numerical noise. Small tolerances reduce the truncation error, but the round-off error can grow a lot. In this particular case, the third value of the tolerance that we tried is actually the optimal one for the method, namely, 1.6681×10^{-6}. This is very interesting because it tells us that adopting a super small tolerance might not yield the best accuracy. Instead, we'll just be wasting integration time.

Alice: We better keep that in mind when we propagate systems of planets. Another interesting result is that RKF4(5) and Do-Pri5(4) exhibit similar performances, although Do-Pri5(4) is a bit more accurate. It's computing the solution with a Runge–Kutta method of order 5, whereas RKF4(5) only uses fourth order. I guess that explains the difference?

Bob: Yes, and DVERK is of order 6 and turns out to be even more accurate.

6.6 Thoughts on Choosing the Integration Method

Bob: Holy cow. We implemented a lot of code over the last few days. My brain is exhausted.

Alice: You're right. A job well done, Bob. I think it's a good time to recap. These are the notes that I took over the past few days:

- Infinitely small time steps or tolerance parameters do not necessarily yield better results!
- A good indicator of propagator accuracy is to check how well preserved are the conservation laws.
- Choosing the right step size critically affects the integrator accuracy and runtime.
- Reducing the time step (or the tolerance parameter with variable step-size methods) reduces the truncation error but increases the round-off error.
- Multistep methods use information from several time steps, generating a polynomial to approximate the solution.
- Implicit methods require solving a set of algebraic equations (typically using predictor-corrector algorithms) and tend to be more stable than explicit methods.

Bob: Thanks for putting those notes together, Alice. They'll be incredibly useful down the road.

Alice: No problem. I'm a bit tired of just propagating two bodies, though. I'm looking forward to moving on to more realistic planetary systems. What do you think?

Bob: Me too. Next, let's tackle an actual N-body system!

6.7 Code Review

Before leaving the study room, Alice and Bob spend some time cleaning up the code that they wrote at Prof. Starmover's office. In particular, they check their implementation of the Butcher tableaus in snippet 6.4 and add comments to explain in more detail all the steps in the main integration driver (snippet 6.5).

Snippet 6.4
Function `butcher_tableaus_rk`: Butcher tableaus for embedded Runge–Kutta methods.

```
def butcher_tableaus_rk( order ):
    # Select integrator
    #   45) Runge-Kutta-Fehleberg 4(5)
    #   54) Dormand-Prince 5(4)
    #   78) Runge-Kutta-Fehlberg 7(8)
    #   65) Verner's method 6(5), DVERK

    # RUNGE-KUTTA-FEHLBERG 4(5)
    if ( order == 45 ):
        # Order
        p  = 4
        pt = 5
        a = np.array([[1./4., 0., 0., 0., 0.],
          [3./32., 9./32., 0., 0., 0.],
          [1932./2197.,-7200./2197., 7296./2197., 0., 0.],
          [439./216.,-8., 3680./513.,-845./4104., 0.],
          [-8./27., 2.,-3544./2565., 1859./4104.,-11./40.]])
        c = np.array([ 1./4., 3./8., 12./13., 1., 0.5])
        b = np.array([ 25./216., 0., 1408./2565., 2197./4104., -0.2, 0])
        bt= np.array([ 16./135., 0., 6656./12825., 28561./56430., -9./50.,
             2./55.])
        #
    # DORMAND-PRINCE 5(4)
    elif ( order == 54 ):
        # Order
        p  = 5
        pt = 4
        q = min([p,pt])
        a = np.array([[1./5., 0., 0., 0., 0., 0],
          [3./40., 9./40., 0., 0., 0., 0],
          [44./45.,-56./15., 32./9., 0., 0., 0],
          [19372./6561.,-25360./2187., 64448./6561.,-212./729., 0., 0],
          [9017./3168.,-355./33., 46732./5247., 49./176.,-5103./18656.,
              0],
          [35./384., 0., 500./1113., 125./192.,-2187./6784., 11./84.]])
        c = np.array([1./5., 3./10., 4./5., 8./9., 1., 1.])
        b = np.array([ 35./384., 0., 500./1113., 125./192., - 2187./6784.,
             11./84., 0])
        bt= np.array([ 5179./57600., 0., 7571./16695., 393./640., -
            92097./339200.,
            187./2100., 1./40.])
        #
```

```
    # RUNGE-KUTTA-FEHLBERG 7(8)
    elif ( order == 78 ):
        # Order
        p  = 7
        pt = 8
        a = np.array([[2./27., 0. , 0., 0., 0., 0., 0., 0., 0., 0.,
            0.],
          [1./36., 1./12., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
          [1./24., 0., 1./8., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
          [5./12., 0.,-25./16., 25./16., 0., 0., 0., 0., 0., 0., 0., 0.],
          [1./20., 0., 0., 1./4., 1./5., 0., 0., 0., 0., 0., 0., 0.],
          [-25./108., 0., 0., 125./108.,-65./27., 125./54., 0., 0., 0.,
              0., 0., 0.],
          [31./300., 0., 0., 0., 61./225.,-2./9., 13./900., 0., 0., 0.,
              0., 0.],
          [2., 0., 0.,-53./6., 704./45.,-107./9., 67./90., 3., 0., 0., 0.,
               0.],
          [-91./108., 0., 0., 23./108.,-976./135., 311./54.,-19./60.,
              17./6,-1./12.,
            0., 0., 0.],
          [2383./4100., 0., 0.,-341./164., 4496./1025.,-301./82.,
              2133./4100.,
            45./82., 45./164, 18./41., 0., 0.],
          [3./205., 0., 0., 0., 0.,-6./41.,-3./205.,-3./41., 3./41.,
              6./41., 0., 0.],
          [-1777./4100., 0., 0.,-341./164., 4496./1025.,-289./82.,
              2193./4100.,
            51./82, 33./164, 19./41., 0., 1.]])
        c = np.array([ 2./27., 1./9., 1./6., 5./12., 1./2., 5./6., 1./6.,
            2./3.,
            1./3., 1., 0., 1.])
        b = np.array([ 41./840., 0., 0., 0., 0., 34./105., 9./35., 9./35.,
             9./280.,
            9./280., 41./840., 0., 0.])
        bt= np.array([ 0., 0., 0., 0., 0., 34./105., 9./35., 9./35.,
            9./280.,
            9./280., 0., 41./840., 41./840.])
        #
    # VERNER 6(5) DVERK
    elif ( order == 65 ):
        # Order
        p  = 6
        pt = 7
        a = np.array([[ 1./6., 0., 0., 0., 0., 0., 0.],
          [ 4./75., 16./75., 0., 0., 0., 0., 0.],
          [ 5./6.,-8./3., 5./2., 0., 0., 0., 0.],
          [-165./64., 55./6.,-425./64., 85./96., 0., 0., 0.],
          [ 12./5.,-8., 4015./612.,-11./36., 88./255., 0., 0.],
          [-8263./15000., 124./75.,-643./680.,-81./250., 2484./10625., 0.,
               0.],
          [ 3501./1720.,-300./43., 297275./52632.,-319./2322.,
              24068./84065., 0.,
            3850./26703.]])
        c = np.array([ 1./6., 4./15., 2./3., 5./6., 1., 1./15., 1.])
```

```
        b = np.array([ 3./40., 0., 875./2244., 23./72., 264./1955., 0.,
            125./11592.,
            43./616.])
        bt= np.array([ 13./160., 0., 2375./5984., 5./16., 12./85., 3./44.,
             0., 0.])

    return a, b, bt, c, p, pt
```

Snippet 6.5

Function `integrate_rk_embedded`: Integrate using an embedded (variable step-size) Runge–Kutta method.

```
def integrate_rk_embedded(x0, t0, tf, atol, rtol, ode, order, params):

    # Check if order is supported:
    if ( not (order == 45 or order == 54 or order == 67 or order == 78) ):
        print("Order␣not␣supported,␣proceeding␣with␣RKF4(5)")
        order = 45

    # Retrieve coefficients:
    a, b, bt, c, p, pt = butcher_tableaus_rk(order)

    # Dimension:
    dim = len(x0)

    # Number of stages:
    stages = len(b)
    ks =     np.zeros((stages, dim))
    q =      min([p, pt])
    dest =   b - bt

    # Safety factors for step-size control:
    facmax = 6.
    facmin = 0.33
    fac =    0.38**(1. / (1. + q))

    # Initial state:
    x = x0

     # Allocate dense output:
    npts = 100000
    sol_state = np.zeros((npts, dim))
    sol_time = np.zeros((npts))

    # ESTIMATE INITIAL STEP SIZE
    # Compute scaling:
    sc = atol + abs(x0) * rtol
    # Evaluate function:
    f0 = ode(x0, t0, params)
    d0 = rk_norm(x0, sc)
    d1 = rk_norm(f0, sc)

    if ( (d0 < 1e-5) or (d1 < 1e-5)):
```

```
        dt0 = 1e-6
    else:
        dt0 = 0.01 * (d0 / d1)

    # Perform one Euler step:
    x1 = x0 + dt0 * f0
    # Call function:
    f1 = ode(x1, t0 + dt0, params)
    d2 = rk_norm(f1 - f0, sc) / dt0

    if ( max(d1, d2) <= 1e-15 ):
        dt1 = max([1e-6, dt0 * 1e-3])
    else:
        dt1 = (0.01 / max([d1, d2]))**(1.0 / (p + 1))

    dt = min([100 * dt0, dt1])

    # Initial values of the vector of times and states:
    sol_time[0] = t0
    sol_state[0,:] = x0

    # Launch integration:
    count = 1
    t = t0
    integrate = True
    while integrate:

        # Evaluate stages:
        ks[0,:]  = ode(x0, t, params)
        for stage in range(1, stages):
            auxvec = x0 * 0
            for j in range(0, stage):
                auxvec += a[stage - 1, j] * ks[j,:]
            ks[stage,:] = ode(x0 + auxvec * dt, t + c[stage-1] * dt,
                params)

        # Advance the state:
        auxvec = x0 * 0
        est = x0 * 0
        for j in range(0, stages):
            auxvec += ks[j,:] * b[j]
            est +=    ks[j,:] * dest[j]
        x   = x0 + dt * auxvec
        est *= dt

        # Estimate error:
        for i in range(0, dim):
            sc[i] = atol[i] + max([abs(x0[i]), abs(x[i])]) * rtol[i]
        err = rk_norm(est, sc)

        # Accept the step:
        if ( err <= 1 ):
            t = t + dt
```

```
            # Store step:
            sol_state[count,:] = x
            sol_time[count] = t

            # Update:
            count += 1
            x0 = x

            if (count == sol_state.shape[0]):
                sol_state = np.concatenate((sol_state, sol_state[0:npts
                    ,:]))
                sol_time = np.concatenate((sol_time, sol_time[0:npts]))

            # Stop integration when tf is reached:
            if ( t >= tf ):
                integrate = False

        dtnew = dt * fac / err**(1. / (1. + q))
        # Check stepsize is adequately bounded:
        if ( abs(dtnew) > facmax * abs(dt) ):
            dt = facmax * dt
        elif ( abs(dtnew) < facmin * abs(dt) ):
            dt = facmin * dt
        elif ( dtnew / (tf - t0) < 1e-12 ):
            dt = (tf - t0) * 1e-12
        else:
            dt = dtnew

        # Correct overshooting:
        if ( t + dt > tf ):
            dt = tf - t

    return sol_time[0:count], sol_state[0:count,:]
```

7 The Three- and *N*-Body Problems

Overview. So far, the code that Alice and Bob developed works for just two bodies. They think the time is right to extend their code to handle an arbitrary number of bodies. Converting the two-body propagator to an N-body simulator only requires modifying the routine implementing the equations of motion. The numerical integrators can handle the extended system without modifying the code. Conservation laws still hold and prove useful for analyzing the solution to generic N-body problems. As illustrative examples, our protagonists solve the Pythagorean three-body problem and some N-body choreographies. Finally, they propagate the Solar System for 1,000 years.

After taking a break for a few days to work on other class assignments, Alice and Bob meet to continue where they left off, namely adapting their two-body integrator to handle additional bodies.

Bob: We're definitely ready to go beyond the two-body problem. Let's do this!

7.1 From the Two-Body to the Three-Body Problem

Alice: The obvious choice for a first step is, as you might have guessed, the *three*-body problem. This involves the addition of only one more body into the mix. But, simple as it may seem, we've been taught in class that the three-body problem has been plaguing researchers for centuries. The general three-body problem has no useful analytic solution. Given a set of generic initial conditions, it is typically not possible to use pen-and-paper calculations to compute the positions and velocities of the particles at any future time. Computers are absolutely necessary.

Bob: Sounds like my kind of problem!

7.1.1 Equations of Motion for the Three-Body Problem

Alice: For starters, we have to update the implementation of the equations of motion to handle three bodies. The function `ode_two_body_first_order` that we implemented in snippet 4.1 can only handle two bodies.

Bob: We just have to implement equation (2.16) but setting $N = 3$, right?

Alice: Yes, but we want this equation to have the form of a first-order system, like the one we had in equation (4.1). With three bodies, our state vector now becomes $\mathbf{x}(t) = [\mathbf{R}_1(t), \mathbf{R}_2(t), \mathbf{R}_3(t), \mathbf{V}_1(t), \mathbf{V}_2(t), \mathbf{V}_3(t)]$. Using a substitution of the general form $r_{ij} = \|\mathbf{R}_i - \mathbf{R}_j\|$, the right-hand side of the differential equations then reads

$$\mathbf{f} = \begin{bmatrix} \mathbf{V}_1 \\ \mathbf{V}_2 \\ \mathbf{V}_3 \\ -Gm_2\dfrac{(\mathbf{R}_1 - \mathbf{R}_2)}{r_{12}^3} - Gm_3\dfrac{(\mathbf{R}_1 - \mathbf{R}_3)}{r_{13}^3} \\ -Gm_1\dfrac{(\mathbf{R}_2 - \mathbf{R}_1)}{r_{21}^3} - Gm_3\dfrac{(\mathbf{R}_2 - \mathbf{R}_3)}{r_{23}^3} \\ -Gm_1\dfrac{(\mathbf{R}_3 - \mathbf{R}_1)}{r_{31}^3} - Gm_2\dfrac{(\mathbf{R}_3 - \mathbf{R}_2)}{r_{32}^3} \end{bmatrix}. \tag{7.1}$$

Bob: Due to the symmetry of the problem, we also know that in general, $r_{ij} = r_{ji}$. Right?

Alice: Yes, that's right. Making that substitution won't reduce the number of terms, so we can leave it as is for now. But, it is of course important to realize that there are fewer independent variables than it appears. So, when we are coding this up, technically we only have to calculate each independent variable once at each time step. Good catch.

Bob: Next, we have to implement this equation in PYTHON. I wrote the new function in snippet 7.1.

Snippet 7.1
Function `ode_three_body_first_order`: Differential equations of the three-body problem.

```
def ode_three_body_first_order( x, t, params ):

    # Retrieve parameters:
    G = params['G']
    masses = params['masses']

    # Retrieve position vectors:
    R1 = x[0:3]
    R2 = x[3:6]
    R3 = x[6:9]

    # Compute relative separation:
    r21 = np.linalg.norm(R2 - R1)**3
    r12 = r21
    r31 = np.linalg.norm(R3 - R1)**3
    r13 = r31
    r32 = np.linalg.norm(R3 - R2)**3
```

```
    r23 = r32

    # Differential equations:
    # - Velocity:
    dxdt[0:3] = x[9:12]
    dxdt[3:6] = x[12:15]
    dxdt[6:9] = x[15:18]
    # - Acceleration:
    dxdt[9:12] = -G * masses[1] * (R1 - R2) / r12 - G * masses[2] * (R1 -
        R3) / r13
    dxdt[12:15]= -G * masses[0] * (R2 - R1) / r21 - G * masses[2] * (R2 -
        R3) / r23
    dxdt[15:18]= -G * masses[0] * (R3 - R1) / r31 - G * masses[1] * (R3 -
        R2) / r32
    return dxdt
```

Alice: Now it's just a matter of updating our code to call `ode_three_body_first_order` instead of `ode_two_body_first_order`. Argh, that's actually going to be a bit annoying because we will have to change it again when we decide to integrate $N > 3$ bodies!

Bob: Wait, not necessarily. I just thought of a neat trick. Instead of hard-coding the name of the function that we want to integrate, we can treat the function as an input variable. Then, we will call the integrator passing a pointer to the function that we want to integrate. We only have to make a single change to replace all instances of `ode_two_body_first_order` in the integrators with a generic name, say `ode`, and update the interfaces to

```
def integrate_euler( x0, t0, tf, dt, ode, params ):
    ...
def integrate_adams_bashforth( x0, t0, tf, dt, ode, params ):
    ...
def integrate_leapfrog( x0, t0, tf, dt, ode, params ):
    ...
def integrate_runge_kutta( x0, t0, tf, dt, ode, params ):
    ...
def integrate_rk_embedded( x0, t0, tf, atol, rtol, ode, order, params ):
    ...
```

Then, to call the integrator, we can simply do the following:

```
# Define function to integrate:
ode = ode_three_body_first_order

# Call integrator pointing to the function to integrate:
sol_time, sol_state = integrate_euler(x, t0, tf, dt, ode, params)
```

Alice: Good job, Bob! That's a nice solution. Actually, thanks to this, we can now use our integrators for other problems; it's just a matter of writing the corresponding force function. I have an assignment for classical mechanics where I have to simulate the motion of a double pendulum. This code will be very useful.

Bob: Why don't we stop by Prof. Starmover's office and see what test problems we should work on?

Alice: Sure, I want to start plotting N-body trajectories as soon as possible!

7.1.2 A Figure-8 with Three Bodies

Alice and Bob arrive at Prof. Starmover's office and find her talking about an upcoming conference with two graduate students.

Professor: Hi Alice, Bob, please come in. We were just finishing.

Alice and Bob come in and grab a couple of chairs, waving at the graduate students as they leave the office.

Alice: We just implemented the equations of motion for the three-body problem and we are ready to test our code with some examples. What would you recommend?

Professor: You will like this one: there is a special periodic solution in which the three bodies describe a figure-8!

Alice: Wow, really?

Professor: All three particles have the same mass and follow the same orbit carving out a perfect figure-8 pattern. The solution has zero total angular momentum.

Bob: That sounds like a fragile orbital configuration, though. I mean, what if one of the particles gets a slight nudge? Won't that launch all three bodies into chaos?

Professor: How "fragile" the orbits are depends on the stability properties of each solution. But, yes, three-body orbits are in general very sensitive. The initial positions and velocities of the particles must be chosen perfectly to ensure the long-term stability of the figure-8 orbital configuration. Otherwise, one of two things will occur: either one of the masses will escape, or two of the particles will undergo a direct collision if the particles have finite sizes.

Bob: So, how exactly is this done? How do we avoid these unstable solutions, in order to identify periodic solutions that remain stable indefinitely?

Professor: The figure-8 solution to the general three-body problem was calculated using computers by Christopher Moore in 1993 (Moore, 1993). Consider three particles of equal mass orbiting one another in a plane. As the particles move, they draw out a braid of three strands in three-dimensional space-time. The strands wind around each other as the particles complete their orbits. If the solution turns out to be periodic, the particles will eventually pass through their initial positions with the same initial velocities. We can then ask: what functional forms of the gravitational potential will actually yield periodic solutions? Moore was able to classify the periodic solutions he found computationally using this simple parameterization for the gravitational potential.

Bob: Good for Moore. He sounds like a smart guy. But what does this have to do with us?

Professor: Moore's work provides us with a nice simple set of initial positions and velocities we can pass to our *N*-body integrator, to check that it's working correctly. If the integrator is doing its job, then we'll see a nice figure-8 orbital configuration that remains stable for long periods of time. If not, then we'll most likely find that one of the particles is ejected from the system before the simulation is done. You can find the explicit initial conditions for a planar configuration of the figure-8 orbit in the paper by Chenciner and Montgomery (2000).

Bob: Okay, this sounds like a good example problem to start with, since we have a theoretical expectation for its time evolution. Let's try to simulate it.

Alice: The basic framework for the new code will be very similar to before. First, we initialize the problem by defining the initial positions and velocities of all three particles. Second, we call on the integrator to propagate the particles forward through time and space. Third, we plot the trajectories of the particles. As before, we can create a function in which we define the initial conditions and physical parameters. For the latter, we'll use the structure `params`. According to Chenciner and Montgomery (2000), we have:

Snippet 7.2
Initial conditions of the figure-8 three-body orbit.

```
def initialize_problem():

    # Number of bodies:
    nbodies = 3

    # The initial conditions:
    R1 = np.array([0.9700436, -0.24308753, 0.0])
    V1 = np.array([0.466203685, 0.43236573, 0.0])
    R2 = np.array([-0.9700436, 0.24308753, 0.0])
    V2 = np.array([0.466203685, 0.43236573, 0.0])
    R3 = np.array([0.0, 0.0, 0.0])
    V3 = np.array([-0.93240737, -0.86473146, 0.0])

    # Store in state vector:
    x = np.concatenate((R1, R2, R3, V1, V2, V3))

    # Mass of the bodies:
    masses = [1.0, 1.0, 1.0]

    # Gravitational constant:
    G = 1.0

    # Final time:
    t0 = 0.0
    tf = 10.0

    # Time step:
```

```
    dt = 0.0001

    # Tolerances:
    atol = x * 0.0 + 1e-8
    rtol = atol

    # Create parameters structure:
    params['G'] = G
    params['masses'] = masses
    params['nbodies'] = nbodies

    # Output:
    return x, t0, tf, dt, params
```

I set the absolute and relative tolerances to $\varepsilon = 10^{-8}$ for the variable step-size integrators.

Bob: We also need to update our plotting function in snippet 4.4 because it expects only two bodies. I'm going to write it in such a way that it handles any number of bodies, since we don't want to have to rewrite it in a few minutes when we have $N > 3$. I improved the function a bit so that it also plots the points where the particles started from. Oh, and I added labels to the axes.

Snippet 7.3

Function `plot_trajectory`: Updated version to handle N bodies.

```
def plot_trajectory( sol_state, params ):
    fig = plt.figure()
    ax = fig.add_subplot(111, aspect='equal')
    # Plot trajectory and initial positions with the same color:
    for ibody in range(0, params['nbodies']):
        traj = ax.plot(sol_state[:, ibody * 3], sol_state[:,1 + ibody *
            3])
        ax.plot(sol_state[0, ibody*3], sol_state[0, 1+ibody*3], "o",\
            color=traj[0].get_color())

    ax.set_xlabel('X')
    ax.set_ylabel('Y')
    plt.show()
```

Alice: Nice. Let me put the pieces together using our RKF4(5) integrator:

```
# Define function to integrate:
ode = ode_three_body_first_order

# The initial conditions and parameters:
x, t0, tf, dt, atol, rtol, params = initialize_problem()

# Integrate orbits:
sol_time, sol_state = integrate_rk_embedded(x, t0, tf, atol, rtol, ode,
    45, params)

# Plot trajectories:
plot_trajectory(sol_state, params)
```

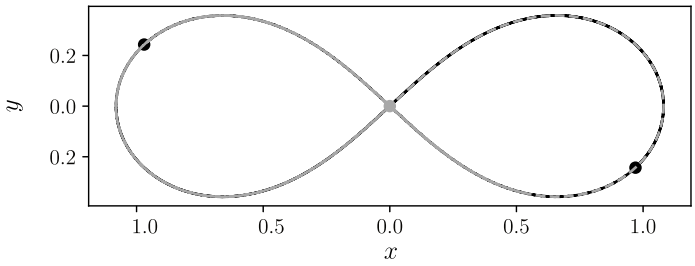


Figure 7.1
The figure-8 orbital configuration.

Bob: Whoa! It worked, look at figure 7.1! That's a figure-8, all right. In fact, I think it looks more like an infinity symbol! It's so stable that we only see one of the trajectories because all three lines match exactly.

Alice: You're right! This means that our implementation is correct, doesn't it?

Professor: It does!

Alice: What happens if we increase the total integration time to, say, 100?

Bob: Let's try it….

Alice: Nothing changed! It looks exactly like before. So that's telling us that this solution is really very stable, over long integration times, right?

Bob: You got it! Pretty cool.

Alice: We should probably give some credit to Christopher Moore and Chenciner and Montgomery.

Bob: Fair enough. They did discover the solution and provided the initial conditions, after all! But the take-away point from all this is that the integrator we have written is working and doing what it's supposed to be doing. And, not to mention, we've written the whole thing in such a way that it's a very simple matter to add in more particles. Before doing that, are there any other interesting three-body configurations we could try out, to continue testing our integrator?

7.1.3 The Pythagorean Three-Body Problem

Professor: Oh, there are many. One that is particularly interesting for understanding how sensitive the three-body problem is to the choice of initial conditions and the integration error is the Pythagorean Problem. In the early 1910s, a gentleman by the name of Burrau set out to calculate by hand the trajectories of all three particles for a very specific set of initial conditions (Burrau, 1913). By hand! He chose to initially place each particle at the vertex of a right-angled triangle. The particles have masses of 3, 4, and 5, in arbitrary units,

and $G = 1$. The length of the opposing side of the triangle to each vertex is chosen to be proportional to the corresponding particle mass.

Burrau then released all three particles from rest (zero velocity) at the exact same time and computed their trajectories by hand. He found that two particles first undergo a close approach and then recede. A second pair of particles then undergo a close approach and subsequently recede. And so on. The time evolution of the system continues in this fashion for a while. Burrau reached the end of his computing capacity before reaching the end of the interaction. Had he continued on, he would've found that eventually one of the particles is permanently ejected and becomes unbound, leaving behind a close binary pair. Szebehely and Peters (1967) provided a very detailed analysis of the complete solution beyond the escape time. You should compare your solution against the one presented in that paper to make sure you get the right result.

Alice: Let me implement the initial conditions first:

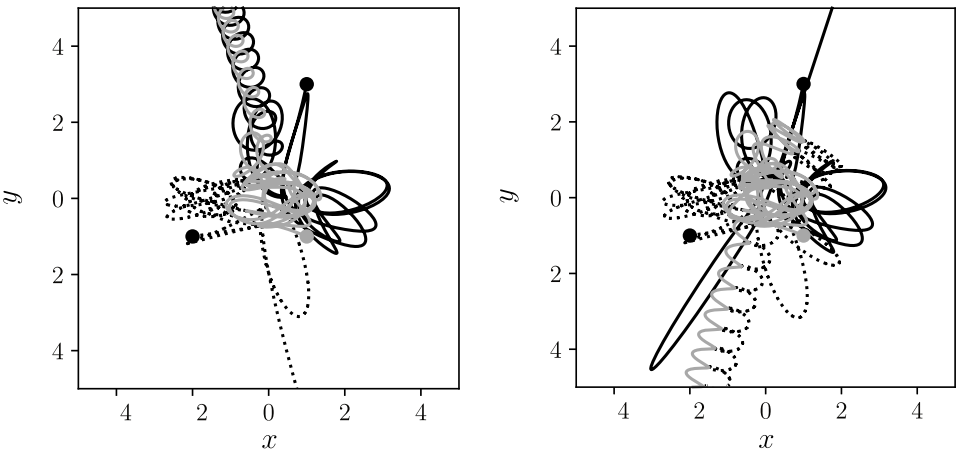
```
 # Positions:
R1 = np.array([1.0, 3.0, 0.0])
R2 = np.array([-2.0, -1.0, 0.0])
R3 = np.array([1.0, -1.0, 0.0])
# Velocities:
V1 = np.array([0.0, 0.0, 0.0])
V2 = np.array([0.0, 0.0, 0.0])
V3 = np.array([0.0, 0.0, 0.0])
# State vector:
x = np.concatenate((R1, R2, R3, V1, V2, V3))
# Mass of the bodies:
masses = [3.0, 4.0, 5.0]
# Final time:
t0 = 0.0
tf = 70.0
```

Bob: Thanks, Alice. Now we can run our script again and we get figure 7.2a. Wait … that doesn't look like the solution reported by Szebehely and Peters (1967). Maybe it's an issue with the tolerance?

Alice: Could be. Let me try again with $\varepsilon = 10^{-13}$ to be sure. Aha! Figure 7.2b nails the solution!

Bob: The orbits are pretty messy. I guess the problem is so sensitive that the numerical errors that we got with $\varepsilon = 10^{-8}$ completely messed up the result.

Professor: That's the reason why I wanted to introduce you to the Pythagorean Problem. The problem is very sensitive and the solution is available in the literature so that you can check if you are doing things right.



(a) $\varepsilon = 10^{-8}$ **(b)** $\varepsilon = 10^{-13}$

Figure 7.2
Solution to the Pythagorean three-body problem.

7.2 The *N*-Body Problem

Alice: I love these orbits. They look so cool!

Bob: Imagine what N bodies will look like! Should we move on to the general N-body problem? I cannot wait!

Professor: I think you should. It shouldn't be hard.

7.2.1 Equations of Motion for the *N*-Body Problem

Alice: Okay, first we need to implement the equations of motion (2.16) for a generic number of bodies N. We need a loop to march through all bodies computing their accelerations and an inner loop to compute the sum over all bodies. And we have to be careful not to compute the denominators $r_{jj} = 0$ because that would break the equations!

Bob: Good catch.

Professor: When dealing with generic N-body problems, it is common to include *test particles*. They are bodies with zero mass, meaning that they do not contribute to the acceleration acting on other bodies.

Bob: Oh, so I guess we could also skip these test particles from the computation, right? I mean, they would not contribute to the acceleration.

Alice: Sure. What do you think about snippet 7.4? Did I miss anything?

Snippet 7.4
Function `ode_n_body_first_order`: Differential equations for the N-body problem.

```
def ode_n_body_first_order( x, t, params ):

    # Retrieve parameters:
    G = params['G']
    masses = params['masses']
    nbodies = params['nbodies']

    # Allocate right-hand side:
    dxdt = x * 0.0

    # Differential equations:
    for j in range(0, nbodies):
        # - Velocities:
        dxdt[j * 3: (j + 1) * 3] = x[(nbodies + j) * 3: (nbodies + j + 1)
            * 3]
        # - Accelerations:
        Rj = x[j * 3: (j + 1) * 3]
        aj = Rj * 0
        for k in range(0, nbodies):
            # Skip equal indices:
            if ( j == k ):
                continue
            # Skip test particles:
            if ( masses[k] == 0 ):
                continue
            Rk = x[k * 3 : (k + 1) * 3]
            rel_sep = Rj - Rk
            aj += -G * masses[k] * rel_sep / np.linalg.norm(rel_sep)**3
        dxdt[(nbodies + j) * 3: (nbodies + j + 1) * 3] = aj

    return dxdt
```

Bob: Let me see…. Looks good to me! The tricky part was how to access the slices of the state vector to get the positions and velocities of the bodies. But I checked the indices and they seem fine.

Alice: We should now point to our new force function before calling the integration function, like

```
ode = ode_n_body_first_order
```

To begin with, we can run again the three-body examples just by setting `nbodies=3`. If our implementation is right, we should get the same exact results as before.

Bob: Give me one second…. Yes! I reproduced both the figure-8 orbit and the solution to the Pythagorean Problem. This is definitely promising!

7.2.2 Double Figure-8 with Five Bodies

Professor: I know you liked the figure-8 solution. But what if I told you that there are also double figure-8 orbits with $N = 5$? You can find the initial conditions in this paper by Simó (2001).

Alice: That's cool! Okay, I modified the content of our function `initialize_problem` to include the initial conditions in the paper. I also set `nbodies=5` inside the function, which I think makes more sense. You can check the code in snippet 7.5.

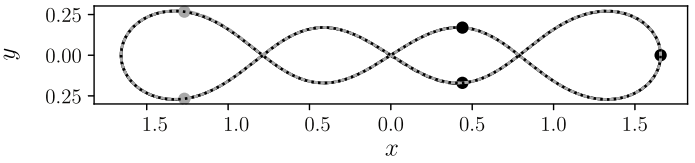
Snippet 7.5
Initial conditions of the double figure-8 problem with five bodies.

```
nbodies = 5
x = np.zeros(nbodies * 6)
# Positions:
x[0:3]   = np.array([1.657666, 0.0, 0.0])
x[3:6]   = np.array([0.439775, -0.169717, 0.0])
x[6:9]   = np.array([-1.268608, -0.267651 ,0.0])
x[9:12]  = np.array([-1.268608, 0.267651, 0.0])
x[12:15] = np.array([0.439775, 0.169717, 0.0])
# Velocities:
x[15:18] = np.array([0.0, -0.593786, 0.0])
x[18:21] = np.array([1.822785, 0.128248, 0.0])
x[21:24] = np.array([1.271564, 0.168645, 0.0])
x[24:27] = np.array([-1.271564, 0.168645, 0.0])
x[27:30] = np.array([-1.822785, 0.128248, 0.0])
# Masses of the bodies:
masses = [1.0, 1.0, 1.0, 1.0, 1.0]
# Gravitational parameter:
G = 1.0
# Final time:
t0 = 0.0
tf = 6.3
```

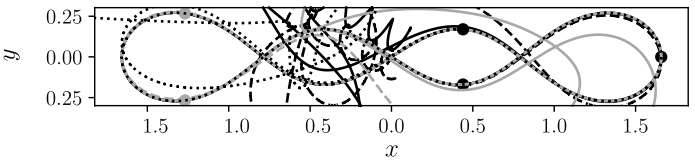
Bob: The main program is simply:

```
# Define function to integrate:
ode = ode_n_body_first_order

# The initial conditions
x, t0, tf, dt, atol, rtol, params = initialize_problem()

# Integrate orbits:
sol_time, sol_state = integrate_rk_embedded(x, t0, tf, atol, rtol, ode,
    45, params)

# Plot trajectories
plot_trajectory(sol_state, params)
```

And there we go! I ran the simulation with $\varepsilon = 10^{-10}$ and plotted the result in Figure 7.3a.



(a) $t_f = 6.3$



(b) $t_f = 15$

Figure 7.3
Double figure-8 orbits with five bodies.

Alice: That looks awesome! It's a double figure-8! Let's increase the integration time to see how stable it is; say we set the final time to 15. The solution is plotted in figure 7.3b. Well, no question that's not stable. It looks like a fine piece of modern art, though!

Professor: Yes, the more bodies you add, the more sensitive the problem becomes. In addition, the initial conditions that you got provide just a few significant figures, which limits the stability of the orbit. In any case, propagating this orbit is more challenging than the three-body one.

Bob: Phew, I thought it was a problem with our integrator!

7.2.3 Propagating the Solar System for 1,000 Years

Professor: Your integrator is in pretty good shape. Anyway, you ran enough simple test cases, don't you think? What do you say we try to propagate the Solar System for, say, 1,000 years?

Bob: Finally! A real problem!

Alice: What makes this such an exciting and real problem?

Bob: The Earth could get ejected from the Solar System or collide with another planet? I'd say that'd be a pretty big problem for us.

Alice: All right, fair enough. I'm all in. Let's do this.

Professor: The first thing you need are the initial conditions. NASA's Jet Propulsion Laboratory maintains the HORIZONS system,[1] with a convenient web interface that provides the position and velocity of any solar-system body at a given epoch. Take a look at the website and get the states of Mercury through Neptune with respect to the Solar System Barycenter, relative to the ecliptic plane. We'll assign them ids 1 through 8. We'll need the Sun too (identified as body 0), but we'll treat it in a special way. That's $N = 9$ bodies in total.

Bob: I'm checking the website. I set the date to 2017-Jun-22 and got the states of the planets. I wrote them in snippet 7.6. The units I used are astronomical units (AU) for the positions and astronomical units per day (AU/d) for the velocities.

Snippet 7.6

Initial conditions of the planets in the Solar System relative to the Solar System Barycenter.

```
nbodies = 9
x = np.zeros(nbodies * 6)
# Positions (AU):
x[3:6]   = np.array([-1.202266214811173E-02,  3.129738317339329E-01,
    2.637608427362013E-02]) # Mercury
x[6:9]   = np.array([6.036656393784035E-01, -4.041148416177896E-01,
    -4.043024664738540E-02])  # Venus
x[9:12]  = np.array([1.241238601620544E-02, -1.011219129247450E+00,
    -9.819526634120970E-05])  # Earth
x[12:15] = np.array([-4.926800380968982E-01,  1.537007440322637E+00,
    4.411949077995760E-02]) # Mars
x[15:18] = np.array([-4.989446787630805E+00, -2.184763688656508E+00,
    1.206579440062781E-01]) # Jupiter
x[18:21] = np.array([-9.681157061545530E-01, -1.000423489517810E+01,
    2.124749402062057E-01])  # Saturn
x[21:24] = np.array([1.806198902787260E+01,  8.416356280190394E+00,
    -2.027374282309190E-01])  # Uranus
x[24:27] = np.array([2.850592355224314E+01, -9.173827312094703E+00,
    -4.680305005676293E-01])  # Neptune
# Velocities (AU/d):
x[30:33] = np.array([-3.374999571504927E-02, -3.223139133573000E-04,
    3.069097338939302E-03])# Mercury
x[33:36] = np.array([1.125511509001221E-02,  1.664344623604423E-02,
    -4.214323452340928E-04])# Venus
x[36:39] = np.array([1.692619974976791E-02,  1.014868174374918E-04,
    -6.047160617924358E-08])# Earth
x[39:42] = np.array([-1.278895103122624E-02, -3.109871472361932E-03,
    2.485576543075108E-04])# Mars
x[42:45] = np.array([2.938669025252137E-03, -6.553859846840747E-03,
    -3.848907559519295E-05])# Jupiter
x[45:48] = np.array([5.246396731312688E-03, -5.546463665223218E-04,
    -1.994279405146075E-04])# Saturn
```

[1] https://ssd.jpl.nasa.gov/?horizons

```
x[48:51] = np.array([-1.689894219169289E-03,  3.381692838015134E-03,
    3.446267721267768E-05])# Uranus
x[51:54] = np.array([9.407596025584859E-04,  3.006460319698939E-03,
    -8.395033744165947E-05])# Neptune
```

Professor: The next thing you need is the masses of the planets, written in solar masses ($M_\odot$). You can find the information online easily.

Alice: Okay, so $m_0 = 1\ M_\odot$ is the mass of the Sun in solar masses. And the rest of the masses are not hard to find:

```
# Masses of the Solar System bodies (Solar masses):
masses = [1., 1.66051140935277e-07, 2.44827371182131e-06, 3.00329789031573
    e-06, 3.22773848604808e-07, 0.000954532562518104, 0.00028579654259599,
     4.3655207025844e-05, 5.1499991953912e-05]
```

Professor: Exactly. As for the Sun, we will compute its position and velocity in such a way that the center of mass of the system is fixed at the origin. This is only optional, since we could have obtained its initial position and velocity vectors in the same way as for the other bodies, but this will simplify the result because the center of mass will not drift.

Bob: From equation (2.50) starting at $i = 0$ for the Sun, we can write

$$\mathbf{r}_{\rm CM} = \frac{\sum_{i=0}^{N-1} m_i \mathbf{R}_i}{M} = \mathbf{0} \implies \mathbf{R}_0 = -\frac{\sum_{i=1}^{N-1} m_i \mathbf{R}_i}{m_0}. \tag{7.2}$$

Alice: Yeah, and $m_0 = 1$. Don't forget that.

Bob: That's right! We can do the same with the velocity and arrive at:

```
# Compute position and velocity of the Sun to fix the c.o.m. at the origin
    :
for ibod in range(1, nbodies):
    x[0:3] -= masses[ibod] * x[ibod * 3: (ibod + 1) * 3]
    x[nbodies * 3: (nbodies + 1) * 3] -= masses[ibod] * x[(nbodies + ibod)
         * 3: (nbodies + ibod + 1) * 3]
```

Professor: Finally, you have to be careful with the units of the gravitational constant, G, and the time. So far, you have used astronomical units as units of length, days as units of time, and solar masses as units of mass. What value of G should you use then?

Alice: Let's see … I know that $G = 6.67408 \times 10^{-20}\ \mathrm{km^3 s^{-2} kg^{-1}}$, but we need to convert it to $\mathrm{AU^3 d^{-2} } M_\odot^{-1}$. It will be something like:

$$\begin{aligned} G &= 6.67408 \times 10^{-20} \left(\frac{86400^2 \times 1.98847541596653 \times 10^{30}}{149597870.7^3} \right) \frac{\mathrm{AU}^3}{\mathrm{d}^2 M_\odot} \\ &= 2.9591220828559 \times 10^{-4}\ \frac{\mathrm{AU}^3}{\mathrm{d}^2 M_\odot}. \end{aligned} \tag{7.3}$$

Bob: And we want to integrate the orbits for 1,000 years but we are using units of days. In the end, we have:

```
# Gravitational constant (AU^3/d^2/M_sun):
G  = 0.00029591220828559
# Final time (d):
t0 = 0.0
tf = 365.25 * 1000
```

Time to propagate! I'll use $\varepsilon = 10^{-10}$ for the tolerance like before. Okay, I plotted the orbits in figure 7.4a. This is exciting!

Alice: The orbits look okay, but let's make sure they're right. The orbits of Neptune, Uranus, Saturn, and Jupiter are easy to distinguish and their orbital distances from the Sun seem to be approximately 30, 20, 10, and 5 AU. Those are indeed the orbital separations of the planets!

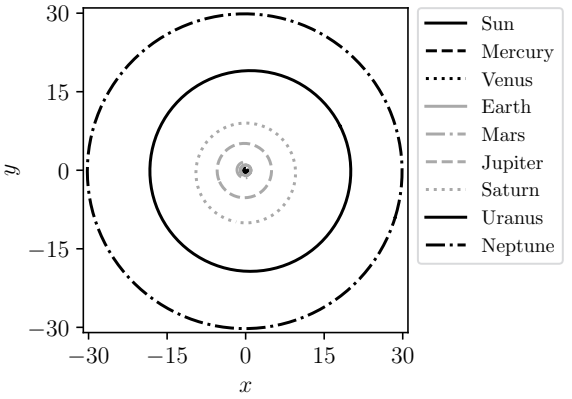
Bob: Whoa, it's amazing to see that the results from our simulations reproduce the real dynamics.

Alice: Definitely. However, this check was only a qualitative estimate. We should check the error in the conservation of the energy to be completely sure.

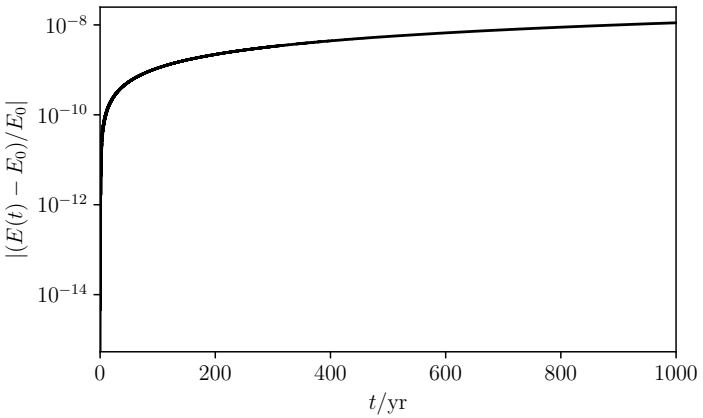
Bob: Wait, how will we compute the energy for N bodies? The function that we had before (snippet 4.5) is only valid for two bodies.

Alice: Hmmm … well, that's not hard. We just need a couple of loops similar to what we did before to compute the accelerations:

```
def compute_energy( sol_state, params ):

    # Retrieve parameters:
    G = params['G']
    masses = params['masses']
    nbodies = params['nbodies']

    # Allocate output:
    energy = sol_state[:, 0] * 0

    count = 0
    for x in sol_state:
        # Compute energy:
        energy[count] = 0
        for i in range(0, nbodies):
            energy[count] += 0.5 * masses[i] \
                * np.linalg.norm(x[(nbodies+i) * 3: (nbodies+1+i) * 3])**2
            for j in range(0, nbodies):
                if ( i == j ): continue
                energy[count] -= 0.5 * G * masses[i] * masses[j] \
                    / np.linalg.norm(x[i*3: (i+1)*3] - x[j*3: (j+1)*3])
        count += 1
    return energy
```



(a) Orbits



(b) Relative energy error

Figure 7.4
Integration of the Solar System for 1,000 years with RKF4(5) and $\varepsilon = 10^{-10}$.

Bob: I see. Now we can finally integrate the problem and check for conservation of the energy! Take a look at figure 7.4b.

Alice: Well…. That doesn't look too good, does it? It's not that the initial error is itself large, but the problem is that it seems to grow continuously over time. The moment we decide to integrate longer time spans, we will most certainly run into trouble with additional error accumulation!

Professor: Good observation, Alice. Although the variable step-size integrators that you have right now are very versatile, there are more accurate integrators for propagating plan-

etary systems. But let's leave that discussion for next week after you digest all these new results.

Bob: Sounds good, I could definitely use a break. See you next week, Prof. Starmover!

Alice: Goodbye, and thanks for the help!

Professor: Sure. Have a good evening.

8 Gauss–Radau Integrator of the Fifteenth Order

Overview. Not satisfied with the performance of their code, Alice and Bob start looking into more advanced methods, backed by Prof. Starmover. High-order integration schemes specifically tailored to orbital motion exhibit an exceptional performance when propagating N-body configurations. In particular, Prof. Starmover presents the Gauss–Radau integration method of the fifteenth order proposed by Everhart (1985) and that resulted in the state-of-the-art IAS15 algorithm developed by Rein and Spiegel (2014). Alice and Bob test the integrator by propagating the evolution of the Solar System in the next 1,000 years. Prof. Starmover explains the algorithm and its implementation in detail, discussing other possible discretization sequences like the Gauss–Lobatto and Gauss–Legendre spacings. A code review is given at the end of the chapter.

Alice and Bob head over to Prof. Starmover's office to inquire further about higher-order integrations.

Bob: How are you, Prof. Starmover? How's it going?

Alice: Hi, Prof. Starmover! We wanted to learn more about performing higher-order integrations and any other techniques you think might be relevant in trying to implement them reliably….

Professor: Great! I completely agree. You are now familiar with how numerical integration methods work, and I think it is time for you to start working on more advanced integrators. If you explore the literature, you will find tens of methods! The next integrator I want to refer you to is the Gauss–Radau method of the fifteenth order proposed by Everhart (1974, 1985). This integrator, further improved by Rein and Spiegel (2014), is widely used in the astronomical community for real applications.

Bob: Sounds good. Is this just a special Runge–Kutta method? If so, where can we find the Butcher tables?

Professor: I am afraid this method requires a bit more work than just implementing new sets of coefficients. But don't lose the spirit!

8.1 Preparing the Second-Order System

Professor: For starters, we will write our integrator to handle second-order systems directly, like we did with the leapfrog method in section 5.4. Let's recover equation (5.29):

$$\ddot{\mathbf{y}} = \mathbf{F}(t, \mathbf{y}). \tag{8.1}$$

where $\mathbf{y}(t) = [\mathbf{R}_1(t), \mathbf{R}_2(t), \ldots, \mathbf{R}_N(t)]$ corresponds to the position vectors of the N bodies and $\mathbf{F}(t, \mathbf{y})$ provides the acceleration $\mathbf{a}_i$ acting on each body.

Alice: Yes, I remember this. For N bodies, the right-hand side will be:

$$\mathbf{F}(t, \mathbf{y}) = [\mathbf{a}_1(\mathbf{y}(t)), \mathbf{a}_2(\mathbf{y}(t)), \ldots, \mathbf{a}_N(\mathbf{y}(t))], \qquad \text{with} \quad \mathbf{a}_i = -\sum_{\substack{j=1 \\ j \neq i}}^{N} G m_j \frac{\mathbf{R}_i - \mathbf{R}_j}{\|\mathbf{R}_i - \mathbf{R}_j\|^3}, \tag{8.2}$$

and $\mathbf{a}_i$ denotes the acceleration experienced by body i.

Bob: In PYTHON this will end up looking something like:

Snippet 8.1
Function `ode_n_body_second_order`: Evaluate the gravitational acceleration from N bodies.

```
def ode_n_body_second_order( t, y, dy, params ):

    # Retrieve parameters:
    G = params['G']
    masses = params['masses']
    nbodies = params['nbodies']

    # Allocate acceleration:
    acc = y * 0.0

    # Differential equations:
    for j in range(0, nbodies):
        Rj = y[j * 3: 3 + j * 3]
        aj = Rj * 0
        for k in range(0, nbodies):
            if ( j == k ):
                continue
            if ( masses[k] == 0 ):
                continue
            Rk = y[k * 3: 3 + k * 3]
            rel_sep = Rj - Rk
            aj += -G * masses[k] * (rel_sep) / np.linalg.norm(rel_sep)**3
        acc[j * 3: 3 + j * 3] = aj

    return acc
```

To save time, I am not computing the contribution of massless particles to the total acceleration.

Alice: Wait, Bob. Why are you passing the velocity dy as an input? You are not using it in the function!

Bob: My bad, let me get it out….

Professor: Actually, it is a good thing that you are passing the velocity as an input. Even though the gravitational terms do not require the velocity, it is a good practice to make it available in case we decide to add other perturbations later on. There are certain forces that *do* depend on the velocities. Like, for example, relativistic corrections.

Bob: Ha, that's absolutely the reason why I decided to include it.

Alice: Oh, yeah. Sure it was….

8.2 Numerical Quadratures: Choosing the Right Sequence

Professor: To solve the problem, we need to integrate equation (8.1) twice. The first integration will give us the velocity, and integrating again will yield the position.

$$\dot{\mathbf{y}}(t) = \int_0^t \mathbf{F}\,\mathrm{d}\tau \equiv \mathbf{G}; \tag{8.3a}$$

$$\mathbf{y}(t) = \int_0^t \mathbf{G}\,\mathrm{d}\tau. \tag{8.3b}$$

If you think about the one-dimensional case, we can reduce this integral problem to numerically approximating a *quadrature*, which is just the area under the curve defined by the integrand. What is the simplest method that you know for evaluating a quadrature numerically?

Alice: The trapezoidal rule? That is pretty simple. The integral of $F(x)$ between a and b is approximated like

$$\int_a^b F(x)\,\mathrm{d}x \approx \frac{\Delta x}{2}[F(x_1) + 2F(x_2) + 2F(x_3) + \ldots + 2F(x_{n-1}) + F(x_n)]. \tag{8.4}$$

Professor: That's right. You evaluate $F(x)$ at a series of nodes x_i that are evenly distributed between a and b. That makes $x_i = a + (i-1)\Delta x$ and $\Delta x = (b-a)/(n-1)$.

As an example, assume that we want to compute the integral of

$$I = \int_{-1}^{1} F(x)\,\mathrm{d}x, \qquad \text{with} \quad F(x) = 3x^2 \sin(x+1). \tag{8.5}$$

First, we can compute the solution analytically. From

$$f(x) = \int F(x)\,\mathrm{d}x = 6x\sin(x+1) - 3(x^2-2)\cos(x+1), \tag{8.6}$$

it follows that

$$I = f(1) - f(-1) = 1.20734405131266. \tag{8.7}$$

Let's now try to arrive at the same result numerically, starting with the trapezoidal rule with $n = 2$. Can you try it?

Alice: It will be:

$$I_2 = \int_{-1}^{1} F(x)\,\mathrm{d}x \approx [F(1) + F(-1)] = 2.727892. \tag{8.8}$$

The result has nothing to do with the actual solution….

Bob: Let me try $n = 4$. I got

$$I_4 = 1.26269913892191. \tag{8.9}$$

Okay, that's a bit better. Let's give $n = 100$ a try:

$$I_{100} = 1.20738512194762. \tag{8.10}$$

We needed quite a lot of points for convergence! And the error is still of the order of 10^{-5}, which is not spectacular.

Professor: Indeed, the trapezoidal rule is not the most efficient method. In practice, this is telling us that trying to integrate a planetary system using the trapezoidal rule might not be the best choice. Before going on, do you agree that this expression,

$$\int_{-1}^{1} F(x)\,\mathrm{d}x \approx \sum_{i=1}^{n} w_i F(x_i), \tag{8.11}$$

is simply a generalization of equation (8.4) in $[-1, 1]$? This is called the Gaussian quadrature and w_i are the weights. It is originally defined in the interval $[-1, 1]$, but it can be generalized to any interval $[a, b]$ by setting

$$\int_{a}^{b} F(x)\,\mathrm{d}x \approx \frac{b-a}{2} \sum_{i=1}^{n} w_i F\left(\frac{b-a}{2} x_i + \frac{b+a}{2}\right). \tag{8.12}$$

Bob: Let me see…. Ah, yes, it's easy to check. It is just a matter of equating equations (8.4) and (8.11) term by term:

$$w_1 = w_n = \frac{\Delta x}{2}, \qquad w_i = \Delta x, \qquad \text{with } i = 2, \ldots, n-1 \quad \text{and} \quad \Delta x = \frac{2}{n-1}. \tag{8.13}$$

And the nodes are $x_i = -1 + (i-1)\Delta x = 2(i-1)/(n-1) - 1$. Why is equation (8.11) better?

Professor: Equation (8.11) tells us that we can approximate integrals by simply evaluating the integrand at certain nodes, multiplying the results by some weights, and summing them up. The trapezoidal rule uses a set of evenly spaced nodes. That is the first thing one would try, but it is not necessarily the best approach. Can you try equation (8.11) using the special

Table 8.1
Nodes and weights for the 5-point Gauss–Legendre quadrature.

i	x_i	w_i
1	–0.9061798459386640	0.2369268850561891
2	–0.5384693101056831	0.4786286704993665
3	0.0000000000000000	0.5688888888888889
4	0.5384693101056831	0.4786286704993665
5	0.9061798459386640	0.2369268850561891

set of nodes and weights in table 8.1? You will only need up to $n = 5$, not $n = 100$ like before.

Alice: Sure, it sounds a bit arbitrary to me, but let's see … I obtained

$$I = 1.2073438732418. \tag{8.14}$$

Wow, that's very close to the solution we were looking for; the relative error is about 10^{-7}! We got much closer than the trapezoidal rule with $n = 100$. And we only used $n = 5$! How did you do that?

Professor: I have to admit it wasn't me. This particular integration method is due to Gauss and Legendre (and they knew what they were doing!). The nodes in table 8.1 are the zeros of the Legendre polynomials $P_n(x)$, that is, the solutions to

$$P_n(x) = 0, \tag{8.15}$$

where n is the degree of the polynomial. That is why this integration method is often called Gauss–Legendre quadrature. Figure 8.1 shows the area under the curve $F(x)$, which is essentially the integral we were interested in computing, and the position of the Legendre nodes. We will talk later about how the coefficients are generated (see section 8.6). For now, you can find the tabulated values for different n in many sources, like Abramowitz and Stegun (1964, p. 916) or the NIST Digital Library of Mathematical Functions.[1]

Alice: I just took a look at the NIST Digital Library. They have a lot of information!

Professor: Yes, I definitely recommend that you keep it in mind whenever you need to use special functions.

Alice: Good to know!

Professor: You might have realized that the sequence in table 8.1 does not include the endpoints of the integration interval.

[1] https://dlmf.nist.gov/3.5#Px2

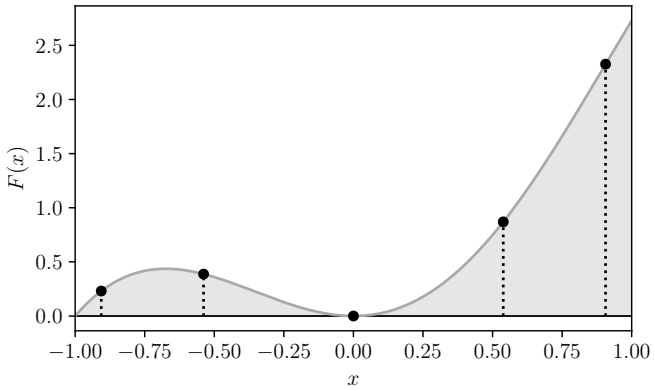


Figure 8.1
Area under the curve $F(x)$ and position of the five Legendre nodes in table 8.1.

Bob: Not really….

Professor: Recall that we are integrating the function over the interval $[-1, 1]$. But the first node is $-0.906\ldots$ and the last one $0.906\ldots$, meaning that we are not really evaluating $F(x)$ at -1 or 1. If we were to use a sequence that also includes the first point, we would end up with a Gauss–Radau sequence. The nodes in this case are the solutions to

$$P_{n-1}(x) + P_n(x) = 0. \tag{8.16}$$

And if we included the last point too, we would have a Gauss–Lobatto sequence. The nodes of the Gauss–Lobatto quadrature are the solutions to

$$xP_{n-1}(x) - P_{n-2}(x) = 0. \tag{8.17}$$

The coefficients are different from one sequence to the other, but the whole idea is the same. It's just a matter of using the right nodes.

Bob: Gauss–Radau! That's the name of the integrator we are implementing, isn't it?

Professor: Yes, Bob. As you might wonder, it is based on a Gauss–Radau sequence for evaluating quadratures.

Alice: I have one question, Prof. Starmover. Why did you choose $n = 5$ and not $n = 4$ or $n = 6$?

Professor: That's a very good question. How many points you choose determines the *order* of the method. Do you remember how the number of stages in equation (5.13) determined the order of the Runge–Kutta integrator? It's a similar concept.

There is a subtle but very important difference, though. The Runge–Kutta methods that we looked at are based on Taylor expansions of the solution. This means that if we trun-

cate the series at order p, then the error will be of the order of the first neglected term, $\mathcal{O}(\Delta t^{p+1})$. For the Gaussian quadrature with Legendre spacings, the series approximation in equation (8.11) is not a Taylor expansion, in the sense that if we retain only n terms, then the integration method is not of order $p = n$ but rather $p = 2n - 1$! The accuracy gain comes from how the spacings are defined. In practice, this means that if we use a Gaussian quadrature evaluating the integrand at n points, we can integrate exactly any polynomial up to degree $2n - 1$.

Say, for example, that we were to compute the integral

$$I = \int_{-1}^{1} P(x)\,\mathrm{d}x, \tag{8.18}$$

where $P(x)$ denotes the polynomial

$$P(x) = 1 - 2x + 3x^2 - 4x^3 + 5x^4 - 6x^5 + 7x^6 - 8x^7 + 9x^8 - 10x^9. \tag{8.19}$$

Bob: The primitive is pretty easy to compute,

$$p(x) = \int P(x)\,\mathrm{d}x = x - x^2 + x^3 - x^4 + x^5 - x^6 + x^7 - x^8 + x^9 - x^{10}, \tag{8.20}$$

and the exact result is

$$I = 10. \tag{8.21}$$

Professor: Indeed, thanks Bob. Can you compute it using the Gauss–Legendre method with five points?

Bob: Sure. If I take the coefficients in table 8.1, I arrive at

$$I_5 = 10.0000000000000. \tag{8.22}$$

This is the exact result! I made a plot of the polynomial and the location of the nodes in figure 8.2. Just for fun, I computed the solution using the trapezoidal rule with $n = 100$ and I got

$$I_{\mathrm{tr},100} = 10.0095209948002. \tag{8.23}$$

It's close but not nearly as good.

Professor: The Gauss–Legendre quadrature worked so well because the polynomial is of order $2 \times 5 - 1 = 9$. We can still integrate it exactly with $n = 5$. Can you please try the same thing with the following polynomial?

$$P(x) = 1 - 2x + 3x^2 - 4x^3 + 5x^4 - 6x^5 + 7x^6 - 8x^7 + 9x^8 - 10x^9 + 11x^{10}. \tag{8.24}$$

It's one order higher.

Bob: The exact solution is

$$I = 12, \tag{8.25}$$

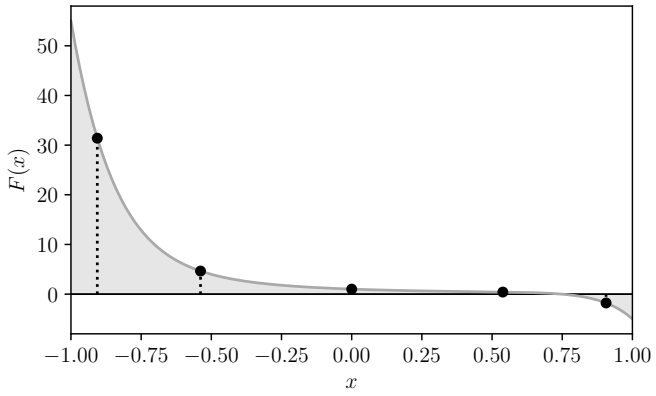


Figure 8.2
Area under the polynomial $P(x)$ and position of five Legendre nodes.

but the Gauss–Legendre method gives

$$I_5 = 11.967750062988. \tag{8.26}$$

Alice: I see now what you meant when you said that a Gaussian quadrature with n points can integrate polynomials of degree $2n - 1$ exactly. In these examples, we obtained the exact result for order 9, but for order 10, we started to see some error.

Bob: Why would you want to use the Gaussian quadrature to integrate a polynomial? It is easier to just compute the primitive and evaluate it at the extrema.

Professor: That's a very good observation, Bob, and you are completely right. If you know the polynomial that you want to integrate, you should evaluate the integral analytically.

Alice: If only the acceleration in the N-body problem were a polynomial…. Wouldn't it be nice? That would simplify everything! Computing the solution would be so easy, just an analytic expression.

8.3 Approximating the Integrand Using a Polynomial

Professor: Yes, it would! I am glad you brought that up because we can actually use the Gaussian quadrature to find the polynomial that provides the best approximation to the integral that we want to compute.

Alice: That'd be great!

Professor: One way to approximate a function is using its Taylor expansion about a reference point. That provides a good approximation around that point, and the accuracy gets

worse as we get farther away from the reference point. But in our case, we don't really want a good approximation about one point; we want the best possible approximation of an integral. The goal is to get an accurate estimate of the trajectory after advancing one time step. That's why the polynomial we seek is different from a Taylor expansion.

Going back to our simple example, our goal now is to find the polynomial

$$P(x) = \sum_{i=0}^{m} a_i x^i \tag{8.27}$$

that provides the best approximation of the integral

$$I = \int_{-1}^{1} F(x)\,\mathrm{d}x \approx \int_{-1}^{1} P(x)\,\mathrm{d}x. \tag{8.28}$$

Again, note that we don't want $P(x)$ to approximate $F(x)$ very well at some x_0. What we really want is to find the coefficients a_i that make the integrals in the expression above be as close to being truly equal as possible. Let's sample the function $F(x)$ at $n = 5$ points to obtain the linear system

$$\begin{bmatrix} F(x_1) \\ F(x_2) \\ F(x_3) \\ F(x_4) \\ F(x_5) \end{bmatrix} = \begin{bmatrix} 1 & x_1 & x_1^2 & x_1^3 & x_1^4 \\ 1 & x_2 & x_2^2 & x_2^3 & x_2^4 \\ 1 & x_3 & x_3^2 & x_3^3 & x_3^4 \\ 1 & x_4 & x_4^2 & x_4^3 & x_4^4 \\ 1 & x_5 & x_5^2 & x_5^3 & x_5^4 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \end{bmatrix}. \tag{8.29}$$

Any guess what the $x_1, \ldots, x_5$ points are?

Alice: The Legendre nodes in table 8.1?

Professor: Yes! Because we already know that they are the optimal points where to sample a function to approximate its integral.

Bob: I already solved for a_i while you were discussing that. I simply inverted the matrix in the equation above to obtain

$$a_0 = 0, \tag{8.30a}$$
$$a_1 = +0.06082368904274, \tag{8.30b}$$
$$a_2 = +2.50028200676281, \tag{8.30c}$$
$$a_3 = +1.33393098020382, \tag{8.30d}$$
$$a_4 = -1.14877699483351. \tag{8.30e}$$

Alice: Now that we have the polynomial approximation, evaluating its integral is very easy. Analytically, it is just

$$I \approx \left(a_0 x + \frac{a_1}{2}x^2 + \frac{a_2}{3}x^3 + \frac{a_3}{4}x^4 + \frac{a_4}{5}x^5\right)\Big|_{-1}^{1} = 1.2073438732418. \tag{8.31}$$

Look at that, it is exactly the result we obtained with the standard Gaussian quadrature in equation (8.14)!

Professor: This is because we are essentially doing the same thing: approximating an integral by sampling the integrand at n Legendre nodes. And the integration is of order $2n-1$. But conceptually, this is much more useful to us because we learned that we can use the Gauss–Legendre spacings to find the polynomial that best approximates our integral of interest.

8.4 Approximating the Force Function

Professor: Now let's go back to our N-body system and to solving equation (8.3). Let's start with the velocities:

$$\dot{\mathbf{y}}(t) = \int_0^t \mathbf{F}(\tau, \mathbf{y})\,\mathrm{d}\tau. \tag{8.32}$$

How would you approach the problem after all we discussed before? Alice, do you remember your observation about the N-body problem?

Alice: Sure … can we try to approximate $\mathbf{F}$ with a polynomial, like we did in section 8.3? Wait, in this problem it would be even better to have such a polynomial approximation! Because we will obtain not only the velocities easily but also the positions.

Bob: Wow, that'd be great!

Professor: And you already know where to start.

Alice: The goal will be to find

$$\ddot{\mathbf{y}} = \mathbf{F} \approx \mathbf{A}_0 + \mathbf{A}_1 t + \mathbf{A}_2 t^2 + \ldots + \mathbf{A}_{n-1} t^{n-1}, \tag{8.33}$$

right? The coefficients are now vectors, and our independent variable is time t. We want the coefficients $\mathbf{A}_i$ to provide the best approximation of the integral (8.32).

Bob: Once we have that, the positions and velocities will be

$$\dot{\mathbf{y}}(t) \approx \dot{\mathbf{y}}_0 + \mathbf{A}_0 t + \mathbf{A}_1 \frac{t^2}{2} + \mathbf{A}_2 \frac{t^3}{3} + \ldots + \mathbf{A}_{n-1}\frac{t^n}{n}, \tag{8.34a}$$

$$\mathbf{y}(t) \approx \mathbf{y}_0 + \dot{\mathbf{y}}_0 t + \mathbf{A}_0 \frac{t^2}{2} + \mathbf{A}_1 \frac{t^3}{6} + \mathbf{A}_2 \frac{t^4}{12} + \ldots + \mathbf{A}_{n-1}\frac{t^{n+1}}{n(n+1)}. \tag{8.34b}$$

We now have to find $\mathbf{A}_i$. For that, we will sample $\mathbf{F}(t, \mathbf{y})$ at the Radau nodes to obtain $\mathbf{F}(t_i, \mathbf{y})$, using equation (8.2).

Alice: Wait, wait … we have a few problems now. First, we are no longer integrating over the interval $[-1,1]$. Second, our function $\mathbf{F}(t,\mathbf{y})$ depends on the position of the bodies at time t. How are we supposed to evaluate the force function $\mathbf{F}$ if we don't know the position of each body? If we knew the solution $\mathbf{y}(t)$, we wouldn't be doing any of this!

Professor: Yes, yes, it will require a bit more work. But you are doing great so far. Let's deal with the problem of the interval first. Instead of the integral over $[-1,1]$, we are interested in $[0,t]$. Actually, over the interval $[0,\Delta t]$ because we will advance step by step. If we introduce the change of variable

$$h = \frac{t}{\Delta t}, \tag{8.35}$$

with $0 \le h \le 1$, we arrive at

$$\dot{\mathbf{y}}(\Delta t) = \Delta t \int_0^1 \mathbf{F}(h\Delta t, \mathbf{y})\,\mathrm{d}h. \tag{8.36}$$

Equation (8.33) is transformed into

$$\begin{aligned}\mathbf{F}(h) &\approx \mathbf{A}_0 + \mathbf{A}_1 h\Delta t + \mathbf{A}_2 h^2\Delta t^2 + \ldots + \mathbf{A}_{n-1} h^{n-1}\Delta t^{n-1} \\ &= \mathbf{B}_0 + \mathbf{B}_1 h + \mathbf{B}_2 h^2 + \ldots + \mathbf{B}_{n-1} h^{n-1}\end{aligned} \tag{8.37}$$

by introducing the new coefficients

$$\mathbf{B}_i = \mathbf{A}_i \Delta t^i. \tag{8.38}$$

Now we have to solve for $\mathbf{B}_i$ instead of $\mathbf{A}_i$. The position and velocity vectors approximated in equation (8.34) become:

$$\dot{\mathbf{y}}(h) = \dot{\mathbf{y}}_0 + \left(\mathbf{B}_0 h + \mathbf{B}_1 \frac{h^2}{2} + \mathbf{B}_2 \frac{h^3}{3} + \ldots + \mathbf{B}_{n-1}\frac{h^n}{n}\right)\Delta t, \tag{8.39a}$$

$$\mathbf{y}(h) = \mathbf{y}_0 + \dot{\mathbf{y}}_0 h\Delta t + \left(\mathbf{B}_0 \frac{h^2}{2} + \mathbf{B}_1 \frac{h^3}{6} + \mathbf{B}_2 \frac{h^4}{12} + \ldots + \mathbf{B}_{n-1}\frac{h^{n+1}}{n(n+1)}\right)\Delta t^2. \tag{8.39b}$$

Alice: Oh, I see, now we are integrating from $h = 0$ to $h = 1$. Can we then use equation (8.12) to shift the integration interval from $[-1,1]$?

Professor: We will essentially do that; we will directly use the nodes h_i shifted to the interval $[0,1]$.

Bob: Okay, so we solved the problem of the integration interval. But I think that was the easy problem. Not knowing the position vectors $\mathbf{y}$ when evaluating the accelerations $\mathbf{F}$ in equation (8.2) looks like a showstopper to me!

Professor: It is not that bad. This is a problem that all *implicit* methods have to deal with, as we discussed in chapter 5. We cannot use an expression like equation (8.29). Instead, we will use a predictor-corrector scheme. A predictor-corrector algorithm is actually pretty simple to understand. You'll initialize the $\mathbf{B}_i$ to some value (like $\mathbf{B}_0 = \mathbf{F}(0)$ and $\mathbf{B}_1 = \ldots =$

$\mathbf{B}_{n-1} = 0$) and you'll check how good the approximation is. Then, you will refine the $\mathbf{B}_i$ coefficients sequentially until you detect that the coefficients are no longer changing after the refinement step, which means that you have converged to the right solution.

By the way, I am surprised that you have not asked me why we are using the Gauss–Radau sequence and not the Gauss–Legendre or Gauss–Lobatto. Remember the difference between Gauss–Legendre, Gauss–Radau, and Gauss–Lobatto?

Alice: Let me check my notes…. The difference was whether or not we included the endpoints of the integration interval in the sequence of nodes.

Professor: Exactly. So? Keep in mind that our integration interval is $h \in [0,1]$ or $t \in [0, \Delta t]$.

Alice: With Gauss–Legendre we didn't include the endpoints. Will we include them now? Which ones?

Bob: We should include $h = 0$, shouldn't we? At the very first step, we know $\mathbf{y}(0)$ from the initial conditions and we should use that information to compute $\mathbf{F}(0, \mathbf{y}_0)$. That's actually the only time when we can evaluate $\mathbf{F}$ exactly.

Professor: Great reasoning, Bob! Yes, you are completely right. We'll keep the first endpoint. What about the last one?

Alice: Well, the solution at the end of the current time interval ($t = \Delta t$) will become the initial conditions for the next step…. We will be evaluating $\mathbf{F}$ twice, won't we? At the end of the current step and at the beginning of the next one. So I'd say no, we don't need that.

8.5 Computing the Coefficients

Professor: That's right! The bottom line is that we will use a Gauss–Radau sequence h_i over the interval $[0,1]$ instead of a Gauss–Legendre sequence over $[-1,1]$ because we want to keep the information from the initial point. As a result, the first point in the sequence will be $h_1 = 0$. The Gauss–Radau sequences can be found in tables too. I will print them out for you in a minute. And we won't use the Gauss–Lobatto sequence in order not to reevaluate $\mathbf{F}$ at the end of each integration step.

We will now focus on the specifics of the integrator proposed by Everhart (1974, 1985). In his papers, he provided a set of formulae for computing $\mathbf{B}_i$ ($i = 1, \ldots, n-1$) using two sets of auxiliary coefficients, c_{ji} and $\mathbf{G}_j$. First, we will rewrite equation (8.37) like

$$\mathbf{F}(h) \approx \mathbf{G}_0 + \mathbf{G}_1 h + \mathbf{G}_2 h(h-h_2) + \mathbf{G}_3 h(h-h_2)(h-h_3) + \ldots + \mathbf{G}_{n-1} h(h-h_2)(h-h_3)\ldots(h-h_{n-1}). \tag{8.40}$$

Then, the $\mathbf{B}_j$ coefficients are given by:

$$\mathbf{B}_1 = c_{11}\mathbf{G}_1 + c_{21}\mathbf{G}_2 + c_{31}\mathbf{G}_3 + \ldots \tag{8.41a}$$

$$\mathbf{B}_2 = \quad c_{22}\mathbf{G}_2 + c_{32}\mathbf{G}_3 + \ldots \tag{8.41b}$$

$$\mathbf{B}_3 = \quad c_{33}\mathbf{G}_3 + \ldots \tag{8.41c}$$

The independent term $\mathbf{B}_0$ is the acceleration at the beginning of the interval,

$$\mathbf{B}_0 = \mathbf{F}(0, \mathbf{y}(0)) = \mathbf{F}(0, \mathbf{y}_0). \tag{8.42}$$

The coefficients $\mathbf{G}_j$ ($j = 1, \ldots, n-1$) are given in nested expressions of the form

$$\mathbf{G}_1 = \quad (\mathbf{F}_2 - \mathbf{F}_1)r_{21}; \tag{8.43a}$$

$$\mathbf{G}_2 = \quad ((\mathbf{F}_3 - \mathbf{F}_1)r_{31} - \mathbf{G}_1)r_{32}; \tag{8.43b}$$

$$\mathbf{G}_3 = \quad (((\mathbf{F}_4 - \mathbf{F}_1)r_{41} - \mathbf{G}_1)r_{42} - \mathbf{G}_2)r_{43}; \tag{8.43c}$$

$$\mathbf{G}_4 = ((((\mathbf{F}_5 - \mathbf{F}_1)r_{51} - \mathbf{G}_1)r_{52} - \mathbf{G}_2)r_{53} - \mathbf{G}_3)r_{54}; \tag{8.43d}$$

$$\mathbf{G}_5 = \ldots$$

where

$$r_{ij} = \frac{1}{h_i - h_j}, \qquad r_{i1} = \frac{1}{h_i}, \tag{8.44}$$

and

$$\mathbf{F}_i = \mathbf{F}(h_i, \mathbf{y}(h_i)) \tag{8.45}$$

is obtained by evaluating equation (8.2). Remember that $h_1, \ldots, h_n$ are the nodes of the Gauss–Radau sequence over the interval $[0, 1]$, with $h_1 = 0$. This means that

$$\mathbf{F}_1 = \mathbf{F}(h_1, \mathbf{y}(h_1)) = \mathbf{F}(0, \mathbf{y}_0). \tag{8.46}$$

Finally,

$$c_{ii} = 1, \tag{8.47a}$$

$$c_{i1} = -h_i c_{i-1,1}, \tag{8.47b}$$

$$c_{ij} = c_{i-1,j-1} - h_i c_{i-1,j}, \tag{8.47c}$$

keeping in mind that in these expressions, $i > 1$ and $j < i$.

Bob: Wait, what was all that?

Professor: Don't panic. The idea is the same as finding an interpolating polynomial using the standard Lagrange polynomials. If I ask you to find the polynomial that interpolates a function $y(x)$ at $\{x_0, x_1, x_2\}$, do you remember the standard procedure?

Bob: Let me look it up … ah, I remember now:

$$p(x) = \frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)}y_0 + \frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)}y_1 + \frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)}y_2. \tag{8.48}$$

Professor: Can you write it like

$$p(x) = g_0 + g_1(x-x_0) + g_2(x-x_0)(x-x_1) \tag{8.49}$$

instead?

Bob: If so, I guess we should be able to find g_0, g_1, and g_2 by equating equations (8.48) and (8.49) term by term:

$$g_0 = y_0; \tag{8.50a}$$

$$g_1 = \frac{x_0-x_1-x_2+1}{(x_0-x_1)(x_0-x_2)}y_0 - \frac{x_2-1}{(x_1-x_0)(x_1-x_2)}y_1 - \frac{x_1-1}{(x_2-x_0)(x_2-x_1)}y_2; \tag{8.50b}$$

$$g_2 = \frac{y_0}{(x_0-x_1)(x_0-x_2)} - \frac{y_1}{(x_0-x_1)(x_1-x_2)} + \frac{y_2}{(x_1-x_2)(x_0-x_2)}. \tag{8.50c}$$

Yes, it worked! That means both equations are equivalent.

Professor: And equation (8.49) has one advantage over equation (8.48). It is really easy to add more terms to the series, whereas if we stick to equation (8.48), we will have to rewrite every coefficient to include extra points.

Bob: Equation (8.40) makes sense now! It is a slightly different way of writing the standard polynomial that interpolates the function $\mathbf{F}(h)$ at $\{0, h_1, \ldots, h_n\}$.

Professor: Exactly. The predictor-corrector works as follows. After computing all c_{ij} and r_{ij} for the integration order of choice, run the predictor-corrector loop for $j = 1, \ldots, n-1$:

1. With the current estimate of $\mathbf{B}_i$, approximate $\mathbf{y}$ and $\dot{\mathbf{y}}$ at h_j and h_{j+1} using equation (8.39).
2. Compute the acceleration at h_j and h_{j+1} ($\mathbf{F}_j$ and $\mathbf{F}_{j+1}$) using equation (8.2) and the values of $\mathbf{y}_j$, $\mathbf{y}_{j+1}$, $\dot{\mathbf{y}}_j$, and $\dot{\mathbf{y}}_{j+1}$ from step 1.
3. Use equation (8.43) to find $\mathbf{G}_j$.
4. Update $\mathbf{B}_j$ by evaluating equation (8.41) using the new value of $\mathbf{G}_j$.

Every time we update $\mathbf{B}_j$, we reduce the error in the approximation of $\mathbf{F}$ until we get the right values of $\mathbf{B}_j$ and the approximation cannot improve further. Let's call $\mathbf{B}_{n-1}^{\text{new}}$ the new value of $\mathbf{B}_{n-1}$ after step 4 and $\mathbf{B}_{n-1}^{\text{old}}$ its previous value before step 4. We will repeat the predictor-corrector loop until we converge, that is, until the difference between $\mathbf{B}_{n-1}^{\text{new}}$ and $\mathbf{B}_{n-1}^{\text{old}}$ is sufficiently small. This means that the last step did not improve the approximation anymore. The change in $\mathbf{B}_{n-1}$ will be

$$\delta\mathbf{B}_{n-1} = \mathbf{B}_{n-1}^{\text{new}} - \mathbf{B}_{n-1}^{\text{old}}. \tag{8.51}$$

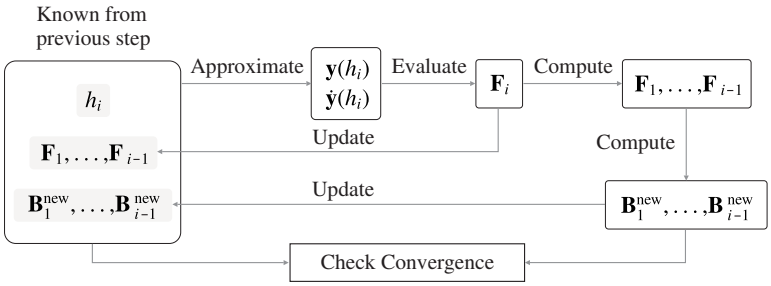


Figure 8.3
Steps for computing the coefficients $\mathbf{B}_i$ at every step i.

We could just check the magnitude of $\delta\mathbf{B}_{n-1}$ to see how much it changed. But it is better to check how important was the last change compared to the magnitude of the acceleration at the last node, $\mathbf{F}_{n-1}$.

Alice: Why don't we check all the coefficients $\mathbf{B}_j$? Why just the last one?

Professor: Each coefficient $\mathbf{B}_j$ is multiplied by h^j. Since $h < 1$, increasing j decreases h^j and therefore h^{n-1} is the smallest among all h^j. Consequently, changes in $\mathbf{B}_{n-1}$ need to be larger than the rest to have a noticeable impact on $\mathbf{F}$.

Alice: Oh, I see. We are interested in checking how large was the maximum change and that corresponds to $\mathbf{B}_{n-1}$.

Professor: That's the idea. To decide if we converged or not, we will evaluate the ratio between the largest components of $|\delta\mathbf{B}_{n-1}|$ and $|\mathbf{F}_{n-1}|$:

$$\varepsilon_{\text{PC}} = \frac{\max|\delta\mathbf{B}_{n-1}|}{\max|\mathbf{F}_{n-1}|} \tag{8.52}$$

and compare it with a certain tolerance, as Rein and Spiegel (2014) suggested. We will repeat steps 1–4 until

$$\varepsilon_{\text{PC}} < tol_{\text{PC}}, \tag{8.53}$$

where tol_{PC} is the tolerance of the predictor-corrector (for example, $tol_{\text{PC}} = 10^{-15}$).

Bob: Okay, these are quite a lot of equations, but we have everything we need, right?

Alice: I could use a diagram here. For every iteration of the predictor-corrector, we'll go through the entire Radau sequence $i = 1,\ldots,n-1$ updating the $\mathbf{B}_i$ coefficients like in figure 8.3 until we converge, that is, $\varepsilon_{\text{PC}} < tol_{\text{PC}}$.

8.6 Radau Sequence and Integration Order

Professor: Are you okay so far? To me, the diagram you made is a good sign that you understood how the integrator works.

Bob: What about the nodes? You said you will give us the values of h_i.

Professor: Yes, I collected the positions of the nodes in table 8.2. I included up to $n = 10$, which will give an integrator of order $2n - 1 = 19$. However, we should choose $n = 8$ to obtain an integrator of the fifteenth order. This integrator has been tested in many practical scenarios and is known to provide very good accuracy and performance.

Alice: Thanks! Still, can you give us some background about how you computed the nodes?

Professor: The nodes are simply the solution to equation (8.16). Solving this equation for x gives us the solution over the interval $[-1, 1]$. Equation (8.12) allows us to shift the interval to $0 \le h \le 1$:

$$h = \frac{1}{2}(x+1). \tag{8.54}$$

To reduce round-off errors, it is a good practice to compute the sequence h_i with many more digits than we actually need.

Bob: Well of course, it would be great to have many more correct digits than we need! But how can we do that? How can we go beyond double precision?

Professor: For starters, you could work in quadruple precision floating-point arithmetic. That will give approximately twice more digits than double precision. But there is another solution even more powerful, which is working with arbitrary precision. Some languages like MAPLE or MATHEMATICA support arbitrary precision natively. For example, in MAPLE, we can define the precision using

```
> Digits := 20;
```

It will give us twenty exact digits.

Snippet 8.2 is a short script in MAPLE that computes the nodes with arbitrary precision using the intrinsic function `fsolve` to solve for x in equation (8.16). It is called with option `fulldigits` to retain all digits in the intermediate computations. Once `x` is known, we apply equation (8.54) to shift the interval. Like many other programming languages, MAPLE provides an implicit function to evaluate the Legendre polynomials $P_n(x)$, `LegendreP(n, x)`.

Snippet 8.2

Procedure `compute_nodes`: Compute the nodes of the Radau sequence.

```
> compute_nodes := proc(n)
    local dx, x0, xf, i, niter, Eq, solution, nodes, inode;
    Eq := LegendreP(n - 1, x) + LegendreP(n, x);
    niter := 100;
    x0 := -1; xf := 1; dx := evalf((xf - x0) / niter);
    nodes := Vector(n);
    inode := 1;
    for i from 0 to niter do
      solution := fsolve(Eq = 0, x = x0, fulldigits);
```

```
      solution := 0.5 * (solution + 1);
      if not has(nodes, solution) then
        nodes[inode] := solution;
        inode := inode + 1;
      end if;
      x0 := x0 + dx
    end do;
    return sort(nodes)
  end proc
```

If you wanted the Gauss–Legendre nodes instead of the Gauss–Radau ones, you could just replace the expression `Eq := LegendreP(n - 1, x) + LegendreP(n, x)` with

```
Eq := LegendreP(n, x);
```

Similarly, to obtain the Gauss–Lobatto nodes, you should use:

```
Eq := x * LegendreP(n - 1, x) - LegendreP(n - 2, x);
```

Anyway, to compute the Gauss–Radau nodes for $n = 5$, just run the previous procedure like:

```
sensitive> compute_nodes(5);
```

$$\begin{bmatrix} 0 \\ 0.13975986434378055215 \\ 0.41640956763108317994 \\ 0.72315698636187617232 \\ 0.94289580388548231780 \end{bmatrix}. \tag{8.55}$$

This is the same sequence that you have in table 8.2. While solving equation (8.16) numerically with `fsolve`, it is important to make sure that we find all nodes, or equivalently that we find all roots of the equation to solve. The trick is to call `fsolve` several times starting from different values of the initial guess `x0`.

8.7 Coding the Main Components of the Integrator

Alice: We have a lot of stuff to implement now! Where should we start?

Bob: While you were discussing about the nodes, I created the function `radau_spacing` with the sequence of Gauss–Radau nodes for order 15 ($n = 8$), using the values in table 8.2. Take a look at snippet 8.7. It returns the sequence of nodes `h[:]` and the value of n, `nh`.

Alice: Great, thanks Bob! To start from the beginning, I'd suggest that we implement equations (8.37) and (8.39).

Table 8.2
Position of the Gauss–Radau nodes over the interval $[0, 1]$ for integration orders 3 to 19 (the integration order is $2n - 1$).

n	i	h_i
2	1	0.00000 00000 00000 00000
2	2	0.66666 66666 66666 66667
3	1	0.00000 00000 00000 00000
3	2	0.35505 10257 21682 19018
3	3	0.84494 89742 78317 80982
4	1	0.00000 00000 00000 00000
4	2	0.21234 05382 39152 94398
4	3	0.59053 31355 59265 28914
4	4	0.91141 20404 87296 05260
5	1	0.00000 00000 00000 00000
5	2	0.13975 98643 43780 55215
5	3	0.41640 95676 31083 17994
5	4	0.72315 69863 61876 17232
5	5	0.94289 58038 85482 31780
6	1	0.00000 00000 00000 00000
6	2	0.09853 50857 98826 42612
6	3	0.30453 57266 46363 90548
6	4	0.56202 51897 52613 85600
6	5	0.80198 65821 26391 82746
6	6	0.96019 01429 48531 25766
7	1	0.00000 00000 00000 00000
7	2	0.07305 43286 80258 88515
7	3	0.23076 61379 69945 49908
7	4	0.44132 84812 28449 86792
7	5	0.66301 53097 18845 70090
7	6	0.85192 14003 31515 70815
7	7	0.97068 35728 40215 10803

n	i	h_i
8	1	0.00000 00000 00000 00000 00000 00000
8	2	0.05626 25605 36922 14646 56521 91032
8	3	0.18024 06917 36892 36498 75799 42809
8	4	0.35262 47171 13169 63737 39077 70171
8	5	0.54715 36263 30555 38300 14485 57652
8	6	0.73421 01772 15410 53152 32106 08306
8	7	0.88532 09468 39095 76809 03597 62932
8	8	0.97752 06135 61287 50189 11745 00429
9	1	0.00000 00000 00000 00000 00000 00000
9	2	0.04463 39552 89969 85073 31210 21858
9	3	0.14436 62570 42145 57148 52185 20228
9	4	0.28682 47571 44430 51894 86862 39749
9	5	0.45481 33151 96573 35096 77277 70047
9	6	0.62806 78354 16727 69756 91460 39518
9	7	0.78569 15206 04369 24164 24587 32418
9	8	0.90867 63921 00206 04399 62585 41926
9	9	0.98222 00848 52636 54818 67948 98962
10	1	0.00000 00000 00000 00000 00000 00000
10	2	0.03625 78128 83209 46094 11643 00768
10	3	0.11807 89787 89998 70019 22851 11994
10	4	0.23717 69848 14960 38531 73066 92854
10	5	0.38188 27653 04705 97536 07702 48396
10	6	0.53802 95989 18989 06511 68568 91319
10	7	0.69033 24200 72362 18294 03795 32770
10	8	0.82388 33438 37004 71813 68242 53928
10	9	0.92561 26102 90803 95536 40818 14044
10	10	0.98558 75903 51123 45136 71732 59189

Professor: That's a good idea, but I can tell you that you don't really need equation (8.37). Whenever you need to evaluate **F**, you will use the actual acceleration given by equation (8.2).

Alice: Oh, I see, I forgot that we will evaluate the actual function and not the approximation. It makes sense, of course. At first we only know equation (8.2) and we will sample it at the nodes to find the approximating coefficients.

Professor: Right. One more thing before implementing equation (8.39). There is a way to write those approximations more efficiently in the form of nested expressions:

$$\dot{\mathbf{y}}(h) = \dot{\mathbf{y}}_0 + h\Delta t\left(\mathbf{F}_1 + h\left(\frac{\mathbf{B}_1}{2} + h\left(\frac{\mathbf{B}_2}{3} + h\left(\frac{\mathbf{B}_3}{4} + h\left(\frac{\mathbf{B}_4}{5} + h\left(\frac{\mathbf{B}_5}{6} + h\left(\frac{\mathbf{B}_6}{7} + h\frac{\mathbf{B}_7}{8}\right)\right)\right)\right)\right)\right), \quad (8.56a)$$

$$\mathbf{y}(h) = \mathbf{y}_0 + h\Delta t\left(\dot{\mathbf{y}}_0 + \frac{h}{2}\Delta t\left(\mathbf{F}_1 + h\left(\frac{\mathbf{B}_1}{3} + h\left(\frac{\mathbf{B}_2}{6} + h\left(\frac{\mathbf{B}_3}{10} + h\left(\frac{\mathbf{B}_4}{15} + h\left(\frac{\mathbf{B}_5}{21} + h\left(\frac{\mathbf{B}_6}{28} + h\frac{\mathbf{B}_7}{36}\right)\right)\right)\right)\right)\right)\right). \quad (8.56b)$$

You'll save a few operations because you are not computing the terms h^i separately. Note that I also replaced $\mathbf{B}_0$ with $\mathbf{F}_1 = \mathbf{F}(0, \mathbf{y}_0)$, the initial acceleration.

Alice: Thanks! Let me write the two functions that approximate the position and the velocity. Take a look at snippet 8.3. These functions assume that we have the coefficients $\mathbf{B}_i$ stored in an array `b`; vector $\mathbf{B}_i$ should be stored like `b[i,:]`. We also need the Gauss–Radau nodes `h`, but we already have the function `radau_spacing` in snippet 8.7 that returns that vector, so we are good to go. And `dt` is the time step Δt.

Snippet 8.3

Functions `approx_pos` and `approx_vel`: Approximate the position and velocity of the particles using equation (8.56).

```
def approx_pos( y0, dy0, F1, h, b, dt ):
    y = y0 + dt * h * (dy0 + dt * h * (F1 + h * (b[0,:] / 3.0 + h * (b
        [1,:] / 6.0 + h * (b[2,:] / 10.0 + h * (b[3,:] / 15.0 + h * (b
        [4,:] / 21.0 + h * (b[5,:] / 28.0 + h * b[6,:] / 36.0)))))) /
        2.0)
    return y

def approx_vel( dy0, F1, h, b, dt ):
    dy = dy0 + dt * h * (F1 + h * (b[0,:] / 2.0 + h * (b[1,:] / 3.0 + h *
        (b[2,:] / 4.0 + h * (b[3,:] / 5.0 + h * (b[4,:] / 6.0 + h * (b
        [5,:] / 7.0 + h *     b[6,:] / 8.0)))))))
    return dy
```

Bob: Looking good! Okay, so we now need to compute `b` (the coefficients $\mathbf{B}_i$). Let's implement equation (8.41). Any suggestions before jumping into it?

Professor: Actually, yes. When I explained the predictor-corrector steps, I mentioned that we will advance along the Gauss–Radau sequence updating $\mathbf{B}_j$, for $j = 1, \ldots, n-1$. Thus, it will be a good idea to implement the function in such a way that you can get $\mathbf{B}_j$ up to a given value of j. Sometimes you might not need all coefficients up to $j = n-1$.

Bob: What do you think about snippet 8.8, then (page 156)? It takes array `b` as an input (with its current value) and it returns this same array with the updated coefficients up to a given step i in the Gauss–Radau sequence, which I called `ih`. I realized that we will need the coefficients $\mathbf{G}_i$ and c_{ij}. Array `g[i,:]` contains $\mathbf{G}_\mathbf{i}$, and `c[i,j]` corresponds to c_{ij}.

Professor: Keep in mind that the c_{ij} coefficients do not change. You can compute them once at the beginning of the integration and use them where needed.

Alice: I computed them in MAPLE with a lot of digits to make sure we don't introduce any errors. Will `Digits := 50` be enough? Snippet (8.9) is the current implementation.

Professor: That's way too many digits, Alice! But that's okay, you can leave it like that. There's no harm in writing the function like that.

Alice: Ah, okay, I'll keep that in mind for next time. I just implemented equation (8.43) following what Bob did in snippet 8.8; pass `ih` as an input variable and then update the coefficients up to i equal to `ih`. Equation (8.43) requires r_{ij} and also the acceleration $\mathbf{F}$ evaluated at the nodes h_j, with $j = 1, \ldots, i$. For now, I am assuming that the function receives array `ddys` as an input, where `ddys[i,:]` is $\mathbf{F}_i$.

Bob: And what about `r`?

Alice: They are also constant, so I computed them at once in MAPLE. I wrote snippet 8.11 with them.

Bob: Those are some huge numbers! I bet you forgot to change `Digits := 50` in MAPLE to something more reasonable.

Alice: I totally forgot about that … well, now there's no doubt we won't lose precision by precomputing their value!

8.8 Advancing One Integration Step

Professor: You have now the main pieces of the integrator. I would recommend that you put together a function that advances one integration step. That is, a function that simply integrates the system from t to $t + \Delta t$. The explanation about the predictor-corrector scheme on page 136 might be useful now.

8.8.1 Predictor-Corrector

Bob: I just went through my notes to find it. The predictor-corrector was used to compute $\mathbf{B}_i$, wasn't it?

Professor: That's right. You will need several iterations of the predictor-corrector until you converge to the right $\mathbf{B}_i$. You can limit the number of iterations to twelve, since in practice you'll typically never reach that limit. It should converge in two or three iterations.

Bob: Okay. We first need to approximate the position and velocity vectors to be able to compute the acceleration at h_i. We will then compute $\mathbf{G}_i$ up to the current i and update $\mathbf{B}_i$ consistently. These steps are repeated until the error in $\mathbf{B}_7$, estimated by equation (8.52), becomes small enough.

I think I got it; it is snippet 8.4.

Snippet 8.4
Predictor-corrector scheme.

```
# Loop for predictor-corrector step:
for ipc in range(0, 12):
    ddys = ddys * 0
    # Advance along the Gauss-Radau sequence:
    for ih in range(0, nh):
        # Estimate position and velocity with current bs and h:
        y  = approx_pos(y0, dy0, ddy0, hs[ih], bs, dt)
        dy = approx_vel(    dy0, ddy0, hs[ih], bs, dt)
        # Evaluate force function and store result:
        ddys[ih,:] = ode(t + hs[ih] * dt, y, dy, params)
        g  = compute_gs(g, r, ddys, ih)
        bs = compute_bs_from_gs(bs, g, ih, c)
    # Estimate convergence of PC:
    db7 = bs[-1,:] - bs0[-1,:]
    if ( max(abs(db7)) / max(abs(ddys[-1,:])) < 1e-16 ):
        break
    bs0 = bs
```

Instead of calling `ode_n_body_second_order` explicitly, I am using the generic function name "`ode`" like we did on page 107. We'll have to remember to add the function pointer as an input variable.

8.8.2 Estimating the Time Step for the Next Iteration

Professor: Don't forget that so far, we have only talked about how to integrate from t to $t+\Delta t$. But this is a variable step-size integrator, meaning that we also need to discuss how to estimate the error of the current step and how to define Δt for the next iteration.

Rein and Spiegel (2014) proposed the following algorithm for estimating the error of the current step. After the predictor-corrector converges, compute the error estimate

$$\tilde{\mathbf{B}}_7 = \frac{\max|\mathbf{B}_7|}{\max|\mathbf{F}(t+\Delta t)|} \tag{8.57}$$

as the ratio of the maximum components of the absolute value of $\mathbf{B}_7$ and the acceleration evaluated at the end of the interval. Then, an alternative estimate of the relative error is

$$\epsilon = \left(\frac{\tilde{\mathbf{B}}_7}{\varepsilon_{\text{tol}}}\right)^{\alpha} \tag{8.58}$$

where the exponent is $\alpha = 1/7$. This error estimate is used to generate the time step for the next iteration,

$$\Delta t_{\text{req}} = \frac{\Delta t}{\epsilon}. \tag{8.59}$$

We will use $\Delta t_{\text{new}} = \Delta t_{\text{req}}$ for the next step unless Δt_{req} is too small (say, $\Delta t_{\text{req}} < 10^{-12}$, when we will simply reduce the previous time step by a certain factor $f < 1$, $\Delta t_{\text{new}} = f\Delta t$) or Δt_{req} is much larger than Δt (if $\Delta t_{\text{req}}/\Delta t > 1/f$, then we'll force $\Delta t_{\text{new}} = \Delta t/f$).

Alice: I have one question about equation (8.58). Am I right if I say that setting $\alpha = 0$ will transform the integrator into a fixed-step integrator? With $\alpha = 0$, ϵ becomes unity and $\Delta t_{\text{req}} = \Delta t$.

Professor: Good catch! Yes, that's an easy trick that you can use to work with a fixed integration step.

Bob: I'm liking this integrator…. So many cool tricks.

Professor: The error estimate in equation (8.57) is useful for deciding whether or not the current step should be accepted. In general, the error of an approximation like equation (8.37) should not be larger than the last term of the series. Thus, we can use $\mathbf{B}_7$ as an upper bound to the error in the approximation of $\mathbf{F}$, especially if we take the maximum value of $\mathbf{B}_7$. If we divide it by the maximum value of $\mathbf{F}$ evaluated at the end of the interval, $t + \Delta t$, we will obtain an estimate of the relative error because $\mathbf{F}$ is precisely the function we want to approximate. Equation (8.58) then tells us how important is the error compared to a certain tolerance: if $\epsilon \leq 1$, then the relative error $\tilde{\mathbf{B}}_7$ will be less than or equal to the tolerance, which means that the step should be accepted.

Alice: So, all we have to do is check if $\epsilon \leq 1$?

Professor: Yes, that's all. But before you code the rest of the integration function, let me refer you to one final refinement. The value of the coefficients $\mathbf{B}_i$ at the current step will be used as an initial guess to compute the coefficients at the next step, and so on. But Everhart (1985) noted that these coefficients can be refined before reusing them at the next step. This refinement will typically reduce the number of iterations required by the predictor-corrector. Think of it as a better prediction of what the values of $\mathbf{B}_i$ in the next step will be. To refine the coefficients, we will first introduce the auxiliary coefficients $\mathbf{E}_i$:

$$\mathbf{E}_1 = q\ (\mathbf{B}_1 + 2\mathbf{B}_2 + 3\mathbf{B}_3 + 4\mathbf{B}_4 + 5\mathbf{B}_5 + 6\mathbf{B}_6 + 7\mathbf{B}_7), \tag{8.60a}$$

$$\mathbf{E}_2 = q^2(\mathbf{B}_2 + 3\mathbf{B}_3 + 6\mathbf{B}_4 + 10\mathbf{B}_5 + 15\mathbf{B}_6 + 21\mathbf{B}_7), \tag{8.60b}$$

$$\mathbf{E}_3 = q^3(\mathbf{B}_3 + 4\mathbf{B}_4 + 10\mathbf{B}_5 + 20\mathbf{B}_6 + 35\mathbf{B}_7), \tag{8.60c}$$

$$\mathbf{E}_4 = q^4(\mathbf{B}_4 + 5\mathbf{B}_5 + 15\mathbf{B}_6 + 35\mathbf{B}_7), \tag{8.60d}$$

$$\mathbf{E}_5 = q^5(\mathbf{B}_5 + 6\mathbf{B}_6 + 21\mathbf{B}_7), \tag{8.60e}$$

$$\mathbf{E}_6 = q^6(\mathbf{B}_6 + 7\mathbf{B}_7), \tag{8.60f}$$

$$\mathbf{E}_7 = q^7\mathbf{B}_7 \tag{8.60g}$$

that we will use to estimate the new $\mathbf{B}_i$ like

$$\mathbf{B}_i^{\text{new}} = \mathbf{E}_i + \Delta\mathbf{B}_i. \tag{8.61}$$

At the very first step, we will make $\Delta\mathbf{B}_i = 0$, and in subsequent steps, $\Delta\mathbf{B}_i = \mathbf{B}_i - \mathbf{E}_i$ is obtained using the values of $\mathbf{B}_i$ and $\mathbf{E}_i$ from the previous step. And q is the ratio between

the current time step and the estimate for the next step,

$$q = \Delta t_{\text{req}} / \Delta t. \tag{8.62}$$

Bob: That means that we will need a flag to specify if we are at the first step. How about using an integer flag `imode`? It will tell us in which *mode* the refinement should operate.

Alice: It is not the best name … but it is good enough. Let's leave it like that. It is hard to find good names for variables!

Bob: I did my best with snippet 8.12. I even evaluated the powers in a fancy way to save some operations! I expect to have q (which I called `q`) and the coefficients $\mathbf{B}_i$ (`b`) and $\mathbf{E}_i$ (`E`) from the previous step available as input. And also the flag `imode`, which will be zero at the very first step and nonzero otherwise.

Alice: I know that we said this a million times already, but now we really have all we need, don't we? I'm afraid of the answer.

Professor: Yes! Go ahead and put together the code for integrating one step.

8.8.3 Putting the Code Together

Alice: Let's start with the name of the function, although we still don't know what input arguments we'll need.

```
def gauss_radau15_step( ... ):
```

Once we are inside, we will need to iterate several times until we accept the current step. This is just like with the Runge–Kutta methods; we don't know how many iterations we'll need, so we can use an infinite loop.

Bob: Agreed. And the first thing to do inside the loop is to run the predictor-corrector. There's really nothing we can do until we get the coefficients $\mathbf{B}_i$!

Alice: That's right. We'll have to add your code from snippet 8.4 inside the `while` loop:

```
    while (True):
        # Loop for predictor-corrector step:
        ....
```

Bob: Okay, now we can estimate $\mathbf{y}$ and $\dot{\mathbf{y}}$ at $t + \Delta t$ (or $h = 1$) and also use equations (8.57) and (8.58) to estimate the error at the current step.

```
        # Advance the solution:
        y =   approx_pos(y0, dy0, ddy0, 1.0, bs, dt)
        dy =  approx_vel(    dy0, ddy0, 1.0, bs, dt)
        ddy = ode(t + dt, y, dy, params)

        # Estimate relative error
        estim_b7 = max(abs(bs[-1,:])) / max(abs(ddy))
        err = (estim_b7 / tol)**(exponent)
```

I realized that we also need $\mathbf{F}(t+\Delta t)$ to estimate the error. I called it `ddy`.

Alice: Don't forget to estimate the step size for the next round!

Bob: Right! Here it is.

```
        # Step-size required for next step:
        dtreq = dt / err
```

Alice: Now that we have an estimate of the error, let's implement the acceptance criterion.

Bob: We should introduce a flag that returns the status of the call. What do you think about `istat`? We can set it to 0 at the beginning and then set it to 1 if we accept the step.

Alice: You and your flags … but that's actually a good idea! Here is the code:

```
        # Accept the step
        if ( err <= 1 ):

            # Report accepted step:
            istat = 1

            # Advance time:
            t = t + dt

            # Store b coefficients before refinement:
            bs0 = bs

            # Refine predictor-corrector coefficients for next pass:
            bs, E = refine_bs(bs, dtreq / dt, E, imode)

            # Normal refine mode:
            imode = 1
```

Bob: Sorry, Alice, but do you mind if I set `istat` to 2 if we reach the final time? I promise I will not introduce any more flags.

```
            # Check if tf was reached:
            if ( t >= tf ):
                istat = 2
```

Alice: Sure, that's okay. The more diagnostics the better.

Bob: Are we done?

Alice: Not yet. We need to adjust the time step for the next step: make sure it is not too small or too large, and correct the overshooting of the last step to ensure we don't go beyond the final time `tf`.

```
        # Step size for next iteration:
        if ( dtreq / dt > 1.0 / fac ):
            dt = dt / fac
        elif ( dtreq < 1e-12 ):
            dt = dt * fac
```

```
        else:
            dt = dtreq

        # Correct overshooting:
        if ( t + dt > tf ):
            dt = tf - t
```

Bob: We are still inside the `while (True)` infinite loop. Look at this; my flags will turn out to be very useful!

```
        # Return if the step was accepted:
        if ( istat > 0 ):
            break
```

Alice: Anyways, I just put the whole thing together in snippet 8.13 (page 159). Bob, do you mind checking again that I didn't miss any required input variables? I remembered to add the pointer `ode` as an input.

Bob: Sure, but what about the output? We will need the states at $t + \Delta t$, `y` and `dy`, the acceleration `ddy`, the new time and the time step (`t` and `dt`), the coefficients `E`, and the refined and unrefined values of $\mathbf{B}_i$, `bs` and `bs0`.

Alice: Aren't you forgetting something? Your flags!

Bob: Of course! We need to return `imode` and `istat`. And also set `istat` equal to zero at the very beginning.

Alice: It feels a bit weird to have so many input variables, doesn't it?

Bob: Yeah, but on the other hand, it is a pretty clear implementation and easy to follow. I'd leave it as is and later, if we decide to optimize the code, we can revisit the structure.

8.9 The Complete Integrator

8.9.1 Initializing Required Variables

Alice: Okay. All we need to do now is call the function `gauss_radau15_step` from snippet 8.13 recursively to integrate from t_0 to t_f. I guess we can reuse some things from our Runge–Kutta integrator, like the strategy that we implemented in snippet 6.3 to store the solution. Let's start by defining some of the constants that we'll need.

```
def integrate_gauss_radau15( y0, dy0, t0, tf, tol, ode, params ):
    # Recommended tolerance: 1e-9

    # Limit the change in time step:
    fac = 0.25

    # For fixed step integration, choose exponent = 0:
    exponent = 1.0 / 7.0
```

```
# Initial dimension to store solution:
nstore = 10000

# Dimension of the system:
dim = len(y0)

# Allocate array to store solution:
sol_pos =  np.zeros((nstore, dim))
sol_vel =  np.zeros((nstore, dim))
sol_time = np.zeros(nstore)

sol_pos[0,:] = y0
sol_vel[0,:] = dy0

# Tolerance of predictor-corrector:
tolpc = 1e-16
```

Bob: We need to make sure that we define all the variables that `gauss_radau15_step` takes as input. In particular, we have to initialize all the coefficients of the method.

```
# Precompute Radau spacings, r, and c:
hs, nh = radau_spacing()
r = compute_rs()
c = compute_cs()

# Initialize:
bs0 =  np.zeros((nh - 1, dim))
bs =   np.zeros((nh - 1, dim))
g =    np.zeros((nh - 1, dim))
t =    t0
E =    np.zeros((nh - 1, dim))
ddys = np.zeros((nh, dim))
```

Alice: The acceleration at t_0 is also required (`ddy0`):

```
# Acceleration at t0:
ddy0 = ode(t0, y0, dy0, params)
```

8.9.2 Initializing the Time Step

Bob: Oh, and we don't know yet the initial time step `dt`.

Alice: Can we reuse the algorithm that we used for the embedded Runge–Kutta method? The one in snippet 6.1?

Professor: Yes, you can definitely use that estimate. The only change I'd suggest is ignoring the scaling factor `sc` and using the infinity norm instead of `rk_norm`.

Bob: The *what*?

Professor: The infinity norm. It is defined as

$$\|\mathbf{x}\|_\infty = \max_i |x_i|. \tag{8.63}$$

You've already used it in equations (8.52) and (8.57), although we didn't give it a name.

Alice: Sounds good. This is what it would look like:

Snippet 8.5
Function `initial_time_step`: Estimate the initial size of the time step.

```
def initial_time_step( y0, dy0, t0, epsb, ode, params ):

    # Integration order:
    p = 15

    # Evaluate right-hand side:
    f0 = ode(t0, y0, dy0, params)
    d0 = max(abs(y0))
    d1 = max(abs(f0))

    if ( (d0 < 1e-5) or (d1 < 1e-5) ):
        dt0 = 1e-6
    else:
        dt0 = 0.01 * (d0 / d1)

    # Perform one Euler step:
    y1  = y0  + dt0 * dy0
    dy1 = dy0 + dt0 * f0
    # Call function
    f1 = ode(dt0, y1, dy1, params)
    d2 = max(abs( (f1 - f0) )) / dt0

    if ( max(d1, d2) <= 1e-15 ):
        dt1 = max([1e-6, dt0 * 1e-3])
    else:
        dt1 = (0.01 / max([d1, d2]))**(1.0 / (p + 1))

    dt = min([100 * dt0, dt1])
    return dt
```

Bob: Wait a second … we are evaluating `ode_n_body_second_order` inside this function and we already did that. Aren't we repeating this call? That's not a smart thing to do.

Alice: I guess you are right. What if we skip the previous call to `ode_n_body_second_order` and have `initial_time_step` return `ddy0`?

Bob: I like it. Let's change the last line in snippet 8.5 to

```
    return dt, f0
```

Alice: We just need to call this new function and to initialize the `integrate` flag and the `istep` counter:

```
    # Initial time step:
    dt, ddy0 = initial_time_step(y0, dy0, t0, tol, ode, params)

    integrate = True
    istep = 1
```

Bob: Don't forget about my flag!

```
    imode = 0
```

Alice: (Sighs)

8.9.3 The Main Loop

Bob: I'm glad we took the time to write the function `gauss_radau15_step`. The main integration loop is very simple!

```
while integrate:

    # Advance one step and return:
    y, dy, ddy, t, dt, g, bs, E, bs0, istat, imode = gauss_radau15_step(y0
        , dy0, ddy0, ddys, dt, t, tf, nh, hs, bs0, bs, E, g, r, c, tol,
        exponent, fac, imode, ode, params)

    # Detect end of integration:
    if (istat == 2):
        integrate = False

    # Store solution:
    sol_pos[istep,:] = y
    sol_vel[istep,:] = dy
    sol_time[istep]  = t

    # Update step:
    istep += 1
    y0 =     y
    dy0 =    dy
    ddy0 =   ddy

    # Increase buffer size if needed:
    if (istep == sol_pos.shape[0]):
        sol_pos =  np.concatenate((sol_pos,sol_pos[0:nstore, :]))
        sol_vel =  np.concatenate((sol_vel,sol_vel[0:nstore, :]))
        sol_time = np.concatenate((sol_time,sol_time[0:nstore]))
```

Bob: The output from the function will be the time vector and the state, obtained by concatenating the position and velocity:

```
return sol_time[0:istep], np.concatenate((sol_pos[0:istep,:], sol_vel[0:
    istep,:]), axis=1)
```

Alice: All right then, snippet 8.14 is the final Gauss–Radau integrator. I'm really excited about it! Thank you so much for your help, Prof. Starmover. We made huge progress! We will now test the integrator with some examples, but we don't want to steal more of your time.

Professor: Let's reconvene next week, at which time you can let me know how the integrator behaved.

Bob: Goodbye, Prof. Starmover!

Alice: Bye!

8.10 Testing the Integrator

Alice and Bob leave Prof. Starmover's office and start testing their new integrator.

Bob: Time to play! Let's see how, or rather *if*, the integrator works. Shall we integrate the Solar System again? I don't feel so great after what happened with the Runge–Kutta–Fehlberg 4(5) method….

8.10.1 Simple Two-Body Problem

Alice: I think it is better to test it first with some simple problems to which we know the solutions. Let's integrate the simple two-body problem that we defined in section 6.5. The initial conditions are:

Snippet 8.6
Initial conditions for an example of the elliptic two-body problem.

```
nbodies = 2
x = np.zeros(nbodies * 6)

# Initial conditions:
x[0:3]  = np.array([-1.0, 0.0, 0.0])
x[3:6]  = np.array([+1.0, 0.0, 0.0])
x[6:9]  = np.array([0.0, -0.48, 0.0])
x[9:12] = np.array([0.0, +0.48, 0.0])

# Mass of the bodies:
masses =[1.0, 1.0]

# Gravitational parameter:
G  = 1.0
```

```
# Initial and final time (one period):
t0 = 0.0
tf = 11.221219176577781

# Store parameters:
params = {'G': G, 'masses': masses, 'nbodies': nbodies}
```

The call to the integrator will be:

```
# Function to integrate:
ode = ode_n_body_second_order

# Launch integration:
tol = 1.0e-9
sol_time, sol_state = integrate_gauss_radau15(x[0:nbodies*3], x[nbodies
    *3:], t0, tf, tol, ode, params)
```

Alice: Okay, I plotted the results in figure 8.4. Figure 8.4a shows the trajectories, and it looks exactly like figure 6.1. That's a good start!

In addition, the final time is exactly the orbital period. This means that we are integrating one revolution and we should end up back where we started. I wrote the final state in table 8.3.

Bob: Nailed it! The final state coincides with the initial conditions. The energy error (figure 8.4b) does not look bad either. It is essentially the machine zero! I think we can say the integrator works.

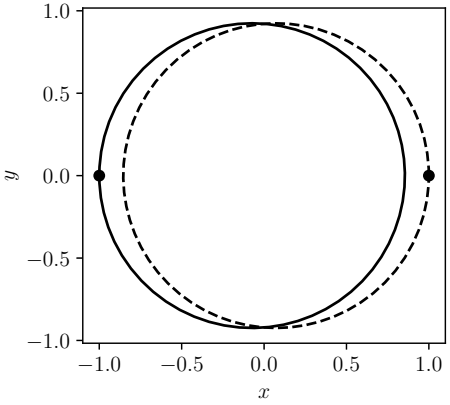
8.10.2 Double Figure-8 Orbits with Five Bodies

Bob: I really liked the idea of double figure-8 orbits using $N = 5$ identified by Simó (2001). Let me give it a try. We already have the initial conditions coded in snippet 7.5.

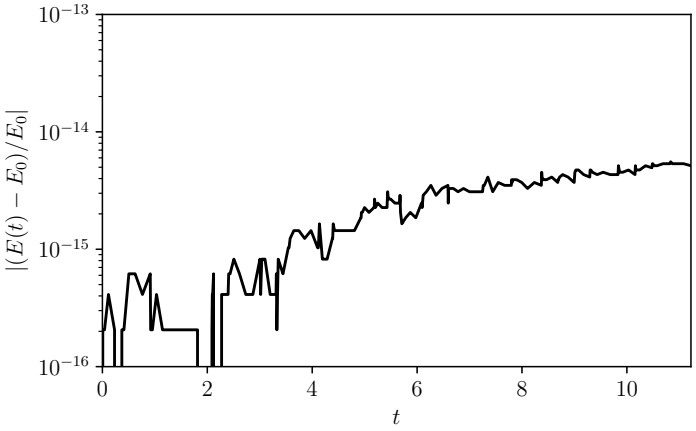
I plotted the solution and the error in figure 8.5. Not bad either! The energy error at the end of the integration is about 10^{-14}. I'm confident our implementation is fine. Do you want to try to integrate the Solar System now?

Table 8.3
Final position and velocity of the particles in the two-body test problem.

Body	x	y	z
1	–1.000000000000006e+00	8.187894806610529e–15	0.000000000000000e+00
2	1.000000000000006e+00	–8.187894806610529e–15	0.000000000000000e+00
	$\dot{x}$	$\dot{y}$	$\dot{z}$
1	–4.864164626638967e–15	–4.799999999999981e–01	0.000000000000000e+00
2	4.864164626638967e–15	4.799999999999981e–01	0.000000000000000e+00



(a) Orbit plot



(b) Relative energy error

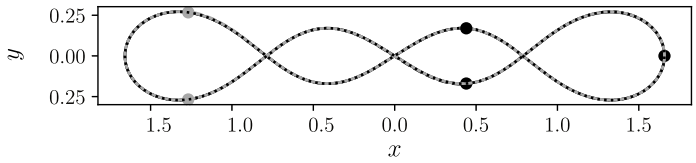
Figure 8.4
Solution to the elliptic two-body problem using GR15.

8.10.3 Propagating the Solar System for 1,000 Years

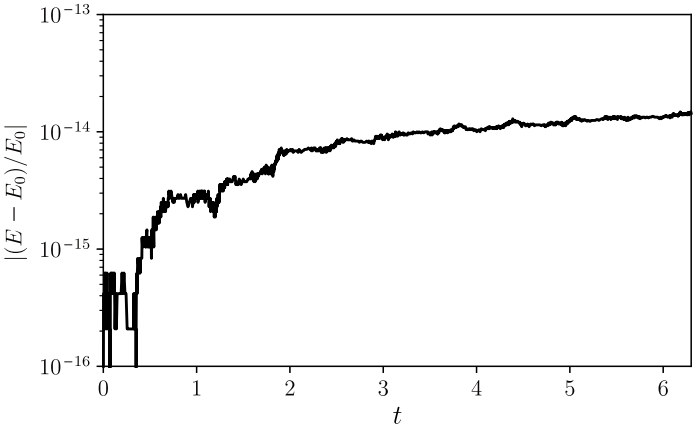
Alice: Sure, let's do it! We can let it run while we grab some lunch. I have some classes I have to attend. I grabbed the initial conditions from snippet 7.6.

After starting the simulation, Alice and Bob take a much-deserved break. They return a few hours later to collect the results.

Alice: Okay, it looks like the simulations have finished. I'm really curious to see how accurate was the integration. With RKF4(5), we only achieved 10^{-9} accuracy. It is not bad



(a) Orbit plot



(b) Relative energy error

Figure 8.5
Solution to the figure-8 problem with $N = 5$.

per se, but I was used to getting close to machine precision! I am plotting the error of the Gauss–Radau integrator (GR15) together with the error that we obtained with RKF4(5), which appears in figure 7.4b too. I created figure (8.6).

Bob: Wow, that really is a huge improvement! Five orders of magnitude! Good job, Gauss–Radau.

Alice: Not only is the error small, but it doesn't seem to be growing as time advances. The energy is conserved in practice through 1,000 years! This is impressive.

Bob: Let's go back to Prof. Starmover's office to show her this!

Alice: I think she has had enough of us for the day, Bob. I'm excited too, but it can wait for our next meeting! Or we could shoot her an email if you want.

Bob: All right, fine. I like the email option.

Alice: Let's do that then.

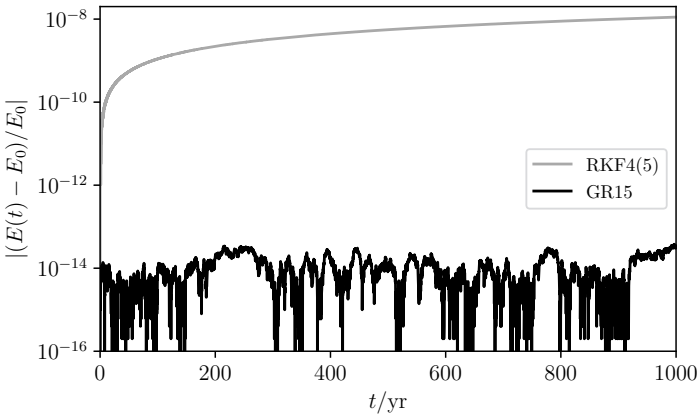


Figure 8.6
Relative error of the integration of the Solar System for 1,000 years.

Bob: There's one thing that is bugging me a bit…. When we integrated the five-body problem, the error seemed to be building. However, when integrating the Solar System (which is a much more complicated system with more bodies and the integration is longer), the error was smaller!

Alice: Hmmm…. Well, thinking about it a bit more, I am not sure that the integration of the Solar System is necessarily a more "complicated" problem. It is true that we have more bodies (the dimension of the system increased) and we are integrating for a longer time span (about six revolutions of Neptune), but the interaction between the planets is not as strong. They describe quasi-circular orbits and their mutual attraction does not have a significant effect. Also, the mass of the Sun (and therefore its gravitational attraction) clearly dominates over the rest. On the other hand, in our toy five-body problem, all particles have similar masses and their mutual interactions are strong. The orbit is not really stable if you remember our first tests! I'm pretty sure that the strength of the interactions plays a more important role in the accuracy than the integration time or the number of bodies.

Bob: Ah, that makes a lot of sense! It looks like we have an integrator that we can finally use to carry out some serious simulations of extrasolar planetary systems!

8.11 Code Review

The GR15 integrator required several auxiliary functions to complete intermediate tasks. Alice and Bob spend some time on their own going through the code to fully understand the details of the implementation.

Snippet 8.7

Function `radau_spacing`: Return the Gauss–Radau sequence for integration order 15.

```
def radau_spacing():
    nh = 8
    h = np.zeros(nh)
    h[0] = 0.0
    h[1] = 0.056262560536922146465652191032
    h[2] = 0.18024069173689236498757994280 9
    h[3] = 0.35262471711316963737390777017 1
    h[4] = 0.54715362633055538300144855765 2
    h[5] = 0.73421017721541053152321060830 6
    h[6] = 0.88532094683909576809035976293 2
    h[7] = 0.97752061356128750189117450042 9

    return h, nh
```

Snippet 8.8

Function `compute_bs_from_gs`: Compute $\mathbf{B}_i$ from $\mathbf{G}_j$ up to `ih` using equation (8.41).

```
def compute_bs_from_gs( b, g, ih, c ):

    if ( ih >= 1 ):
        b[0,:] = c[0,0]*g[0,:] + c[1,0]*g[1,:] + c[2,0]*g[2,:] + c[3,0]*g
            [3,:]    + c[4,0]*g[4,:] + c[5,0]*g[5,:] + c[6,0]*g[6,:]
    if ( ih >= 2 ):
        b[1,:] = c[1,1]*g[1,:] + c[2,1]*g[2,:] + c[3,1]*g[3,:] + c[4,1]*g
            [4,:]    + c[5,1]*g[5,:] + c[6,1]*g[6,:]
    if ( ih >= 3 ):
        b[2,:] = c[2,2]*g[2,:] + c[3,2]*g[3,:] + c[4,2]*g[4,:] + c[5,2]*g
            [5,:]    + c[6,2]*g[6,:]
    if ( ih >= 4 ):
        b[3,:] = c[3,3]*g[3,:] + c[4,3]*g[4,:] + c[5,3]*g[5,:] + c[6,3]*g
            [6,:]
    if ( ih >= 5 ):
        b[4,:] = c[4,4]*g[4,:] + c[5,4]*g[5,:] + c[6,4]*g[6,:]
    if ( ih >= 6 ):
        b[5,:] = c[5,5]*g[5,:] + c[6,5]*g[6,:]
    if ( ih >= 7 ):
        b[6,:] = c[6,6]*g[6,:]

    return b
```

Snippet 8.9

Function `compute_cs`: Precompute the coefficients c_{ij} in equation (8.47).

```
def compute_cs():
    c = np.zeros((8, 8))
    for i in range(0, 8);
        c[i,i] = 1.0
    c[1,0] = -0.056262560536922146465652 2

    c[2,0] =  0.010140802830063629986481803995496414174134953110 8
    c[2,1] = -0.236503252273814511453232 1
```

```
    c[3,0] = -0.00357589772925161759493445892845671873620404645937 28
    c[3,1] =  0.09353769525946206589574845561035371499343547051116
    c[3,2] = -0.58912796938698414882713 99

    c[4,0] =  0.00195656540994722107690056723796686106481798381409 13
    c[4,1] = -0.05475538688906868644080843067105502260202838258449 5
    c[4,2] =  0.41588120008230686168862193041156933067050816537030
    c[4,3] = -1.13628159571753953182858 85

    c[5,0] = -0.00143653023637089154244595541941532471344385719621 98
    c[5,1] =  0.04215852772126870770729734781320320298022813539585 8
    c[5,2] = -0.36009959650205681228976647408968845289781580280782
    c[5,3] =  1.25015071184069102585054411868575276940775655160 84
    c[5,4] = -1.87049177293295006335179 91

    c[6,0] =  0.00127179030902686774929431176229642208894846661475 01
    c[6,1] = -0.03876035791590677036990462684990189910850215835438 3
    c[6,2] =  0.36096224345284598322533983078129066420907893718190
    c[6,3] = -1.46688420840042696437015534613784801487616555997 54
    c[6,4] =  2.90613625930842930142379143711739467053842124792 46
    c[6,5] = -2.75581271977204583144215 89

    c[7,0] = -0.00124320124320124320124320138490387192371339402381 63
    c[7,1] =  0.03916083916083916083916084122758265723928915988756 3
    c[7,2] = -0.39160839160839160839160841545895262429018228668896
    c[7,3] =  1.79487179487179487179487190278667387118625513376 29
    c[7,4] = -4.30769230769230769230769252318539007235033385863 35
    c[7,5] =  5.60000000000000000000000019611293002337688038455 26
    c[7,6] = -3.73333333333333333333333 34
    return c
```

Snippet 8.10

Function `compute_gs`: Compute $\mathbf{G}_i$ up to `ih` using equation (8.43). The acceleration $\mathbf{F}_i$ is stored in `ddys[i,:]`.

```
def compute_gs( g, r, ddys, ih ):

    # Retrieve required accelerations:
    F1 = ddys[0,:]
    F2 = ddys[1,:]
    F3 = ddys[2,:]
    F4 = ddys[3,:]
    F5 = ddys[4,:]
    F6 = ddys[5,:]
    F7 = ddys[6,:]
    F8 = ddys[7,:]

    # Update g's using the accelerations:
    if ( ih >= 2 ):
        g[1,:] = ((F3 - F1) * r[2,0] - g[0,:]) * r[2,1]
    if ( ih >= 3 ):
```

```
        g[2,:] = (((F4 - F1) * r[3,0] - g[0,:]) * r[3,1] - g[1,:]) * r
            [3,2]
    if ( ih >= 4 ):
        g[3,:] = ((((F5 - F1) * r[4,0] - g[0,:]) * r[4,1] - g[1,:]) * r
            [4,2] -    g[2,:]) * r[4,3]
    if ( ih >= 5 ):
        g[4,:] = (((((F6 - F1) * r[5,0] - g[0,:]) * r[5,1] - g[1,:]) * r
            [5,2] -   g[2,:]) * r[5,3] - g[3,:]) * r[5,4]
    if ( ih >= 6 ):
        g[5,:] = ((((((F7 - F1) * r[6,0] - g[0,:]) * r[6,1] - g[1,:]) * r
            [6,2] -  g[2,:]) * r[6,3] - g[3,:]) * r[6,4] - g[4,:]) * r
            [6,5]
    if ( ih >= 7 ):
        g[6,:] = (((((((F8 - F1) * r[7,0] - g[0,:]) * r[7,1] - g[1,:]) * r
            [7,2] - g[2,:]) * r[7,3] - g[3,:]) * r[7,4] - g[4,:]) * r[7,5]
             - g[5,:]) *     r[7,6]

    return g
```

Snippet 8.11

Function `compute_rs`: Return the values of the coefficients r_{ij} defined in equation (8.44).

```
def compute_rs():

    r = np.zeros((8,8))

    r[1,0] = 17.773808914078000840752659565672904106978971632681
    r[2,0] = 5.548136718537216505692821614076506175857933694139 8
    r[3,0] = 2.835876078644438678252010442804243740087900314794 9
    r[4,0] = 1.827640267517597829794607758737120438565162845715 4
    r[5,0] = 1.362007816062469496937000629244565099419737192831 8
    r[6,0] = 1.129533875336789902732286154272859350976814876910 5
    r[7,0] = 1.022996329823486745838611907193963677902415913410 3

    r[2,1] = 8.065938648381886688537125668968715441226741618020 7
    r[3,1] = 3.374249976962635259942035818826746044833008769674 3
    r[4,1] = 2.037111835358584782794915916156655492184179259040 4
    r[5,1] = 1.475040217560411547921848248016740402474012743135 8
    r[6,1] = 1.206187666058445616625203629964622779147420352780 1
    r[7,1] = 1.085472193938642384046724317256891386203011867982 7

    r[3,2] = 5.801001559264061482328677889391888015574397916425 1
    r[4,2] = 2.725442211808226283774272200349133472971145028880 7
    r[5,2] = 1.805153580140251260439114743544867958657441408069 3
    r[6,2] = 1.418278263734739153771378367485832843371364069251 8
    r[7,2] = 1.254264622281877765990542246586824958686236972582 6

    r[4,3] = 5.140624105810934228636319909150443792933518966830 4
    r[5,3] = 2.620644926387035081154181603193307469673022772981 2
    r[6,3] = 1.877242496186810097216992028310965833542744608441 1
    r[7,3] = 1.600266549490816260991671694916115036632325915440 8

    r[5,4] = 5.345976899871107514121490963227789804577033666035 4
    r[6,4] = 2.957116017290455747807104020424555650835277692976 2
```

```
    r[7,4] = 2.32359830021969422283253454510916680736089558350 34

    r[6,5] = 6.617662013702424487447128489119392573703329149 1748
    r[7,5] = 4.109975778344559086238576182406878214472308263 3980

    r[7,6] = 10.846026190236844684706431007823415424143683137181
    return r
```

Snippet 8.12
Function `refine_bs`: Refine the coefficients $\mathbf{B}_i$ using equation (8.60).

```
def refine_bs( b, q, E, imode ):
    if ( imode != 0 ):
        bd = b - E
    else:
        bd = b * 0

    # Compute the powers of q:
    q2 = q * q
    q3 = q2 * q
    q4 = q3 * q
    q5 = q4 * q
    q6 = q5 * q
    q7 = q6 * q

    E[0,:] = q * (b[6,:] * 7.0 + b[5,:] * 6.0 + b[4,:] * 5.0 + b[3,:] *
        4.0       + b[2,:] * 3.0 + b[1,:] * 2.0 + b[0,:])
    E[1,:] = q2 * (b[6,:] * 21.0 + b[5,:] * 15.0 + b[4,:] * 10.0 + b[3,:]
        * 6.0   + b[2,:] * 3.0 + b[1,:])
    E[2,:] = q3 * (b[6,:] * 35.0 + b[5,:] * 20.0 + b[4,:] * 10.0 + b[3,:]
        * 4.0   + b[2,:])
    E[3,:] = q4 * (b[6,:] * 35.0 + b[5,:] * 15.0 + b[4,:] * 5.0 + b[3,:])
    E[4,:] = q5 * (b[6,:] * 21.0 + b[5,:] * 6.0 + b[4,:])
    E[5,:] = q6 * (b[6,:] * 7.0 + b[5,:])
    E[6,:] = q7 * b[6,:]

    b = E + bd
    return b, E
```

Snippet 8.13
Function `gauss_radau15_step`: Advance one integration step using the Gauss–Radau integrator.

```
def gauss_radau15_step(y0, dy0, ddy0, ddys, dt, t, tf, nh, hs, bs0, bs, E,
     g, r, c, tol, exponent, fac, imode, ode, params):

    istat = 0
    while (True):
        # Variable number of iterations in PC:
        for ipc in range(0, 12):
            ddys = ddys * 0

            # Advance along the Radau sequence:
            for ih in range(0, nh):
```

```
            # Estimate position and velocity with bs and current h:
            y  = approx_pos(y0, dy0, ddy0, hs[ih], bs, dt)
            dy = approx_vel(    dy0, ddy0, hs[ih], bs, dt)

            # Evaluate force function and store:
            ddys[ih,:] = ode(t + hs[ih] * dt, y, dy, params)
            g  = compute_gs(g, r, ddys, ih)
            bs = compute_bs_from_gs(bs, g, ih, c)

        # Estimate convergence of PC:
        db7 = bs[-1,:] - bs0[-1,:]
        if ( max(abs(db7)) / max(abs(ddys[-1,:])) < 1e-16 ):
            break
        bs0 = bs

    # Advance the solution:
    y  = approx_pos(y0, dy0, ddy0, 1., bs, dt)
    dy = approx_vel(    dy0, ddy0, 1., bs, dt)
    ddy= ode(t + dt, y, dy, params)

    # Estimate relative error
    estim_b7 = max(abs(bs[-1,:])) / max(abs(ddy))
    err      = (estim_b7 / tol)**(exponent)

    # Step-size required for next step:
    dtreq = dt / err

    # Accept the step
    if (err <= 1):

        # Report accepted step:
        istat = 1

        # Advance time:
        t = t + dt

        # Update b coefficients:
        bs0   = bs

        # Refine predictor-corrector coefficients for next pass:
        bs, E = refine_bs(bs, dtreq / dt, E, imode)

        # Normal refine mode:
        imode = 1

        # Check if tf was reached:
        if ( t >= tf ):
            istat = 2

    # Step size for next iteration:
    if ( dtreq / dt > 1.0 / fac ):
        dt = dt / fac
    elif ( dtreq < 1e-12 ):
        dt = dt * fac
```

```
        else:
            dt = dtreq

        # Correct overshooting:
        if ( t + dt > t f):
            dt = tf - t

        # Return if the step was accepted:
        if ( istat > 0 ):
            break

    return y, dy, ddy, t, dt, g, bs, E, bs0, istat, imode
```

Snippet 8.14
Function `integrate_gauss_radau15`: Gauss–Radau integrator of the fifteenth order.

```
def integrate_gauss_radau15( y0, dy0, t0, tf, tol, ode, params ):
    # Recommended tolerance: 1e-9

    # Limit the change in time step:
    fac  = 0.25

    # For fixed step integration, choose exponent = 0
    exponent = 1.0 / 7.0

    nstore = 10000
    # Dimension of the system
    dim = len(y0)

    sol_pos = np.zeros((nstore,dim))
    sol_vel = np.zeros((nstore,dim))
    sol_time= np.zeros(nstore)

    sol_pos[0,:] = y0
    sol_vel[0,:] = dy0

    # Tolerance Predictor-Corrector
    tolpc = 1e-16

    # Pre-compute Radau spacings, r, and c:
    hs, nh = radau_spacing()
    r = compute_rs()
    c = compute_cs()


    # Initialize
    bs0 = np.zeros((nh-1,dim))
    bs  = np.zeros((nh-1,dim))
    g   = np.zeros((nh-1,dim))
    t   = t0
    E   = np.zeros((nh-1,dim))
    ddys= np.zeros((nh,dim))

    # Initial time step
```

```
    dt, ddy0 = initial_time_step(y0, dy0, t0, tol, ode, params)

    integrate = True
    istep = 1
    imode = 0
    while integrate:

        # Advance one step and return:
        y, dy, ddy, t, dt, g, bs, E, bs0, istat, imode =
            gauss_radau15_step(y0, dy0, ddy0, ddys, dt,
            t, tf, nh, hs, bs0, bs, E, g, r, c, tol, exponent, fac, imode,
                ode, params)

        # Detect end of integration:
        if (istat == 2):
            integrate = False

        # Store solution
        sol_pos[istep,:] = y
        sol_vel[istep,:] = dy
        sol_time[istep]  = t
        print(istep, t, tf)

        # Update step
        istep+= 1
        y0    = y
        dy0   = dy
        ddy0  = ddy

        if (istep == sol_pos.shape[0]):
            sol_pos  = np.concatenate((sol_pos,sol_pos[0:nstore, :]))
            sol_vel  = np.concatenate((sol_vel,sol_vel[0:nstore, :]))
            sol_time = np.concatenate((sol_time,sol_time[0:nstore]))

    return sol_time[0:istep], np.concatenate((sol_pos[0:istep,:], sol_vel
        [0:istep,:]), axis=1)
```

9 Symplectic Map for Long-Term Integration

Overview. After having propagated the Solar System for 1,000 years, Bob and Alice are excited to start working with longer propagations using their brand-new Gauss–Radau 15 integrator. However, they soon realize that they will need either another kind of integrator or to optimize their current code if they want to run long-term dynamical simulations. The long-term integration of a planetary system is a challenging problem for conventional numerical integrators. The accumulation of errors from millions of operations depreciates the accuracy of the solution, and many hours of computing time may be required to complete the simulation. In this context, approximate symplectic maps can speed up the propagation substantially. Prof. Starmover introduces to Alice and Bob the integration algorithm proposed by Wisdom and Holman (1991), and they use it to propagate the Solar System for 1 Myr. The chapter ends with code review.

Alice and Bob meet the following day.

Alice: Good morning, Bob! Ready for some orbit propagations?

Bob: Yeah! I was pretty tired last night, but it was definitely worth the effort. The Gauss–Radau integrator looks very promising. Actually, while I was coming here this morning, I thought that we should try simulating the Solar System for a million years. Imagine how cool that would be!

Alice: I have to admit it does sound cool…. Let's aim for that goal, a million-year propagation.

Bob: Let's do it! I just changed the time span to 10^6 and I am ready to go.

Alice: Go for it! Oh … hmmm … wait, we haven't really thought about how long this is going to take. We can do a simple check. Can you please propagate the orbits for just one year and measure the runtime?

Bob: Sure. It took about sixteen seconds to propagate one year.

Alice: So, if we propagate 10^6 years, it will take … 185 days!

Bob: You have to be kidding, right? That's about six months! I don't know about you, but I will not wait six months for this simulation to finish.

Alice: I'm afraid we will not be able to use this integrator for such a long propagation.

Bob: Yeah, but trust me, I'm sure I can optimize the code and speed everything up!

Alice: I do trust you, Bob, but six months is six months….

9.1 Understanding Time Scales

Bob: True, I felt like the Gauss–Radau 15 was the solution to any problem we would ever encounter!

Alice: Yeah, I know…. It worked really well, but it might not be the best solution for propagating the system forward for millions and millions of years. There are a lot of intermediate steps that seem to slow down the integration when we face long time scales. We'll have to discuss these issues with Prof. Starmover.

Alice and Bob head into Prof. Starmover's office to go over the results they obtained using their new integrator, hoping that she will have new ideas about faster methods for propagating orbits for long periods of time.

Professor: Alice, Bob, please come in and have a seat. How did the tests with the Gauss–Radau integrator go?

Alice: We were able to simulate the evolution of the Solar System over the next 1,000 years! That was pretty exciting.

Bob: And the energy was conserved almost to machine precision over the entire propagation. But then we tried increasing the time span to a million years and realized that it would basically take forever. The integration was going to take months, literally.

Professor: Well, let's dig a bit into that. The integrator starts from t_0 and advances to t_f taking steps of size Δt, which changes according to the error estimates. If Δt is very small compared to the time interval $t_f - t_0$, then the integrator will need to take many steps to reach the final time. We could be talking about millions of steps. At each step, the force function is evaluated several times during the predictor-corrector phase, and that takes computing time. Especially when multiplied by a factor of several millions. That's the reason why integrations can take a very long time.

Alice: That makes sense. I'm not sure off the top of my head what we chose for the time step, but it shouldn't be hard to check. Right, Bob?

Bob: Oh, sure. I will just add a `print` statement right after computing the step size for the next iteration. I will print it together with the integration time:

```
print(t, dt)
```

Alice: Great! Let's rerun the simulation….

Bob: Wait, I don't want to start the integration again. It takes forever! Shouldn't we reduce the integration time to just a few years?

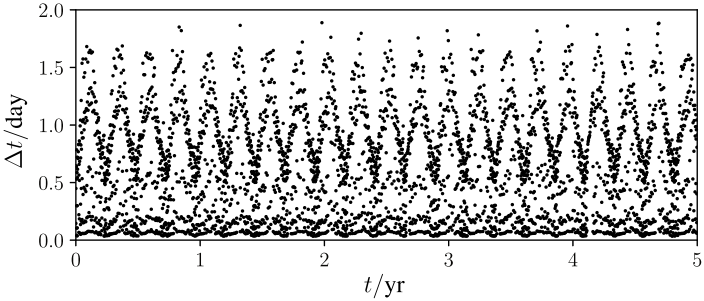


Figure 9.1
Integration step size when propagating the Solar System with Gauss–Radau 15.

Alice: Yes, that should not affect the step-size control. The evolution of the time step will be the same.

Bob: Okay. Let's start by integrating for five years and then see what we get. Figure 9.1 shows the size of the time step as a function of the integration time. The step size turns out to be less than a day, most of the time!

Alice: Interesting…. If we think about the orbit of the Earth, then the integrator is taking over 300 steps to integrate the orbit. That's almost one step per degree. That doesn't sound very impressive; it actually sounds like a lot of steps for such a simple orbit. It's almost circular and changes very little.

Professor: That's a very good point, Alice. The size of the time step is actually driven by the fastest *time scale* relevant to the problem. Time scales are the characteristic times over which a physical system evolves. For example, in the Solar System, there are different time scales that are associated with the orbital period of each planet. Take a look at figure 9.2; it shows how the x component of the motion evolves over time for Mercury and Jupiter. Do you see how different the frequencies are? Keep in mind that the orbital period of Mercury is only eighty-eight days, whereas the orbital period of Jupiter is twelve years. As a result, in twelve years, Jupiter will have completed just one revolution while Mercury will have rotated almost fifty times around the Sun. If the integrator takes large time steps (well suited for integrating the orbit of Jupiter), it will fail to capture accurately the evolution of Mercury. That's why during the integration, the step size will become small enough to model the fastest motion accurately, in this case the motion of Mercury. We wouldn't need such small time steps if we were propagating Jupiter's orbit alone, but that's the price we have to pay.

Let's run a simple experiment to check this. Could you please integrate the Solar System without Mercury and Venus and record the step size?

Bob: Let me change the initial conditions…. I had to shift the components of the array a bit, but I got it. I plotted the result in figure 9.3.

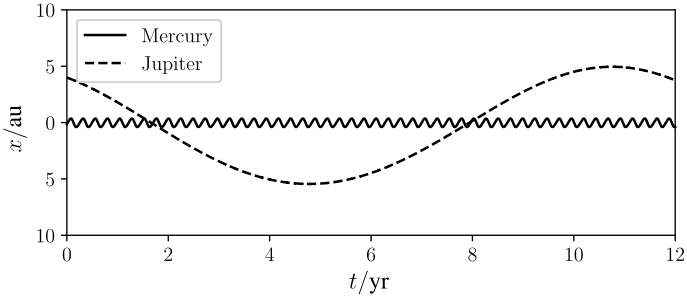


Figure 9.2
Time scales associated with the motion of Mercury and Jupiter.

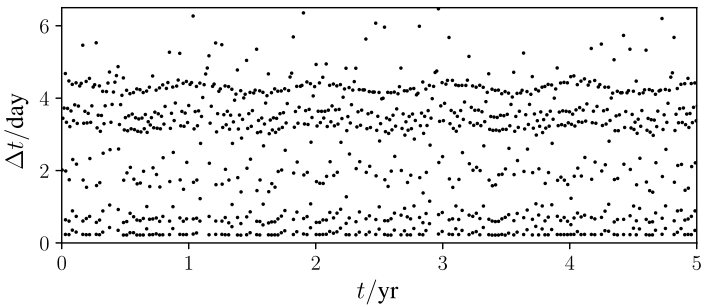


Figure 9.3
Integration step size when propagating the Solar System without Mercury and Venus.

Alice: Interesting, the average time step is now about four times larger than before!

Professor: Can you guess why? What is the fastest time scale now?

Alice: Without Mercury and Venus, it would be the Earth.

Professor: Right. And how does the period of the Earth compare to the period of Mercury?

Alice: It's $365/88 \approx 4.1$ times longer. Ah, I see your point. The fastest time scale is now four times slower and the average time step became almost four times larger. Pretty cool!

9.2 Long-Term Evolution

Bob: Do we always need to take many steps per orbit? Can't we integrate several orbits in one single step? That would speed things up quite a lot.

Alice: I agree. If you think about it, it's not like the orbits of the planets are changing a lot after one revolution. It would be a mess if the orbit of the Earth changed significantly every year!

Professor: Yes, it typically takes hundreds of thousands of years to see significant changes in the orbits of the planets. In this context, it is important to recover the definition of orbital elements and secular evolution (see chapter 2). Orbital elements are *constants* of the two-body problem that define the orbit geometry and orientation. We only need six numbers to completely characterize a two-body orbit. Actually, we only need five elements to define the orbit in space $(a, e, i, \omega, \Omega)$ because the anomaly (true, mean, eccentric, or hyperbolic) tells us where the planet is along the orbit as a function of time. However, if we want to define the orbit in Cartesian coordinates $(x, y, z, \dot{x}, \dot{y}, \dot{z})$, we will need a long table of data with the coordinates as a function of time because they are not constant. The key concept is that orbital elements remain constant, whereas coordinates change "rapidly," with a frequency given by the orbital period. Another way to think about this is that elements define the curve that represents the orbit, whereas Cartesian coordinates just provide single points along the orbit.

If instead of a simple two-body orbit we have a multiplanet system like the Solar System, the orbital elements will no longer be constant. But, if the mutual attraction between planets is small compared to the attraction from the central body, the orbital elements will change very slowly. Since your integrator works in Cartesian coordinates, it takes many steps per orbit even though the elements hardly change at all, as Alice pointed out.

Bob: Eureka! Using orbital elements will solve the problem.

Professor: Well, yes and no. Orbital elements work well when there are no violent changes in the orbits and when it is clear what is orbiting around what. Using orbital elements can get very complicated if you don't have a clear hierarchy (a large central body and smaller bodies orbiting around it). This scenario might happen in a star cluster, when the moon of a planet escapes the planet's Hill sphere, or when asteroids get captured by a planet. In addition, if the orbital perturbations are very strong, the orbital elements might change as fast as the coordinates themselves, so there will be no advantage to using orbital elements at all. You also have to keep in mind that working with elements requires intermediate transformations from orbital elements to Cartesian coordinates that can slow down the propagation.

Alice: What should we do then?

Professor: Do you remember the discussion about secular evolution (see chapter 2)? We went through the equations that model the long-term evolution of the system, obviating the orbital frequency. That is the key when one is interested in the long-term evolution of a planetary system. For example, if you are interested in knowing how the Solar System evolves over the next million years, you should not worry about periodic changes that oc-

cur on a daily basis. There are two well-known techniques for getting rid of short-period variations:

1. Averaging orbital motion
2. Simplifying the dynamical model

I should mention some caveats of these statements, though. In particular, for a general multiplanet system (not the Solar System), mean motion resonances might be important, and they don't necessarily operate on short time scales. Also, there can be an interplay between mean-motion resonances, evection resonances, and secular evolution in the long-term evolution that is currently not well understood.

Bob: I have a question, Prof. Starmover…. What are *resonances*?

Professor: It's a phenomenon that appears when the frequency of a periodic force acting on the system matches the frequency at which the system responds. The classical example is a child on a swing; if you want the swing to go higher and higher, when should you give it a push?

Alice: When it reaches its highest point, right?

Professor: That's right, and that means that the frequencies at which the swing moves and at which you push are the same, leading to resonance. Resonances result in the amplification of the amplitude of the oscillations. In orbital mechanics, if the frequency of a third-body perturbation matches the orbital frequency of a body, it might result in eccentricity variations that increase in amplitude, for example.

9.2.1 Orbital Averaging

Professor: Since we are interested in understanding the evolution of the orbits after millions of revolutions, we could just study what is the average change in the orbital elements after each revolution instead of modeling each revolution accurately. To understand the average change of a function $f(t)$ over one orbital period P, we will take the integral:

$$\langle f(t)\rangle = \frac{1}{P}\int_0^P f\,\mathrm{d}t = \frac{1}{2\pi}\int_0^{2\pi} f\,\mathrm{d}\mathcal{M}. \tag{9.1}$$

The second identity comes from the definition of the mean anomaly, $\mathcal{M} = n(t-\tau)$. But let's take a very simple example to see how averaging works. What is the long-term behavior of the following system?

$$\frac{\mathrm{d}f}{\mathrm{d}t} = t + \sin t = g_1(t) + g_2(t). \tag{9.2}$$

Alice: This system is easy to integrate by hand,

$$f(t) = \frac{t^2}{2} - \cos t + (f_0 + 1). \tag{9.3}$$

In the limit $t \to \infty$, the first term will be very large, while $\cos t$ only takes values between -1 and 1. Thus, I'd say that $f(t)$ will go to infinity with a scaling like $t^2/2$.

Professor: Very good, Alice. But let's now try to arrive at the same conclusion without explicitly solving the integral because we cannot do that in the N-body problem. There is a periodic term ($g_2(t) = \sin t$) with period $P = 2\pi$ responsible for the fast oscillatory motion. We can now look at the contribution of each term averaged after one period,

$$\langle g_1(t)\rangle = \frac{1}{P}\int_0^P g_1(t)\,\mathrm{d}t = \frac{1}{2\pi}\int_0^{2\pi} t\,\mathrm{d}t = \frac{t^2}{4\pi}\bigg|_0^{2\pi} = \pi; \tag{9.4a}$$

$$\langle g_2(t)\rangle = \frac{1}{P}\int_0^P g_2(t)\,\mathrm{d}t = \frac{1}{2\pi}\int_0^{2\pi} \sin t\,\mathrm{d}t = -\frac{1}{2\pi}\cos t\bigg|_0^{2\pi} = 0. \tag{9.4b}$$

I used the symbol $\langle\square\rangle$ to denote the average. What averaging tells us is that $g_2(t)$ does not contribute to the long-term evolution of the system because its effects are canceled out every period. Thus, we can ignore $g_2(t)$ and integrate equation (9.2) using $g_1(t)$ only. This will provide the secular evolution:

$$f_{\mathrm{sec}}(t) = \int_0^t g_1(t) = \frac{t^2}{2}, \tag{9.5}$$

which governs the long-term dynamics. We could always add the short-period variations to the solution if we need them for accuracy, like

$$f_{\mathrm{sp}}(t) = \int_0^t g_2(t) = -\cos t, \tag{9.6}$$

although that would require some extra work when integrating the periodic component. This would provide a more accurate solution, though:

$$f(t) = f_{\mathrm{sec}}(t) + f_{\mathrm{sp}}(t), \tag{9.7}$$

but we might not care about the short-period terms.

This is of course a very simplified example, but the same idea can be applied to a planetary system. We would retain the secular terms to understand how the orbital elements evolve over time. Since there would be several frequencies, we would find short- and long-period terms that would be needed depending on how accurate we want to be. The result is an approximate solution written as a mathematical series, leading to analytic and semianalytic theories to propagate orbital motion. Historically, before the invention of computers, this was the method of choice to integrate orbits. For example, Delaunay (1860) published an amazingly accurate analytic solution for the motion of the Moon including 460 periodic terms, which significantly improved *lunar theory* at the time. You can find more information about orbital averaging in some excellent books like Brouwer and Clemence (1961), Battin (1999, chap. 10), and Gurfil and Seidelmann (2016, chaps. 12–13). Nowadays, analytic theories (sometimes called *general perturbation methods*) are still used to study the

very long-term evolution of dynamical systems or as fast propagators aboard spacecraft computers.

Alice: Are we averaging the equations of motion then?

Professor: We will focus on the second approach, using a simplified approximate model. But if you are interested, I can show you how to average the Hamiltonian with an example (see appendix D.2).

Bob: Well, let's move on and maybe we can revisit it later.

9.2.2 Approximate Model

Professor: The idea is based on the same principle: obviating rapid oscillations and retaining only the terms that contribute the most. When orbital elements change very slowly, it wouldn't be unreasonable to assume that they are constant over a short period of time. Then, we will correct them to account for the effects of perturbations and assume them to be constant again.

Bob: That would simplify things a lot! If the orbital elements were constant, everything would become very easy. We could use the expressions in section 2.3.2 or appendix B and we would be done.

Professor: You're right, Bob, and that's precisely why this approach speeds the integration up by several orders of magnitude. This, of course, comes at a price: we are losing accuracy because of the underlying assumptions. But if we are interested in understanding the long-term evolution of the system, this method is very convenient. We'll assume that each planet describes a two-body orbit with respect to the center of attraction. We'll say that each planet *drifts* during that phase because we will consider no external perturbations.

Alice: I see. Accuracy always takes more time, while speed sacrifices some accuracy. I guess the question now is how to update the orbital elements after each step.

9.3 Wisdom–Holman Integrator

Professor: Indeed, and that's what Wisdom and Holman (1991) addressed in their work. They suggested modeling the effect of the mutual attraction between planets as an instantaneous change in the velocity of the particles. This *velocity kick*, $\Delta\mathbf{v}$, leaves the position vectors unchanged. The resulting strategy ensures that the evolution of the position vectors is smooth and that there are no jumps. Changing the velocity accounts for the cumulative effect of the perturbing acceleration, $\mathbf{a}_p$, during the time step. Essentially,

$$\Delta\mathbf{v} = \mathbf{a}_p \Delta t. \tag{9.8}$$

There is a very elegant theoretical background behind this method, but we will focus on its practical implementation.

Bob: What do you mean by *perturbing* acceleration?

Professor: We know that the planets in the Solar System orbit around the Sun, which is the center of attraction. We could consider the two-body motion of the Sun and Mercury, the Sun and Venus, the Sun and the Earth, and so on. and obtain pretty good approximations of each orbit. But, to be precise, we should note that each planet is also attracted by the other planets, although their gravitational influence is of course not as strong as that of the Sun (if so, planets would be orbiting each other!). This effect modifies—or *perturbs*—the two-body solution slightly. Thus, we define perturbations as all the accelerations acting on a body that are different from the attraction of the central body. Take the Earth as an example; the attraction from the Sun is the main two-body term, while the attraction from Mars, Jupiter, and so on are perturbations.

9.3.1 Extending the Leapfrog Method

Bob: Makes sense. I was tempted to say that this is an easy method to implement and that we have everything we need, but every time I say that, we end up with a ton of new equations…. This time, I'm not falling for it. So, what are we missing?

Professor: Don't worry, Bob, we'll go step by step. First, there's the question of where to apply the velocity kick. If you remember the basics of the integration methods in chapter 5, you will recall that evaluating the right-hand side of the differential equations at the beginning, middle, or end of the time interval made a big difference. The Euler method used the solution at the beginning of the interval, and the performance improved significantly when we constructed the leapfrog scheme by taking the acceleration at the midpoint of the interval.

Bob: That's right, I remember that equation (5.32) defines the solution at the next step using the value of the acceleration at the middle. We used that rather curious notation for defining the velocity at step $n + 1/2$,

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \dot{\mathbf{y}}_{n+1/2}\Delta t. \tag{9.9}$$

Professor: Yes, and remember that this is not only more accurate but also leads to a symplectic integration scheme.

Bob: Right! The energy error doesn't grow over time.

Professor: The Wisdom–Holman (WH) integrator—or mapping—adopts a somewhat similar approach, which also results in a symplectic scheme. Say that we want to advance from t_n to $t_{n+1} = t_n + \Delta t$. To do that, we will propagate the dynamics assuming two-body orbits and correct the velocities to account for the cumulative effects of the perturbations over the

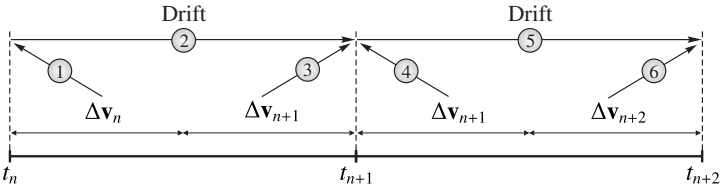


Figure 9.4
Schematic representation of the Wisdom–Holman integrator for two consecutive steps.

time interval Δt. The key idea is to split the velocity kick in half; the first half accounts for the effect of the acceleration at t_n over the first half of the time step,

$$\Delta\mathbf{v}_n \equiv \Delta\dot{\mathbf{y}}_n = \mathbf{a}_{p,n}\frac{\Delta t}{2}, \tag{9.10}$$

and the second half accounts for the effect of the acceleration at t_{n+1} over the second half of the time step,

$$\Delta\mathbf{v}_{n+1} \equiv \Delta\dot{\mathbf{y}}_{n+1} = \mathbf{a}_{p,n+1}\frac{\Delta t}{2}. \tag{9.11}$$

These two kicks combined model the effects of the perturbations over the course of the current time step Δt.

The integrator works as follows. To advance from time step t_n to t_{n+1}, given the positions and velocities at t_n ($\mathbf{y}_n$ and $\dot{\mathbf{y}}_n$), we perform the following procedure:

1. Compute the perturbation at t_n, $\mathbf{a}_{p,n}$, and use it to compute the velocity kick in equation (9.10).
2. Update the velocities at t_n using $\Delta\mathbf{v}_n \equiv \Delta\dot{\mathbf{y}}_n$, that is,
$$\dot{\mathbf{y}}_n \to \dot{\mathbf{y}}_n + \Delta\dot{\mathbf{y}}_n. \tag{9.12}$$
3. Use the positions $\mathbf{y}_n$ and updated velocities $\dot{\mathbf{y}}_n$ as initial conditions to propagate the two-body orbits (drift phase) from t_n to t_{n+1}. The result is $\mathbf{y}_{n+1}$ and $\dot{\mathbf{y}}_{n+1}$.
4. Compute the perturbation at t_{n+1}, $\mathbf{a}_{p,n+1}$, and use it to compute the velocity kick in equation (9.11).
5. Update the velocities at t_{n+1} using $\Delta\mathbf{v}_{n+1} \equiv \Delta\dot{\mathbf{y}}_{n+1}$, that is,
$$\dot{\mathbf{y}}_{n+1} \to \dot{\mathbf{y}}_{n+1} + \Delta\dot{\mathbf{y}}_{n+1}. \tag{9.13}$$

These steps lead to the positions and velocities at t_{n+1}, $\mathbf{y}_{n+1}$, and $\dot{\mathbf{y}}_{n+1}$. We will repeat them sequentially until reaching the final time. The diagram in figure 9.4 might help you understand the strategy. The numbers and the arrows indicate the order in which the steps are taken.

Bob: That helps a lot…. But wait, aren't steps 3 and 4 exactly the same? We are using the same kick $\mathbf{a}_{p,n+1}\Delta t/2$ in both of them, aren't we?

Professor: That's correct, Bob. When you advance several steps in sequence, you could combine the last kick of the current step with the first kick of the next step. Moreover, even though we are using a kick-drift-kick (KDK) strategy, we could obtain exactly the same results using a drift-kick-drift (DKD) strategy because of how consecutive steps are combined. Using DKD, we will drift for half a time step, update the velocity with a velocity kick $\mathbf{a}_{p,n+1/2}\Delta t$, and drift for the second half of the time step. Levison and Duncan (2013) used a KDK approach when developing the SWIFT[1] software package, while Rein and Tamayo (2015) adopted a DKD scheme for WHFAST.[2]

Bob: Alice? You've been really quiet for a while….

Alice: There's something bugging me about the drift phase…. If I got it right, we assume that each planet describes a two-body orbit around the Sun. But what about the motion of the Sun itself? The planets attract the Sun too, right? What if the central star were much smaller? Or even better, what if we had a binary star? I think we are missing something here.

Professor: That's a good point, Alice, but notice that I didn't specify what the center of attraction was when I talked about the drift phase. I did that on purpose because it is an important part of the method. We need to introduce the *Jacobi coordinates* to do things properly, especially when we start working with the Hamiltonian. For now, and to answer your question, keep in mind that we will not simply assume two-body orbits about the Sun.

Alice: Phew, I feel better. I thought I wasn't following you.

Professor: I'm glad that you brought that up and that it's more clear now. It will make more sense once you understand how the Jacobi coordinates work. But before going into further detail, let me show you the skeleton of what a single WH step would look like. Take a look at snippet 9.1, which implements the function `wh_advance_step`. We will work in Jacobi coordinates and use their Cartesian counterparts to compute the perturbing acceleration.

The input variable `x` is an array containing the position and velocity vectors of all bodies, just like the one you've been using in snippet 4.3. The only difference is that the positions and velocities are now given in Jacobi coordinates. For convenience, we will sort the bodies by their distance to the center of mass of the system, although this is not strictly required. The other interesting input is `accel`, the perturbing acceleration affecting each body (in Jacobi coordinates). You'll see that I am using an auxiliary variable `eta` that I will explain in a minute.

[1] https://www.boulder.swri.edu/~hal/swift.html

[2] https://rebound.readthedocs.io/en/latest/index.html

Snippet 9.1
Function `wh_advance_step`: Advance one step using the Wisdom–Holman integrator.

```
def wh_advance_step( x, t, dt, masses, nbodies, accel, G, eta):

    # Allocate array:
    jacobi = np.zeros(nbodies * 6)
    jacobi = x

    # Kick for dt / 2 using the acceleration at t:
    jacobi = wh_kick(jacobi, 0.5 * dt, nbodies, accel)

    # Drift for dt:
    jacobi = wh_drift(jacobi, dt, masses, nbodies, G, eta)

    # Convert from Jacobi to Cartesian coordinates for computing the
        acceleration:
    cart = jacobi2cart(jacobi, masses, nbodies, eta)

    # Compute acceleration at t + dt:
    accel = compute_accel(cart, jacobi, masses, nbodies, G, eta)

    # Kick for dt / 2 using the acceleration at t + dt:
    jacobi = wh_kick(jacobi, 0.5 * dt, nbodies, accel)

    # Return the Jacobi coordinates and the acceleration:
    return jacobi, accel
```

Bob: Well, the structure is pretty clear. And I see better now why we needed the second kick. With the "drifted" coordinates, we compute the acceleration at $t_n + \Delta t$, use it to determine the velocity kick, and then correct the velocity with the effect of the perturbation over the remaining $\Delta t/2$. And figure 9.4 shows that we can then call this function again and again using its output (`jacobi` and `accel`) as input (`x` and `accel`).

9.3.2 Jacobi Coordinates

Professor: That's exactly what we will do. Let's start by defining what the Jacobi coordinates are. We'll need to name the bodies first. Body 0 is the central body, body 1 is the innermost body of the system, body 2 is the second innermost, and so on. Keep in mind that we started counting at 0 so the index of the last body will be $N-1$. Don't get confused because we still have N bodies! It is just how we numbered them.

The Jacobi coordinates of body i are the position and velocity relative to the center of mass of the inner bodies (0 through $i-1$). To write it formally, let me use $\tilde{\mathbf{r}}$ and $\tilde{\mathbf{v}}$ to denote the Jacobi coordinates, while $\mathbf{R}$ and $\mathbf{V}$ are the usual inertial Cartesian coordinates relative to a certain fixed center. We then have

$$\tilde{\mathbf{r}}_i = \mathbf{R}_i - \mathbf{R}_{\mathrm{cm},i-1} \qquad \text{and} \qquad \tilde{\mathbf{v}}_i = \mathbf{V}_i - \mathbf{V}_{\mathrm{cm},i-1} \qquad \text{for} \quad 0 < i \le N-1. \tag{9.14}$$

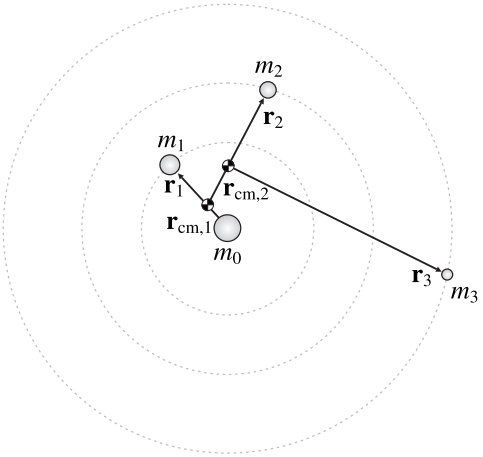


Figure 9.5
Graphical representation of the Jacobi coordinates.

In this equation, $\mathbf{R}_{\mathrm{cm},i-1}$ and $\mathbf{V}_{\mathrm{cm},i-1}$ are the position and velocity of the center of mass of all bodies from 0 up to $i-1$,

$$\mathbf{R}_{\mathrm{cm},i-1} = \frac{1}{\eta_{i-1}} \sum_{j=0}^{i-1} m_j \mathbf{R}_j \qquad \text{and} \qquad \mathbf{V}_{\mathrm{cm},i-1} = \frac{1}{\eta_{i-1}} \sum_{j=0}^{i-1} m_j \mathbf{V}_j, \tag{9.15}$$

and η_{i-1} denotes the total mass of the $i-1$ innermost bodies:

$$\eta_{i-1} = \sum_{j=0}^{i-1} m_j. \tag{9.16}$$

Bob: How should we treat bodies 0 and 1? We cannot define any interior bodies for them….

Professor: By definition, the Jacobi coordinates of body 0 are the position and velocity of the center of mass of the *entire* system,

$$\tilde{\mathbf{r}}_0 \equiv \mathbf{R}_{\mathrm{cm},N-1}, \qquad \text{and} \qquad \tilde{\mathbf{v}}_0 \equiv \mathbf{V}_{\mathrm{cm},N-1}. \tag{9.17}$$

As for body 1, we can actually still apply the definition in equation (9.14), just noting that $\mathbf{R}_{\mathrm{cm},0} \equiv \mathbf{R}_0$ and $\mathbf{V}_{\mathrm{cm},0} \equiv \mathbf{V}_0$. Take a look at figure 9.5. It might help visualizing the construction.

Bob: Oh, you're right about the case $i = 1$. It wasn't special at all.

Professor: As for the implementation of the transformation, I would recommend that you take a look at the paper by Rein and Tamayo (2015). They present a compact and efficient way to write the transformation.

Bob: Okay, I looked at the transformation and it reads

$$\begin{aligned}
&\mathbf{R}_{\rm cm} = m_0\mathbf{R}_0\\
&\mathbf{V}_{\rm cm} = m_0\mathbf{V}_0\\
&\textbf{for } i = 1,\ldots,N-1 \textbf{ do}\\
&\quad \tilde{\mathbf{r}}_i = \mathbf{r}_i - \mathbf{R}_{\rm cm}/\eta_{i-1}\\
&\quad \tilde{\mathbf{v}}_i = \mathbf{v}_i - \mathbf{V}_{\rm cm}/\eta_{i-1}\\
&\quad \mathbf{R}_{\rm cm} = \mathbf{R}_{\rm cm}(1+m_i/\eta_{i-1}) + m_i\tilde{\mathbf{r}}_i\\
&\quad \mathbf{V}_{\rm cm} = \mathbf{V}_{\rm cm}(1+m_i/\eta_{i-1}) + m_i\tilde{\mathbf{v}}_i\\
&\textbf{end do}\\
&\tilde{\mathbf{r}}_0 = \mathbf{R}_{\rm cm}/\eta_{N-1}\\
&\tilde{\mathbf{v}}_0 = \mathbf{V}_{\rm cm}/\eta_{N-1}
\end{aligned} \tag{9.18}$$

It updates the position and velocity of the center of mass sequentially. I wrote the transformation from Cartesian coordinates to Jacobi coordinates in snippet 9.5 (page 192). I'm assuming that the list η_i is computed somewhere else because it does not change throughout the integration.

Alice: Not that I don't trust your coding skills, Bob, but we should check the result with a simple example. You coded this very quickly!

Bob: Sure…. You're right, of course.

Alice: Okay, let's define a simple four-body system with masses $m_0 = 5$, $m_1 = 4$, $m_2 = 3$, and $m_3 = 2$. Let's also define the Cartesian positions and velocities of the bodies as follows:

$$\begin{aligned}
&\mathbf{R}_0 = [0,0,0], \quad &&\mathbf{R}_1 = [1,1,0], \quad &&\mathbf{R}_2 = [2,-1,0], \quad &&\mathbf{R}_3 = [4,3,0],\\
&\mathbf{V}_0 = [0,0,0], \quad &&\mathbf{V}_1 = [1,2,0], \quad &&\mathbf{V}_2 = [-1,0,0], \quad &&\mathbf{V}_3 = [-1,2,0].
\end{aligned} \tag{9.19}$$

Professor: Any reason why you set $\mathbf{R}_0$ and $\mathbf{V}_0$ to zero, Alice?

Alice: I just like it that way, I guess, but they could take any values, couldn't they?

Professor: That's correct, I just wanted to make sure you didn't see it as a requirement.

Alice: No problem. The Jacobi coordinates should be

$$\tilde{\mathbf{r}}_0 = \frac{5\mathbf{R}_0 + 4\mathbf{R}_1 + 3\mathbf{R}_2 + 2\mathbf{R}_2}{5+4+3+2} = [1.2857, 0.5, 0]; \tag{9.20a}$$

$$\tilde{\mathbf{r}}_1 = \mathbf{R}_1 - \mathbf{R}_0 = [1,1,0]; \tag{9.20b}$$

$$\tilde{\mathbf{r}}_2 = \mathbf{R}_2 - \frac{5\mathbf{R}_0 + 4\mathbf{R}_1}{5+4} = [1.5556, -1.4444, 0]; \tag{9.20c}$$

$$\tilde{\mathbf{r}}_3 = \mathbf{R}_3 - \frac{5\mathbf{R}_0 + 4\mathbf{R}_1 + 3\mathbf{R}_2}{5+4+3} = [3.1667, 2.9167, 0]. \tag{9.20d}$$

And if everything worked out fine so far, the velocities should be:

$$\tilde{\mathbf{v}}_0 = \frac{5\mathbf{V}_0 + 4\mathbf{V}_1 + 3\mathbf{V}_2 + 2\mathbf{V}_2}{5+4+3+2} = [-0.0714, 0.8571, 0]; \tag{9.21a}$$

$$\tilde{\mathbf{v}}_1 = \mathbf{V}_1 - \mathbf{V}_0 = [1, 2, 0]; \tag{9.21b}$$

$$\tilde{\mathbf{v}}_2 = \mathbf{V}_2 - \frac{5\mathbf{V}_0 + 4\mathbf{V}_1}{5+4} = [-1.4444, -0.8889, 0]; \tag{9.21c}$$

$$\tilde{\mathbf{v}}_3 = \mathbf{V}_3 - \frac{5\mathbf{V}_0 + 4\mathbf{V}_1 + 3\mathbf{V}_2}{5+4+3} = [-1.0833, 1.3333, 0]. \tag{9.21d}$$

Bob: Let's see…. Let me code the initial conditions:

```
nbodies = 4
x0 = np.array([0.0, 0.0, 0.0,  1.0, 1.0, 0.0,  2.0, -1.0, 0.0,  4.0, 3.0,
    0.0,  0.0, 0.0, 0.0,  1.0, 2.0, 0.0,  -1.0, 0.0, 0.0,  -1.0, 2.0,
    0.0])
masses = np.array([5.0, 4.0, 3.0, 2.0])
```

and then run our new function:

```
eta = np.zeros(nbodies)
eta[0] = masses[0]
for ibod in range(1, nbodies):
    eta[ibod] = masses[ibod] + eta[ibod - 1]

jacobi = cart2jacobi(x0, masses, nbodies, eta)
print(jacobi)
```

after computing the list η_i. Guess what? I nailed the result!

```
[ 1.28571429  0.5         0.          1.          1.          0.
  1.55555556 -1.44444444  0.          3.16666667  2.91666667  0.
 -0.07142857  0.85714286  0.          1.          2.          0.
 -1.44444444 -0.88888889  0.         -1.08333333  1.33333333  0.]
```

Alice: Cool!

Bob: Now it's your turn to derive the transformation from Jacobi to Cartesian coordinates.

Alice: Challenge accepted! Inverting equation (9.14) is easy:

$$\mathbf{R}_i = \tilde{\mathbf{r}}_i + \mathbf{R}_{\mathrm{cm},i-1}, \qquad \text{and} \qquad \mathbf{V}_i = \tilde{\mathbf{v}}_i + \mathbf{V}_{\mathrm{cm},i-1}. \tag{9.22}$$

Now we need to write the position and velocity of the center of mass in terms of the Jacobi coordinates. Let's see … for starters, we know that

$$\mathbf{R}_{\mathrm{cm},N-1} \equiv \tilde{\mathbf{r}}_0 \qquad \text{and} \qquad \mathbf{V}_{\mathrm{cm},N-1} \equiv \tilde{\mathbf{v}}_0. \tag{9.23}$$

Professor: One hint: how does $\mathbf{R}_{\mathrm{cm},i}$ relate to $\mathbf{R}_{\mathrm{cm},i-1}$?

Alice: From equation (9.15), we can write

$$\left.\begin{aligned}\eta_i \mathbf{R}_{\mathrm{cm},i} &= \textstyle\sum_{j=0}^{i} m_j \mathbf{R}_j \\ \eta_{i-1} \mathbf{R}_{\mathrm{cm},i-1} &= \textstyle\sum_{j=0}^{i-1} m_j \mathbf{R}_j\end{aligned}\right\} \implies \eta_i \mathbf{R}_{\mathrm{cm},i} - \eta_{i-1} \mathbf{R}_{\mathrm{cm},i-1} = m_i \mathbf{R}_i. \tag{9.24}$$

Oooh, I see where we are going with this. If we now plug in equation (9.22) we get

$$\eta_i \mathbf{R}_{\mathrm{cm},i} - \eta_{i-1} \mathbf{R}_{\mathrm{cm},i-1} = m_i \tilde{\mathbf{r}}_i + m_i \mathbf{R}_{\mathrm{cm},i-1} \implies \eta_i \mathbf{R}_{\mathrm{cm},i-1} = \eta_i \mathbf{R}_{\mathrm{cm},i} - m_i \tilde{\mathbf{r}}_i. \tag{9.25}$$

Bob: So? Uh…. Where are we going exactly?

Alice: Well, we can now compute $\mathbf{R}_{\mathrm{cm},i-1}$ from $\mathbf{R}_{\mathrm{cm},i}$ and $\tilde{\mathbf{r}}_i$. Since we know $\mathbf{R}_{\mathrm{cm},N-1}$ and the Jacobi coordinates, we can just compute all coordinates in reverse order starting from $i = N-1$ through $i = 0$, thanks to

$$\mathbf{R}_i = \tilde{\mathbf{r}}_i + \mathbf{R}_{\mathrm{cm},i} - \frac{m_i}{\eta_i}\tilde{\mathbf{r}}_i \qquad \text{and} \qquad \mathbf{V}_i = \tilde{\mathbf{v}}_i + \mathbf{V}_{\mathrm{cm},i} - \frac{m_i}{\eta_i}\tilde{\mathbf{v}}_i. \tag{9.26}$$

Professor: Good job, Alice!

Alice: Let me check how Rein and Tamayo (2015) implemented the transformation to make it efficient:

$$\begin{aligned}
&\mathbf{R}_{\mathrm{cm}} = \eta_{N-1}\tilde{\mathbf{r}}_0 \\
&\mathbf{V}_{\mathrm{cm}} = \eta_{N-1}\tilde{\mathbf{v}}_0 \\
&\textbf{for } i = N-1, \ldots, 1 \textbf{ do} \\
&\quad \mathbf{R}_{\mathrm{cm}} = (\mathbf{R}_{\mathrm{cm}} - m_i\tilde{\mathbf{r}}_i)/\eta_i \\
&\quad \mathbf{V}_{\mathrm{cm}} = (\mathbf{V}_{\mathrm{cm}} - m_i\tilde{\mathbf{v}}_i)/\eta_i \\
&\quad \mathbf{R}_i = \tilde{\mathbf{r}}_i + \mathbf{R}_{\mathrm{cm}} \\
&\quad \mathbf{V}_i = \tilde{\mathbf{v}}_i + \mathbf{V}_{\mathrm{cm}} \\
&\quad \mathbf{R}_{\mathrm{cm}} = \eta_{i-1}\mathbf{R}_{\mathrm{cm}} \\
&\quad \mathbf{V}_{\mathrm{cm}} = \eta_{i-1}\mathbf{V}_{\mathrm{cm}} \\
&\textbf{end do} \\
&\mathbf{R}_0 = \mathbf{R}_{\mathrm{cm}}/m_0 \\
&\mathbf{V}_0 = \mathbf{V}_{\mathrm{cm}}/m_0
\end{aligned} \tag{9.27}$$

I coded it in snippet 9.4. Bob, do you want to check the inverse transformation?

Bob: Let me run it and check the result:

```
cart = jacobi2cart(jacobi, masses, nbodies, eta)
print(cart)
```

This is what I got:

```
[ 0.  0.  0.  1.  1.  0.  2. -1.  0.  4.  3.  0.  0.  0.  0.  1.  2.  0.
 -1.  0.  0. -1.  2.  0.]
```

It does match the initial conditions!

9.3.3 Hamiltonian Splitting

Professor: It's very important that you test the transformations as you implement them. That will save you a lot of trouble in the future! Okay, now that you've sorted out the transformations, we need to compute the acceleration acting on each body. To do this, we will follow the Hamiltonian derivation introduced by Wisdom and Holman (1991). Incidentally, this exercise will also reveal why using Jacobi coordinates is important.

First, let's recover the Hamiltonian of Kepler's two-body problem using the definition in equation (2.55) and the potential in equation (2.61):

$$\mathcal{H}_{\text{Kep}} = T + V = \frac{p^2}{2m} - \frac{Gm_0 m}{r}. \tag{9.28}$$

Recall that $p = mv$ is the momentum. This Hamiltonian governs the motion of a body of mass m orbiting a central body of mass m_0. Next, we can write the Hamiltonian of the N-body problem using the kinetic and potential energies from equation (2.48):

$$\mathcal{H} = \sum_{i=0}^{N-1} \frac{p_i^2}{2m_i} - \frac{1}{2}\sum_{i=0}^{N-1}\sum_{\substack{j=0 \\ j\neq i}}^{N-1} G\frac{m_i m_j}{r_{ij}} = \sum_{i=0}^{N-1} \frac{p_i^2}{2m_i} - \sum_{i=0}^{N-2}\sum_{j=i+1}^{N-1} G\frac{m_i m_j}{r_{ij}}. \tag{9.29}$$

The potential depends on $r_{ij} = \|\mathbf{r}_{ij}\| = \|\mathbf{R}_j - \mathbf{R}_i\|$, meaning that the position vectors of the bodies are coupled. We don't have separate terms for each $\mathbf{R}_i$ since they appear in pairs inside each r_{ij}. This makes things complicated when we try to derive Hamilton's equations because we have to differentiate the Hamiltonian with respect to each of the position vectors $\mathbf{R}_i$.

Wisdom and Holman (1991) split the N-body Hamiltonian into a Keplerian part, where there are no interactions between bodies (no r_{ij} terms), plus a second part of the Hamiltonian that accounts for the mutual interaction between bodies. To achieve this, we'll need to use the Jacobi coordinates.

Bob: Why is that?

Professor: Because the Jacobi coordinates allow us to model the motion of an exterior body like a Keplerian orbit around the center of mass of the interior bodies. What we need to do is write the kinetic energy in Jacobi coordinates. Can you do that?

Alice: Let me see…. Essentially, we need the magnitude of the velocity in equation (9.22).

Bob: Can't you just multiply the velocity by itself? What we really need is the magnitude squared, and $(\mathbf{V}_i \cdot \mathbf{V}_i) = V_i^2$.

Alice: Sure, let's do that….

Professor: Let me stop you there. Why don't you take a closer look at equation (9.24) and see if it simplifies things? Your approach will work, of course, but it will take a lot of work.

Alice: Sounds good to me. For the velocities, we have

$$m_i \mathbf{V}_i = \eta_i \mathbf{V}_{\mathrm{cm},i} - \eta_{i-1} \mathbf{V}_{\mathrm{cm},i-1}. \tag{9.30}$$

If we just plug equation (9.22) into this expression, we obtain

$$m_i(\tilde{\mathbf{v}}_i + \mathbf{V}_{\mathrm{cm},i-1}) = \eta_i \mathbf{V}_{\mathrm{cm},i} - \eta_{i-1} \mathbf{V}_{\mathrm{cm},i-1} \implies m_i \tilde{\mathbf{v}}_i = \eta_i (\mathbf{V}_{\mathrm{cm},i} - \mathbf{V}_{\mathrm{cm},i-1}). \tag{9.31}$$

Aha, this is looking good. I can now square equations (9.30) and (9.31) and divide by m_i to get the right units (mass multiplied by velocity squared):

$$m_i V_i^2 = \frac{1}{m_i}(\eta_i \mathbf{V}_{\mathrm{cm},i} - \eta_{i-1} \mathbf{V}_{\mathrm{cm},i-1})^2 = \frac{1}{m_i}(\eta_i^2 V_{\mathrm{cm},i}^2 - 2\eta_i \eta_{i-1} \mathbf{V}_{\mathrm{cm},i} \cdot \mathbf{V}_{\mathrm{cm},i-1} + \eta_{i-1}^2 V_{\mathrm{cm},i-1}^2); \tag{9.32a}$$

$$m_i \tilde{v}_i^2 = \frac{\eta_i^2}{m_i}(\mathbf{V}_{\mathrm{cm},i} - \mathbf{V}_{\mathrm{cm},i-1})^2 = \frac{\eta_i^2}{m_i}(V_{\mathrm{cm},i}^2 - 2\mathbf{V}_{\mathrm{cm},i} \cdot \mathbf{V}_{\mathrm{cm},i-1} + V_{\mathrm{cm},i-1}^2). \tag{9.32b}$$

Bob: If you multiply the second expression by η_{i-1}/η_i, you can then subtract both expressions and cancel out the annoying $(\mathbf{V}_{\mathrm{cm},i} \cdot \mathbf{V}_{\mathrm{cm},i-1})$ term.

Alice: Good idea, Bob:

$$m_i\left(V_i^2 - \tilde{v}_i^2 \frac{\eta_{i-1}}{\eta_i}\right) = \eta_i V_{\mathrm{cm},i}^2 - \eta_{i-1} V_{\mathrm{cm},i-1}^2 \implies m_i V_i^2 = \eta_i V_{\mathrm{cm},i}^2 - \eta_{i-1} V_{\mathrm{cm},i-1}^2 + \frac{\eta_{i-1}}{\eta_i} m_i \tilde{v}_i^2. \tag{9.33}$$

We now have to sum up the contribution from all bodies,

$$2T = \sum_{i=0}^{N-1} m_i V_i^2 = \sum_{i=1}^{N-1}\left(\eta_i V_{\mathrm{cm},i}^2 - \eta_{i-1} V_{\mathrm{cm},i-1}^2 + m_i \tilde{v}_i^2 \frac{\eta_{i-1}}{\eta_i}\right) = \sum_{i=1}^{N-1} \frac{\eta_{i-1}}{\eta_i} m_i \tilde{v}_i^2 + \eta_{N-1} V_{\mathrm{cm},N-1}^2. \tag{9.34}$$

Professor: Looks good to me! In this expression, η_{N-1} is the total mass of the system and $V_{\mathrm{cm},N-1}$ is the velocity of the center of mass of the entire system. Under the assumption that the system is fixed in space, we will have $V_{\mathrm{cm},N-1} = 0$ and the last term will disappear. Alice, please introduce the auxiliary variables

$$\tilde{m}_i = \frac{\eta_{i-1}}{\eta_i} m_i, \qquad \text{for} \quad 0 < i < N \tag{9.35a}$$

$$\tilde{m}_0 = \eta_{N-1} \tag{9.35b}$$

and see what we get. Recall also the identities we wrote in equation (9.17).

Alice: Okay, now we obtain

$$2T = \sum_{i=0}^{N-1} m_i V_i^2 = \sum_{i=1}^{N-1} \tilde{m}_i \tilde{v}_i^2 + \tilde{m}_0 \tilde{v}_0^2. \tag{9.36}$$

I will write the kinetic energy in terms of the momenta $\tilde{\mathbf{p}} = \tilde{m}\tilde{\mathbf{v}}$,

$$T = \sum_{i=0}^{N-1} \frac{p_i^2}{2m_i} = \sum_{i=1}^{N-1} \frac{\tilde{p}_i^2}{2\tilde{m}_i} + \frac{\tilde{p}_0^2}{2\tilde{m}_0}. \tag{9.37}$$

Professor: I'll recommend that you take a look at Plummer (1918, p. 184) or Murray and Dermott (1999, p. 443) if you want more details about the transformation.

Alice and Bob take a moment to peruse the provided references.

Bob: Neat! We can now rewrite the first part of the Hamiltonian in equation (9.29) using Jacobi coordinates for $\tilde{p}_i = \tilde{m}_i \tilde{v}_i$:

$$\mathcal{H} = \frac{\tilde{p}_0^2}{2\tilde{m}_0} + \sum_{i=1}^{N-1} \frac{\tilde{p}_i^2}{2\tilde{m}_i} - \sum_{i=0}^{N-2} \sum_{j=i+1}^{N-1} G \frac{m_i m_j}{r_{ij}}. \tag{9.38}$$

But we still have the coupled terms….

Professor: Yes, we still need one final trick. If we add and subtract the term (Wisdom and Holman, 1991; Murray and Dermott, 1999, p. 443)

$$\sum_{i=1}^{N} G \frac{m_0 m_i}{\tilde{r}_i} = \sum_{i=1}^{N} G \frac{(m_0 \eta_i / \eta_{i-1}) \tilde{m}_i}{\tilde{r}_i} = \sum_{i=1}^{N} G \frac{\tilde{M}_i \tilde{m}_i}{\tilde{r}_i}, \tag{9.39}$$

using $\tilde{M}_i = (\eta_i / \eta_{i-1}) m_0$, we arrive at

$$\mathcal{H} = \frac{\tilde{p}_0^2}{2\tilde{m}_0} + \sum_{i=1}^{N-1} \left(\frac{\tilde{p}_i^2}{2\tilde{m}_i} - G \frac{\tilde{M}_i \tilde{m}_i}{\tilde{r}_i} \right) + \sum_{i=1}^{N-1} G \frac{m_0 m_i}{\tilde{r}_i} - \sum_{i=0}^{N-2} \sum_{j=i+1}^{N-1} G \frac{m_i m_j}{r_{ij}}. \tag{9.40}$$

Alice: There it is, the Keplerian Hamiltonian in Jacobi variables!

$$\mathcal{H}_{\text{Kep}} = \sum_{i=1}^{N-1} \left(\frac{\tilde{p}_i^2}{2\tilde{m}_i} - G \frac{\tilde{M}_i \tilde{m}_i}{\tilde{r}_i} \right). \tag{9.41}$$

I guess the second part is what you called the *interaction Hamiltonian*, which models the interaction between planets:

$$\mathcal{H}_{\text{inter}} = \sum_{i=1}^{N-1} G \frac{m_0 m_i}{\tilde{r}_i} - \sum_{i=0}^{N-2} \sum_{j=i+1}^{N-1} G \frac{m_i m_j}{r_{ij}}. \tag{9.42}$$

Professor: That's exactly right, Alice. The velocity kick accounts for $\mathcal{H}_{\text{inter}}$, whereas the drift phase accounts for $\mathcal{H}_{\text{Kep}}$.

9.3.3.1 Motion of the center of mass

Bob: Hey, one second. What about the first term, $\tilde{p}_0^2/2\tilde{m}_0$?

Professor: Let's call it

$$\mathcal{H}_0 = \frac{\tilde{p}_0^2}{2\tilde{m}_0} \tag{9.43}$$

and see what its contribution is. If you recall equations (9.17) and (9.35), you will see that $\tilde{\mathbf{p}}_0 = \eta_{N-1}\mathbf{V}_{\text{cm},N-1}$ is the total momentum of the system. And we know that it remains constant if there are no external perturbations, although it might not be zero if the center of mass is moving with constant velocity. This means that the Hamiltonian in equation (9.40) will still be constant if we ignore this term. Another way to look into this is to realize that the Hamiltonian does not depend on $\tilde{\mathbf{r}}_0$. As a result, if you evaluate Hamilton's equation for that term, you will get

$$\frac{\partial \mathcal{H}}{\partial \tilde{\mathbf{r}}_0} = -\frac{\mathrm{d}\tilde{\mathbf{p}}_0}{\mathrm{d}t} = 0, \tag{9.44}$$

which proves that $\tilde{\mathbf{p}}_0$ is a constant of motion.

We will ignore $\mathcal{H}_0$ in the following derivations because it does not contribute to the dynamics of the bodies. It just changes the Hamiltonian by a constant factor.

9.3.3.2 Keplerian Hamiltonian

Bob: Okay, so we agreed that equation (9.28) models the two-body dynamics of a body of mass m orbiting a central mass m_0. Then, equation (9.41) should model the two-body dynamics of each body i with mass $\tilde{m}_i$ orbiting a central mass $\tilde{M}_i$, shouldn't it?

Alice: Ah, that explains why we needed the Jacobi coordinates in the first place and also how the drift phase will work. We get an equivalent two-body problem for each body.

Professor: That's it. I'm glad you got it so quickly. Thanks to using Jacobi coordinates and having arrived at the Hamiltonian (9.41), we can now use the two-body equations for the relative motion of each body i that we derived in chapter 2, as long as we use the masses $\tilde{m}_i$ and $\tilde{M}_i$ and the coordinates $\tilde{r}_i$. Essentially, the Keplerian part is equivalent to solving the differential equation (Murray and Dermott, 1999, p. 444):

$$\frac{\mathrm{d}^2\tilde{\mathbf{r}}_i}{\mathrm{d}t^2} + \frac{G\tilde{M}_i}{\tilde{r}_i^3}\tilde{\mathbf{r}}_i = 0, \tag{9.45}$$

instead of the usual equation (2.19),

$$\frac{\mathrm{d}^2\mathbf{r}_i}{\mathrm{d}t^2} + \frac{G(m_0+m_i)}{r_i^3}\mathbf{r}_i = 0. \tag{9.46}$$

In just a moment, we will see how to drift the orbits, that is, how to propagate the two-body part (see section 9.3.5). Just keep in mind that we will use the Jacobi coordinates and $\tilde{M}_i$ as the central mass.

Alice: Noted!

9.3.3.3 Interaction Hamiltonian

Professor: But before that, we need to determine the acceleration on each body due to the interaction Hamiltonian. We are working with Jacobi coordinates so we seek the acceleration in Jacobi coordinates, $\mathrm{d}\tilde{\mathbf{v}}_i/\mathrm{d}t$. We will obtain it from Hamilton's equation:

$$\frac{\partial \mathcal{H}_{\text{inter}}}{\partial \tilde{\mathbf{r}}_i} = -\frac{\mathrm{d}\tilde{\mathbf{p}}_i}{\mathrm{d}t} = -\tilde{m}_i \frac{\mathrm{d}\tilde{\mathbf{v}}_i}{\mathrm{d}t}. \tag{9.47}$$

Bob: Sounds good. We then have to differentiate $\mathcal{H}_{\text{inter}}$ with respect to $\tilde{\mathbf{r}}_i$. I just realized from equation (9.42) that $\mathcal{H}_{\text{inter}}$ can be easily divided into two parts: one that is written in Jacobi coordinates and one that is written in Cartesian coordinates,

$$\mathcal{H}_{\text{inter}} = \mathcal{H}_{\text{Jac}} + \mathcal{H}_{\text{Cart}}, \tag{9.48}$$

with

$$\mathcal{H}_{\text{Jac}} = \sum_{i=1}^{N-1} G\frac{m_0 m_i}{\tilde{r}_i}; \tag{9.49a}$$

$$\mathcal{H}_{\text{Cart}} = -\sum_{i=0}^{N-2}\sum_{j=i+1}^{N-1} G\frac{m_i m_j}{r_{ij}}. \tag{9.49b}$$

Alice: Good catch, Bob.

Bob: Yep, no problem.

Alice: Differentiating $\mathcal{H}_{\text{Jac}}$ is now pretty easy,

$$\frac{\partial \mathcal{H}_{\text{Jac}}}{\partial \tilde{\mathbf{r}}_i} = -\frac{\mathrm{d}\tilde{\mathbf{p}}_i}{\mathrm{d}t} = -G\frac{m_0 m_i}{\tilde{r}_i^3}\tilde{\mathbf{r}}_i. \tag{9.50}$$

Similarly, we can find the derivatives of $\mathcal{H}_{\text{Cart}}$ with respect to $\mathbf{R}_i$ and look for a way to transform the acceleration from Cartesian to Jacobi coordinates:

$$\frac{\partial \mathcal{H}_{\text{Cart}}}{\partial \mathbf{R}_i} = -\frac{\mathrm{d}\mathbf{p}_i}{\mathrm{d}t} = -\sum_{\substack{j=0 \\ j\neq i}}^{N-1} \frac{G m_i m_j}{r_{ij}^3}\mathbf{r}_{ij}, \tag{9.51}$$

recalling that $\mathbf{r}_{ij} = \mathbf{R}_j - \mathbf{R}_i$.

Bob: That's a good plan. We can start by differentiating the velocity in equation (9.14) with respect to time:

$$\frac{\mathrm{d}\tilde{\mathbf{v}}_i}{\mathrm{d}t} = \frac{\mathrm{d}\mathbf{V}_i}{\mathrm{d}t} - \frac{\mathrm{d}}{\mathrm{d}t}\mathbf{V}_{\mathrm{cm},i-1} = \frac{\mathrm{d}\mathbf{V}_i}{\mathrm{d}t} - \frac{1}{\eta_{i-1}}\sum_{j=0}^{i-1} m_j \frac{\mathrm{d}\mathbf{V}_j}{\mathrm{d}t} \tag{9.52}$$

and then transform the velocities to the corresponding momenta by multiplying by $\tilde{m}_i = (\eta_{i-1}/\eta_i)m_i$:

$$\frac{\mathrm{d}\tilde{\mathbf{p}}_i}{\mathrm{d}t} = \frac{\eta_{i-1}}{\eta_i}\frac{\mathrm{d}\mathbf{p}_i}{\mathrm{d}t} - \frac{m_i}{\eta_i}\sum_{j=0}^{i-1}\frac{\mathrm{d}\mathbf{p}_j}{\mathrm{d}t}. \tag{9.53}$$

Alice: Great, we can now use this equation to transform the acceleration we obtained in equation (9.51) and add it to equation (9.50), and that should yield the acceleration, including the contributions from both $\mathcal{H}_{\mathrm{Jac}}$ and $\mathcal{H}_{\mathrm{Cart}}$:

$$\begin{aligned}\frac{\mathrm{d}\tilde{\mathbf{v}}_i}{\mathrm{d}t} &= \frac{\eta_i}{\eta_{i-1}}\frac{Gm_0}{\tilde{r}_i^3}\tilde{\mathbf{r}}_i + \frac{1}{\tilde{m}_i}\left(\frac{\eta_{i-1}}{\eta_i}\sum_{\substack{j=0\\ j\neq i}}^{N-1}\frac{Gm_im_j}{r_{ij}^3}\mathbf{r}_{ij} - \frac{m_i}{\eta_i}\sum_{j=0}^{i-1}\sum_{\substack{k=0\\ k\neq j}}^{N-1}\frac{Gm_jm_k}{r_{jk}^3}\mathbf{r}_{jk}\right)\\ &= \frac{G\tilde{M}_i}{\tilde{r}_i^3}\tilde{\mathbf{r}}_i + \sum_{\substack{j=0\\ j\neq i}}^{N-1}\frac{Gm_j}{r_{ij}^3}\mathbf{r}_{ij} - \frac{1}{\eta_{i-1}}\sum_{j=0}^{i-1}\sum_{\substack{k=0\\ k\neq j}}^{N-1}\frac{Gm_jm_k}{r_{jk}^3}\mathbf{r}_{jk}.\end{aligned} \tag{9.54}$$

Professor: We're almost done! Keep in mind that this equation is valid for $0 < i \leq N-1$ (not for $i = 0$) to avoid affecting the center of mass of the entire system. Let me also point out that this acceleration is equivalent to the perturbing acceleration that we introduced in equation (9.8) only that it is defined in Jacobi coordinates, $\tilde{\mathbf{a}}_p$. Murray and Dermott (1999, p. 445) showed that some terms can be simplified to provide:

$$\tilde{\mathbf{a}}_{p,i} \equiv \frac{\mathrm{d}\tilde{\mathbf{v}}_i}{\mathrm{d}t} = G\tilde{M}_i\left(\frac{\tilde{\mathbf{r}}_i}{\tilde{r}_i^3} - \frac{\mathbf{r}_{0i}}{r_{0i}^3}\right) - \frac{\eta_i}{\eta_{i-1}}\sum_{j=1}^{i-1}\frac{Gm_j}{r_{ji}^3}\mathbf{r}_{ji} + \sum_{j=i+1}^{N-1}\frac{Gm_j}{r_{ij}^3}\mathbf{r}_{ij} - \frac{1}{\eta_{i-1}}\sum_{j=0}^{i-1}\sum_{k=i+1}^{N-1}\frac{Gm_jm_k}{r_{jk}^3}\mathbf{r}_{jk}. \tag{9.55}$$

You can now implement the function `compute_accel`. Note that you will need the states in both Cartesian and Jacobi coordinates.

Bob: Finally! I coded equation (9.55) into snippet 9.6. I used the same input notation that you introduced in snippet 9.1.

9.3.4 Velocity Kick

Professor: It's time to implement the velocity kick, now that we know how to compute the acceleration. Keep in mind that we'll be working in Jacobi coordinates, although the function that you will be coding is not affected by the choice of coordinates.

Bob: Done. Equation (9.8) was straightforward to implement. I'm leaving the position vectors untouched, and I am only modifying the velocity components (stored from `nbodies * 3` through the end of the array). The function is in snippet 9.2.

Snippet 9.2
Function `wh_kick`: Compute the velocity kick.

```
def wh_kick( x, dt, nbodies, accel ):

    # Allocate output array:
    kick = np.zeros(nbodies * 6)
    kick[0:] = x[0:]

    # Apply kick (velocities only):
    kick[nbodies * 3: ] += accel * dt

    return kick
```

9.3.5 Drift Phase: Two-Body Propagator

Professor: After splitting the Hamiltonian, we saw that $\mathcal{H}_{\rm Kep}$ resulted in independent two-body problems for each body, written in Jacobi coordinates like equation (9.45). To implement the drift phase, you can use the function `propagate_kepler` that you wrote in appendix B (snippet B.2). It implements a Keplerian propagator using the f and g functions instead of the typical transformations from and to orbital elements.

Alice: Ah, I remember that function. We just have to be careful with what variables we use when calling the function. For each individual two-body problem, the central mass will be $\tilde{M}_i = (\eta_i/\eta_{i-1})m_0$ and the initial conditions will be the position and velocity vectors in Jacobi coordinates, $\tilde{\mathbf{r}}$ and $\tilde{\mathbf{v}}$. I implemented the drift phase in snippet 9.3 assuming that the input `x` is in Jacobi coordinates. How does it look?

Snippet 9.3
Function `wh_drift`: Drift the state of all bodies (Keplerian propagation).

```
def wh_drift( x, dt, masses, nbodies, G, eta ):

    # Drifted state:
    drift = np.zeros(nbodies * 6)

    # Propagate each body assuming Keplerian motion:
    for ibod in range(1, nbodies):

        # Compute equivalent GM:
        gm = masses[0] * eta[ibod] / eta[ibod - 1] * G

        # Initial conditions:
        pos0 = x[ibod * 3: (ibod + 1) * 3]
        vel0 = x[(nbodies + ibod) * 3: (nbodies + ibod + 1) * 3]
```

```
        # Propagate:
        pos, vel = propagate_kepler(0.0, dt, pos0, vel0, gm)

        # Store states:
        drift[ibod * 3: (ibod + 1) * 3] = pos
        drift[(nbodies + ibod) * 3: (nbodies + ibod + 1) * 3] = vel

    return drift
```

Professor: That looks great, Alice!

9.3.6 Building the Integrator

Alice: We now have all the pieces required by snippet 9.1! Pretty exciting.

Bob: Yeah! Putting together the integrator in snippet 9.7 is now pretty simple. First, we will allocate an auxiliary solution array. Since the integrator uses a fixed time step, we can know exactly how many points the solution will have. Then, we need to compute all the η_i terms, transform the initial state to Jacobi coordinates, and compute the initial acceleration. We will then call `wh_advance_step` (snippet 9.1) sequentially until we reach t_f. I introduced the variable `skip_step` that controls the resolution of the solution; we will only save the solution once every `skip_step` steps.

9.3.7 Testing the Code

Bob: Time to play with the integrator! I think we should start with a simple test to make sure everything works.

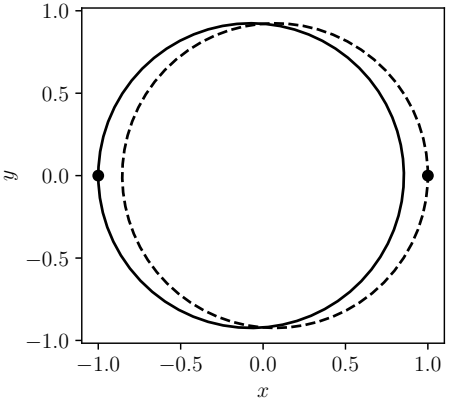
Alice: How about the two-body orbit we defined in snippet 8.6? We know how the solution should look. We might want to propagate the orbits for three or four periods instead of just one, to make sure that the energy error is not growing. This is a symplectic integrator after all!

Bob: I like it. I plotted the result of propagating three revolutions with $\Delta t = 0.01$ in figure 9.6.

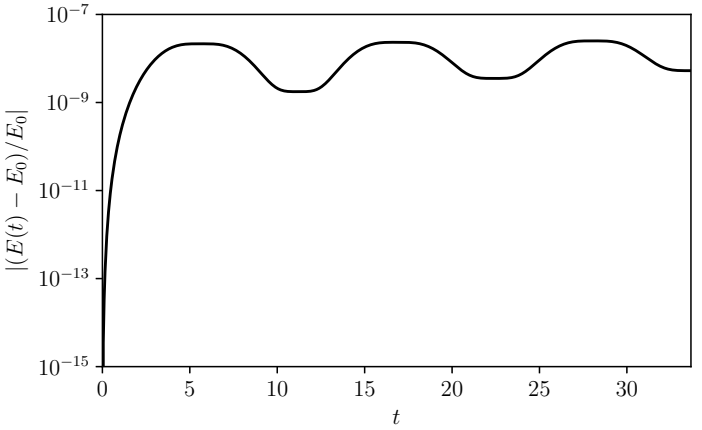
Alice: The orbits look perfectly fine; the bodies go back to where they started from, and the trajectories match those in figure 8.4a. Oh, and the energy error oscillates, but there is no secular growth. Looking good!

9.4 Propagating the Solar System for a Million Years

Bob: Let's now try to integrate the Solar System for a million years again. First we need the gravitational constant and the number of bodies:



(a) Orbit plot



(b) Relative energy error

Figure 9.6
Solution to the elliptic two-body problem using the WH integrator.

```
# Gravitational constant (AU^3/kg/d^2):
G = 1.48813611629e-34

# Number of bodies:
nbodies = 9
```

We can simply retrieve the inertial positions and velocities of the Sun and all the planets with respect to the Solar System Barycenter from HORIZONS,[3] with units of AU and AU/d. We obviated the motion of the center of mass, so we don't need to worry about it.

```
# Initial conditions:
pos0 = np.array([\
  -7.136455199990743e-03,-2.795724239370603e-03, 2.061367448662713e-04,\
  -1.372300608220831e-01,-4.500833421231408e-01,-2.439217068677536e-02,\
  -7.254387515234633e-01,-3.545003280048490e-02, 4.122031886718398e-02,\
  -1.842715550033945e-01, 9.644459624383047e-01, 2.020509819625348e-04,\
   1.383579466546939e+00,-1.621204162199320e-02,-3.426152623891661e-02,\
   3.994040946651115e+00, 2.935780101608396e+00,-1.015793842267007e-01,\
   6.399272401977961e+00, 6.567193895707603e+00,-3.688702929216678e-01,\
   1.442472018423088e+01,-1.373711773058018e+01,-2.379347191117984e-01,\
   1.680490957344363e+01,-2.499455859992035e+01, 1.274294623716956e-01])
vel0 = np.array([\
   5.378459295718575e-06,-7.406912670374715e-06,-9.433299568131420e-08,\
   2.137177414060989e-02,-6.455396674574269e-03,-2.487957667804378e-03,\
   8.034959858448681e-04,-2.030262561468278e-02,-3.235458387131530e-04,\
  -1.720224660756253e-02,-3.166189066385779e-03, 1.065953235594308e-08,\
   6.768779529388747e-04, 1.517984118242539e-02, 3.015574313127102e-04,\
  -4.562948182122576e-03, 6.435847813493556e-03, 7.548674581187792e-05,\
  -4.286973417088009e-03, 3.882908784079374e-03, 1.028535168332451e-04,\
   2.683483979188155e-03, 2.665288849251202e-03,-2.486624055378518e-05,\
   2.584652466233836e-03, 1.769493937752207e-03,-9.600383320403503e-05])

# Generate state vector:
x0 = np.concatenate((pos0, vel0))

# Masses:
masses = np.array([ 1.988475415966534e+30, 3.301096181046679e+23,
    4.867466257521635e+24, 5.972365261370794e+24, 6.417120205436417e+23,
    1.898187239616558e+27, 5.684766319852324e+26, 8.682168328818365e+25,
    1.024339999008106e+26,])
```

The bodies are, in order, Sun, Mercury, Venus, Earth, Mars, Jupiter, Saturn, Uranus, and Neptune. Time to define the times. Oh, wait … what time step should we use?

Alice: Well, the fastest orbit we are integrating is Mercury's orbit, which has a period of eighty-eight days. Following our previous discussion about the characteristic time scales of the problem, I would suggest choosing a time step that divides this orbit into approximately 100 segments. Say, $\Delta t = 1$ d?

[3] https://ssd.jpl.nasa.gov/?horizons

Bob: I like it. Also, let's store the solution every 1,000 steps.

```
# Define time interval and time step:
t0 = 0.0
tf = 365.25 * 1e6
dt = 1.0

# Store solution:
skip_step = 1000
```

We can finally call the integration function:

```
# Propagate using WH mapping:
time, state = propagate_wh(x0, t0, tf, dt, masses, nbodies, skip_step, G)
```

Do you want to do the honors, Alice?

Alice: Of course! I will launch the simulation.

Professor: I'm very happy to see all your progress. The simulation will take a while, so why don't we leave it here and reconvene next week?

Bob: Sure, see you next week then!

Alice: Bye, Prof. Starmover!

Alice and Bob leave Prof. Starmover's office and go home for the day.

One week later, the propagation of the Solar System finishes and Alice and Bob head back into Prof. Starmover's office to discuss the results.

Bob: Good morning, Prof. Starmover!

Alice: Hi, Prof. Starmover.

Professor: Hello to you both. So, did you collect the results from your simulation?

Alice: Yes, Bob was about to plot the energy error. We are quite excited to see how it went!

Bob: Okay, I'm plotting the energy fluctuation in figure 9.7. Look at that; it remained below 2×10^{-10} for the entire integration! That's great news.

Alice: Definitely. I'm actually very surprised to see that it worked so well. This means that we can actually trust our solution.

Professor: Congrats, you did a great job. You should keep in mind that a small energy error is a necessary but not sufficient condition for accuracy. This is especially true when you integrate bodies with small or negligible mass because their contribution to the total energy is so small that their orbits could be completely wrong and you wouldn't notice the impact on the total energy. You can find a dedicated discussion in the paper by Mikkola and Innanen (2002), in case you want more details. There's still one other check that you can do with the tools that you already have, which is comparing your integrated solution with the Laplace–Lagrange theory that we discussed in section 2.5.3. We derived an approximation

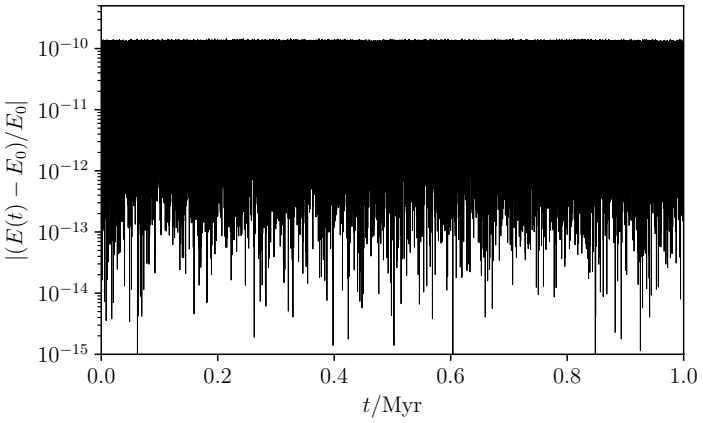


Figure 9.7
Energy error during the integration of the Solar System.

of how the planets in the Solar System evolve in time. This way, you will not only check that your solution is correct but also convince yourselves that analytic theories provide very useful approximations.

Bob: Let's see, we need to convert from Cartesian coordinates to orbital elements using the expressions in section 2.5.2. That's actually not too bad; I plotted the result in figure 9.8.

Alice: We have to compare this figure with figure 2.5. First, we can see that the eccentricity of Mercury completes a little more than one complete cycle in both figures, decreasing from about 0.2. Its inclination oscillates and reaches four clearly differentiated maximum values, the highest being 10 deg. The eccentricity of Uranus does not complete one full cycle and reaches a maximum value of about 0.075 after about 0.9 Myr. It is interesting to see that the integration now captures shorter-period variations. The inclination of Uranus, on the other hand, completes two full periods remaining below 5 deg. Mars also exhibits relatively large oscillations and a secular trend that decreases the eccentricity. The inclination reaches about 8 deg after about 0.6 Myr. Jupiter and Saturn are pretty stable, the latter showing slightly larger variations in eccentricity. Finally, Neptune's eccentricity remains small throughout the entire interval, with figure 9.8 capturing variations in shorter time scales. Everything seems to match! At least qualitatively.

Professor: I think you have pretty powerful tools now to tackle many research projects.

Bob: This is exciting! Thanks for all your help, Prof. Starmover!

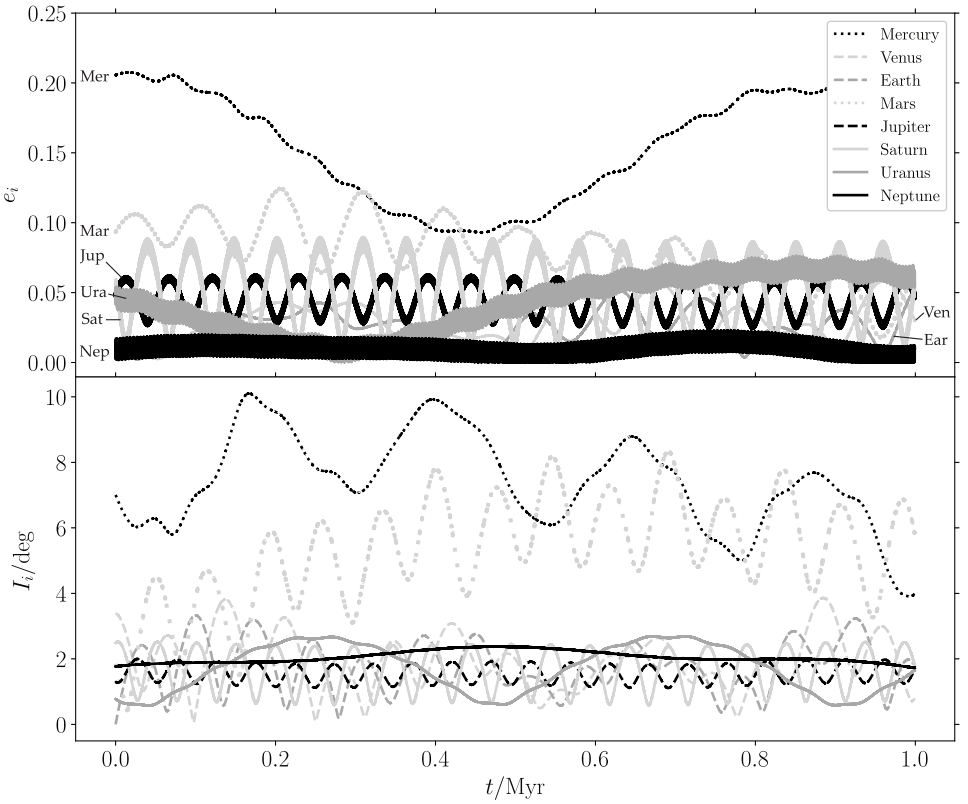


Figure 9.8
Evolution of the Solar System over a million years.

9.5 Code Review

Alice and Bob leave Prof. Starmover's office and find a place to sit near the library to review the code that they wrote.

Snippet 9.4
Function `jacobi2cart`: Convert from Jacobi to Cartesian coordinates following Rein and Tamayo (2015).

```
def jacobi2cart( x, masses, nbodies, eta ):

    # Allocate output:
    cart = np.zeros(nbodies * 6)

    R = x[0: 3] * eta[-1]
    V = x[nbodies * 3: (nbodies + 1) * 3] * eta[-1]
    for i in range(nbodies - 1, 0, -1):
```

```
        R = (R - masses[i] * x[i * 3: (i + 1) * 3]) / eta[i]
        V = (V - masses[i] * x[(nbodies + i) * 3: (nbodies + i + 1) * 3])
            / eta[i]
        cart[i * 3: (i + 1) * 3] = x[i * 3: (i + 1) * 3] + R
        cart[(nbodies + i) * 3: (nbodies + i + 1) * 3] = \
          x[(nbodies + i) * 3: (nbodies + i + 1) * 3] + V
        R = eta[i - 1] * R
        V = eta[i - 1] * V
    cart[0: 3] = R / masses[0]
    cart[nbodies * 3: (nbodies + 1) * 3] = V / masses[0]

    return cart
```

Snippet 9.5
Function `cart2jacobi`: Convert from Cartesian to Jacobi coordinates using the transformation from Rein and Tamayo (2015).

```
def cart2jacobi( x, masses, nbodies, eta ):

    # Allocate output:
    jacobi = np.zeros(nbodies * 6)

    R = masses[0] * x[0: 3]
    V = masses[0] * x[nbodies * 3: (nbodies + 1) * 3]
    for i in range(1, nbodies):
        jacobi[i * 3: (i + 1) * 3] = x[i * 3: (i + 1) * 3] - R / eta[i -
            1]
        jacobi[(nbodies + i) * 3: (nbodies + i + 1) * 3] = \
          x[(nbodies + i) * 3: (nbodies + i + 1) * 3] - V / eta[i - 1]
        R = R * (1 + masses[i] / eta[i - 1]) + masses[i] * jacobi[i * 3: (
            i + 1) * 3]
        V = V * (1 + masses[i] / eta[i - 1]) + masses[i] \
          * jacobi[(nbodies + i) * 3: (nbodies + i + 1) * 3]

    jacobi[0: 3] = R / eta[-1]
    jacobi[nbodies * 3: (nbodies + 1) * 3] = V / eta[-1]

    return jacobi
```

Snippet 9.6
Function `compute_accel`: Compute perturbing acceleration in Jacobi coordinates.

```
def compute_accel( cart, jac, masses, nbodies, G, eta ):

    # Allocate output array:
    accel = np.zeros(nbodies * 3)

    aux = np.zeros(3)
    for i in range(1, nbodies):

        r0i = cart[i * 3: (i + 1) * 3] - cart[0: 3]
        accel[i * 3: (i + 1) * 3] = masses[0] * eta[i] / eta[i - 1] \
```

```
          * (jac[i * 3: (i + 1) * 3] / np.linalg.norm(jac[i * 3: (i + 1) *
              3])**3 \
             - r0i / np.linalg.norm(r0i)**3)

        aux *= 0.0
        for j in range(1, i):
            rji = cart[i * 3: (i + 1) * 3] - cart[j * 3: (j + 1) * 3]
            aux += masses[j] * rji / np.linalg.norm(rji)**3
        accel[i * 3: (i + 1) * 3] += - eta[i] / eta[i - 1] * aux

        aux *= 0.0
        for j in range(i + 1, nbodies):
            rij = cart[j * 3: (j + 1) * 3] - cart[i * 3: (i + 1) * 3]
            aux += masses[j] * rij / np.linalg.norm(rij)**3
        accel[i * 3: (i + 1) * 3] += aux

        aux *= 0.0
        for j in range(0, i):
            for k in range(i + 1, nbodies):
                rjk = cart[k * 3: (k + 1) * 3] - cart[j * 3: (j + 1) * 3]
                aux += masses[j] * masses[k] * rjk / np.linalg.norm(rjk)
                    **3
        accel[i * 3: (i + 1) * 3] += - aux / eta[i - 1]

    accel *= G

    return accel
```

Snippet 9.7

Function `propagate_wh`: Propagate N-body system using the Wisdom–Holman integrator.

```
def propagate_wh( x0, t0, tf, dt, masses, nbodies, skip_step, G ):

    # Initialize array buffer to store solution:
    nbuffer = int(np.floor((tf - t0) / (dt * skip_step))) + 1
    sol_state = np.zeros((nbuffer, nbodies * 6))
    sol_time = np.zeros(nbuffer)

    # Allocate:
    cart = np.zeros(nbodies * 6)
    cart[0:] = x0[0:]

    # Store initial solution:
    sol_state[0, :] = cart
    sol_time[0] = t0

    # Compute etas (interior masses):
    eta = np.zeros(nbodies)
    eta[0] = masses[0]
    for ibod in range(1, nbodies):
        eta[ibod] = masses[ibod] + eta[ibod - 1]

    # Compute Jacobi coordinates and initial acceleration:
    jacobi = cart2jacobi(cart, masses, nbodies, eta)
```

```
    accel = compute_accel(cart, jacobi, masses, nbodies, G, eta)

    # Main loop:
    istep = 0
    isol = 1
    t = t0
    drift1 = True
    extend2 = True
    while t < tf:

        # Advance one step:
        jacobi, accel = wh_advance_step(jacobi, t, dt, masses, nbodies,
            accel, G, eta)

        # Advance time:
        t += dt

        # Advance step counter:
        istep += 1

        # Store the solution:
        if ( istep % skip_step == 0):

            # Convert to Cartesian:
            cart = jacobi2cart(jacobi, masses, nbodies, eta)

            # Store the solution:
            sol_state[isol, :] = cart
            sol_time[isol] = t
            isol += 1

    return sol_time[0: isol], sol_state[0: isol, :]
```

10 Building a Production Code

Overview. After having spent quite some hours implementing various integrators, Alice and Bob are happy that they now have these integrators at their disposal. They have been thinking about how to apply these integrators to production runs. However, the code is getting longer and longer after long-term accumulation of all kinds of routines. They decide to perform a clean-up before pushing the code to its limits. These efforts subsequently become a complete overhaul of the software architecture, in which they decide to adopt the object-oriented programming paradigm for their code. In addition, they use native C and even GPU programming to significantly speed up their code.

10.1 An Object-Oriented Discussion

Alice: Now we have a handful of different integrators, and we have arranged them in a relatively modular way, such that the input/output (I/O) and plotting routines are shared among them. But, the source code file is getting longer and longer, which makes it increasingly difficult to read and edit.

Bob: We are basically running into a typical software engineering issue. Many sophisticated software packages, such as the operating systems that we are using every day, have far longer source codes than our integrators. Lengthy source codes are not only difficult to debug but also challenging for teamwork. Imagine that someone else wants to extend our existing code by adding one more integrator; they have to read through our code to learn how to get the input from our routines, how to return the integration data with proper data structure, how to call our plotting methods, and so on, which requires significant knowledge of the code. If our codes were tens of thousands of lines long, it would be nearly impossible for any new friend who wants to join our team to know what's going on. We should make our code more modular.

Alice: Agreed, I think…. But so far we have already put different routines into different methods. What more can we do? Is it possible to separate different routines into different source files?

Bob: I kind of like this idea, but I think we can do more than that. How about reconstructing our source code into the so-called *object-oriented programming (OOP)* paradigm? The

fundamental idea of OOP is that a set of attributes and routines can be encapsulated into an *object*, which is an independent unit capable of bookkeeping itself. It interacts with other objects through its interface.

Alice: Hmmm, interesting…. I am not sure how this would help.

Bob: Before directly addressing your question, let me offer a simple example: when you drive, do you need to know how the engine works?

Alice: Well, I know how the engine of my car works in general. But this is not necessary to drive a car. To drive a car, you just need to know how to operate the clutch, brake, throttle, and the steering wheel…. Well, maybe you also need to know how to read the speedometer, or else you may get a fine!

Bob: Exactly! So in this example, the car is an object, which can be controlled/operated via the interface, namely the clutch, brake, throttle, and the steering wheel. It hides its own complexity, because we do not need to understand how the complicated engine works. It maintains its own stateful data-like speed and gas level, which are read-only attributes for the drivers. Also, all cars use nearly the same interface, regardless of whether they are powered by gasoline, diesel, or electricity. In other words, the internal implementations are independent from the interface.

Alice: Ah, I see. So in programming we can do similar things to encapsulate our routines into objects?

Bob: Absolutely. OOP is a feature supported natively in many modern programming languages, such as PYTHON and C++. In fact, the geeky name C++ derives from the C programming language with additional features such as OOP and templates. But languages like C and FORTRAN do not offer native support for OOP.

Alice: It was a good decision, then, to use the PYTHON programming language to code up our integrators. So according to your car analogy, we should encapsulate our integrators into objects and only allow them to be controlled by a unified interface?

Bob: Precisely. We will need to define a common interface for different integrators. In the terminology of OOP, this process is called *abstraction*. By the way, we should also encapsulate other things, such as the data input/output routines, so that the integrator simply has to pass the generated data to the I/O module, and the I/O module should take care of the low-level algorithms by itself.

Alice: But different integrators can be fundamentally *different*. If we abstract them into a unified interface, they will bear the same function names. Would that be a problem? How do we know which function to invoke?

Bob: That's a great question. This problem is elegantly solved in the OOP by the `inheritance` and `override` mechanisms. In a nutshell, the `inheritance` mechanism allows a *subclass* to inherit the properties/attributes and methods/functions of a *superclass*.

As such, we could define a superclass with the common properties and behaviors that an integrator may have and create a bunch of subclass integrators that contain the peculiarities of individual integrator algorithms. The peculiarities of individual integrators bear the same method names as the superclass, but the `override` mechanism allows them to override the behaviors of the superclass.

For instance, every integrator should have a method[1] `integrate()`. Of course, different algorithms have different ways of integrating an N-body system. So we could define a superclass `Integrator` without having to implement the `integrate()` method. All subclasses derived from `Integrator` should then override the `integrate()` method with their own algorithms.

Alice: Sounds like a great programming philosophy. But will it make the code more difficult to read?

Bob: Hmmm, I think it is the other way around. OOP will, in general, improve the readability of the code, because this paradigm makes the code more modular. Having a modular code also greatly benefits the subsequent debugging/maintenance.

Alice: I am convinced that the OOP paradigm is optimal for us. Let's do it!

10.2 Requirement Analysis: What Do We Want Our Code to Be?

Bob: Alice, I think we should get a piece of paper and sketch out the structure of our code.

Alice: Sounds like we are doing some serious software engineering. Sure thing! Pen and paper. Here we go.

Bob: Okay, so we are ready to convert our code into a professional software distribution. Before starting, what do we want our code to do? What do you have in mind regarding the selling points of our code?

Alice: Hmmm…. I feel like I am working in an IT company now. Fair enough. Let's say, we want our code to be a general-purpose numerical package for N-body dynamics. I think that one possible selling point is that we offer several different integrator algorithms and allow the users to make their own choice based on the actual problems that they want to tackle.

Bob: I agree. And I am proud of our state-of-the-art integrators, such as the fifteenth-order Gauss–Radau integrator (Everhart, 1985) and the Wisdom–Holman integrator (Wisdom and Holman, 1991). I believe that we can optimize this code to the domain of planetary system dynamics. Since you just mentioned the term "general-purpose," I guess we should

[1] In the terminology of object-oriented programming, a function/routine is usually called a *method*.

make sure that our code is extensible; that is, we want to be able to add other integrators conveniently later on. The code is currently specialized to planetary system dynamics, but with more integrators being incorporated later on, it will be useful for a much wider range of applications.

Alice: Good point. So extensibility is also an objective of our work. Who would be our potential users?

Bob: Ideally, we would like to have as many users as possible, of course. In practice, our code would be interesting to people doing planetary system dynamics, I guess. Yesterday, Prof. Starmover told me that there are several existing codes for this domain already, such as MERCURY6 (Chambers, 1999), SWIFT (Levison and Duncan, 1994), REBOUND (Rein and Liu, 2012), and so on. However, my impression is that most of these codes are single-threaded, running on single CPU cores. Our code would be a valuable contribution to the field, if we can manage to parallelize it to handle relatively large N systems (such as planetary systems with a lot of asteroids/comets and star clusters). On the other hand, if we can incorporate some nongravitational forces into the code, such as post-Newtonian (PN) corrections, it may be useful for the gravitational waves community!

Alice: It is going to be a wonderful feeling to see that our code is useful to other researchers. You just mentioned the term "parallelize," and it seems to be an important one. What exactly is that?

Bob: Good catch! By saying "parallelize the code," I meant that we could make use of multiple CPU or GPU cores to simultaneously perform the numerical integration and therefore achieve faster speeds. In general, parallelization can be achieved if a task can be split into several independent subtasks (preferably with similar workloads) and have multiple processors to work on these subtasks. For example, in a system of N particles, the pairwise accelerations exerted on one particle due to the other $N - 1$ particles can be calculated in parallel. Eventually, the result of the original task can be obtained by gathering and reducing the results of these subtasks. When we finish the implementation of the integrator on a single CPU, we could measure its performance and locate the bottlenecks and probably improve the performance of the code by parallelizing the bottleneck.

Alice: Wow, making use of multiple processors sounds cool! Let's definitely discuss it later. For now, instead of saying "our code," I think we'd better give it a name. Let's say, the "Alice–Bob Integrator"?

Bob: Absolutely. But we are going to build a framework for a number of integrators. How about "Alice–Bob Integrator Environment," or ABIE?

Alice: ABIE. I love it!

10.3 Constructing the Software Framework

10.3.1 Defining the Software Architecture

Bob: I learned from my software engineering class that a software package can be divided into three layers: the presentation layer is on the top, providing the user interface and translating the data between the user and the program; the logic layer is in the middle, which performs calculations and manages decision making about the data; the bottom layer is the data layer for data storage and retrieval.

Alice: I also took that course, but I honestly never thought I'd have the chance to actually put it into practice so soon … if ever. Okay, right … three layers. So, if we apply this architecture to ABIE, then the logic layer would probably be the integrators, and the data layer would be the particle data structure. What about the presentation layer? It seems that now we only interact with ABIE through the UNIX command lines. Oh, we do make plots using `matplotlib`[2]—perhaps this is considered part of the presentation layer as well?

Bob: I agree. The data layer should also include the facility for saving simulation data to files and for loading initial conditions. This is the so-called *data persistence layer*.

Alice: Okay….

Bob: The logic layer consists of our integrators. The integrators will be incorporated into the code via a unified API (application programming interface), meaning that all integrators can be invoked in exactly the same way regardless of the actual algorithm.

Alice: Got it. I'm with you so far.

Bob: As for the presentation layer, it is rather slim in our current implementation, but maybe we could implement some visualization facilities later on. We could even provide a PYTHON programming interface, so that ABIE can be run programmatically.

Alice and Bob sketch out the previously discussed software architecture for ABIE on a piece of paper….

Bob: Here it is, in all its considerable glory! The pyramid of ABIE (see figure 10.1)! The upper layer depends on the lower layer, and the lower layer is independent of the upper layer. So, the higher the layer, the fewer components in that layer. It follows a hierarchical structure. You see, the user runs ABIE through the command-line interface (CLI) or the PYTHON API, which in turn causes the integrator API to be invoked. The integrator API delegates the tasks to the actual integrator chosen by the user. Of course, the integrator should be supplied with initial conditions, and the simulation data it generates should be stored properly. This relies on the data layer at the bottom. This layer contains logic that optimizes access of the particle data from the system memory and from the hard drive. For

[2] https://matplotlib.org

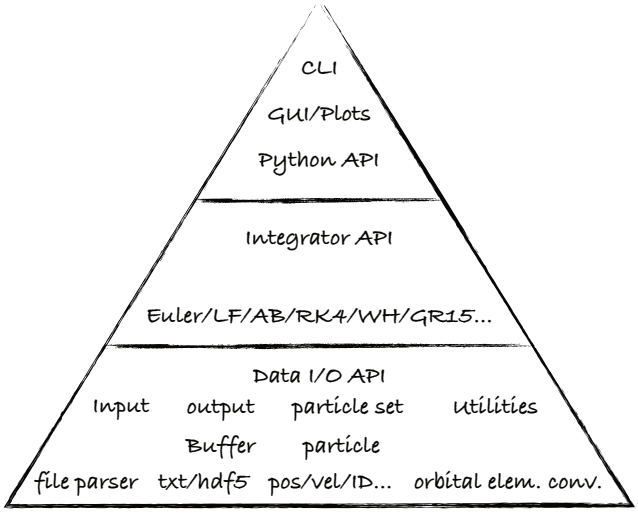


Figure 10.1
The software architecture of ABIE. The top layer is the presentation layer, consisting of a command-line interface (CLI), a simple graphical user interface (GUI) for plotting, and a PYTHON application programming interface (API); the middle layer is the logic layer, consisting of an abstract integrator API and the concrete implementation of various integrator algorithms; the bottom layer is the data layer, consisting of the input/output facility, the in-memory data structure for particles and particle sets, the utilities of orbital element conversions, and so on.

example, the layer contains a buffer for caching simulation data, and so the file writing action is only triggered if the buffer is full or the simulation is finished.

Alice: Neat! Wow, the structure of ABIE is now much more clear. So which layer should we start working on first?

Bob: We should start working on the data layer. The two upper layers are both based on it.

Alice: Great, let's get down to it!

10.3.2 Implementing the Data Layer

10.3.2.1 Define the data structure for a single particle

Alice: So, by "data," do you mean particle data? Then I guess we need to define our particles.

Bob: Right. Let's start with the definition of a particle in our N-body code.

Alice: A particle contains attributes such as position, velocity, mass, and radius. Anything else?

Bob: I agree that these are the "essential" physical quantities for describing a particle. I would suggest that we add a few more: the `name` of the particle is an identification tag that the user can assign. For example, a particle can be called "Sun." The `hash` is a long integer unique to each particle and will not change during the simulation, which allows the user to uniquely identify a particle without ambiguity.

Alice: So the `name` of a particle doesn't have to be unique?

Bob: That's right. I believe that the user has the liberty to call a particle with whatever name she or he likes, even if there's more than one particle sharing the same name. Also, a particle may not have a name if the user doesn't make such an assignment. Therefore, we need to generate a machine-friendly "name" for every particle when they are initialized, which is the `hash` field.

Alice: What about the `ID` of a particle? From what I can remember from class, the `ID` of a particle is basically its index in the global array.

Bob: That's a good catch. As you just noted, the `ID` is the index in the *global* array. I don't think that it is an "intrinsic" property of a particle. For example, a particle with `ID=3` happens to be placed in the fourth (if the array starts from 0) or third (if the array starts from 1) position in the global particle array. But suppose that another particle with `ID=2` is deleted because it is ejected from the system; then, the `ID=3` particle could get shifted leftward in the global array and become `ID=2`. Therefore, the `ID` index is insufficient to uniquely identify a particle.

Alice: Brilliant. So we could define a particle like this:

Snippet 10.1
First version of the `Particle` class.

```
class Particle(object):
    def __init__(self, ptype=0, mass=0.0, pos=np.zeros(3), vel=np.zeros(3)
        , radius=0.0, name=None, primary=None):
        self.ptype = ptype  # particle type. 0: regular particle; 1:
            massless; 2: low-mass
        self.mass = mass  # mass
        self.radius = radius  # radius
        self.name = name  # user-assigned name, optional
        self.hash = hash(self)
        self.pos = pos
        self.vel = vel
```

Here, `self` is a keyword in PYTHON OOP that represents an instance of a class. Therefore, in our case, the attributes starting with `self.` belong to an instance of the `Particle` class. Setting the attributes of one `Particle` instance does not affect the attributes of another `Particle` instance.

Bob: Exactly! And the class definition looks great. I like the way that you group (x, y, z) as `pos` and v_x, v_y, v_z as `vel`. This looks very concise. I do see that there is a danger here: the attributes `self.pos` and `self.vel` expect 1D vectors with length=3. These attributes are public members, meaning that everyone can modify them. But here there is no mechanism to guarantee that the data type passed to these attributes is correct. For instance, if a user mistakenly passes a scalar property to `self.pos`, the program will crash.

To mitigate this danger, I would suggest that we write `self.pos` as `self.__pos`, and likewise `self.vel` as `self.__vel`, and then create a getter/setter for each of them:

Snippet 10.2
Second version of the `Particle` class.

```
class Particle(object):
    def __init__(self, ptype=0, mass=0.0, pos=np.zeros(3), vel=np.zeros(3)
        , radius=0.0, name=None, primary=None):
        self.ptype = ptype  # particle type. 0: regular particle; 1:
            massless; 2: low-mass
        self.mass = mass  # mass
        self.radius = radius  # radius
        self.name = name  # user-assigned name, optional
        self.hash = hash(self)
        self.__pos = pos
        self.__vel = vel

    @property
    def pos(self):
        return self.__pos

    @property
    def vel(self):
        return self.__vel

    @pos.setter
    def pos(self, pos_vec):
        if type(pos_vec).__module__ == np.__name__:
            if pos_vec.size == 3:
                self.__pos = pos_vec
            else:
                raise ValueError('Position␣vector␣must␣be␣a␣len=3␣vector.'
                    )
        else:
            raise TypeError('Position␣vector␣must␣be␣a␣numpy␣vector␣with␣
                len=3.')

    @vel.setter
    def vel(self, vel_vec):
        if type(vel_vec).__module__ == np.__name__:
            if vel_vec.size == 3:
                self.__vel = vel_vec
            else:
                raise ValueError('Velocity␣vector␣must␣be␣len=3␣vector.')
        else:
```

```
            raise TypeError('Velocity␣vector␣must␣be␣a␣numpy␣vector␣with␣
                len=3.')
```

In the above code, the setters allow `self.__pos` and `self.__vel` to be accessed as `self.pos` and `self.vel`, respectively. But when the user attempts to set their values, the setters will first perform a data type check to make sure that they are `numpy` arrays and then make sure that they have the desired array dimension.

Alice: Fantastic. I didn't know that PYTHON can do this!

Bob: Elegant, isn't it?

Alice: Yep. Sure is.

10.3.2.2 Define a particle set

Bob: Now that we have defined the `Particle` class, the next thing is to define a container and manager for a collection of `Particle` objects, that is, the `Particles` class.

Alice: What would be the expected functionality of the container? Does it provide facilities for adding, removing, and accessing particles?

Bob: Yes. These are pretty much all we need, I think. Now that we have added the `name` and `hash` attributes to the `Particle` class, we also want our container to support accessing particles using their names and hash values. Let's code it up:

```
class Particles(object):
    """
    A particle container.
    """

    def __init__(self, const_g):
        self.__particles = []
        self.__N = 0
        self.CONST_G = const_g

    @property
    def N(self):
        return self.__N

    @property
    def particles(self):
        return self

    def __getitem__(self, item):
        if isinstance(item, (int, long)):
            if item < len(self.__particles):
                return self.__particles[item]
            else:
                raise ValueError('Particle␣#%d␣does␣not␣exist!' % item)
        elif isinstance(item, basestring):
            if item in self.__names:
                return self.__particles[self.__names[item]]
```

```
            else:
                raise ValueError('Particle␣%s␣does␣not␣exist!' % item)
        return None
```

So here I am using PYTHON's built-in data structure `list` as the underlying data structure of our particle set, which automatically allows us to access an element (in our case, a `Particle` object) by its index. However, we can't really afford to take the risk that this list will be modified mistakenly. So I again write this attribute in a "protected" way (`self.__particles`) and provide a getter and a setter to allow the user to access it. The setter will modify the protected list only if the data type and data shape (i.e., array size) are correct.

The method `__getitem__()` does something magical. This is a standard built-in PYTHON function, which provides access to an element in a collection by a given key. For example, if `particles` is a collection of particles, and `i` is the index of a certain particle, then by using `particles[i]`, we are actually calling `particles.__getitem__(i)`. Now, the magical thing is that since we are using OOP, we could actually customize its behavior by overriding this particular method. In the code snippet above, we ask the method to do a few extra things compared to the standard implementation: we will first check whether the key is an integer or a long integer, in which case we believe that the key is the index of the particle. If the key is actually a string, then we could make a reasonable guess that it is the name of the particle, and therefore we convert the given name into the index and return the relevant particle.

Alice: Wow, that's quite elegant! I didn't know that the behavior of accessing this syntax `particles[key]` could be customized. Clever. I remember in my `C` programming class that if I give a noninteger to an array, the code will probably go wild….

Bob: I experienced something similar when it was first presented to me. Bear in mind, though, that PYTHON was invented in 1990, whereas C was invented back in 1972. Anyways, now that we have basically implemented the underlying data structure of a particle set and the method for accessing it, we can now work on the facilities for adding/removing particles.

Alice: Agreed. I guess that it is straightforward to implement. Since the underlying data structure of the particle set is based on the `list` in PYTHON, we just need to call the add/remove method of `list` accordingly. Based on what we have done so far, I would suggest that we add two more methods like this:

Snippet 10.3
First version of the adding/removing particle methods.

```
class Particles(object):
    ...
    def add_particle(self, particle):
        if isinstance(particle, Particle) and (particle not in self.
            particles):
```

```
            # make sure that the data type is correct and the particle is
                not yet in the collection
            self.__particles.append(particle)

    def remove_particle(self, particle):
        if isinstance(particle, Particle) and (particle in self.particles)
            :
            # make sure that the data type is correct and the particle
                already exists
            self.__particles.remove(particle)
    ...
```

Inspired by what you just said, I added some "protection" to the code: a particle can only be added to the set if it's indeed a `Particle` and is not yet in the set; it can only be deleted if it's of the correct data type and is already in the set.

Bob: Looks great! You are fast becoming a real programmer, Alice! I think that should work. For adding particles, in our current implementation, we need to create a `Particle` object first before we can use the `add_particle()` method. Why don't we create a shortcut method here, so that a particle can be created automatically with some given set of properties (e.g., position, velocity, mass, name)?

Alice: That's a great idea. As it should, the data layer is handling everything related to the data!

Bob: Yes, that's our goal. So I would call this method `add()`:

Snippet 10.4
Second version of the adding/removing particle methods.

```
class Particles(object):
    ...
    def add_particle(self, particle):
        ...

    def remove_particle(self, particle):
        ...

    def add(self, pos, vel, mass, radius=0, name=None):
        particle = Particle(mass=mass, pos=pos, vel=vel, name=name, radius
            =radius)
        self.add_particle(particle)
    ...
```

I guess we want to treat particles as point masses by default, so I use PYTHON's default argument facility to set the `radius` attribute to zero by default. This means that if the user doesn't provide a value for the `radius` attribute, it will be set to zero automatically.

Alice: Oh really? We can even have a default argument? That's really nice! In this case, how about adding the six orbital elements as default parameters? Sometimes it is easier to add particles using orbital elements rather than using Cartesian coordinates. For example,

we know that the semimajor axis of the Earth is 1 AU. How nice it would be if we could let the user label the particle like `add(mass=3.e-6, a=1.0, name='Earth')`. And, yes, it makes sense that we treat particles as point masses for simplicity, at least for now.

Bob: I like the idea of adding orbital elements support to the `add()` method. In doing so, though, we have to implement the routine for the conversion between Cartesian coordinates and orbital elements. We will do it in a minute, but now let me extend our `add()` method:

Snippet 10.5
Third version of the adding/removing particles method.

```
class Particles(object):
    ...
    def add_particle(self, particle):
        ...

    def remove_particle(self, particle):
        ...

    def add(self, mass, pos=None, vel=None, radius=0, a=None, e=0.0, i
        =0.0, Omega=None, omega=None, f=None, primary=None, name=None):
        if a is not None:
            # orbital elements given. Calculate the Cartesian coordinates
                accordingly
            pos, vel = Tools.from_orbital_elements_to_cartesian(mp=mass,
                ms=primary.mass, semimajor_axis=a, eccentricity=e,
                inclination=i, longitude_of_ascending_node=Omega,
                argument_of_periapsis=omega, true_anomaly=f, G=self.
                CONST_G)
            particle = Particle(mass=mass, pos=pos, vel=vel, name=name,
                radius=radius)
            self.add_particle(particle)
        else if pos is not None and vel is not None:
            particle = Particle(mass=mass, pos=pos, vel=vel, name=name,
                radius=radius)
            self.add_particle(particle)
        else:
            raise ValueError('Either␣the␣pos/vel␣or␣the␣orbital␣elements␣
                should␣be␣specified!')
    ...
```

It looks pretty straightforward, don't you think? Just a few more lines to detect whether or not a user has provided the orbital elements. If so, then the position and velocity of the particle will be calculated from the elements; otherwise, the position/velocity data will be used directly. If neither coordinates nor elements are provided, the code will raise an exception.

Alice: Good. So we will need to create a `Tools` class and put the orbital elements conversion routines over there. I think that the routine should be smart enough to generate random values for the longitude of the ascending node (`Omega`), argument of periapsis (`omega`), and the true anomaly (`f`) if the values are not given by the user (in fact, the mean anomaly is

a more convenient quantity, since it is a linear function of time). It is more reasonable to assign random values to them instead of assuming that they are zeros, because otherwise newly created particles will always be at the same orbital phase, which is not physical. I could work on this part.

Bob: Great.

Alice implements the conversion routine between orbital elements and Cartesian coordinates (for brevity, the code snippet is only available on GitHub).

10.3.2.3 Output buffer

Alice: For a production code, we need to save the simulation data to the files on the hard drives. What kind of format should we choose? Text files? Binary files?

Bob: This is the essential question for building the data persistence layer! Well, text files and binary files have their own pros and cons. Text files are readable by humans, but the numerical data in text format are not really the internal representation of the machine, so they could pose a bottleneck when we have a lot of data to read/write. Moreover, text files are usually less compact and therefore require more storage space. For example, consider the value of π; we could use a 64-bit (8 bytes) `double` to preserve up to sixteen decimals, but we need 18 bytes if we are to store it as text ("3." costs 2 bytes, and the sixteen decimals costs 16 bytes). Binary files, on the other hand, have relatively opaque internal structures and can be machine dependent. A binary file generated by machine A may not be correctly loaded by machine B if they have different endianness and/or operating systems. These problems can be circumvented by using some machine-independent I/O libraries, though. So I guess I am voting for binary data format in combination with a certain I/O library, such as `HDF5`.

Alice: Looks like you've got a systematic theory for output data formats up your sleeve. I guess a computer science student *is* a computer science student…. What is *endianness* and what is `HDF5`?

Bob: Ha, don't I also sound like an astronomer sometimes?

Alice: No, not really.

Bob: Ouch. Well, endianness is basically the sequential order in which bytes are arranged in the computer memory. Computers only know 0 and 1, so for a value like 100, the corresponding binary would be `01100100`. Some computers "write" from left to right, so the binary value is indeed `01100100` (which are called big-endian machines, as they store the most significant byte first). But some other computers have a right-to-left "writing habit" (small-endian machines), so they write the binary value as `00100110`. So, for a given binary sequence, the values may be misinterpreted unless the endianness is specified.

HDF5 stands for "Hierarchical Data Format, version 5." It is a data format optimized for storing large amounts of numerical data. It allows data sets to be arranged in hierarchical ways, hence the name "hierarchical." This is very similar to how we store our files on the hard drive—files are stored in different directories, and a directory can have multiple subdirectories. In HDF5, the directories are called "groups," and the data sets are analogous to files. I really like this feature. As we catalog data sets into different groups, the internal file structure becomes much more clear.

Alice: Sounds really neat. Does it work with PYTHON?

Bob: Definitely. The format is supported by a wide range of programming languages, including PYTHON. In fact, the PYTHON interface of HDF5 is really nice. For example, if we have a numpy array called my_array, it can be written to a HDF5 file really easily:

Snippet 10.6
Minimum code for writing an array into a HDF5 file.

```
import h5py
with h5py.File('data.hdf5', 'w') as h5f:
    h5f.create_dataset('my_array', data=my_array)
```

And for reading the file, also super easy:

Snippet 10.7
Minimum code for reading an array into a HDF5 file.

```
import h5py
with h5py.File('data.hdf5', 'r') as h5f:
    my_array = h5f['/my_array'].value
```

That's it! We don't need to care about how the data sets are actually stored and loaded. We don't even need to care about the shape of the data set—storing and loading a 1D, 2D, 3D,…, *n*D data set has exactly the same syntax. The data set is platform independent, so we could send our simulation data files to any collaborators without caring about their hardware or operating systems. The HDF5 library guarantees that the data will be consistent. It works really well, actually.

Alice: Impressive! So I think we should build the HDF5 input/output facility into ABIE's data persistence layer. But, why the data persistence layer? I thought that we just need to insert the file writing lines into our integrator whenever we need to output the data. This is basically just calling the HDF5 library right then and there….

Bob: Yes, you are right. In principle, we could do that. But in practice, this will lead to a necessary coupling between our integrators and the HDF5 library. I guess we would instead build a thin layer of data I/O API. This layer decides the I/O logic, for example, which I/O library to use (we may want ABIE to support more formats later), when should an input/output action be carried out, and so on.

In fact, I think we want to implement a buffer in the output. Our integrators continue to generate data, but as the integrator advances for one step, only a small amount of new data will be generated. If we are to write such a small amount of data to the disk every time, it is likely to cause performance issues. This is due to the fact that hard drives (including solid-state drive, SSD) are much slower than the system memory, so if we ask the hard drive to write a small volume of data whenever the integrator produces, it is going to be a serious bottleneck. Rather, we could cache these data in the system RAM until they are piled up to a certain amount and then dump them to the file all at once.

Alice: That's really smart! We cache the data to preallocated `numpy` arrays?

Bob: Exactly. In doing so, we need to define a parameter `buf_len`, which is the length of the buffer. The `numpy` arrays are preallocated in such a way that they have sufficient space to store the integrator output for `buf_len` times. When the buffer is full, it will be flushed to the file. So, let's start coding the output buffer!

Snippet 10.8
Output buffer.

```
import h5py
import numpy as np


class DataIO(object):
    def __init__(self, buf_len=1024, output_file_name='data.hdf5'):
        self.buf_len = buf_len
        self.buf_cursor = 0
        self.output_file_name = output_file_name
        self.buf_initialized = False
        self.buf_t = None  # for the time vector
        self.buf_pos = None
        self.buf_vel = None
        self.buf_mass = None
        self.buf_radius = None
        self.buf_hashes = None
        self.h5_step_id = 0

    def initialize_buffer(self, n_particles):
        if self.buf_initialized is False:
            buf_len = self.buf_len
            self.buf_t = np.zeros(buf_len) * np.nan
            self.buf_mass = np.zeros((buf_len, n_particles)) * np.nan
            self.buf_hashes = np.zeros((buf_len, n_particles), dtype=np.
                int) * np.nan
            self.buf_radius = np.zeros((buf_len, n_particles)) * np.nan
            self.buf_pos = np.zeros((buf_len, 3 * n_particles)) * np.nan
            self.buf_vel = np.zeros((buf_len, 3 * n_particles)) * np.nan
            self.buf_cursor = 0
            self.buf_initialized = True

    def store_state(self, t, pos, vel, masses, radii=None):
```

```
        # return if the snapshot time is the same as store_t, which means
            that this time is already stored
        if self.store_t == t:
            return

        if self.buf_cursor == self.buf_len:
            # the buffer is full, trigger store
            self.flush()
            self.buf_cursor = 0

        self.buf_t[self.buf_cursor] = t
        self.buf_pos[self.buf_cursor] = pos
        self.buf_vel[self.buf_cursor] = vel
        self.buf_mass[self.buf_cursor] = mass

    def close(self):
        if self.buf_cursor > 0:
            # there are still some buffer data to be written to the HDF5
                file
            self.flush()
        if self.h5_file is not None:
            self.h5_file.close()
            self.h5_file = None
```

The `self.buf_cursor` keeps track of the next available buffer slot for the next batch of integrator data. With this API, the integrator just needs to call the `store_stage()` method, and the `Data_IO` layer will take care of the rest and achieve an optimal performance.

10.3.3 Implementing the Logic Layer

Bob: Finally! We finished coding up the infrastructure. Now it's time to implement the logic layer.

Alice: That is, creating an API for all our integrators.

Bob: Exactly. So we need to abstract the common attributes and methods shared by different integrators. In my opinion, all integrators generally have attributes such as starting time, current time, ending time, universal gravitational constant G, a particle set, and, of course, the `integrate()` method to launch the integration. Maybe it looks like this:

Snippet 10.9
First version of the `Integrator` API.

```
class Integrator(object):

    def __init__(self):
        self.t = 0.0  # current model time
        self.h = 0.01  # integration time step
        self.t_end = 1.0  # the ending time
        self.particles = None  # particle set
        self.CONST_G = 1.0  # the gravitational constant
```

```
    def integrate(self):
        pass  # the logic of the integration to be filled later
```

So the method `self.integrate()` evolves `self.particles` from the current time `self.t` to the termination time `self.t_end` with a fixed integration time step `self.h`. We will leave out the implementation of `integrate()` to the subclasses.

Alice: It looks like a nice idea, but I am still not sure how it will work. If we simply call the `integrate()` method of the `Integrator` class, it will do nothing, because we haven't implemented any logic there. Say, if we implement the LeapFrog integrator as a subclass, should we call the `integrate()` method of a subclass instead?

Bob: Yes and no.

Alice: What?

Bob: What you said is correct. If we just directly invoke the `integrate()` method in the class `Integrator`, it won't do anything. However, consider that when we are running a simulation, we only use one particular integrator (for example, Gauss–Radau15) at a time, so we could define one more property, say, `active_integrator`, to indicate which integrator is being used for the simulation. And then when invoking the `integrate()` method of the superclass `Integrator`, the actual integration will be delegated to the active integrator.

Alice: Wow, delegating the computing tasks! I love this idea. It looks like we are building into our code a framework for calling different integrators, where each integrator can be treated as a module in our code.

Bob: Yep! We wanted to make our integrators as modules, or plug-ins, to the framework of ABIE. In the future, incorporating more integrators into ABIE is like developing plug-ins. Cool?

Alice: Cool! It is interesting that doing astrophysics can sometimes mean doing plug-in development. This time, it is my turn to take the initiative to start coding!

Alice takes the reins of the keyboard from Bob.

Alice: I would first convert our existing integrators into modules. Let me start from the two simplest integrators, `Euler` and `LeapFrog`. I'll make sure that they are subclasses of `Integrator`:

Snippet 10.10
`integrator_euler.py` (abridged).

```
from integrator import Integrator

__integrator__ = 'Euler'


class Euler(Integrator):
```

```
    def __init__(self):
        super(Euler, self).__init__()
        self.__initialized = False

    def integrate(self, to_time=None):
        # for brevity, the actual algorithm is not shown here. Please
            refer to the previous chapters
        ...
```

Snippet 10.11
`integrator_leap_frog.py` (abridged).

```
from integrator import Integrator

__integrator__ = 'LeapFrog'


class LeapFrog(Integrator):

    def __init__(self):
        super(LeapFrog, self).__init__()
        self.__initialized = False

    def integrate(self, to_time=None):
        # for brevity, the actual algorithm is not shown here. Please
            refer to the previous chapters
        ...
```

Then, I will import all subclasses that we just implemented to `integrator.py`:

Snippet 10.12
Second version of the `Integrator` API.

```
from integrator_euler import Euler
from integrator_leap_frog import LeapFrog


class Integrator(object):

    def __init__(self):
        self.t = 0.0  # current model time
        self.h = 0.01  # integration time step
        self.t_end = 1.0  # the ending time
        self.particles = None  # particle set
        self.CONST_G = 1.0  # the gravitational constant
        self.__active_integrator = None

    def integrate(self):
        if self.__active_integrator is not None:
            self.__active_integrator.integrate()

    @property
    def active_integrator(self, int_name):
```

```
        if int_name == 'Euler':
            self.__active_integrator = Euler()
        elif int_name == 'LeapFrog':
            self.__active_integrator = LeapFrog()
        elif ...
```

I also created a setter for the attribute `active_integrator`, which allows the user to set the active integrator using just the name.

Bob: This is a great feature, Alice! I just have one minor comment: the way that you incorporate integrators into ABIE is hard-coded. Whenever a new integrator is implemented, we will need to add extra import lines into the class `Integrator`. Why don't we make this happen automatically instead? PYTHON allows us to dynamically import a class at runtime. So let's say that our integrators are implemented in the files `integrator_*.py`. We could scan the ABIE code directory to discover and import them all:

Snippet 10.13
Third version of the `Integrator` API.

```
__mpa_dir__ = os.path.dirname(os.path.abspath(__file__))
__user_shell_dir__ = os.getcwd()
class Integrator(object):
    ...
    @staticmethod
    def load_integrators():
        """
        Load integrator modules using the reflection mechanism
        :return: a dict of integrator class objects, mapping the name of
            the integrator to the class object
        """
        mod_dict = dict()
        module_candidates = glob.glob(os.path.join(__mpa_dir__, '
            integrator_*.py'))
        sys.path.append(__mpa_dir__)  # append the python path
        if __mpa_dir__ != __user_shell_dir__:
            # load the integrator module (if any) also from the current
                user shell directory
            module_cwd = glob.glob(os.path.join(__user_shell_dir__, '
                integrator_*.py'))
            for m_cwd in module_cwd:
                module_candidates.append(m_cwd)
            sys.path.append(__user_shell_dir__)  # append the python path
        for mod_name in module_candidates:
            mod_name = os.path.basename(mod_name)
            mod = __import__(mod_name.split('.')[0])
            if hasattr(mod, '__integrator__'):
                # if is a valid ABIE module, register it as a module
                mod_dict[mod.__integrator__] = mod
        return mod_dict
    ...
```

So the method `load_integrators()` is a static method, meaning that it can be invoked as `Integrator.load_integrator()` without initializing any instance. When called, it will scan all integrator classes with the file name pattern `integrator_.py`, then determine whether they are actually a valid integrator by testing the existence of the attribute `__integrator__`. The attribute `__integrator__` contains the name of the integrator, which is then used to build the `dict` that associates the name of the integrator with the class of the integrator. The `dict` structure is basically a hash table: if we give it the name of the integrator, it will return us the class of the integrator, which can then be used for carrying out the integration.

Alice: Great! So we just need to place our integrators in the ABIE code directory, name them `integrator_*.py`, and make sure that they contain the attribute `__integrator__`, and that's all. ABIE will be able to detect and load them.

Bob: Not only in the ABIE code directory, but also in the current directory of the user's terminal under which ABIE is launched. This is actually based on the *reflection*[3] software programming paradigm, which, according to Wikipedia, is the ability of a computer program to examine, introspect, and modify its own structure and behavior at runtime.

Alice: Fantastic. I will now port our other integrators to this paradigm.
(For brevity, the code is only available on GitHub.)

10.3.4 Implementing the Presentation Layer

Alice: All set! We have all our integrators in the ABIE framework now. I also tested them.

Bob: In order to run the code, it is time for us to build the top layer of our software, that is, the presentation layer. This layer interfaces between a computer program and the user, so that we can interact with the software through it.

10.3.4.1 Building a command-line interface

Alice: Didn't we already have a primitive command-line interface for each integrator before? Maybe we could generalize it.

Bob: Yeah, I'm also thinking of starting from there. I think that as the first step, we wanted to build a simple command-line interface for ABIE to accept user input such as the path/filename of the initial conditions file, the total integration time, the integration time step, the name of the chosen integrator, and some other parameters.

Alice: Command-line interface? I've always been fascinated that many UNIX terminal commands are followed by long argument lists, and now we're doing this for our ABIE! I

[3] https://en.wikipedia.org/wiki/Reflection_(computer_programming)

learned from my C programming class that the command-line arguments can be obtained through the `argv[]` array in the `main()` function:

Snippet 10.14
Obtaining the command-line arguments from the `main` function.

```
int main(int argc, char *argv[]){
    // argc is the number of arguments
    // argv[] is the string array of the arguments
    return 0;
}
```

Is there any equivalence in PYTHON?

Bob: Uh…. Yes! We could use a built-in PYTHON package `argparse` to do that:

Snippet 10.15
Command-line argument parsing with PYTHON.

```
import argparse

parser = argparse.ArgumentParser()
parser.add_argument('-c', '--config', help='config␣file', default=None)
parser.add_argument('-o', '--output_file', dest='output_file', help='
    output␣data␣file', default='data.hdf5')
parser.add_argument('-t', '--t_end', type=float, dest='t_end', help='
    Termination␣time')
parser.add_argument('-d', '--dt', type=float, dest='dt', help='Integration
    ␣time␣step␣(optional␣for␣certain␣integrators)', default=None)
parser.add_argument('-s', '--store_dt', type=float, dest='store_dt', help=
    'output␣time␣step', default=100)
parser.add_argument('-i', '--integrator', dest='integrator', help='Name␣of
    ␣the␣integrator␣[GaussRadau15|WisdomHolman|RungeKutta|AdamsBashForth|
    LeapFrog|Euler]', default='GaussRadau15')

args = parser.parse_args()
```

The `argparse` package is fairly easy to use. We just need to create an `ArgumentParser` instance, add the arguments one by one, and then call the `parse_args()` method. That's it. The package will automatically generate the help information. If we run the code above using the `-h` argument, it will generate the following help message:

```
 $ python abie.py -h
usage: abie.py [-h] [-c CONFIG] [-o OUTPUT_FILE] [-t T_END] [-d DT] [-s
    STORE_DT] [-i INTEGRATOR]

optional arguments:
  -h, --help            show this help message and exit
  -c CONFIG, --config CONFIG
                        config file
  -o OUTPUT_FILE, --output_file OUTPUT_FILE
                        output data file
  -t T_END, --t_end T_END
                        Termination time
```

```
-d DT, --dt DT        Integration time step (optional for certain
    integrators)
-s STORE_DT, --store_dt STORE_DT
                      output time step
-i INTEGRATOR, --integrator INTEGRATOR
                      Name of the integrator [GaussRadau15|WisdomHolman|
                          RungeKutta|AdamsBashForth|LeapFrog|Euler]
```

Alice: Wow, this is super nice! I really like that we can define the data type of the command-line arguments and their help messages. How do we access the parsed result?

Bob: Simple. All results are encapsulated in the structure `args` after we called `args = parser.parse_args()`. The options are arranged as attributes of this `args` object, which we defined in the `parser.add_argument()` method as the `dest` argument. So, for example, to access the termination time of the simulation, we just need to write it as `args.t_end`. And since the `t_end` argument is defined as `float`, we automatically get a `float` value when calling `args.t_end`.

Alice: Great. Then the remaining job is rather simple. Just call the proper method according to the values collected from the `ArgumentParser`. That's it.

Bob: Exactly.

For brevity, the source code for calling certain functions using the `ArgumentParser` values is only available on GitHub.

10.3.4.2 Building a PYTHON interface

Bob: With the command-line interface that we have built just now, we can run ABIE as a standalone application (just like most N-body codes). I thought that sometimes it would be even nicer to be able to run ABIE programmatically as a PYTHON library.

Alice: I didn't think about that. Wait … why is there a need?

Bob: Well, I read that astrophysical simulations are now getting increasingly complicated: multiple physical processes are taken into account, which usually requires multiple codes to work together. For example, to evolve the star formation process in a giant molecular cloud, we may need a hydrodynamics code for the gas, an N-body code for the dynamical interactions between stars, and a radiative transfer code for the heating/cooling processes of the cloud. If these codes all have a common PYTHON interface, it would be much easier to make them work together, since in this case, it is easier to pass the data or message from one to another. In fact, there is an astrophysical software framework called AMUSE, which stands for the "Astrophysical Multipurpose Software Environment" that does exactly this. AMUSE treats existing codes as modules and uses PYTHON to "glue" them together. Discussing AMUSE in detail goes a bit beyond our immediate needs and the scope of our project, but I personally think that it is worth providing a PYTHON interface for ABIE.

Alice: Sounds interesting. Multiphysics simulations…. Can you give an example of what ABIE's PYTHON interface would look like?

Bob: Sure. Let's consider this simple example:

Snippet 10.16
A minimum example of running ABIE programmatically.

```
from abie import ABIE

# create a new simulation
sim = ABIE()

# choose the GaussRadau15 as the integrator
sim.integrator = 'GaussRadau15'

# set the units by choosing the gravitational constant G
sim.CONST_G = 1.0  # units: AU, year/2*pi, MSun

# create a simple Sun-Earth-Moon system
sim.add(mass=1.0, name='Sun')
sim.add(mass=3.e-6, a=1.0, name='Earth', primary='Sun')
sim.add(mass=3.6943e-8, a=0.00257, name='Moon', primary='Earth')


# Evolve the system for 1.e6/(2*pi) years
sim.integrate(1.e6)
```

Alice: Now I see your point. This is very nice, because we don't need an initial conditions file or command-line arguments anymore. The user can fully control ABIE through this simple script. What's more, it allows a simulation to be created programmatically during the runtime, which opens up the possibility for multiphysics simulations.

Bob: Precisely! How nice to make our code "programmable." I think that we can actually achieve this goal relatively easily. Just like the way we used to implement the command-line arguments, we just need to call the relevant methods according to the value of the attributes passed by the user in their script. For example, when the user calls the `sim.add()` method, we just need to pass those arguments to the `Particles` container in the data layer; when the user sets the `sim.integrator` attribute, we just need to call the logic layer to set the integrator. Of course, we'll also need to pass the context (such as the setting of the gravitational constant G) to the chosen integrator.

Alice: I see. And so if the user, for example, wants the particle set of the current simulation, we just need to pass the particle set of the currently active integrator to the ABIE PYTHON API, right?

Bob: Yes, you are exactly correct.

Alice and Bob implement the PYTHON API of ABIE together.
For brevity, the code is only available on GitHub.

10.4 Accelerating the Code Using Native C

Alice: We're all set. I tested ABIE with its command-line interface and its PYTHON interface. They both work fine!

Bob: It's like our baby is finally born.

Alice: Don't be too delighted just yet. I found that it takes up to one minute to integrate the Solar System for 1,000 years using the Gauss–Radau integrator. So wouldn't this correspond to a very slow, and hence painful, birth? In contrast, popular codes such as MERCURY6 can do this within just one second. This is not entirely a fair comparison, though, since MERCURY6 uses a second-order mixed-variable symplectic (Chambers, 1999) integrator by default, which is a different propagator from our Gauss–Radau integrator. But still, think about a factor of 10^4 difference in computing time….

Bob: Really?! That slow?

Alice: Yep. Didn't we already utilize `numpy` in our code? My impression is that `numpy` is natively implemented in C and it is quite fast.

Bob: Yeah, you're right. As near as I can tell, the problem is that PYTHON is a dynamic language, and therefore a lot of data structures need to be created and deleted dynamically at runtime. For example, consider the following code:

```
import numpy as np
x = np.linspace(0, 100, 1001)
y = np.sin(x) + np.cos(x)
```

It is, of course, very concise and clear, but there are a lot of things happening behind the scenes: on line 2, an array with a length of 1,001 elements must be allocated and then filled linearly from 0 to 100 with a step size of 0.1. On line 3, `numpy` will have to allocate two more 1D arrays with the same length to store the values of `np.sin(x)` and `np.cos(x)`, and then create a third 1D array, also at the same length, to store the values of `y`. Afterward, `numpy` should free the space for the intermediate arrays at various points during the integration. Do you think that the dynamical management of data structures will result in significant overhead in computing time?

Alice: Yeah, I think you're right. The overhead could be significant. I might go so far as to venture that things could get downright ungodly. I'm afraid that this is the cost of using such a highly dynamic language. After all, there is no such thing as a free lunch. I think that the bottleneck of ABIE is the execution of the integrators. Maybe we should seek to compile the integrators to native machine codes?

Bob: I am aware of a few options to achieve that goal, actually. We could rewrite the entire integration algorithm *manually* in C and then call the routine from PYTHON, or we could use some PYTHON library to convert the PYTHON script into some low-level machine

codes *automatically*. The former approach can be achieved using packages like `ctypes`[4] and `swig`; the latter approach can be achieved using `cython`[5] or `numba`,[6] which are also PYTHON packages.

Alice: Wow, already plenty of choices! It seems that many people have encountered our situation and have come up with their own solutions. From what I've read, the `ctypes` method seems to be quite straightforward. Could you elaborate a bit more about other methods?

Bob: Happy to. The mechanism that `numba` uses is such that when we decorate a PYTHON method with a certain decorator [7] (in this case, `numba` provides the `@jit` decorator), it will know that this particular method should be compiled natively, and therefore it will generate native machine codes accordingly. `cython` is actually an extension of the PYTHON language, which allows static type declarations of variables and methods. Recall that PYTHON is an interpreting, dynamics language and so has the convenience to use a variable without declaring it a priori. But, as you said, there is no free lunch—this convenience comes with a cost, as the PYTHON integrator has to do a lot of extra checking to make sure that there is no memory leaking (i.e., a failure to free the parts of memory that are no longer needed by the program). If we are willing to sacrifice some convenience by making static declarations, we could free the interpreter from repeatedly performing those checks, which in turn improves the performance by reducing the computation overhead. `ctypes` and `swig`[8] work by allowing a native library (which usually runs faster than PYTHON) to be invoked from PYTHON. In this case, we will have to reimplement our integrators in C, compile them as a library, and let ABIE call it.

Alice: Nice introduction, plenty of new terms, which will take me some time to make sense of. Which method would you prefer?

Bob: I personally prefer to reimplement the algorithms in C and use `ctypes` to call them. In doing so, the code will be completely transparent to us, and we will have full control over it. And `ctypes` is part of PYTHON, so it doesn't depend on the third-party libraries. We can be assured that it always works as long as PYTHON is installed.

Alice: Oh, I see. Sounds great. I'd like our code to have the minimum amount of dependencies possible, since we want to make the installation of ABIE as simple as possible. And, having full control over the code is great.

---

[4] https://docs.python.org/3/library/ctypes.html

[5] https://cython.org

[6] https://numba.pydata.org

[7] In PYTHON, a decorator is a syntax and a mechanism to modify the functionality of a method, which starts with "`@`." For details, please refer to https://wiki.python.org/moin/PythonDecorators.

[8] http://www.swig.org

Bob: Great, let's go for `ctypes`.

10.4.1 Building a C Library

10.4.1.1 Source code structure and data types

Bob: So we are now basically turning ABIE into a "sandwich": the data layer is written in PYTHON, the logic layer (which consists of the integrators) is implemented in C, and the presentation layer is again written in PYTHON. Therefore, I think our C-based logic layer should be designed to accept exactly the same argument as the PYTHON-based logic layer, and also return the same data structure. In this way, we could then easily replace the logic layer with the C implementation without having to touch the other two layers.

Alice: Sounds like a great idea. Now I am seeing the advantage of the three-layer architecture we have adopted for ABIE. Most codes in astronomy/astrophysics are entirely written in one language, but I am a big fan of our hybrid code. It basically combines the conciseness of PYTHON and the efficiency of C. But how exactly does it work?

Bob: Let me show you a simple example. We have a C function to compute the Fibonacci series, which is defined recursively as

$$F_n = F_{n-1} + F_{n-2}, \tag{10.1}$$

with the boundary conditions $F_0 = 0$ and $F_1 = 1$. If we would like to calculate F_n in C, we could write something like

Snippet 10.17
Function `fibonacci.c`: Compute the Fibonacci sequence.

```
int fibonacci(int n) {
    if (n == 0) return 0;
    else if (n == 1) return 1;
    else return fibonacci(n+1) + fibonacci(n+2);
}
```

In order to call this function from PYTHON, we'll have to compile it as a shared object:

```
    gcc -shared -fPIC fibonacci.c -o libfib.so
```

This generates the shared library `libfib.so`. Our next task is to call it from PYTHON:

Snippet 10.18
`test_ctypes.py`: Call a C function from PYTHON using `ctypes`.

```
import ctypes
lib = ctypes.cdll.LoadLibrary('libfib.so')

# call the C function
n = 10
print(lib.fibonacci(ctypes.c_int(n)))
```

That's it.

Alice: Wow, that looks quite straightforward! So the idea is that one should compile the C code as a library instead of a standalone code, and let PYTHON invoke it. Nice!

Bob: Exactly, and in doing so, we will need to decide what data we should pass to the C library, what integration data we should get back, and how we will do this. Now let's first define the function signatures of our integrators:

```
#define real double
size_t integrator_gauss_radau15(real *pos, real *vel, real *m_vec, real *
    r_vec, size_t N, real _G, real _t, real _t_end, real _dt);
size_t integrator_runge_kutta(real *pos, real *vel, real *m_vec, real *
    r_vec, size_t N, real G, double _t, double _t_end, double _dt);
size_t integrator_wisdom_holman(real *pos, real *vel, real *m_vec, real *
    r_vec, size_t N, real _G, real _t, real _t_end, real _dt);
```

We will call these C functions with the particle data and any other needed information (e.g., the gravitational constant G, the integration time, the time step, and so on), and the function should return to us the state of the system (positions, velocities) at the designated time.

Alice: Why do you define the `double` data type as `real`?

Bob: Good question. By using this macro definition, we can easily change the precision of the code. For example, we can simply change the first line of the code to

```
#define real long double
```

Our code may be able to benefit from the extended double precision, depending on the underlying hardware and the compiler. When using the GNU C compiler on the Intel x86 CPU, the `long double` is an 80-bit float-point number, which has higher precision than the more commonly used `double` data type. This will be useful for some N-body systems. For example, when evolving a highly eccentric orbit, the integration at the orbital periapsis usually requires very high precision, and in this case, using a `long double` precision may be helpful.

Alice: Is float-point precision a limiting factor for the integrator accuracy?

Bob: It depends. Generally, yes, the accuracy of the integrator is limited by the machine float-point number precision. We cannot build an integrator with energy/angular momentum conservation better than the maximum precision of the built-in data types, for obvious reasons. When using the `double` data type, the best precision is $\sim 10^{-16}$ (see also a related discussion in section 5.6 of chapter 5). That's why we've never seen the energy conservation of our current PYTHON-based integrators perform better than this value. When using the `float` data type, the best precision is generally $\sim 10^{-8}$. On the other hand, the accuracy also depends on the integrator itself. Imagine that we use the forward Euler integrator; then, the energy error will quickly accumulate regardless of the machine precision.

Alice: That makes sense. If we use the `long double` data type, what kind of accuracy can we get, at least in theory?

Bob: It again depends on the hardware and the compiler. But for our computers, we use the `x86/x64` CPU and the GNU C compiler. The `long double` is 80-bit; 1 bit is used for the sign, 15 bits are used for the exponent, and 64 bits are used for the fraction. So the best accuracy would be $2^{-64} \sim 10^{-20}$, which is about four orders of magnitude better than the `double` type. Some compilers may consider `long double` as 128-bit `double double`, which should be even more precise.

Alice: Wow, that's nice! Why wouldn't people use `long double` all the time? Is it slower or is it using more memory than the "standard" `double`?

Bob: Yes, apparently it uses more memory. Nowadays, memory chips are cheap, and this may not be a real problem. But I guess the 80-bit float number is not really native to the machine (most machines have 64-bit architecture nowadays), so it is expected to be slower. Anyway, I think it would be nice to have this option built into our implementation. But I don't have the performance benchmark numbers in mind. Wait, we can just experiment with our ABIE!

Alice: Sure thing. So let's work on "translating" our PYTHON code to C!

Bob: Yes, we should do that. But before we proceed, I propose that we define the code structure. Since we have multiple integrators, it's better to put them in different files. We could probably put each integrator in a source code file `integrator_*.c`, with a corresponding header file `integrator_*.h`. Then, we set up `common.c` and `common.h` for the definitions of commonly shared variables and functions.

Alice: Sounds good to me.

Alice and Bob code up the C version of the integrators…. *Source code only available on GitHub.*

Seven days go by, with Alice and Bob working diligently throughout.

10.4.1.2 Compile the source code as a C library

Bob: I can't believe that it took us a whole week to just "translate" the PYTHON-based integrators into C!

Alice: Translation is never an easy business. But it's finally time to test it!

Bob: Yeah. We'll have to compile them into a library. Since we have multiple source code files, I think we should create a `Makefile`.

Alice: What is a `Makefile`?

Bob: It's basically a definition of compilation rules, which tells the computer how to compile and link the program. In general, it is structured like this:

```
target: dependencies
    commands for compilation
```

As you can see, the structure of a `Makefile` is pretty straightforward. A target (for example, an executable that we hope to create) depends on certain source files, and the commands for compiling the dependencies to the target are defined on the second line in this example.

So for ABIE, the `Makefile` looks like this:

```
# Using `long double` instead of `double` for improved precision
LONGDOUBLE = 0

OBJS = common.o integrator_runge_kutta.o integrator_gauss_radau15.o
    integrator_wisdom_holman.o

DEPS = common.h

ifeq ($(LONGDOUBLE), 1)
    CFLAGS += -DLONGDOUBLE -fPIC -O3 -std=c99
else
    CFLAGS += -fPIC -O3 -std=c99
endif

libabie.so: $(OBJS)
    $(CC) -o $@ $(OBJS) $(CFLAGS) -shared
    rm *.o

%.o: %.c $(DEPS)
    $(CC) -c -o $@ $< $(CFLAGS)

clean:
    rm *.o *.so *.pyc
```

The library can be compiled by simply typing `make` in the terminal.

Alice: Neat! It works!

Alice and Bob spend quite some time struggling with compilation errors. There are also several runtime errors, such as segmentation faults, that need to be addressed. The library was initially producing wrong results, but after a painstaking debugging process, they finally manage to make it work. The results produced by the C library are now fully consistent with the original PYTHON code. In the end, another week has passed.

Bob: Finally! It works!

Alice: Yeah, finally! My brain is all about debugging now. How does the performance look?

Bob: Let's test it. Here I have a very simple system: a Jupiter-mass planet orbiting a solar-mass star at 1 AU:

Snippet 10.19
Minimum performance benchmark code `abie_benchmark.py`.

```
from abie import ABIE

# create a new simulation
sim = ABIE()

# select the integrator
sim.integrator = 'WisdomHolman'

# Toggle the acceleration method between ctypes and numpy
sim.acceleration_method = 'ctypes'
# sim.acceleration_method = 'numpy'

# Define the units
sim.CONST_G = 1.0

# Add the particles
sim.add(mass=1.0, x=0.0, y=0.0, z=0.0, vx=0.0, vy=0.0, vz=0.0, name='Sun')
sim.add(mass=1.e-3, a=1, name='Planet')

# Integration time step
sim.h = 0.1

# Launch the integration
sim.integrate(10000)

# Stop and clean up
sim.stop()
```

When we use our newly built C library:

```
 $ time python abie_benchmark.py
t = 100.100000, N = 2, dE/E0 = -2.1684e-16
...
t = 9803.300000, N = 2, dE/E0 = -1.0842e-15
t = 9903.300000, N = 2, dE/E0 = -6.50521e-16
t = 10003.300000, N = 2, dE/E0 = -1.0842e-15
python abie_benchmark.py  0.19s user 0.09s system 98% cpu 0.282 total
```

It takes 0.19 seconds. Now let me set the `acceleration_method` to `numpy`, so that the code will use our original PYTHON routine:

```
 $ time python abie_benchmark.py
t = 100.100000, N = 2, dE/E0 = -2.1684e-16
...
t = 9803.300000, N = 2, dE/E0 = -1.0842e-15
t = 9903.300000, N = 2, dE/E0 = -6.50521e-16
t = 10003.300000, N = 2, dE/E0 = -1.0842e-15
python abie_benchmark.py  10.85s user 0.10s system 99% cpu 10.969 total
```

It takes about 11 seconds to carry out the same integration. This means that the C library is a factor of ~50 speedup compared to the PYTHON version!!

Alice: Wow, that's *really* amazing! Our one-week debugging effort has really paid off.

Bob: Not bad at all, huh? Our ABIE is now fast enough to carry out real astrophysical simulations!

Alice: Let's celebrate!

10.4.2 Further Performance Speedup with CUDA/GPU

10.4.2.1 Pushing the code to the limit

Alice: I am rather happy that our ABIE can tackle planetary systems with a few interacting bodies (i.e., small N systems). What if we use the code to simulate the Solar System with the Oort cloud? We would need to include a lot of (massless) particles.

Bob: Sounds like a very cool application! Well, let's find ABIE's limit. We'll create a simplified Solar System with only the gas giants (Jupiter, Saturn, Uranus, and Neptune), plus an Oort cloud consisting of 1,000 massless particles.

Snippet 10.20
Integrate a simplified Solar System with 1,000 Oort cloud particles.

```
from abie import ABIE
import numpy as np

# create a new simulation
sim = ABIE()

# select the integrator
sim.integrator = 'GaussRadau15'

# Use the native-C library
sim.acceleration_method = 'ctypes'

# Define the units
sim.CONST_G = 4 * np.pi ** 2

# Add the Sun and planets
sim.add(mass=1.0, x=0.0, y=0.0, z=0.0, vx=0.0, vy=0.0, vz=0.0, name='Sun')
sim.add(mass=9.5458e-4, a=5.2, e=0.05, name='Jupiter')
sim.add(mass=2.858e-4, a=9.537, e=0.05, name='Saturn')
sim.add(mass=4.366e-5, a=19.189, e=0.05, name='Uranus')
sim.add(mass=5.15e-5, a=30.07, e=0.0086, name='Neptune')

# Add the Oort Cloud particles
n_oc = 1000
semi = np.random.uniform(500, 10000, n_oc)
ecc = np.random.uniform(0.3, 0.99, n_oc)
inc = np.random.uniform(0, 2*np.pi, n_oc)
for i in range(n_oc):
    sim.add(mass=0, a=semi[i], e=ecc[i], i=inc[i], name=('test_particle_%d
        ' % i))
print(sim.particles)
```

```
# Launch the integration
sim.integrate(1000)

# Stop and clean up
sim.stop()
```

Bob: Let's see how fast it is:

```
 $ time python oort_cloud.py
...
t = 100.436731, N = 1005, dE/E0 = -6.06663e-15
...
t = 900.496336, N = 1005, dE/E0 = -3.03332e-15
t = 1000.892486, N = 1005, dE/E0 = -4.04442e-15
python oort_cloud.py  124.16s user 0.19s system 99% cpu 2:04.47 total
```

Alice: Wow! It takes more than two minutes just to advance the system for 1,000 years. I can't imagine how long it would take to integrate an Oort cloud of 10^6 particles for a few Gyr….

Bob: Yeah, that's a real problem. But this is not really a surprise, because we are using a direct N-body algorithm, and so the computational time scales with the number of particles N as $\mathcal{O}(N^2)$. A tree code could be faster. For example, the Barnes–Hut Tree (Barnes and Hut, 1986) is an $\mathcal{O}(N \log N)$ algorithm. Could we use it for our large N problems?

Alice: It depends, but in our case, we need a direct N-body algorithm to more accurately resolve close encounters, whereas a tree code is mainly for collisionless systems such as galaxies. Actually, I think that integrating a large number of massless particles would still be cheap, because we do not need to calculate their gravity contributions. So we should optimize our double loops in the force calculation routine for massless particles.

Bob: Yes, you certainly have a great point that massless particles are cheap. Integrating them only requires an $\mathcal{O}(N)$ operation. In our current code, a quick fix may be done by checking whether the mass of a particle is zero, and if so then we skip the calculation of its contribution. But still, the mass checking has to be performed N^2 times for a system of N particles, so a better approach is to maintain a list of massive particles and another list of test particles and only loop over the list of massive particles to sum up their gravity on all other particles.

But if we could not treat certain particles as massless (e.g., simulating a star cluster or the coagulation of planetesimals), then we can instead try to parallelize the code. I think that the most expensive calculation in each time step is the calculation of pairwise accelerations, which is an $\mathcal{O}(N^2)$ operation. Other parts, such as kicking and drifting the particles, are cheap because they are $\mathcal{O}(N)$ operations. We can parallelize the acceleration calculation on either the CPU or GPU, or even both. But since we expect to have large N, it makes more sense to implement the acceleration calculation on the GPU.

Alice: Why?

Bob: Well, a modern GPU card typically has thousands of cores, whereas a modern multicore CPU only has a few cores. While a GPU core is not as powerful as a CPU core, the hardware architecture of the GPU is optimized to perform a large number of independent and relatively simple calculations in parallel. I think that our problem is well suited for GPUs, because we can divide the calculation of pairwise accelerations into N subtasks, each of which calculates the acceleration acting on one particle due to the other $N-1$ particles. These subtasks are independent of each other, meaning that the calculations performed by one subtask do not require any data from another subtask.

Alice: What exactly is a GPU, and why it is optimized for performing large numbers of simple tasks in parallel?

Bob: The term "GPU" stands for "graphics processing unit" and, as its name suggests, a GPU is specialized to accelerate the generation and processing of graphical data for our computer screens. Our computer screens have millions of pixels, which can be described as a 2D matrix (in the term of computer graphics, this matrix is called the frame buffer). Displaying an image or a shape on the screen means setting certain values (e.g., colors) on certain parts of the frame buffer, and transforming an image or a shape means applying a transformation matrix to the frame buffer to perform element-wise transformations. The frame buffer needs to be updated dozens of times per second to make sure that the screen looks smooth enough to human eyes, so these linear algebra transformations should be performed in a highly efficient way. A GPU is designed to perform these transformations as efficiently as possible, and a fundamental approach to achieve this is parallelization—make sure that these transformations are distributed evenly on many cores and have them executed simultaneously. This is the reason why a GPU usually has many cores. In fact, a modern GPU can easily have a few thousand cores.

Alice: So, if we port our calculation of gravity to these GPU cores, can we gain a speedup?

Bob: Yes, I think so! Well, at least I hope so. In the beginning, GPUs were really only designed for processing graphics, so in order to perform more general-purpose calculations, one would need to structure the calculation in such a way that the GPU is "tricked" into thinking it is dealing with standard graphical data. But in recent years, with the rapid development of general-purpose GPU (GPGPU) computing technology, special APIs have been developed to use GPUs as computing devices. Mature APIs for GPGPU include CUDA (developed by Nvidia) and OpenCL (originally developed by Apple, now an open standard). These APIs hide the details of the hardware and therefore give us the convenience to implement our algorithm in a more intuitive way.

Alice: Using GPUs to calculate the gravity in parallel—sounds like a very cool idea. Can we do this on our laptops?

Bob: Hmmm, it depends. Our laptop may not have dedicated hardware for that. For example, if we are going to use CUDA, we'll need an Nvidia[9] GPU. OpenCL does not require a specific GPU brand (for example, a GPU manufactured by AMD[10] would also work), but we'd still need a proper GPU to gain the desired speedup. But even if we had a proper GPU, I don't know how to do that.

Alice: I think we should ask Prof. Starmover for her help.

Bob: Agreed.

Alice and Bob go to Prof. Starmover's office.

10.4.2.2 The CUDA architecture

Professor: What's up, guys?

Bob: Doing pretty well! We rebuilt our N-body integrators using the object-oriented programming paradigm. It's now an integrator framework, and adding a new integrator in the future would be like adding a plug-in to the framework. We call the framework "Alice–Bob Integrator Environment," or ABIE.

Alice: And we rewrote the integrators using native C as a library. When calling the integrator library from Python, we obtained a speedup of fifty times compared to the original Python implementation. We think we're ready to use our code ABIE to do some astrophysical projects. We also managed to push the code to its limit—after adding 1,000 test particles in the hope of simulating the Oort cloud, the integrator became so slow that long-term integration becomes impractical.

Bob: Right, so here we are to get help from you with exploring possible options to further speed up the code. We've been thinking about using GPUs for calculating the pairwise gravity terms, which is an $\mathcal{O}(N^2)$ operation, but we have no experience with GPU programming or the actual hardware….

Professor: This is excellent progress! Very well done! It makes a lot of sense to put some software engineering practices into your code. The results so far sound very promising, and I'm not surprised to hear that the code has difficulty handling large numbers of (test) particles. I'm very impressed that you guys pushed the code to its current limit and have even started to think about accelerating the code with GPUs. Look! There is a GPU machine right here in my office. It has an Nvidia Titan RTX GPU with 4,608 cores. I can create accounts on this machine for you guys so that you can play with the GPU!

Alice: That would be outstanding!

---

[9] Registered trademark of Nvidia Corporation. Homepage: https://www.nvidia.com/

[10] Registered trademark of Advanced Micro Devices, Inc. Homepage: https://www.amd.com

Bob: Thank you so much, Prof. Starmover! I am soooo excited to play with the GPU. Wait … I don't know how to program on the GPU….

Professor: No problem! I'll give you a crash course on GPU programming, using the case study of accelerating the calculation of gravity.

So, since we have an Nvidia GPU here, my introduction would be biased toward CUDA. Regardless, the idea is transferrable to other GPGPU platforms, such as OpenCL. The fundamental idea behind many-core computing is to divide a task into a number of subtasks and utilize the cores to calculate the subtasks simultaneously. To achieve this, a computing paradigm called *single instruction, multiple data (SIMD)* is often used. Consider the following snippet, which uses the SIMD paradigm to add up two vectors in parallel:

Snippet 10.21
A simple CUDA kernel of adding two vectors.

```
__global__ void vectorSum(float *sum, const float *a, const float *b, int
    size) {
    int idx = threadIdx.x + blockDim.x*blockIdx.x;
    if (idx >= size) return;
    sum[idx] = a[idx] + b[idx];
}
```

As you can see, the same function `vectorSum` will be executed simultaneously on multiple GPU cores. However, each GPU core will be performing calculations on different data. This works because the CUDA runtime automatically assigns a thread ID to the variable `threadIdx.x` and a block ID to the variable `blockIdx.x`, and so the value of `idx`, which is calculated according to the values of `threadIdx.x`, `blockIdx.x`, and `blockDim.x`, will be different in each GPU thread. With the different `idx` values, the algorithm fetches data from the corresponding part of the data arrays `a` and `b`, adds up the value, and passes the summation back to the corresponding index of the result array `sum`.

Bob: What is a block and what is `blockDim.x`?

Professor: Good questions! The short answer is that CUDA arranges GPU processors and the GPU memory in a hierarchical way, such that multiple processors are grouped into a block.

Prof. Starmover draws the memory model of CUDA on the whiteboard….

Look (Prof. Starmover points to the whiteboard, which is shown in figure 10.2), each processor has a very high speed but small *local memory*; a block uses a shared memory, shared by the processors corresponding to the same block. The shared memory is larger than the per-processor local memory but still small (of the order of tens of KB on our Titan RTX GPU). Larger amounts of data will only fit into the global memory, which is much larger but also much slower. Multiple blocks are combined into a grid.

So, coming back to your question, Bob, it is due to the hierarchical architecture that a thread has its own `threadIdx`, which identifies its position in a block, and also a `blockIdx`,

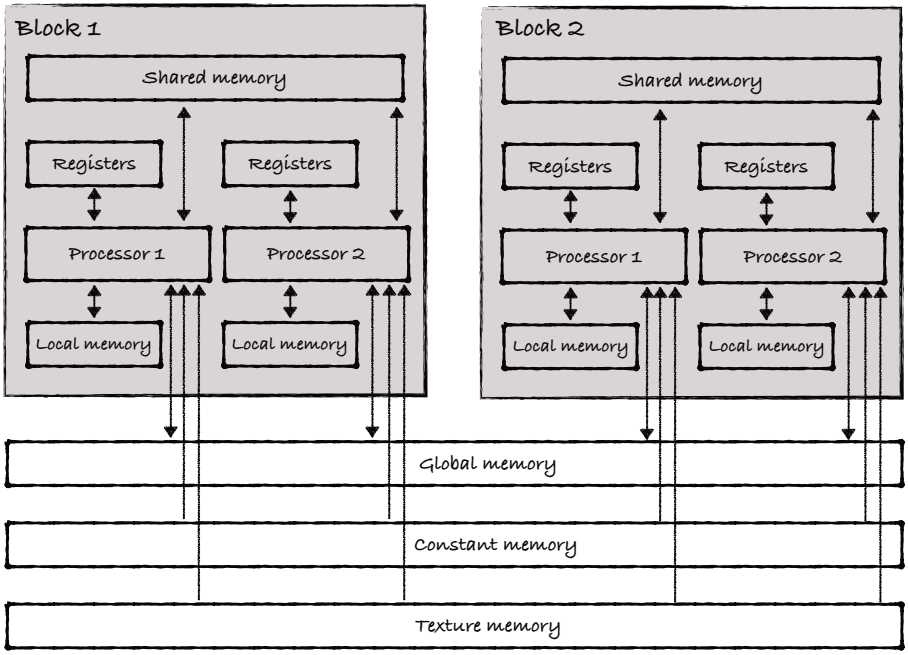


Figure 10.2
The CUDA memory model. Processors are grouped in units of blocks. A GPU processor has exclusive access to its own registers and local memory; the processors in the same block have shared access to the block-wide shared memory, but the shared memory of one block is not accessible from another block. All blocks have read/write access to the global memory, but the constant memory and the texture memory are read-only. Multiple blocks are grouped into a grid (not shown).

which identifies the position of its home block in the grid. The block size (the number of threads per block) is automatically assigned by the CUDA runtime as `blockDim` in the code snippet above. A block or a grid can be 1D, 2D, or 3D, depending on the data structure. In our simple example above, we are adding up two 1D vectors, so it is natural to create 1D blocks, for which the thread can be identified as `threadIdx.x`. If we are going to handle 2D arrays, then sometimes it is more convenient to make 2D blocks to match the shape of the underlying data, in which case a thread can be identified as `(threadIdx.x, threadIdx.y)`, and so on.

Alice: That *is* complicated…. The architecture is elaborate, but why would CUDA make the memory hierarchical? Wouldn't it be easier to create a large memory and let the processors access it directly, just like with CPUs?

Professor: That's another good question! It would certainly be conceptually easier if the GPU memory is not hierarchical. The main reason for making this hierarchical structure is for the sake of performance. Consider the following analogy: we have N workers in a factory. If each worker has to visit the warehouse every single time he or she does a job, then the warehouse will be pretty busy. Moreover, it is possible that two workers may simultaneously request access to the same object in the warehouse, in which case one worker will have to wait until the other worker is finished with the object and puts it back in its original place. The waiting worker does nothing during that time, so the performance is undermined significantly. Another point is that the trip from the worker's office to the warehouse and back also takes time. However, if each worker has his or her own small storage space such as an office, then he or she can put the most frequently accessed objects in there without having to visit the warehouse too often. This measure saves both the travel time between the office and the warehouse and any potential waiting time in the event of simultaneous access conflict.

Alice: Ah, that makes sense! Basically, the local memory serves as the cache.

Professor: Exactly! A bit more than just a cache—a routine executing on a processor (which is called a *kernel* in the GPU terminology) can store a local variable in its registers and local memory and only report its final result back to the shared memory and global memory.

Bob: Wait a minute. I just noticed that there's no loop in the simple CUDA kernel you showed us, Prof. Starmover. Wouldn't it be necessary to loop over the elements of the vectors `a` and `b` one by one to compute their sum?

Professor: Ah, good catch! We don't explicitly write the loop here because CUDA launches N threads to calculate the addition of two vectors with N elements. The CUDA runtime automatically assigns the corresponding values of `threadIdx`, `blockIdx`, and `gridIdx`, so that we can work out the index for selecting the correct elements from the vectors. The CUDA runtime automatically allocates processors to execute these threads, so we don't need to worry about which processor performs which threads. The kernel can be launched using a special syntax:

```
vectorAdd<<<n_blocks, threads_per_block>>>(sum, a, b);
```

Here, `n_blocks` is the number of blocks, `threads_per_block` is the number of threads in each block, and the three arguments in the bracket are the standard arguments for the kernel. I assume that `sum`, `a`, and `b` are already properly allocated before launching the kernel. In a standard CPU program, we usually use the `malloc()` function to allocate memory and the `free()` function to release memory. The CPU memory is not directly usable on the GPU, so on the GPU, we need to use

```
// allocate an array
float *dev_a;
cudaMalloc((void **) &dev_a, N * sizeof(float));

// do something

// free the memory
cudaFree(dev_a);
```

Apart from the `cuda` prefix, it is very similar to the standard way of allocating CPU memory. In the `CUDA` terminology, the CPU memory is called *host memory*, and the GPU memory is called *device memory*. To transfer data from CPU to GPU or the other way around, we'll need to use the `cudaMemcpy()` function.

Bob: Wow, there are lots of peculiar things going on here!

Alice: I am getting a headache. Is it worth the effort?

Professor: Let's implement our GPU-accelerated pairwise gravity calculation and see whether the speedup is worth the effort.

Bob: I'm very curious about the results.

Alice: Me too!

10.4.2.3 Implementing the gravity calculation with CUDA

Professor: Let's get started. We learned from the previous example of vector add that we do not need to write a loop explicitly in our kernel. However, in the case of our gravity calculations, we need to sum up the superposition of the acceleration of $N-1$ particles exerted upon each particle. This means that we need to perform a loop in the code. A naive implementation (i.e., without any optimization) could look like this:

Snippet 10.22
A naive implementation of gravity calculation on GPU.

```
__global__ void gpuforce(double4 *p, double G, int n, double3 *acc) {
    // obtain the thread index
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < n) {
        // for each particle, initialize the sum of acceleration to zero
        double Fx = 0.0f; double Fy = 0.0f; double Fz = 0.0f;

        // calculate the sum of the accelerations acting on particle i
            due to the other particles
        for (int j = 0; j < n; j++) {
            double m = p[j].w; // mass of particle #j
            // skip the particle itself and ignore the contribution of
                massless particles
            if (i == j || m == 0) continue;
            double dx = p[i].x - p[j].x;
            double dy = p[i].y - p[j].y;
```

```
            double dz = p[i].z - p[j].z;
            double distSqr = dx*dx + dy*dy + dz*dz;
            // Avoid singularity
            if (distSqr == 0.0) continue;
            double invDist = rsqrt(distSqr);
            double invDist3 = invDist * invDist * invDist;

            Fx -= (G * m * dx * invDist3);
            Fy -= (G * m * dy * invDist3);
            Fz -= (G * m * dz * invDist3);
        }
        acc[i].x = Fx; acc[i].y = Fy; acc[i].z = Fz;
    }
}
```

Here, the data of each particle are stored in a `double4` object, which has a structure of four `double`: (x, y, z, m). The collection of particle data is the `double4 *p` vector. We also need the gravitational constant G and the number of particles n to perform the calculation. The final results are written back to the `double3 *acc` vector.

Bob: The structure looks quite clear. Oh, I forgot to ask you earlier—what does the `__global__` keyword do?

Professor: `__global__` is a qualifier that makes a CUDA function become a kernel, which means that it can be called from the host (i.e., in the context where CPU codes are running). In contrast, a function with `__device__` qualifier can only be called on the GPU side (i.e., on the device).

Alice: I'm following you as well. Why wouldn't we set the mass array apart? I mean, using something like

```
__global__ void gpuforce(double3 *p, double *masses, double G, int n,
    double3 *acc);
```

Professor: You got it, Alice! It is more efficient to put the position and the mass together, so that each time when the GPU loads the data, the thread has all the data it needs. If we separate the position vector and the mass vector, the runtime needs to fetch the position *and* then fetch the mass in order to calculate the acceleration. This will introduce quite some overhead.

Bob: Nice. Shall we test it?

Professor: Sure. I'm curious too. So, we aim to replace the CPU gravity calculation routine with that of the GPU. Maybe we can make a switch like this in the host code:

```
int calculate_accelerations(const real pos[], const real vel[], size_t N,
    real G, const real masses[], const real radii[], real acc[]) {
#ifdef GPU
    gravity_gpu(pos, N, G, masses, radii, acc);
#else
    gravity_cpu(pos, N, G, masses, radii, acc);
```

```
#endif
    return 0;
}
```

In this snippet, we are using the conditional compilation feature. If we pass a flag `-DGPU` to the compiler, it will compile the `gravity_gpu()` clause and ignore the other clause; if we do not pass the flag to the compiler, then by default it will ignore the GPU clause and only compile the `gravity_cpu()` line.

Bob: I like the idea that both `gravity_cpu()` and `gravity_gpu()` have the same interface and take exactly the same arguments. And I guess the reason that we use conditional compilation here is that we want to make sure that our code can be compiled successfully on machines without CUDA support, right?

Professor: You got it exactly right—people without GPUs should also be able to use our code without a problem—it will just run slower for large-N systems. We should be careful when introducing additional dependencies of libraries since this will make installing the code a lot harder.

All right, now I'm very curious how fast the code will be, with this naive implementation. I'll finish the function `gravity_gpu()`, which makes sure that arrays on the GPU are properly allocated and initialized, and the kernel is launched with proper parameters. Here we go:

Snippet 10.23
The driver of the GPU kernel.

```
#define BLOCK_SIZE 256
extern "C" {
    int inited = 0;

    void gpu_init(int N) {
        if (inited) return; // return if already initialized
        int err = 0;
        err = cudaMalloc(&pos_dev, N * sizeof(double4));
        if (err > 0) {printf("cudaMalloc␣err␣=␣%d\n", err); exit(0); }
        err = cudaMalloc(&acc_dev, N * sizeof(double3));
        if (err > 0) {printf("cudaMalloc␣err␣=␣%d\n", err); exit(0); }
        inited = 1;
        printf("GPU␣force␣opened.\n");
    }

    void gpu_finalize() {
        printf("Closing␣CPU␣force...");
        if (pos_dev != NULL) cudaFree(pos_dev);
        if (acc_dev != NULL) cudaFree(acc_dev);
        printf("done.\n");
    }

    int gravity_gpu(const real vec[], size_t N, real G, const real masses
        [], const real radii[], real acc[]) {
```

```
        // rearrange the memory to put positions and masses together
        double * pos_host = (double *)malloc(N * 4 * sizeof(double));
        for (size_t i = 0; i < N; i++) {
            pos_host[4 * i] = vec[3 * i];
            pos_host[4 * i + 1] = vec[3 * i + 1];
            pos_host[4 * i + 2] = vec[3 * i + 2];
            pos_host[4 * i + 3] = masses[i];
        }

        cudaError_t err;
        gpu_init(N); // initialize the GPU if necessary

        // copy the particle data from host to device, capture the error
            if any
        err = cudaMemcpy(pos_dev, pos_host, N*sizeof(double4),
            cudaMemcpyHostToDevice);
        if (err > 0) {printf("cudaMemcpy␣err␣=␣%d,␣host_to_dev\n", err);
            exit(0); }

        int BLOCK_SIZE = 256;
        int nBlocks = (N + BLOCK_SIZE - 1) / BLOCK_SIZE;

        // launch the kernel
        gpuforce<<<nBlocks, actual_block_size>>>(pos_dev, (double) G, (int
            ) N, acc_dev);


        err = cudaGetLastError();
        if (err != cudaSuccess) {printf("Error:␣%d␣%s\n", err,
            cudaGetErrorString(err)); exit(0);}

        // copy the data from device back to the host
        err = cudaMemcpy(acc, acc_dev, N*sizeof(double3),
            cudaMemcpyDeviceToHost);
        if (err != cudaSuccess) {printf("cudaMemcpy␣err␣=␣%d,␣%s\n", err,
            cudaGetErrorString(err)); exit(0); }

        free(pos_host);

        return 0;
    }
}
```

The program looks clumsy, because we have to do a lot of bookkeeping for the GPU-related functions (such as monitoring the return code, copying the data, allocating/freeing the memory, restructuring the data, and so on). But let's leave it like that for now and compile the code.

Bob: Why do we wrap all these routines in the `extern "C"` block?

Professor: That's because the CUDA compiler `nvcc` will consider them as C++ functions if they aren't wrapped in the `extern "C"` block. When the functions are compiled as C++

functions, the names of these functions will likely be *mangled*,[11] making them inaccessible to the C functions we've built so far.

Bob: I get it. I guess it's time to engineer the `Makefile` so that the GPU code can be compiled properly. I'm excited.

Professor: All right. Let's put everything related to the GPU in a CUDA source file `gpuforce.cu` and add it to the `Makefile`:

```
# Use CUDA/GPU to perform the calculation of the accelerations
GPU = 0

# CUDA installation path
CUDA_SDK_PATH=/usr/local/cuda

ifeq ($(GPU), 1)
    OBJS = common.o integrator_runge_kutta.o integrator_gauss_radau15.o
        integrator_wisdom_holman.o additional_forces.o gpuforce.o
    CFLAGS += -DGPU
else
    OBJS = common.o integrator_runge_kutta.o integrator_gauss_radau15.o
        integrator_wisdom_holman.o additional_forces.o
endif

ifeq ($(GPU), 1)
    NVCC = nvcc
else
    NVCC = $(CC)
endif

libabie.so: $(OBJS)
ifeq ($(GPU), 1)
    $(CC) -o $@ $(OBJS) $(CFLAGS) -shared -L${CUDA_SDK_PATH}/lib64 -I${
        CUDA_SDK_PATH}/include -lcudart -lstdc++
    # rm *.o
else
    $(CC) -o $@ $(OBJS) $(CFLAGS) -shared
    rm *.o
endif

gpuforce.o: gpuforce.cu
ifeq ($(GPU), 1)
    $(NVCC) -Xcompiler -fPIC -DGPU -O3 -g $^ -c -o $@
endif

%.o: %.c $(DEPS)
    $(CC) -c -o $@ $< $(CFLAGS)

%.o: %.cpp $(DEPS)
    $(CC) -c -o $@ $< $(CFLAGS)
```

[11] For details, see https://en.wikipedia.org/wiki/Name_mangling.

```
clean:
    rm *.o *.so *.pyc
```

We use `nvcc` to compile the GPU-related routines if the flag `GPU` is switched on. The compiled binary object file will be used along with other object files to build the library `libabie.so`.

Alice: Let me be the person to compile the code!

```
 $ make clean; make
cc -c -o common.o common.c -DGPU -fPIC -O3 -std=c99 -g
cc -c -o integrator_runge_kutta.o integrator_runge_kutta.c -DGPU -fPIC -O3
     -std=c99 -g
cc -c -o integrator_gauss_radau15.o integrator_gauss_radau15.c -DGPU -fPIC
     -O3 -std=c99 -g
cc -c -o integrator_wisdom_holman.o integrator_wisdom_holman.c -DGPU -fPIC
     -O3 -std=c99 -g
cc -c -o additional_forces.o additional_forces.c -DGPU -fPIC -O3 -std=c99
    -g
nvcc -Xcompiler -fPIC -DGPU -O3 -g gpuforce.cu -c -o gpuforce.o
nvcc -Xcompiler -fPIC -DGPU -O3 -g bodysystemcuda.cu -c -o bodysystemcuda.
    o
cc -o libabie.so common.o integrator_runge_kutta.o
    integrator_gauss_radau15.o integrator_wisdom_holman.o
    additional_forces.o gpuforce.o -DGPU -fPIC -O3 -std=c99 -g -shared -L/
    usr/local/cuda/lib64 -I/usr/local/cuda/include -lcudart -lstdc++
```

Yes, it compiles successfully!

Bob: Give it a shot!

Alice: Sure thing!

```
$ time python oort_cloud.py
Setting the integrator to GaussRadau15
Initializing the code...GPU force opened.
Initialized.
t = 100.440368, N = 1005, dE/E0 = -6.06663e-15
t = 200.204867, N = 1005, dE/E0 = -6.26885e-15
...
t = 1000.892490, N = 1005, dE/E0 = -6.06663e-15
python oort_cloud.py  47.49s user 9.98s system 101% cpu 56.606 total
```

Well, the execution time of the GPU code is about one-third of the original CPU code execution time. It's faster already!

Bob: I'm happy to see that but, honestly, not terribly impressed. Only three times the speed of a CPU?

Professor: Well, the comparison isn't entirely fair. We have to do many more things when launching the GPU kernel, in particular putting the positions and masses together and copying the memory back and forth. This is the main bottleneck. Since our Oort

cloud particles have very large semimajor axes ranging from 5,000 to 10^5 AU, their orbital periods are very long. The code is mainly slowed down by the short orbital period of the innermost planet—in our case, Jupiter. If you remove the four gas planets and keep only the Sun and the Oort cloud particles, the code will be able to adopt a much longer integration time scale. This means it will reduce the frequency of calling `gravity_gpu()` and along with it the frequency of working on the overheads I mentioned.

Alice: Okay, so as an experiment, let's try to remove the planets.

Alice comments out the lines that add in the planets. In order to make sure that the energy conservation check works, Alice modifies the mass of the Oort cloud particles by giving them each a small mass of $10^{-15} M_\odot$. After rerunning the simulations with and without the GPU, here are the results….

Alice: With the GPU:

```
$ time python oort_cloud.py
Setting the integrator to GaussRadau15
Initializing the code...Initialized.
t = 126.789580, N = 1001, dE/E0 = -1.59331e-15
...
t = 1076.761464, N = 1001, dE/E0 = -4.58077e-15
python oort_cloud.py  2.21s user 1.29s system 130% cpu 2.675 total
```

And without the GPU:

```
$ time python oort_cloud.py
Setting the integrator to GaussRadau15
Initializing the code...Initialized.
t = 126.789580, N = 1001, dE/E0 = -1.59331e-15
...
t = 1076.761464, N = 1001, dE/E0 = -4.58077e-15
python oort_cloud.py  18.41s user 0.83s system 104% cpu 18.355 total
```

So the GPU routine is nine times faster than the CPU routine.

Bob: That's cool! We improved the performance by about one order of magnitude! My guess is that if we increase N, the factor will be even bigger. How about trying $N = 5,000$? As a quick test, let's just run it for 100 years.

Alice: Sure! And here are the results. With the GPU:

```
$ time python oort_cloud.py
Setting the integrator to GaussRadau15
Initializing the code...GPU force opened.
Initialized.
t = 34.952500, N = 5001, dE/E0 = -4.39244e-13
t = 43.690600, N = 5001, dE/E0 = -2.77439e-13
t = 52.428700, N = 5001, dE/E0 = -5.54466e-14
t = 61.166800, N = 5001, dE/E0 = -7.18951e-13
t = 96.119300, N = 5001, dE/E0 = -5.51992e-13
t = 104.857400, N = 5001, dE/E0 = -4.94691e-14
python oort_cloud.py  17.21s user 4.42s system 104% cpu 20.799 total
```

And without the GPU:

```
$ time python oort_cloud.py
Setting the integrator to GaussRadau15
Initializing the code...Initialized.
t = 34.952500, N = 5001, dE/E0 = -4.39244e-13
t = 43.690600, N = 5001, dE/E0 = -2.77439e-13
t = 52.428700, N = 5001, dE/E0 = -5.54466e-14
t = 61.166800, N = 5001, dE/E0 = -7.18951e-13
t = 96.119300, N = 5001, dE/E0 = -5.51992e-13
t = 104.857400, N = 5001, dE/E0 = -4.94691e-14
python oort_cloud.py  478.51s user 1.20s system 100% cpu 7:58.77 total
```

Wow, the GPU routine is now twenty-eight times faster than the CPU routine!

Bob: Confirmed! So if N is too small, the CPU implementation could actually be faster. We could check later.

Professor: I'm happy to see that even with a naive implementation, we can gain in some cases significant speedup. And I think we know what to improve next, don't we? Here are some issues that could be further improved upon:

1. Adjust the memory layout of the `C` library so to put the position and mass of each particle together. This would free us from having to reorganize the memory layout at each time step.
2. Make use of the shared memory in each block to cache the positions of a subset of particles. Currently, we use the global memory to store the entire particle set, which is a very inefficient way of using the CUDA architecture.
3. Use the 32-bit `float` data type whenever the precision of `float` is sufficient. Operations with the `double` data type are much slower than the `float` data type on the consumer-class GPUs, because these GPUs are originally optimized for gaming purposes, and for the rendering of video games, 32-bit `float` is usually sufficient.
4. Tune the block size to achieve optimal performance.

Bob: This is very helpful advice. Thank you very much, Prof. Starmover!

Alice: Yes, thank you so much, Prof. Starmover!

Bob and Alice play with systems composed of different numbers of particles N.

Bob: Look! We again sped up the code by about an order of magnitude. In combination with the previous speedup we already achieved (the native `C` library is fifty times faster than the original PYTHON implementation), we sped up ABIE by a factor of 1,000!

Alice: It's incredible how much we can extract from the potential of a computer. I'm really excited to start using ABIE for an actual astrophysics project.

Bob: Me too! But first, we should give ourselves a pat on the back for a job well done with our GPU-accelerated code—ABIE.

Alice: Definitely!

Alice takes a step back, pulls her arm way back with an open-palmed hand, then slaps Bob on the back. SMACK!

11 Defining the Project

Overview. In this chapter, Alice and Bob choose a research project to apply their new integrator to. After some brainstorming with Prof. Starmover, they decide to tackle the subject of the stability of exomoons around circumbinary planets. Prof. Starmover also explains the concept of a computer cluster and grants Bob and Alice access to her own cluster to improve the efficiency when performing their simulations.

Bob: We celebrated last night. Now we are ready to move forward with our project.

Alice: I had fun. Definitely. I'm glad we went out to celebrate the successes we've had so far. And, I can honestly say that I have never in my life witnessed dancing like that. And, for as long as I live, I will never forget it.

Bob: I was that good?

Alice: Oh dear Lord, no. You were that bad.

Bob: Great, now I have to come to terms with being a bad dancer.

Alice: Well, if it makes you feel any better, I have no idea how to dance. So we're in the same boat.

Bob: I feel better. I'm excited to get into choosing a project! Let's do that!

11.1 What Project to Tackle?

Alice: Dare I say, we're ready to apply our integrator to an *actual* astrophysics problem.

Bob: Right on! Uh…. But how will we define a project to tackle?

Alice: Any ideas?

Bob: Not really.

Alice: Hmmmm…. Well, we started with systems composed of two objects. Then we moved to three-body and larger-N systems. But we never really worried about what each of the particles actually represents.

Bob: What do you mean?

Alice: Like, what are the particles? Are they stars, planets, moons, asteroids, comets, alien mega-structures, and so on?

Bob: I like this alien mega-structure idea.

Alice: I thought we wanted to try our hands at *real* science.

Bob: Sigh. All right, fair enough. Let me think for a second….

Alice: Your head hasn't exploded yet, so I will take that as a good sign.

Bob: We know of binary *star* systems, right?

Alice: Yes.

Bob: What about *triple* star systems?

Alice: Yep, they are known to exist and be in stable configurations. They tend to be hierarchical, in the sense that one of the orbits is very compact, and the other is very wide. This is needed to ensure long-term stability; effectively, the outer star needs to be far enough away to perceive the inner binary star system as a single object.

Bob: Gotcha. That makes sense. Okay, so, could you ever have a binary star system with a planet as the outer stable triple companion?

Alice: Of course! Those have been observed and are called circumbinary planets.

Bob: Really?! Okay, okay, I'm getting excited. Do any of those systems have any other bodies orbiting them, like other planets or even moons?

Alice: I am not sure about exomoons in circumbinary systems. I see no reason why it wouldn't be possible, at least theoretically. Nobody has as of yet ever detected an exomoon, so there is no way to know for sure at the moment. But, in theory, what Prof. Starmover taught us suggests it should also be possible. You've reached the limits of my knowledge, I'm afraid.

Bob: Outstanding! This is amazing!

Alice: What the hell are you talking about?

Bob: Tatooine!

Alice's eyes open wide as she casts a deep smile, a devilish grin spreading across her face. Tapping one finger against her chin pensively, as if lost in a sea of possibilities, she responds:

Alice: Oh, yes. I see your point, and I see it well. This is indeed an exciting moment for us….

Bob and Alice: STAR WARS!

Bob: I'm pumped! We can test the stability of the Tatooine system, Luke Skywalker's original safe haven from the Empire! Luke Skywalker! *THE* Luke!

Alice: This is going to be epic!

Bob: Yeah!

Alice: Let's get down to it and figure out how we are going to make this happen. The integrator is done and ready to go. We have that covered. I guess that leaves deciding on the initial conditions?

Bob: Yeah, that makes sense to me. We will definitely need those. Do you know of any observed systems that meet our Tatooine-motivated criteria?

Alice: Not off the top of my head. Let's go ask Prof. Starmover!

11.2 Circumbinary Planets

Professor: You're back. How's the integrator coming along?

Alice: It's done! We've run a few basic quality control tests, and it has performed very well.

Bob: We think we're ready to take it out on the road, open that puppy up, and see what it can really do.

Alice: What Bob is trying to say is that we're looking for a project to apply our new integrator to. We have a few basic ideas we're pretty excited about, but we wanted to see if you could help guide us toward actually observed systems that approximately meet our chosen project criteria.

Professor: What have you decided on as your focus?

Alice: We'd like to tackle an observed *circumbinary* planet and explore the parameter space for stability. We are also considering including additional bodies, like more planets or even exomoons.

Professor: I like it! I know of a few systems that match that description. That I know of, there are now of order 10 circumbinary planets that have been reported in the literature. I will email you the specific references, which will give you the initial conditions for input to your new code.

Bob: That would be awesome!

Alice: Great, with that, we'll be ready to run some additional tests of our code, applied to actually observed circumbinary planets (CBPs).

Bob: Cool!

Professor: I know there has already been some theoretical work exploring the stability of circumbinary planets, but not much. In terms of exomoons, though, I think you are wide open. I can't think of any studies that have looked at that.

Bob and Alice: YES!

Alice: That settles it. Exomoons in CBPs it is!

Professor: I recommend first integrating the observed parameters of the CBPs to make sure they are stable on their own, before including an exomoon.

Bob: Good idea. That way, we can compare the evolution of the host CBP with and without the moon, to try to understand how the presence of a moon could change the evolution.

Professor: Exactly. But, if you happen to find that the observed parameters of any of the CBPs correspond to unstable orbital architectures, that could be an important result all on its own.

Alice: Would that suggest that there must be something wrong with the observations?

Professor: Either that, or the integrations you carry out could have a bug. But, if you have already tested your code rigorously, then it makes sense to ask if there could be something wrong with the reported observational data. A scenario such as this should then prompt you to consider *both* scenarios in more depth, to better understand its origins. But the first step toward a major discovery is often associated with unexpected features of applying a given analysis methodology.

Bob: I think that's it. We're ready to go!

Professor: There's one last thing. Actually, there are a few more things, but I am getting at one in particular. How many simulations do you intend to perform in total? You will likely have to somehow sample from a range of orbital inclinations, semimajor axes, and even eccentricities. In total, this might add up to thousands of simulations and could even exceed a million depending on the problem.

Bob: Uh … Alice will take care of this one. Alice?

Alice: I will?! Uh…. Well, honestly, it sounds like we're going to need a lot more laptops.

Bob: I only have the one!

Professor: Ha ha. Don't worry. I can grant you access to the department cluster.

Bob: Yes! Now I'm excited!

Alice: I don't get it. What's a "cluster"?

Professor: A computer cluster is effectively a set of connected computers that can more or less work together as a single operating system. Simply put, you can partition different simulations to different computers all working at the same time. So it is the next best thing to having a whole lot more laptops.

Alice: Oh, I get it! I think….

Professor: Clusters use a set of nodes, which are effectively equivalent to individual laptops, to all perform the same basic task, controlled by the same software. In your case, the task is running your N-body integrator for different sets of initial conditions.

Alice: Okay, I definitely get it. It's actually pretty simple.

Bob: Yep, but also super cool!

Professor: I will contact the system administrator for our cluster and get him to grant you access. This will require a username and password, which will be provided to you. Look out for an email from the administrator, which will tell you how and when you can officially gain access to the cluster and start running simulations in parallel!

Alice and Bob return to their workspace, excited at the prospect of running their simulations on the department cluster.

Bob: Okay, so we know where to get the initial conditions now, courtesy of Prof. Starmover. Not to mention we are getting access to the department cluster!

Alice: Right. We're almost ready to implement the initial conditions and do some test runs.

Bob: Let's do it!

12 Setting Up the Project

Overview. Alice and Bob set up the simulations of moons around circumbinary planets with their ABIE code. They run some simulations to test the configuration. After this, they find out how to run the simulations on a large computer cluster.

12.1 Initial Conditions

12.1.1 The Circumbinary Planetary System

Bob: Let's set up our simulations for our moons-around-circumbinary planets project!

Alice: Yes, let's get started! First, we need to know the parameters of the *Kepler* circumbinary planets.

Bob: Prof. Starmover emailed us the relevant citations from the scientific literature. For now, let's just go to Wikipedia and look them up. It will be faster, and we can cross-check the values against the citations from Prof. Starmover later.

Alice: Wait a minute! Wikipedia is a very nice resource to quickly look up facts, but it is not always reliable. This is, I guess, the downside; everyone can edit it, so you never know if the content is accurate (although, usually, vandalism is quickly spotted by the community and fixed by restoring to a previous version of the webpage). Astronomy-related articles are usually not very controversial so should be fine most of the time, but let's stick to the peer-reviewed discovery papers.

Bob: Okay, fair enough. Let's focus on one system first.

Alice: Sure, let's take Kepler-16. I have pulled up the discovery paper here, Doyle et al. (2011).

Bob: Wow, a *Science* paper. Isn't that journal supposed to be only for very special discoveries?

Alice: Yes, but in practice it doesn't always turn out that way. In this case, though, Kepler-16 was the first transiting circumbinary planet, so a big deal back in 2011!

Bob: They have a nice table here, their table 1. It lists all their fitted parameters.

Alice: Yep. The stellar binary has masses $M_1 = 0.6897\,\mathrm{M}_\odot$, $M_2 = 0.20255\,\mathrm{M}_\odot$, and the planet has a mass of $M_\mathrm{p} = 0.333\,M_\mathrm{J}$. The semimajor axis of the stellar binary is $a_\mathrm{bin} = 0.22431$ AU, and the semimajor axis of the planet around the binary is $a_\mathrm{p} = 0.7048$ AU. The eccentricities are $e_\mathrm{bin} = 0.15944$ and $e_\mathrm{p} = 0.0069$, respectively.

Bob: You are giving a lot of significant figures there, Alice. Can we really trust all those?

Alice: Well, maybe not the last significant figure or two, but the parameters are generally known very precisely, especially for astronomical standards. I think the general reason is that the complex transit geometry of the circumbinary system results in very specific timing effects that contain a lot of information.

Bob: Fair enough. I see that the orbital orientations are also given in terms of the inclination, argument of periapsis, and longitude of the ascending node. But with respect to which plane are these orbital elements valid?

Alice: If I remember right, in observational papers, the orbital elements are usually defined with respect to the plane of the sky. That is also clearly the case here, since the inclinations are close to 90°, that is, we see the orbits edge-on, which you would expect for a transiting system. In other words, if the orbits weren't close to being edge-on, we wouldn't be able to observe transits.

Bob: I see. The inclinations listed are $i_\mathrm{bin} = 90.3401°$ and $i_\mathrm{p} = 90.0322°$. So the mutual inclination between the orbit of the binary and the orbit of the planet is $i_\mathrm{bin} - i_\mathrm{p} = 0.3079°$?

Alice: Actually, I think that this is not necessarily the case. The two orbits could be mutually rotated with respect to each other, which is quantified by the longitudes of the ascending nodes. This means that you cannot simply add or subtract the inclination angles. Remember that Prof. Starmover told us about a formula to calculate the mutual inclination between two orbits?

Bob: Oh yeah, you're right!

Alice: Let me look at my notes. Ah, here it is (equation 2.60):

$$\cos(i_{12}) = \cos(i_1)\cos(i_2) + \sin(i_1)\sin(i_2)\cos(\Omega_1 - \Omega_2). \tag{12.1}$$

Here, 1 and 2 refer to the two orbital planes, i_{12} is the mutual inclination, and Ω_i is the longitude of the ascending node of orbit i. If $\Omega_1 = \Omega_2$, that is, if the orbits have the same nodal rotation, then

$$\cos(i_{12}) = \cos(i_1)\cos(i_2) + \sin(i_1)\sin(i_2) = \cos(i_1 - i_2), \tag{12.2}$$

where the last step is just a trigonometric identity. So, in this case, your value of the mutual inclination would be correct. Looking at the discovery paper, the binary has a longitude of the ascending node defined as zero, that is, $\Omega_\mathrm{bin} = 0$, and the planet has $\Omega_\mathrm{p} = 0.003°$, which is very close to, but not exactly, zero.

Bob: Okay, let's plug in the numbers. I get $i_{\mathrm{bin,p}} = 0.307915°$. Hmmm … this is almost the same as $i_{\mathrm{bin}} - i_{\mathrm{p}} = 0.3079°$. But I understand that this could be quite different if the difference in the nodal angles were not so small, so let us keep equation (12.1) in mind.

Alice: Right. Note that the mutual inclination is very small. This makes sense, since otherwise, the binary and planet would not be transiting each other as viewed from Earth, or space in the case of *Kepler*.

Bob: Let's now put this system into our ABIE integrator. This should be very easy using our orbital element routines that we have already implemented.

Bob opens a new empty file and starts typing.

```
sim = ABIE()
sim.integrator = "GaussRadau15"
sim.integrator.acceleration_method = "ctypes"
sim.CONST_G = 4*np.pi**2
sim.t_end = t_end
sim.store_dt = t_end/N_steps

sim.add(mass=M_1,x=0.0, y=0.0, z=0.0, vx=0.0, vy=0.0, vz=0.0, name="
    primary_star", radius = R_1)
sim.add(mass=M_2, a=a_bin, e=e_bin, i=i_bin, omega=omega_bin, Omega=
    Omega_bin, f=f_bin, name="secondary_star", primary="primary_star",
    radius = R_2)
sim.particles["primary_star"].primary = "secondary_star"

sim.add(mass=M_p, a=a_p, e=e_p, i=i_p, omega=omega_p, Omega=Omega_p, f=f_p
    , name="planet", primary=["primary_star","secondary_star"], radius =
    R_p)
```

We are putting the primary star at the origin, and we define the secondary star with respect to the primary using the binary orbital elements. We then need to specify the "primary" of the primary star after adding the secondary star for the orbital elements of the primary star to be correctly calculated, with `sim.particles["primary_star"].primary = "secondary_star"`. We cannot yet do this when adding the primary star, since at that time, the secondary star has not yet been defined. Finally, we add the circumbinary planet, which has a primary that is the center of mass of the binary. This is specified by providing a list as the primary object of the planet, and this list contains the names of the objects used to define the center of mass position. This approach is also consistent with the orbital elements that we retrieved from the Kepler-16 discovery paper, which are Jacobian orbital elements.

Alice: This is definitely an elegant way to set up the system. We only need to know the hierarchy of the system and the orbital elements. We don't need to worry about the details of converting the orbital elements to Cartesian coordinates, and the output files also automatically include the corresponding orbital elements as a function of time.

Bob: Of course, we need to supply the gravitational constant. Here, I chose it to be $G = 4\pi^2$, which implies that the unit of time is one year (for example, the orbital period of the Earth in these units with length in AU and mass in solar masses is $P = 2\pi\sqrt{a_\oplus^3/(GM_\odot)} = 2\pi\sqrt{1^3/(4\pi^2\times 1)} = 1$). We should also define the integration time, `t_end`, and the number of output steps, `N_steps`. What do you think is a reasonable integration time?

Alice: I think we should scale `t_end` to the longest dynamical time scale of the system, which is the orbital period of the planet around the binary. So let us set

$$t_{\rm end} = N_{\rm int} P_{\rm p} = N_{\rm int}\, 2\pi \sqrt{\frac{a_{\rm p}^3}{G(M_{\rm bin}+M_{\rm p})}} \simeq 0.626\, N_{\rm int}\ {\rm yr}. \tag{12.3}$$

Here, I substituted the parameters of the Kepler-16 system. The binary period, by the way, is $P_{\rm bin} \simeq 0.11\,{\rm yr}$. Let's take $N_{\rm int}$ to be a large number, but not too large; otherwise, we will be waiting forever for the simulations to finish!

Bob: I guess we need to make a compromise. Maybe take $N_{\rm int} = 1{,}000$, and check if integrating for shorter times gives different results?

Alice: Sounds good to me!

12.1.2 Adding the Moon

12.1.2.1 The lunar mass

Bob: The properties of our target circumbinary planetary system are well constrained by observations. However, when it comes to the moon around the planet, there are many more free parameters. First of all, we need to specify the lunar mass.

Alice: Well, as far as I know, there are no confirmed exomoon detections to date, just candidates. This means that we do not have any actual observational constraints for the exomoons we simulate. So maybe we should take the Solar System as an inspiration?

Bob: Okay. Some well-known examples are Earth–Moon, Io–Jupiter, and Pluto–Charon. Let's look those up.

Bob opens up a web browser to Wikipedia. Alice looks at Bob, an exaggerated frown stretching across her face.

Bob: What? We want to get an idea of the masses of the objects in the Solar System, right? Wikipedia should be fine for that.

Alice: Okay, I grant you that.

Bob: I'll do a more detailed search after to find a better resource to cite. Ah, we have $q_{\rm m} \simeq 0.012$ for the Earth–Moon system, $q_{\rm m} \simeq 4.7\times 10^{-5}$ for Io–Jupiter, and $q_{\rm m} = 0.12$ for Pluto–Charon, where $q_{\rm m} = M_{\rm m}/M_{\rm p}$, and $M_{\rm m}$ is the mass of the moon.

Alice: That is a large range of mass ratios. Why don't we take two extreme values, $q_{\rm m} = 10^{-3}$ and $q_{\rm m} = 10^{-1}$? If there turns out to be a strong dependence of our results on the mass ratio, then we can always decide to sample more values.

Bob: Okay. Next up is the orbit of the moon around the planet. I don't think we have the freedom to make the orbit of the moon arbitrarily wide. There will be some critical separation at which the moon is no longer bound to the planet, but instead bound to the host stars, or even not bound to the massive bodies at all. Isn't that right?

Alice: Right. I recall from my dynamics class that there is this concept of the Hill radius or Hill sphere, but I forget the details. I suggest we ask Prof. Starmover.

Bob: Okay, but let's keep it short and not get lost in random details. You have a tendency to divert Prof. Starmover's attention from the goal….

Alice: Excuse me!? Please. First of all, it takes more than one person to divert that woman's train of thought from wherever she wants it go. Second, you're just as guilty as I am!

Bob: All right, fine! I admit it. I ask random tangential questions. But she says such fascinating stuff!

Alice: I know. It's a real burden, let me tell you. Let's agree to share the burden, and both try not to occupy too much of her time.

Bob: All right, I'm in.

12.1.2.2 The Hill radius

Alice and Bob enter Prof. Starmover's office.

Alice: Hello, Prof. Starmover. We were somewhere in the middle of setting up our simulations and decided the concept of the Hill radius might be useful for defining the initial conditions of the moon around the planet. Could you tell us more about this, assuming we are on the right track?

Professor: Sure, sounds like you are! The Hill radius, $r_{\rm H}$, decides how large the radius of influence is of a small body orbiting a more massive body. If a test particle (i.e., something with negligible mass) orbits around the small body, then it can be considered bound to that object if its orbital distance is less than $r_{\rm H}$. The canonical expression for $r_{\rm H}$ is

$$r_{\rm H} = r\left(\frac{m}{3M}\right)^{1/3}, \tag{12.4}$$

where r is the orbital distance of the small body (mass m) around the more massive body (M). Here, we need to assume $M \gg m$ and circular orbits. So $r_{\rm H} \ll r$. Also, note that in your case, the "more massive body" is in fact a binary, but let us for the moment forget

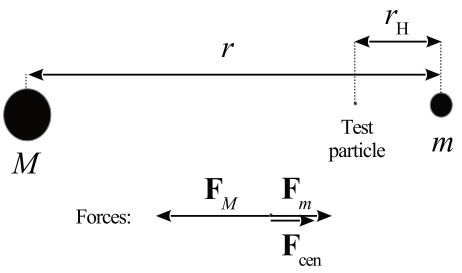


Figure 12.1
Derivation of the Hill radius.

about this subtlety. In your case, we have

$$r_H = a_p(1-e_p)\left(\frac{M_p}{3(M_1+M_2)}\right)^{1/3}. \tag{12.5}$$

Alice: I understand why you set the mass of the small body to M_p and that of the massive body to $M_1 + M_2$, but why did you replace r with $a_p(1-e_p)$?

Professor: Admittedly, this is a bit of an ad hoc replacement. As I mentioned, equation (12.4) only strictly applies if the orbits are circular. In your case, the orbit of the circumbinary planet around the binary can have any eccentricity, so to take this into account, I simply replaced r with the shortest distance of the small body to the massive body, which is the periapsis distance $a_p(1-e_p)$. The idea is that perturbations from the massive body are largest at periapsis, so this will set the Hill radius. Nevertheless, be aware that equation (12.5) is approximate, even in the case when the stellar binary is a point mass. So I would not be surprised if it only works to within a factor of a few.

Alice: I see. Can you tell us how to derive equation (12.4)?

Prof. Starmover draws on the blackboard (figure 12.1).

Professor: Let there be a small body m, separated from the massive body M at a distance of r on a circular orbit. The test particle is orbiting the small body, also on a circular orbit, and we consider the point in time when the test particle is closest to the massive body. All we need to do is consider the balance of forces and take into account that we are in the frame of reference corotating with the small body around the massive body. This frame is rotating with an angular velocity $\Omega = \sqrt{GM/r^3}$, that is, the Keplerian angular velocity (recall that I assumed that $M \gg m$).

The massive body attracts the test particle with a force $\mathbf{F}_M$, and the small body with a force $\mathbf{F}_m$, in opposite directions. From Newton's law of gravitation, we have $F_M = GMm_{test}/(r-$

$r_{\rm H})^2$, and $F_m = Gmm_{\rm test}/r_{\rm H}^2$. Here, $m_{\rm test}$ is the test particle's mass, which, as we will see, cancels out.

Also, there is the centrifugal force acting on the test particle in the direction toward m. As you may recall from mechanics class, in a rotating frame, there is a force called the centrifugal force. This is not a "true" force but an apparent force that arises from the fact that a rotating frame is not inertial. Basically, objects in inertial frames with zero net force acting on them move in straight lines with constant velocity (Newton's first law), but from the point of view of a rotating frame, this appears as though the object is being flung out, just like laundry in a centrifuge will stick to the walls.

The magnitude of the centrifugal force is given by $F_{\rm cen} = m_{\rm test}\Omega^2(r - r_{\rm H})$. So the force balance reads

$$-\frac{GMm_{\rm test}}{(r-r_{\rm H})^2} + \frac{Gmm_{\rm test}}{r_{\rm H}^2} + m_{\rm test}\frac{GM}{r^3}(r - r_{\rm H}) = 0. \tag{12.6}$$

The test particle's mass $m_{\rm test}$ drops out, as does G. Rewriting the terms a bit, we have

$$-\frac{M}{r^2}\left(1 - \frac{r_{\rm H}}{r}\right)^{-2} + \frac{m}{r_{\rm H}^2} + \frac{M}{r^2}\left(1 - \frac{r_{\rm H}}{r}\right) = 0. \tag{12.7}$$

Next, we use a Taylor expansion to simplify the first term, that is, $(1 + x)^\alpha \simeq 1 + \alpha x$ if $x \ll 1$, and here $x = r_{\rm H}/r$. So,

$$-\frac{M}{r^2}\left(1 + 2\frac{r_{\rm H}}{r}\right) + \frac{m}{r_{\rm H}^2} + \frac{M}{r^2}\left(1 - \frac{r_{\rm H}}{r}\right) = \frac{m}{r_{\rm H}^2} - 3\frac{r_{\rm H}}{r}\frac{M}{r^2} \simeq 0, \tag{12.8}$$

which is easily solved for $r_{\rm H}$ and yields equation (12.4).

Alice: Nice, a simple derivation! Thank you, Prof. Starmover.

Bob: Let's get back to coding and setting up the simulations….

12.1.2.3 The other parameters

Alice and Bob leave Prof. Starmover's office and return to their workspace.

Bob: Let's add the moon to the code in our script.

```
sim.add(mass=M_m, a=a_m, e=e_m, i=i_m, omega=omega_m, Omega=Omega_m,
    f=f_m, name="moon", primary="planet", radius=R_m)
```

All we need to do is specify the orbital elements and the primary body, which is the planet.

Alice: So far, we have decided on the mass $M_{\rm m}$ and that the semimajor axis $a_{\rm m}$ should be of the order of the Hill radius. For Kepler-16, I get $r_{\rm H} \simeq 0.034$ AU. We still need to specify the eccentricity $e_{\rm m}$, inclination $i_{\rm m}$, argument of periapsis $\omega_{\rm m}$, longitude of the ascending node $\Omega_{\rm m}$, and the orbital phase (true anomaly $\theta_{\rm m}$). I think we can set the initial eccentricity to zero (or, at least, a tiny value like 10^{-10} to avoid any potential numerical issues with orbital element conversion) and try different values later to see if the initial eccentricity

matters. A zero eccentricity is also convenient because the dependence on the argument of periapsis then goes away.

Bob: So we are left with the inclination, longitude of the ascending node, and orbital phase.

Alice: For the orbital phase, I think we should just take a range of values, say, fixed ones. For $i_{\rm m}$ and $\Omega_{\rm m}$, they combined determine the relative orientation of the moon with respect to the orbit of the planet around the binary.

Bob: Right, this is accomplished via equation (12.1).

Alice: What we could do is set the difference between the longitudes of the ascending nodes of the planet and the moon to be π, that is, $\Omega_{\rm p} - \Omega_{\rm m} = \pi$, such that equation (12.1) reduces to

$$\cos(i_{\rm pm}) = \cos(i_{\rm p})\cos(i_{\rm m}) - \sin(i_{\rm p})\sin(i_{\rm m}) = \cos(i_{\rm p} + i_{\rm m}), \tag{12.9}$$

that is, we choose the nodal orientation of the lunar orbit relative to the planetary orbit such that the angular momentum vectors lie in the same plane, and we can simply add the individual inclinations to get the mutual inclination. So we can use this to get the mutual inclination from the initial inclinations $i_{\rm p}$ and $i_{\rm m}$.

Bob: So what you are saying is we take $\Omega_{\rm m}$ to be $\Omega_{\rm m} = \pi + \Omega_{\rm p}$, and take a range of mutual inclinations $i_{\rm pm}$, which translates into a range of initial values for $i_{\rm m}$?

Alice: Right. The mutual inclination $i_{\rm pm}$ can range between 0 and π radians. Zero mutual inclination means that the moon and planet are in coplanar orbits, and the sense of rotation is in the same direction (also known as prograde orientation). If $i_{\rm pm} = \pi/2$, then the orbits are orthogonal with respect to each other. A mutual inclination of π means that the orbits are again coplanar, but their sense of rotation is in the opposite direction (also known as retrograde orientation).

Bob: That sounded suspiciously like gibberish to me, but I'll take your word for it. I hate to say it, but you know your stuff, Alice.

Alice: Wow! How painful was that to say aloud?

Bob: Very.

Alice: I appreciate the effort.

Bob: How gracious of you….

Alice punches Bob in the shoulder amiably.

12.2 The First Simulation

12.2.1 Trial and Error

Bob: Prof. Starmover argued that the critical separation for stability is expected to be of the order of the Hill radius, which you calculated to be $r_{\rm H} \simeq 0.034$ AU. Let's set $a_{\rm m} = 2r_{\rm H}$ for now, and see what happens!

Alice: Sure!

Bob runs his script and makes plots of the orbital elements (figure 12.2).

Bob: This looks weird. The semimajor axis of the moon quickly becomes negative, and the eccentricity becomes enormous. The periapsis distance, $r_{\rm p,m} = a_{\rm m}(1 - e_{\rm m})$, oscillates.

Alice: Well, at least the binary and planet are doing what they are supposed to: their orbital elements are constant as a function of time and reflect the initial conditions. Maybe we should plot the Cartesian coordinates of the objects to see what is going on?

Bob: Okay. I have done that in the (x, z)-plane in figure 12.3. We can see the stars orbiting around each other and the planet orbiting around the stars. The moon is initially trying to follow the planet, but this quickly goes awry, and the moon flies away as shown in the right-most panel of figure 12.3.

Alice: Right. So the moon becomes unbound from the planet and apparently even from the binary. Now that I think about it, it makes sense that the orbital elements behave oddly: a negative semimajor axis means a positive energy—the moon is unbound from the planet. And the eccentricity is larger than unity, indicating that the orbit is hyperbolic, that is, unbound.

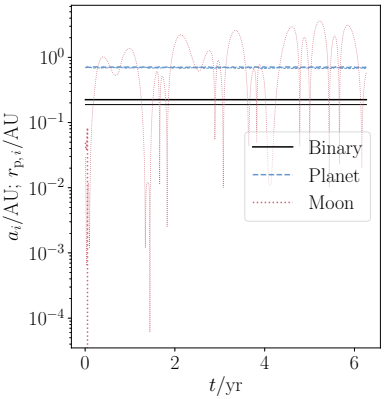
Bob: I see. So, apparently, the moon is unstable at $2r_{\rm H}$. One thing I don't quite understand is why the entire system seems to be moving in one particular direction.

Alice: Hmm. Well, it does not seem to affect the orbital elements of the binary and the planet; they are fine as shown in figure 12.2.

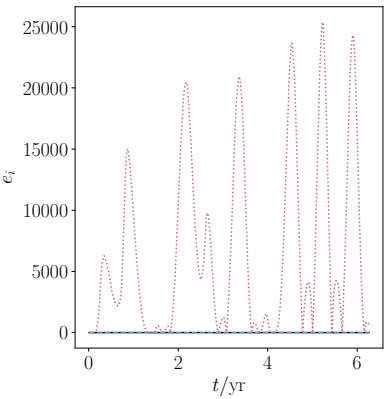
Bob: Could it be related to the center of mass? I remember hearing from Prof. Starmover that the center of mass moves at a constant velocity.

Alice: You may be onto something. The center of mass velocity does not necessarily have to be zero. In our case, we have set the primary star initially at the origin with zero velocity, and the secondary star initially has some position and velocity relative to the origin according to the orbital elements that we supplied. Let us write down the center of mass velocity,

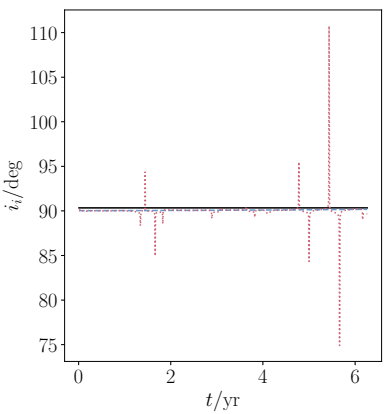
$$\mathbf{v}_{\rm CM} = \frac{M_1\mathbf{V}_1 + M_2\mathbf{V}_2 + M_{\rm p}\mathbf{V}_{\rm p} + M_{\rm m}\mathbf{V}_{\rm m}}{M_1 + M_2 + M_{\rm p} + M_{\rm m}}. \tag{12.10}$$



(a) Semimajor axes a_i (thicker lines) and periapsis distances $r_{\mathrm{p},i} = a_i(1 - e_i)$ (thinner lines)

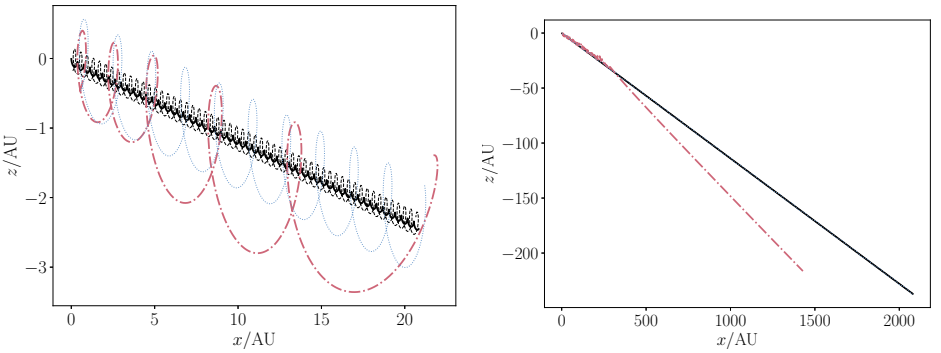


(b) Eccentricities e_i



(c) Inclinations i_i

Figure 12.2
Orbital elements as a function of time for the Kepler-16 system with $a_{\mathrm{m}} = 2r_{\mathrm{H}} \simeq 0.068$ AU and $i_{\mathrm{pm}} = 0°$. Black, blue, and red lines correspond to the binary, planet, and moon, respectively. The moon quickly becomes unbound.



(a) After 10 orbits of the planet around the binary **(b)** After 1,000 orbits of the planet around the binary

Figure 12.3
The stars (black lines), planet (blue line), and the moon (red line) in the (x, z)-plane for Kepler-16, using the same parameters as in figure 12.2. Left (right)-hand panel: after 10 (1,000) orbits of the planet around the binary.

Bob: Okay, so we have $\mathbf{V}_1 = \mathbf{0}$. Also, I think we can neglect the contribution from the planet and the moon given their small masses compared to the stars. This gives us

$$\mathbf{v}_{\rm CM} \simeq \frac{M_2}{M_1 + M_2}\mathbf{V}_2, \tag{12.11}$$

and this should be approximately constant. If I print `sim.particles[1]`, I can easily get the initial value of $\mathbf{V}_2$ from ABIE:

```
print(sim.particles[1])
Particle(m=0.20255, x=-0.0214617, y=0.0011119, z=-0.187317, vx=14.622, vy
    =0.00994414, vz=-1.67524, r=0, name='secondary_star', hash=503495493)
```

Alice: If we take into account the mass ratio $M_2/(M_1 + M_2)$, this means that

$$\mathbf{v}_{\rm CM} \simeq \frac{0.20255}{0.6897 + 0.20255}(14.6\,\hat{\mathbf{x}} - 1.68\,\hat{\mathbf{z}})\ {\rm AU/yr} \simeq (3.3\,\hat{\mathbf{x}} - 0.38\,\hat{\mathbf{z}})\ {\rm AU/yr}, \tag{12.12}$$

neglecting the component of the velocity in the y-direction.

Bob: So the center of mass is moving in the positive x- and negative z-direction, which is consistent with figure 12.3.

Alice: I think we can be even more quantitative. In the right-most panel of figure 12.3, we are integrating for $1{,}000\,P_{\rm p}$, right?

Bob: Yes, so that's $\Delta t \simeq 626$ yr.

Alice: So the center of mass will have traversed

$$\Delta\mathbf{r}_{\rm CM} = \mathbf{v}_{\rm CM}\Delta t \simeq (2.0\times 10^3\,\hat{\mathbf{x}} - 2.4\times 10^2\,\hat{\mathbf{z}})\,{\rm AU}. \tag{12.13}$$

Bob: That is indeed approximately the displacement of the center of mass shown in the right-most panel of figure 12.3!

Alice: Looks like we understand this!

12.2.2 Deciding on the Grid Parameters of the Moon

Bob: Let's try a smaller initial separation for the orbit of the moon about the planet.

Alice: Sounds good. How about $a_{\rm m} = r_{\rm H} \simeq 0.034$ AU?

Bob: Okay. I have plotted the results in figure 12.4.

Alice: Still not stable.

Bob: Yes, but it is bound for a short amount of time, during the first ≈ 50 yr. So if we reduce $a_{\rm m}$ even further, it should at some point become stable.

Alice: We'll see. Let's try $a_{\rm m} = 0.01$ AU.

Bob: Okay, see figure 12.5.

Alice: Finally! The moon now remains stable over 1,000 orbits of the planet.

Bob: The eccentricity of the moon is oscillating slightly, which I guess is fine since there are still perturbations from the binary. But the planet's eccentricity is also oscillating.

Alice: I think that is fine, too. I read that most of the *Kepler* circumbinary planets are close to being dynamically unstable, so some oscillation in the eccentricity is to be expected. Note that there is also some oscillation in the semimajor axis of the planet, although the amplitude is small and the average value remains constant.

Bob: Just for fun, I have also plotted the trajectories in figure 12.6.

Alice: Looks good! The moon keeps following the circumbinary planet, consistent with the information that we got from the orbital elements.

Bob: Let me try $a_{\rm m} = 0.02$ AU. Hmm, again unstable.

Alice: Given all that we have just learned, I propose we decide on the range 0.005–0.02 AU for $a_{\rm m}$ when we do a more detailed and systematic study with a large number of simulations.

Bob: Sure. Makes sense to me. We can always adjust the ranges later if need be.

12.3 Running on a Cluster

Alice: So, to summarize, for the Kepler-16 system, we have decided to consider for now semimajor axes $a_{\rm m}$ in the range 0.005–0.02 AU and inclinations $i_{\rm pm}$ between 0 and π radians. For each combination of $a_{\rm m}$ and $i_{\rm pm}$, we take a range of initial orbital phases of the moon around the planet.

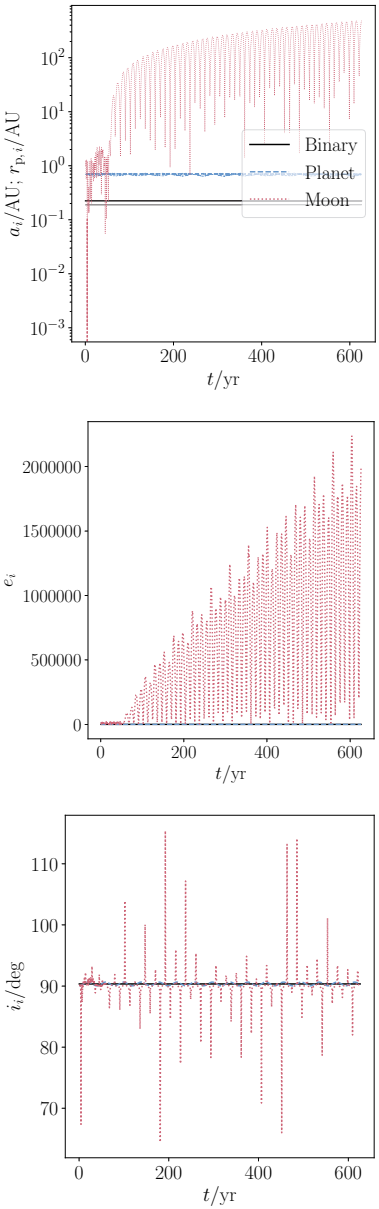


Figure 12.4
Orbital elements as a function of time for the Kepler-16 system with $a_{\mathrm{m}} = r_{\mathrm{H}} \simeq 0.034$ AU and $i_{\mathrm{pm}} = 0°$. The moon still becomes unbound, although it takes a little longer.

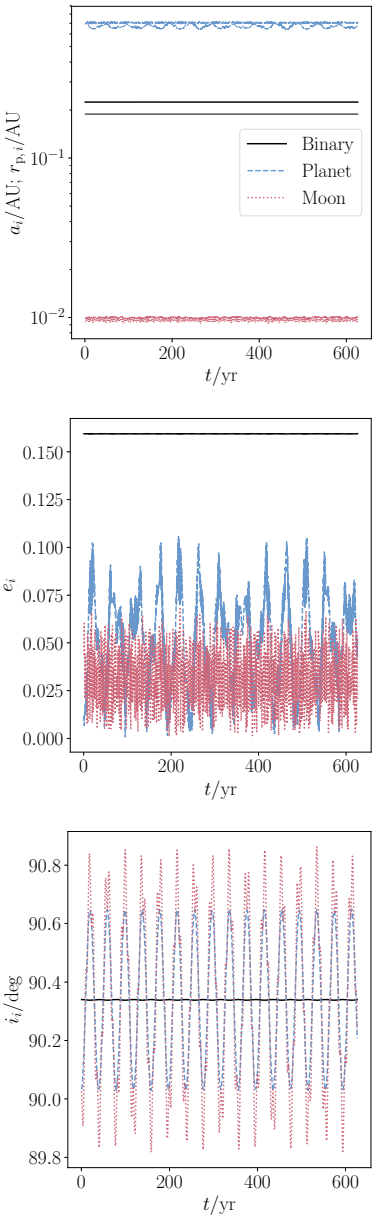


Figure 12.5
Orbital elements as a function of time for the Kepler-16 system with $a_{\mathrm{m}} = 0.01$ AU and $i_{\mathrm{pm}} = 0°$. The moon now remains bound.

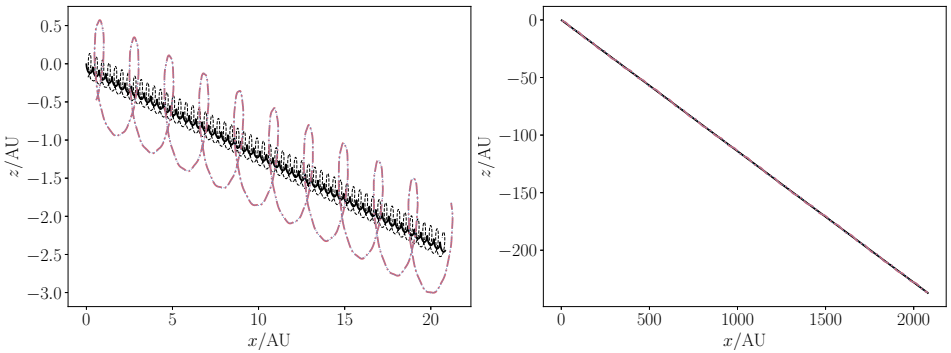


Figure 12.6
The stars (black lines), planet (blue line), and the moon (red line) in the (x, z)-plane for Kepler-16 and the same parameters as in figure 12.5, that is, $a_{\rm m} = 0.01$ AU and $i_{\rm pm} = 0°$. Left (right)-hand panel: after 10 (1,000) orbits of the planet around the binary.

Bob: If we took, say, forty values of $a_{\rm m}$, forty for $i_{\rm pm}$, and ten different fixed phases (evenly spaced mean anomaly between 0 and 2π), this would come out to 16,000 simulations in total. That's quite a lot!

Alice: I guess we could run them on our laptops, but it would take a while. Why not run the simulations on the cluster that Prof. Starmover offered?

Bob: Sure. Let's go to Prof. Starmover's office and ask how we should proceed.

Alice and Bob head over to Prof. Starmover's office. Bob barges in without knocking.

Bob: Prof. Starmover, you offered us access to a cluster on which to do our simulations. Could you help us get started?

Alice: Jeez, Bob, try knocking first. Apologies, Prof. Starmover. Bob is excited. We're pretty much ready to perform larger numbers of simulations on the cluster.

Professor: Outstanding! Very nice work, you two. You have been given an SSH account that you can use to log on to the cluster. Once you do, you can use `scp` or `rsync` to copy the ABIE code to the cluster and run it there. But only for testing purposes! We have a job scheduler on the cluster to make sure that the cluster's resources are allocated fairly. You do not want to overload the login node, since this could cause problems with the job scheduler and, potentially, cause the entire cluster to crash (believe me, I know…).

Alice: How do we use the job scheduler?

Professor: Generally, there are a number of software packages commonly used on clusters, such as Moab and Slurm. Our cluster uses the latter, so I will focus on Slurm, although usage is generally the same for these packages. Basically, all you need to do is to write a script that specifies what kind of resources you would like to use, which libraries,

and, of course, the actual execution of your program. Let me give a bare minimum example of a SLURM job submission script.

```
#!/bin/bash
#SBATCH --job-name=kepler16
#SBATCH --output=kepler16-%j.out
#SBATCH --error=kepler16-%j.err
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --time=00:10:00

cd $MYPATH
./python kepler16.py
```

Note that all SLURM-related directives are indicated with the `#SBATCH` command. First, the job name is used to identify the job, for example, when you list the current queue of the scheduler. I specified the name of the output and error files; `%j` is the job id that is assigned once you submit your job. The number of nodes is specified with the `--nodes` flag and the number of threads per node with `--ntasks-per-node`. In this example, we are running just one system, so we are requesting one thread on one node. Also, you need to specify the job runtime with `--time`. In this case, I requested ten minutes, which should be more than enough for just one system. Note that if your job is still running after the allotted time, then it will be killed, so make sure you reserve enough time. On the other hand, do not overestimate the runtime too much, since a job with a long requested runtime will be given lower priority. Also, clusters may have policies on the allowed maximum runtime, often depending on the number of nodes or cores requested. For example, you might be allowed to use a single core indefinitely, but if you are asking for, say, 70 percent of the cluster's resources, then there likely is a maximum time (say, twenty-four hours). Lastly, I `cd` to the path where the PYTHON script resides (you will have to specify this to your situation, of course), and then I just run it.

To submit your job script, simply type `sbatch script.sh`, where `script.sh` is the name of the SLURM job submission script. You will then get a message with the identification number of the job. To see the current status of the cluster, type `squeue`, and you will get a table with an overview of the jobs that are currently running, queued, or on hold. If there are a lot of jobs, you can restrict the list to include only your own jobs by adding to `squeue` the command line option `-u MYNAME`, where `MYNAME` is your user name.

Alice: Sounds simple enough! But in our case, we have many systems to run—would we really have to submit separate job scripts for each system (implying 16,000 jobs)?

Professor: In principle, you could, but I would definitely not recommend it. I am not sure how well the job scheduler would handle it (you risk crashing the system); moreover, there could be a limit on the number of submitted/active jobs per user. There are several options; one of them is to use MPI through the MPI4PY package for PYTHON. MPI stands for message passing interface and is a standard used for parallel computing. In your case,

the problem is "embarrassingly parallel," that is, all the systems can be run independently of each other and there is no communication between them. It is therefore very easy to set up. Let me give an example of PYTHON pseudocode (i.e., you will have to fill in some details yourself).

```
from mpi4py import MPI
import math

simulations = ...

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
N_threads = comm.Get_size()
N_jobs = len(simulations)
N_jobs_per_thread = int(math.ceil( float(N_jobs)/float(N_threads) ))

indices = range(rank,N_jobs,N_threads)

for index_rank,index_sim in enumerate(indices):
    sim = simulations[index_sim]
    sim.initialize()
    sim.integrate()
```

To run this script with MPI, you can use the `mpiexec` or `srun` command, and you need to specify the number of processes; the script will be executed by each thread separately. The identification of the current thread, that is, its rank, is retrieved with `comm.Get_rank()`. `simulations` is just a PYTHON list of instances of ABIE that have been set up previously. So, in your case, `simulations` would contain 16,000 instances of ABIE simulations, with (slightly) varying initial conditions, that you need to set up somewhere. I retrieve the number of threads available to MPI using `comm.Get_size()` and calculate the number of jobs per thread (of course, I need to round this number to an integer). Then, I determine the indices of the systems in `simulations` that the current thread is required to run by generating an appropriate sequence with `range()`. Finally, all the systems for this particular thread are run within a for loop.

So, if this script is called `run_grid.py`, you would run it in parallel on four cores with `mpiexec -n 4 run_grid.py`, and you should modify the corresponding line in the submission script (i.e., instead of `./python kepler16.py`). By the way, four cores would be appropriate for a quadcore processor, but on a cluster, you would typically have many more threads available. On our cluster, in particular, there are 28 cores available per node, so if you requested ten nodes, you could use 280 cores.

Alice: Wow, that should really help to speed up the calculations!

Bob: No kidding. Can I set it up and run the simulations on the cluster?

Alice: Of course, indulge yourself! Let me know when I can help.

13 Running and Analyzing the Simulations

Overview. Alice and Bob encounter unexpected behavior in their simulations if the moon and the planet are initially highly mutually inclined. They speak with Prof. Starmover, who tells them about Lidov–Kozai oscillations. Alice and Bob then realize that they should check for collisions in their simulations. After rerunning their simulations, they create stability maps in the $(a_{\mathrm{m}}, i_{\mathrm{pm}})$ plane and Prof. Starmover further helps with the interpretation, telling them about mean-motion resonances.

13.1 Problems with the Simulations

Bob: I ran the Kepler-16 simulations on the cluster last night, using both the Wisdom–Holman and the Gauss–Radau integrators.

Alice: Great! What are the results?

Bob: First, I need to talk about some issues that I encountered. I noticed that in the simulations with i_{pm} close to 90° (i.e., high inclinations of the moon with respect to the planet), the relative energy errors became relatively large, larger than 10^{-10}. This was especially the case with the Wisdom–Holman integrator. In some cases with the Gauss–Radau integrator, the code even completely stopped—I got the error message that the time steps are too small, and the code stalled. Increasing the precision of the code by reducing the `epsb` parameter for the Gauss–Radau integrator and using `long double` instead of `double` in the code did improve the energy errors, but still the code stalled if i_{pm} was close enough to 90°.

Alice: That does sounds very weird.

Bob: I also noticed that the eccentricity of the moon with respect to the planet became very high, that is, the moon was approaching the planet at a very close distance. I suspect that this is why the Wisdom–Holman integrator in particular did not perform well since it uses constant time steps. This means that if the eccentricity becomes very high, then the time step can no longer be short enough to accurately resolve the periapsis passage. The Gauss–Radau integrator, on the other hand, uses adaptive time steps. It still encounters problems, however, when the eccentricity becomes so high that the required time step is smaller than machine precision.

Alice: I wonder if these high eccentricities are physical or whether there is a problem with our integrator. If it is physical, we should maybe think about avoiding the time-step issues by, for example, checking for physical collisions between the planet and the moon.

Bob: Do you know what is the best way to do that?

Alice: Not really, but Prof. Starmover definitely would!

Bob jumps up from his seat.

Bob: To the professor!

13.1.1 Lidov–Kozai Oscillations

A bit disgruntled, Alice and Bob enter Prof. Starmover's office.

Alice: Prof. Starmover, we are having issues with our integrations. At high inclinations between the moon and the planet, the energy errors become relatively large, and if we choose $i_{\rm pm}$ very close to 90°, our code even stalls due to time steps that are too small. We were wondering if this could be a physical effect or just a problem with our integrator.

Professor: First of all, whenever you encounter a problem with energy errors or time steps, you should try to increase the precision of the code. Or, try a different integrator.

Bob: We have tried all that, but the code still has major issues at high inclinations.

Professor: Well, can you show me an example when $i_{\rm pm}$ is high?

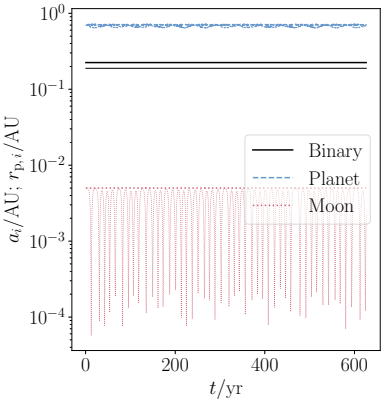
Bob: Sure. In figure 13.1, I show the orbital elements for an integration with $a_{\rm m}$ = 0.005 AU and $i_{\rm pm} = 81°$. Note that the eccentricity becomes very high; the periapsis distance of the moon to the planet becomes as small as 10^{-4} AU. Also, the inclination of the moon is changing all the time.

Professor: This sure looks like you are getting Lidov–Kozai (LK) oscillations in the orbit of the moon.

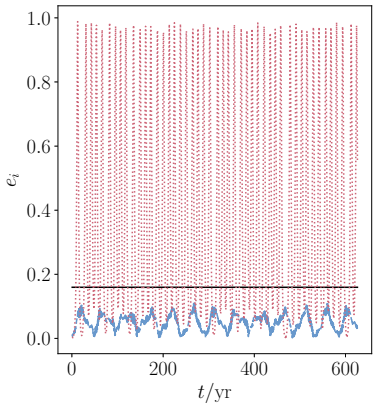
Alice: Wait, what are those?

Bob: They sound ominous.

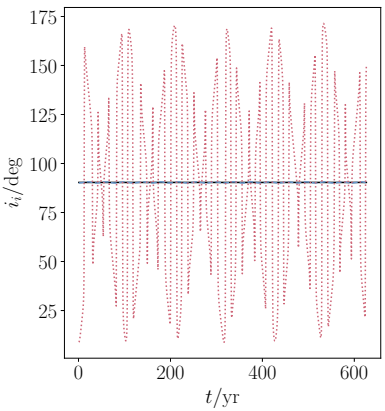
Professor: There is a lot to say about them. In short, if you have a hierarchical three-body problem in which the inner binary is orbited by an inclined third body, then the torque of the third body will make the inner orbit eccentric, whereas the semimajor axis remains fixed. This is a periodic phenomenon in which the eccentricity is highest whenever the mutual inclination is lowest and vice versa. The closer the initial mutual inclination is to 90°, the higher the maximum eccentricity reached. Technically, the eccentricity will reach unity as the mutual inclination reaches 90°. This is likely the source of your problems. Although you are dealing with a four-body system, for many intents and purposes, we can consider the stellar binary to be a point mass. So, effectively, we have a three-body system in which the moon–planet system (the inner binary) is orbited by a massive third body (the



(a) Semimajor axes a_i (thicker lines) and periapsis distances $r_{\mathrm{p},i} = a_i(1 - e_i)$ (thinner lines)



(b) Eccentricity e_i



(c) Inclinations i_i

Figure 13.1
Orbital elements as a function of time for the Kepler-16 system ($a_{\mathrm{m}} = 0.005$ AU and $i_{\mathrm{pm}} = 81°$). Black, blue, and red lines correspond to the binary, planet, and moon, respectively. The moon becomes highly eccentric and changes its inclination due to Lidov–Kozai oscillations.

stellar binary). When $i_{\rm pm}$ is close to 90°, this means that the "inner" and "outer" orbits are mutually highly inclined, and this leads to very high eccentricities. In turn, the integrator gets into trouble because high eccentricities imply small periapsis distances, so it will need to make very small time steps near periapsis.

Bob: This indeed seems consistent with the time-step errors that we get.

Professor: Yes. We can investigate this a bit more quantitatively. The period of LK oscillations is given, to within a factor of a few, by

$$t_{\rm LK} \sim \frac{P_{\rm p}^2}{P_{\rm m}} \frac{M_{\rm p}+M_{\rm m}+M_1+M_2}{M_1+M_2} \left(1-e_{\rm p}^2\right)^{3/2} \simeq 20\ {\rm yr}, \tag{13.1}$$

where I assumed that $a_{\rm m} = 0.005$, the same as in Bob's figure 13.1. So, within 200 yr, we should expect on the order of ten oscillations. Looking at figure 13.1, this is indeed the case: there are fifteen oscillations between 0 and 200 yr.

Next, we can look at the maximum eccentricity or, equivalently, the smallest periapsis distance $r_{\rm p,m} = a_{\rm p}(1-e_{\rm p})$. With a number of simplifications (in particular, zero initial eccentricity), one can show that the maximum eccentricity reached during LK oscillations is

$$e_{\rm max} = \sqrt{1-\frac{5}{3}\cos^2(i_{\rm rel})}, \tag{13.2}$$

where $i_{\rm rel}$ is the initial mutual inclination between the inner and outer orbits. So, for $i_{\rm pm} = 81°$ as in figure 13.1, we get $e_{\rm max} \simeq 0.979$, implying a periapsis distance of $r_{\rm p,m} \simeq 1.0 \times 10^{-4}$ AU. This is consistent with your simulations.

Alice: Great! Could you tell us more, in particular, how to derive these analytical relations?

Bob: Here we go again….

Professor: Maybe if you came back some other time (appendix D), I could discuss that in more detail.

Alice: Sounds good to me!

Bob: Sounds exhausting to me.

13.1.2 Strong Interactions

Bob: So, to summarize, our problems are related to a physical phenomenon. But how do we deal with it in practice?

Professor: Well, in reality, an extremely inclined moon that is excited to very high eccentricity would get very close to the planet and experience tidal effects or even be tidally disrupted if it gets too close, probably before actually physically colliding. So, in reality,

the eccentricity would not reach extremely large values, and I think it would be appropriate to stop the integrations whenever such a "strong" interaction occurs.

Bob: Thanks to the transit method, we at least know the radius of the planet accurately so we could check for physical collisions, which we have already implemented in ABIE.

Professor: Yes, you could do that. But there could be strong interactions before an actual physical collision would occur. In particular, the moon is expected to be tidally disrupted if it gets closer than its Roche radius, which can be computed as

$$r_{\mathrm{Roche}} \simeq 1.523 \left(\frac{M_{\mathrm{p}}}{\rho_{\mathrm{m}}} \right)^{1/3} \simeq 1.8\, R_{\mathrm{J}} \left(\frac{\rho_{\mathrm{m}}}{1\,\mathrm{g\,cm^{-3}}} \right)^{-1/3}, \tag{13.3}$$

where ρ_{m} is the density of the moon. In the numerical estimate, I substituted the value of M_{p} for Kepler-16 and assumed a density of $1\,\mathrm{g\,cm^{-3}}$ (for reference, the mean density of the Moon is $3.34\,\mathrm{g\,cm^{-3}}$ and $5.51\,\mathrm{g\,cm^{-3}}$ for the Earth). This value, $1.8\,R_{\mathrm{J}}$, is larger than the radius of the planet, $0.7538\,R_{\mathrm{J}}$, so we would expect that the moon should be tidally disrupted before colliding with the planet. However, if the moon had a high density, for example, $10\,\mathrm{g\,cm^{-3}}$, then r_{Roche} would only be $\simeq 0.84\,R_{\mathrm{J}}$, close to the radius of the planet. Taking into account that the moon has a finite radius, the moon might not be tidally disrupted but could collide with the planet instead.

In any case, we don't know the density of the moon. I think it would be reasonable to just check for physical collisions and bear in mind that, depending on the density of the moon, it could be tidally disrupted in some cases, whereas you do not check for this scenario in the integrations.

Alice: Okay, so we have a way to deal with our problem by essentially avoiding the super-high inclination cases.

Professor: Exactly right.

13.1.3 Short-Range Forces

Professor: Now that we are talking about strong interactions, I should mention that there are also other effects that can play an important role in real physical systems of planets and moons. In principle, these effects can be modeled as additional accelerations acting on the moon, and they are highly sensitive to the distance of the moon to the planet (or to the stars). Therefore, these effects are also known as short-range forces.

Alice: Should we add them to our code and include them in our simulations?

Professor: A complication is that most of the short-range forces depend on the details of the moon and of the planet, not all of which are known. For your project, I think it suffices for you to be aware of which forces are potentially relevant by looking at the associated time scales, and what their consequences might be.

Bob: For starters, what kind of short-range forces exist that are relevant for our system?

Professor: I can think of three flavors of short-range forces. First, there are deviations from the Newtonian equations of motion due to general relativity. For planetary systems, these deviations are described to good approximation by adding correction terms to the acceleration, known as post-Newtonian (PN) forces or accelerations. In the so-called PN approximation, the general relativistic equations of motion are essentially expanded in terms of v/c, where v is the orbital velocity, and c is the speed of light. The zeroth-order PN term is proportional to $(v/c)^0$, that is, it is independent of v, and is simply the Newtonian acceleration. The next nonzero term turns out to be proportional to $(v/c)^2$, and we call this the first-order PN term.

Bob: What does this term do?

Professor: The first PN term leads to precession of the line of apsides of the orbit. There is a well-known example of this phenomenon in the Solar System: the innermost planet, Mercury, precesses due to the Newtonian perturbations from the other planets (mainly due to Jupiter) but also due to the first-order PN term (at a rate of about forty-three seconds of arc per century). In fact, before Einstein published his famous theory of general relativity in 1915, there was a long-standing problem that the observed precession of the orbit of Mercury was discrepant from the Newtonian value arising from perturbations from the other planets (i.e., as predicted by Laplace–Lagrange theory). In the nineteenth century, the French mathematician Le Verrier hypothesized that there exists an additional planet within the orbit of Mercury, which would explain the discrepancy. He named this hypothetical planet "Vulcan" (since it would indeed be a fiery planet very close to the Sun). Since Le Verrier had successfully predicted the existence of Neptune using similar arguments in 1846, the hypothetical planet Vulcan was given serious attention, and much effort was made to observe it. Some old Solar System maps from the nineteenth century even included the planet Vulcan. However, despite these efforts, the planet was never found, and Einstein's prediction of Mercury's precession by precisely the discrepant amount was a major triumph for the theory of general relativity.

Alice: This story sounds reminiscent of Planet Nine, which has been hypothesized to exist in the outer Solar System based on the clustering of trans-Neptunian objects in their orbital orientation.

Professor: Yes, exactly right, Alice. Let me emphasize that the existence of Planet Nine is still heavily debated and controversial. There are currently active searches for it, which have so far turned out to be unsuccessful. However, this may not be so surprising given the difficulty of detecting a faint planet in the outer reaches of the Solar System. The future will hopefully tell. Returning to Mercury, the anomalous behavior is due to a deviation of Newton's laws of motion, rather than due to unseen objects.

Now, PN precession occurs in the orbit of the moon around the planet on a time scale given by

$$t_{1\mathrm{PN}} = \frac{1}{6}\left(1 - e_{\mathrm{m}}^2\right)\frac{a_{\mathrm{m}}c^2}{GM_{\mathrm{p}}}P_{\mathrm{m}} \simeq 5.3\,\mathrm{Myr}, \tag{13.4}$$

where I substituted the values $a_{\mathrm{m}} = 0.005$ AU and $e_{\mathrm{m}} = 0$ (note that PN precession applies to any orbit, so this includes the orbit of the planet around the binary and the stars around each other, but here I will only consider the orbit of the moon around the planet).

Bob: Why the "1PN" in the subscript of the PN time scale, $t_{1\mathrm{PN}}$?

Professor: The PN precession I am talking about is associated with the first-order PN term, hence the "1".

Another short-range force is due to the tidal deformation of the planet by the moon (I will neglect the converse effect of the tidal deformation of the moon by the planet). Just like our Moon causes tides in the oceans on Earth, the moon could raise tidal bulges on the planet. These bulges give rise to accelerations in addition to the Newtonian point-mass acceleration. The additional accelerations can be classified into two categories: those that lead to dissipation of energy within the planet at the expense of orbital energy (so the moon will spiral into the planet) and those that lead to precession of the line of apsides. I will focus on the latter effect, which is known as short-range precession due to tidal bulges. The associated precession time scale (i.e., precession period), for a circular orbit of the moon, $e_{\mathrm{m}} = 0$, is given by

$$t_{\mathrm{TB}} = \frac{8}{15}\frac{1}{n_{\mathrm{m}}}\frac{M_{\mathrm{p}}}{M_{\mathrm{m}}}k_{\mathrm{AM,p}}^{-1}\left(\frac{a_{\mathrm{m}}}{R_{\mathrm{p}}}\right)^5 \simeq 3.5\,\mathrm{Myr}. \tag{13.5}$$

Here, $n_{\mathrm{m}} \equiv 2\pi/P_{\mathrm{m}}$ is the mean motion of the orbit of the moon, and $k_{\mathrm{AM,p}}$ is the apsidal motion constant, which is dimensionless and depends on the density structure of the planet. A typical value for a planet would be $k_{\mathrm{AM,p}} = 0.25$, which I used for the numerical estimate (and, again, I assumed $a_{\mathrm{m}} = 0.005$ AU).

Finally, there is a short-range force due to the rotation of the planet (ignoring the converse effect if the moon is spinning). Due to the spin of the planet, its shape is not perfectly spherical—the equatorial radius is larger than the polar radius. Similar to tidal bulges, this effect gives rise to accelerations in addition to the Newtonian point-mass acceleration, which now depend on the planet's spin rate. The effect is precession of the line of apsides, on a time scale given by (again, assuming $e_{\mathrm{m}} = 0$)

$$t_{\mathrm{rot}} = \frac{1}{n_{\mathrm{m}}}\left(1 + \frac{M_{\mathrm{m}}}{M_{\mathrm{p}}}\right)^{-1}k_{\mathrm{AM,p}}^{-1}\left(\frac{a_{\mathrm{m}}}{R_{\mathrm{p}}}\right)^5\left(\frac{n_{\mathrm{m}}}{\Omega_{\mathrm{p}}}\right)^2 \simeq 5.4\,\mathrm{yr}. \tag{13.6}$$

Here, Ω_{p} is the spin rate of the planet (which is $2\pi/P_{\mathrm{spin,p}}$, where $P_{\mathrm{spin,p}}$ is the spin period). Note that there are no tidal bulges if the moon is considered massless (as you can see from equation 13.5, $t_{\mathrm{TB}} \to \infty$ as $M_{\mathrm{m}} \to 0$), but there is still precession due to the rotation of the

planet ($t_{\rm rot}$ remains finite as $M_{\rm m} \to 0$). For the numerical estimate in equation (13.6), I again set $a_{\rm m} = 0.005$ AU and assumed that the planet is spinning at half its breakup rotation speed, which is $\Omega_{\rm p,crit} = \sqrt{GM_{\rm p}/R_{\rm p}^3}$.

Bob: What is the breakup rotation speed?

Professor: The breakup rotation speed is the rotation speed at which the planet would physically break apart because the centrifugal force at the surface is the same as its self-gravity. Although we do not (currently) know the rotation period of the *Kepler* circumbinary planets, we know they cannot exceed the "speed limit" $\Omega_{\rm p,crit}$. So my assumed value of one half $\Omega_{\rm p,crit}$ is quite an extreme one.

Alice: I get that there are all these precession time scales. How do we use them to say anything useful about our simulations?

Professor: This ties into the LK oscillations. During these oscillations, not only will the eccentricity and inclination oscillate, but also the arguments of periapsis (and longitudes of the ascending nodes) of both the inner and outer orbits. It turns out that, if there is an additional source of precession, in particular in the inner orbit, then the LK oscillations can be reduced in their amplitude (i.e., the maximum eccentricity attained is lower) or even completely quenched (i.e., there is no eccentricity excitation). One can generally distinguish between three regimes, depending on the LK time scale ($t_{\rm LK}$) and the short-range precession time scale ($t_{\rm SRF}$):

1. $t_{\rm LK} \ll t_{\rm SRF}$—short-range forces are completely unimportant and can be safely ignored;
2. $t_{\rm LK} \gg t_{\rm SRF}$—short-range forces dominate precession in the inner orbit and completely quench LK oscillations;
3. $t_{\rm LK} \sim t_{\rm SRF}$—precession due to short-range forces is comparable in importance to precession associated with LK oscillations; there can be a reduction of the maximum eccentricity but also an *enhancement* of the eccentricity (in some so-called resonant cases, where the precession time scales are commensurate with each other).

Alice: Looking at your numbers, for the tightest orbits of the moon around the planet (for which short-range forces are most important), relativistic precession and tidal bulges are completely unimportant, whereas precession due to rotation could be important.

Professor: Yes, but note that $t_{\rm rot}$ (equation 13.6) depends inversely and quadratically on $\Omega_{\rm p}$, so precession due to rotation quickly becomes unimportant as the planet is spinning less rapidly.

Alice: Okay. So, to summarize: we can say that, with reasonable assumptions about the moon and planet, short-range forces are probably not very important.

Bob: Which means we do not have to implement all these complicated terms!

Alice: Yep, good for us!

Bob: Alice and Bob for the win!

13.2 A Suite of Integrations for Kepler-16

13.2.1 Stability Maps

Alice and Bob meet again the following day.

Bob: Last night, I reran the simulations with collision detection enabled, checking for physical collisions between the moon and the planet or the stars. I assumed the observed radii for the planet and the stars and zero radius for the moon (since its radius would likely be much smaller than that of the planet). I used the Gauss–Radau integrator with a minimum time step of 10^{-13} and a tolerance parameter of $\epsilon = 10^{-8}$.

Alice: Nice! How did the simulations turn out?

Bob: Good news: I no longer get the issue with the time steps being too small.

Alice: That's great!

Bob: Now, how do we process all our data in a meaningful way?

Alice: For starters, we could consider the moon to be stable for a given integration time if its orbit remains bound to the planet within that time. If not, then we flag it as unstable.

Bob: That sounds reasonable. We could then make plots in the $(a_{\rm m}, i_{\rm pm})$ parameter space indicating stable and unstable orbits. If the orbit is stable, we mark it with, say, a green dot. If it is unstable, we mark it with, say, a red cross. If there is a collision, we indicate that with another symbol.

Alice: Okay, but we have different initial phases of the moon around the planet (we chose ten values). So each point in the $(a_{\rm m}, i_{\rm pm})$-plane actually represents ten simulations. We could simply make ten plots, but this becomes messy very quickly.

Bob: I guess you're right. Maybe we can somehow combine all the data into one $(a_{\rm m}, i_{\rm pm})$ plot?

Alice: I think that there will be regions that are "completely stable," that is, the moon is stable regardless of the initial orbital phase, in particular if $a_{\rm m}$ is small. We mark these with green dots. Conversely, if $a_{\rm m}$ is large, then the moon will likely be unstable irrespective of the initial phase, which we mark with red crosses. In the intermediate regime, I imagine that some initial phases could yield stable orbits, whereas others could yield unstable ones. We could mark these with a different symbol, for example, yellow open circles.

Bob: I like it. That's a good solution. We also have to consider collisions. I suggest we show collisions with the planet with "–" symbols and collisions with the stars with "$\star$" symbols (small for collisions with the secondary star and large for collisions with the primary star).

Alice: Fine, but there is another complication. For each point in the (a_m, i_{pm}) parameter space, there could be collisions with the planet for some phases, but collisions with the stars for others.

Bob: I suggest that we simply show the collision marker corresponding to the most common collision outcome among the realizations of the phases. Actually, we could also indicate the frequency of collisions at each (a_m, i_{pm}) point with color coding: yellow to red for occurrences between one and ten.

Alice: I like that idea. In the end, we will have to make these plots for different integration times. But if the dependence on time is not too strong, we have a way to show all the data in a single plot.

Bob: Yep! Give me some time to write a PYTHON script to make the stability maps.

13.2.2 Interpretation

13.2.2.1 LK evolution and MMCs

Alice and Bob once again meet the next day.

Bob: Okay, I have processed the data. I have made three stability maps for three integration times (400, 700, and 1,000 P_p), which I show in figure 13.2.

Alice: Nice! It almost looks artistic!

Bob: Yeah, I guess so. First of all, there seems to be little dependence on the integration time.

Alice: Agreed. There are a few yellow points that appear over time on the boundary layer, which is to be expected. In some cases, the moon could be stable for a short time and become unstable later on. We have seen this behavior before. But on the whole, it seems like $400\,P_p$ is already enough to set most of the stability map.

Bob: Yep. There is a striking feature at high inclinations, with i_{pm} close to 90°. There are a lot of collisions with the planet at these inclinations, even if a_m is small.

Alice: These must be the LK oscillations that Prof. Starmover told us about. At high inclinations, these oscillations lead to high eccentricities in the orbit of the moon around the planet. As the inclination approaches 90°, the periapsis distance apparently becomes small enough for the moon to collide with the planet.

Bob: Yes, makes sense. However, for large a_m and i_{pm} close to 90°, the collisions disappear and instead the moon becomes unstable without colliding.

Alice: Right, I am not sure why, though. Maybe we should ask Prof. Starmover. There's also another thing I do not quite understand. It looks like the stability regions are more or less symmetric around 90°, that is, there does not appear to be a major difference be-

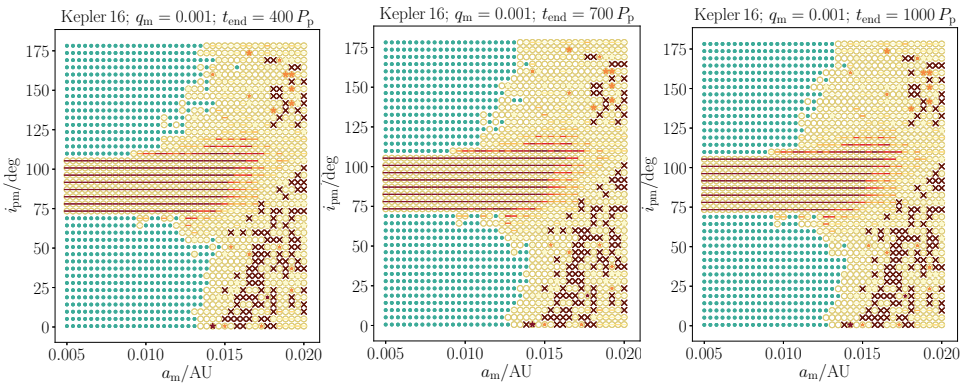


Figure 13.2

Stability map for the Kepler-16 system. In each panel, each point represents ten realizations with different initial lunar mean anomaly, which are used to define stable (green filled circles), completely unstable (red crosses), and marginally stable (yellow open circles) systems. Collisions of the moon with the planet and stars are indicated with "−" and "⋆," respectively, where the color encodes the frequency with respect to the ten realizations of the initial lunar mean anomaly (yellow to red for one to ten). The frequencies of the most common collision type among the mean anomalies are shown. For collisions of the moon with the stars, the large and small "⋆" symbols correspond to collisions with the primary and secondary star, respectively. The three panels in each row correspond to different integration times: 400, 700, and 1,000 P_p, where P_p is the orbital period of the planet around the stellar binary. The mass ratio of the moon to the planet is $q_m = M_m/M_p = 0.001$.

tween prograde and retrograde orbits. However, I thought that prograde orbits are generally understood to be less stable than retrograde orbits.

Bob: Why would that be?

Alice: If the orbit is retrograde, then the relative velocities between the moon and the other bodies (the planet and the two stars) are typically larger compared to prograde orbits. This means that the other bodies have less of an effect on the moon, since for the perturbation on the moon to be large, not only does the force matter, but also the effective duration over which it can act.

Bob: Okay, sounds plausible.

Alice: Yes, but apparently this is not the case here. Let's ask Prof. Starmover if she has an answer to our puzzles.

Alice and Bob walk to Prof. Starmover's office and show her the stability map.

Professor: This looks very nice, Alice and Bob!

Bob: Thank you! I am feeling like an artist, but without needing any real artistic skills. We understand that LK oscillations lead to collisions of the moon with the planet at high initial inclinations. However, this feature disappears for larger $a_{\rm m}$. Do you have any idea why?

Professor: Well, you have to consider that the LK mechanism is a secular effect, that is, it takes place on time scales that are much longer than the orbital periods. In your case, $t_{\rm LK} \sim (P_{\rm p}/P_{\rm m})P_{\rm p}$ (see equation 13.1), which is much greater than $P_{\rm p}$. I think that in most cases, instability happens on short time scales, comparable to $P_{\rm p}$. So at the larger $a_{\rm m}$, the LK mechanism does not "get a chance" to act, since the moon is stripped from its host planet on a shorter time scale. At smaller $a_{\rm m}$, the moon in principle remains bound to the planet, but due to the LK mechanism, it collides with the planet.

Bob: So, either way, a highly inclined moon is doomed from the start!

Professor: Yes, you could say that. By the way, you can quantify this effect by calculating the boundary between collisions and no collisions due to the LK mechanism. To get this boundary, all you need to do is equate the smallest periapsis distance due to LK oscillations, which you can get from the maximum eccentricity (equation 13.2), to the physical radius of the planet, that is,

$$a_{\rm m}\left(1-\sqrt{1-\frac{5}{3}\cos^2(i_{\rm pm})}\right)=R_{\rm p}. \tag{13.7}$$

You can show this relation as lines in the $(a_{\rm m}, i_{\rm pm})$ plane, and it should be consistent with your collision-instability boundaries at high inclinations, in the case of small $a_{\rm m}$.

Bob: Sounds good. We will add these lines to our plots.

Alice: Another thing we do not understand, Prof. Starmover, is why prograde orbits seem to be about equally stable compared to retrograde ones. Aren't retrograde orbits supposed to be more stable than prograde ones?

Professor: You are right that this is usually the case. However, you do have to consider that the planet–moon system is not orbited by a single object but by a stellar binary that presents a quadrupole moment, that is, it is effectively an extended object. This, I think, complicates matters. In particular, I think that it is the cause of the relative instability of retrograde orbits and that it is related to a mean-motion commensurability (MMC).

Bob slides his hand over the top of his head, as though something flew over it.

Bob: That went right over my head. You will have to elaborate, Prof. Starmover.

Professor: Of course. MMCs are commensurabilities of the orbital motion in a system. That is just a fancy way of saying that some of the orbital periods have rational proportions to each other. A commensurability of the motion is often manifested as a dynamical effect called a mean-motion *resonance* (MMR). The classic picture of an MMR in a system is a single star with two planets that have commensurable orbital periods. For example, a star with one planet with a period of ten days and another with a period of twenty days. In this case, there is a 2:1 MMR. If we assume, for simplicity, circular orbits, it means that the planets will encounter each other at a relatively close distance each time the twenty-day planet makes a complete revolution. These encounters can have a cumulative effect, similar to a pendulum that is excited at its natural frequency. The cumulative effect can be destabilizing (in most cases) or stabilizing. If they are stabilizing, they usually only lead to a periodic change in the orbital elements, in particular the semimajor axis and eccentricity. This is actually very useful, since these oscillations lead to changes in the transit time of transiting exoplanets. For example, there could be a single planet close to the star, occulting the stellar light and causing transits. There could be a second planet farther away and not detectable via transits, but due to the MMR, it affects the timing of the transit signals of the inner planet. These transit timing variations (TTVs) depend on the properties of the second planet, so by measuring the TTVs, one can indirectly infer the properties of the second planet. An increasing number of planets are being detected by this method.

Bob: That's a cool technique!

Professor: Indeed. In your case, there are not two orbits (two planets around a single star) but three (the stellar binary, the planet around the binary, and the moon around the planet). Let's compare the orbital periods.

Prof. Starmover takes out her calculator.

Professor: The stellar binary has a period of $P_{\rm bin} \simeq 0.11$ yr, the planet a period of $P_{\rm p} \simeq 0.63$ yr, and the period of the moon depends, of course, on $a_{\rm m}$; for $a_{\rm m} = 0.005$ AU, it is $P_{\rm m} \simeq 0.02$ yr, and for $a_{\rm m} = 0.015$ AU, it is $P_{\rm m} \simeq 0.10$ yr.

Alice: Based on these numbers, the periods of the binary and the planet are not particularly close to each other and not commensurate. But the orbits of the binary and the moon (around the planet) could be commensurate with each other if $a_{\rm m}$ is close to 0.015 AU.

Professor: Yes, so I think that it is no coincidence that the general boundary between stable and unstable systems is close to 0.015 AU. You could quantify this by setting $P_{\rm bin} = \alpha P_{\rm m}$, where α is a dimensionless number. Solving for $a_{\rm m}$, this will give you

$$a_{\rm m,MMC} = \alpha^{-2/3}\, a_{\rm bin} \left(\frac{M_{\rm p}}{M_1 + M_2} \right)^{1/3} \simeq 0.016 \text{ AU}, \tag{13.8}$$

where for the numerical value I took $\alpha = 1$, that is, the 1:1 MMC. If the MMC is manifested as an MMR, then the effect is expected to be strongest for the 1:1 MMC, that is, the binary and moon are orbiting at the same frequency.

Bob: This looks a lot like the Hill radius, $r_{\rm H}$, with the mass ratio $M_{\rm p}/(M_1 + M_2)$ to the one-third power (see equation 12.4).

Professor: Yes, but there is an important difference: $r_{\rm H}$ scales with $a_{\rm p}$ (or $a_{\rm p}[1 - e_{\rm p}]$), whereas $a_{\rm m,MMC}$ scales with $a_{\rm bin}$.

Bob: I will add this line to the plots, including the LK collision boundary lines.

Professor: Great. My hypothesis is that, assuming the MMC is manifested as an MMR, it results in large variations in the orbital elements of the moon around the planet, until they get so large that the moon gets close to the binary at apoapsis, which destabilizes it. This would apply to both prograde and retrograde orbits, thus explaining the symmetry in the simulations around 90°. To test this, I would suggest that you do simulations in which you replace the stellar binary by a single point mass. If my explanation makes sense, I expect to see a significant difference at retrograde orientations—in the single-star case, they should be more stable compared to prograde orientations.

Bob: Okay. While we are at it, we could also do a simulation with a more massive moon, say, $q_{\rm m} = 0.1$.

Alice: Sounds good! Let's meet again tomorrow.

13.2.2.2 Testing the hypotheses

The next day, Alice and Bob meet again.

Bob: I have carried out simulations with $q_{\rm m} = 0.1$ and with $q_{\rm m} = 0.001$ and the stellar binary replaced by a point mass. I have plotted the stability maps in figure 13.3 and also plotted the lines that we talked about.

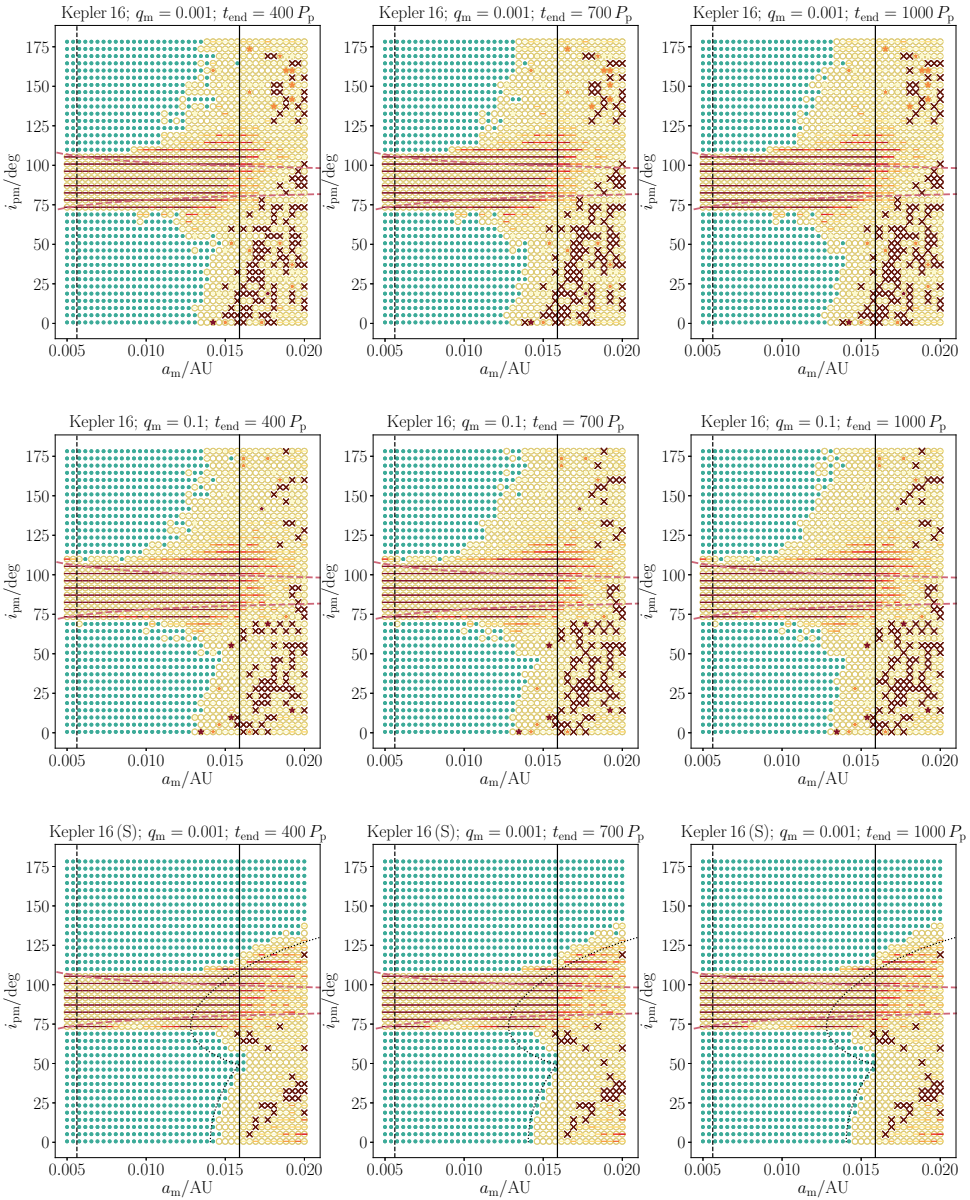


Figure 13.3

Stability maps for the Kepler-16 system. Top row: $q_m = 0.001$; middle row: $q_m = 0.1$; bottom row: $q_m = 0.001$, with the stellar binary replaced by a point mass. The vertical black solid lines show the location of the 1:1 MMC of the moon with the binary (equation 13.8); the vertical black dashed lines show the locations of the 2:1 and 1:2 MMCs. The red dashed lines show the boundary for collisions of the moon with the planet due to LK evolution, computed according to equation (13.7). In the third row, the black dotted lines show the Hill stability boundary for the single-star case according to Grishin et al. (2017).

The LK-collision boundaries in the maps are described by equation (13.7), shown with the red dashed lines. Also, the black solid lines show the 1:1 MMC, and they indeed approximately capture the boundaries between stable and unstable orbits.

Alice: Nice! Let's first compare the $q_m = 0.001$ and $q_m = 0.1$ results. There seem to be very few differences.

Bob: Yes. If you look carefully, the stability boundaries show an overall shift toward larger a_m as the mass of the moon increases from $0.001\,M_p$ ($q_m = 0.001$) to $0.1\,M_p$ ($q_m = 0.1$).

Alice: That makes sense, since a more massive moon is expected to be more stable.

Bob: Indeed. Now let's focus on the differences between the single and binary star cases (first vs. third rows in figure 13.3). Prof. Starmover had predicted that retrograde orbits would be more stable in the single-star case. This indeed turns out to be correct: the single-star stability map is no longer symmetric around 90°; retrograde orbits at the larger semi-major axes are now stable.

Alice: So this seems to corroborate Prof. Starmover's idea that the binary destabilizes retrograde orbits.

Bob: Yep. I also plotted for the single-star case in the third row of figure 13.3 some analytic fits with black dotted lines. These fits are from a paper I found, Grishin et al. (2017), in which the Hill stability boundary is generalized to arbitrary inclinations.

Alice: Those fits work well for small inclinations. The agreement becomes worse at higher inclinations, but the overall qualitative trend is still consistent.

Bob: That might be because our definition of stability is different from that of Grishin et al. (2017). Moreover, Grishin et al. (2017) also noted that their fits do not work well at high inclinations.

Alice: Okay. Now that we have looked at Kepler-16, I think it is time we carry out the simulations for the other *Kepler* systems!

13.3 The Other *Kepler* Systems

13.3.1 Stability Maps

After taking the weekend to rest, Alice and Bob reconvene to continue working on their project.

Bob: It's taken a while to run all these systems, but I've finally got the results. I show the maps in figure 13.4 for the binary-star case and in figure 13.5 for the single-star case. In both cases, $q_m = 0.001$, and the integration time is $1{,}000\,P_p$. Similarly to Kepler-16, there are few differences between the results with $q_m = 0.001$ and $q_m = 0.1$, and there is also little difference between integration times of 400, 700, and 1,000 P_p. I also ran a couple of

cases with larger initial eccentricity for the moon, $e_{\rm m}$, but once again found no significant differences.

Alice: You've certainly been busy!

Bob: Nah, I started most of the simulations late on Friday. They've been running all weekend.

Alice takes a moment to look at the plots.

Alice: Generally, the plots look very similar to those of Kepler-16. There seem to be two notable exceptions: Kepler-47c and Kepler-1647. For those, the boundary is not well described by the 1:1 MMC, equation (13.8), and there is no symmetry between the prograde and retrograde orientations, that is, retrograde orbits are more stable.

Bob: Yes. First of all, let me mention that Kepler-47 actually has two confirmed circumbinary planets: Kepler-47b and Kepler-47c.[1] Here, I have considered them separately in the simulations, that is, ignoring the other planet. I have tested this by doing additional simulations with the two circumbinary planets (i.e., a five-body system) but found no differences.

Alice: Good catch!

Bob: Another thing I should mention is that Kepler-64 is technically a quadruple-star system: the binary-circumbinary planet system is orbited by a distant stellar binary. However, the stellar binary is very widely separated, with a semimajor axis of $\sim 1{,}000$ AU, so I think it is safe to ignore the distant stellar binary.

Alice: Seems reasonable.

Bob: Getting back to why the maps of Kepler-47c and 1647 look different: maybe this is because the circumbinary planet is relatively distant from the stellar binary?

Alice: I think that would make sense: a more distant binary would imply that the effects of the 1:1 MMC are weaker. This seems to be supported by the single-star cases: there are little differences between the single- and binary-star cases for Kepler-47c and Kepler-1647.

13.3.2 Summary Plots

Bob: We now have all sorts of stability maps. I wonder if we can summarize the results into just one plot.

Alice: I think I have an idea how to do this. For a given inclination, we can define a critical semimajor axis, $a_{\rm m,crit}$, as the maximum $a_{\rm m}$ for which all ten realizations of the initial phase of the moon are stable, that is, the largest value of $a_{\rm m}$ for which we get green dots in the stability maps.

[1] Recently, a third circumbinary planet was confirmed in the Kepler-47 system (see Orosz et al. 2019).

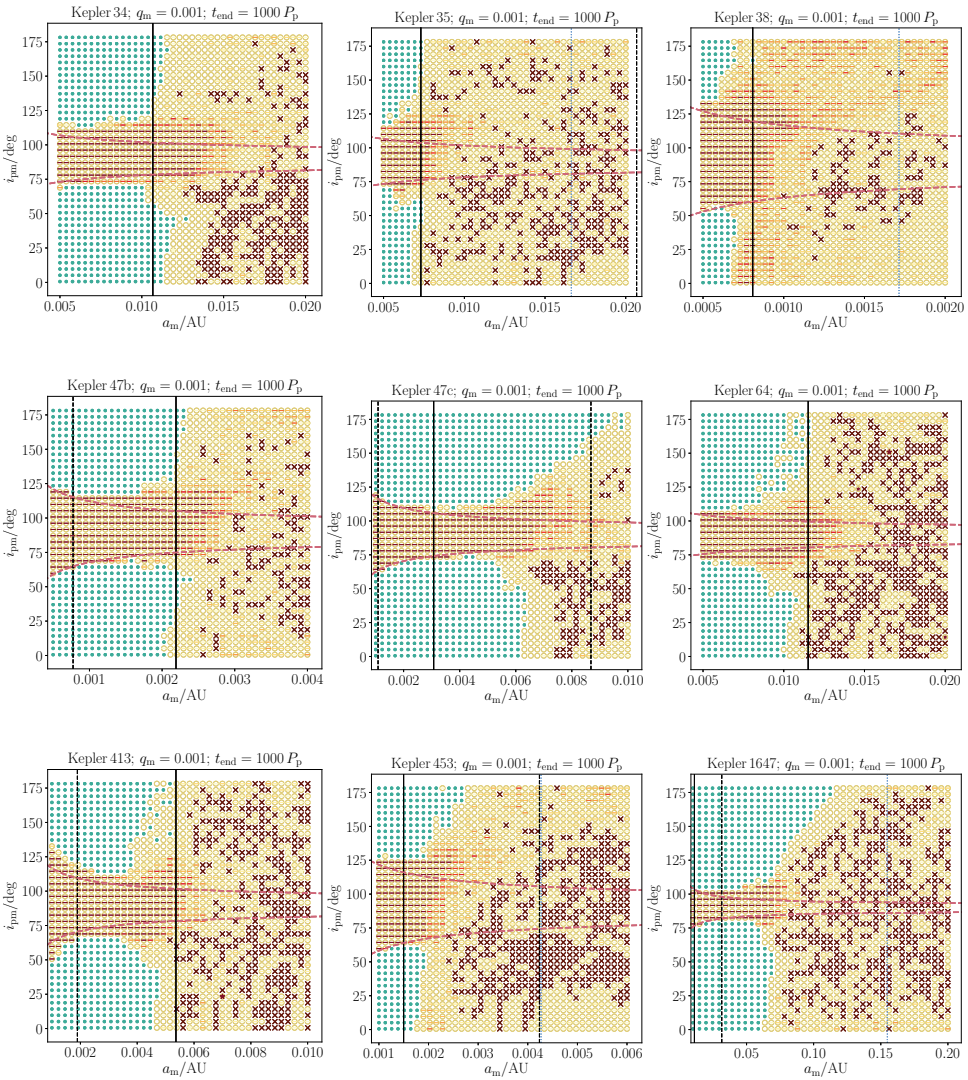


Figure 13.4
Stability maps for nine *Kepler* CBP systems, similar to figure 13.3. The name of the system is indicated above each panel. In all cases, $q_m = 0.001$, and the integration time is 1,000 P_p. The vertical black solid lines show the location of the 1:1 MMC of the moon with the binary (equation 13.8); the vertical black dashed lines show the locations of the 2:1 and 1:2 MMCs. The red dashed lines show the boundary for collisions of the moon with the planet due to LK evolution, computed according to equation (13.7). The vertical blue dashed lines (not visible in all panels) show the Hill radius, equation (12.5).

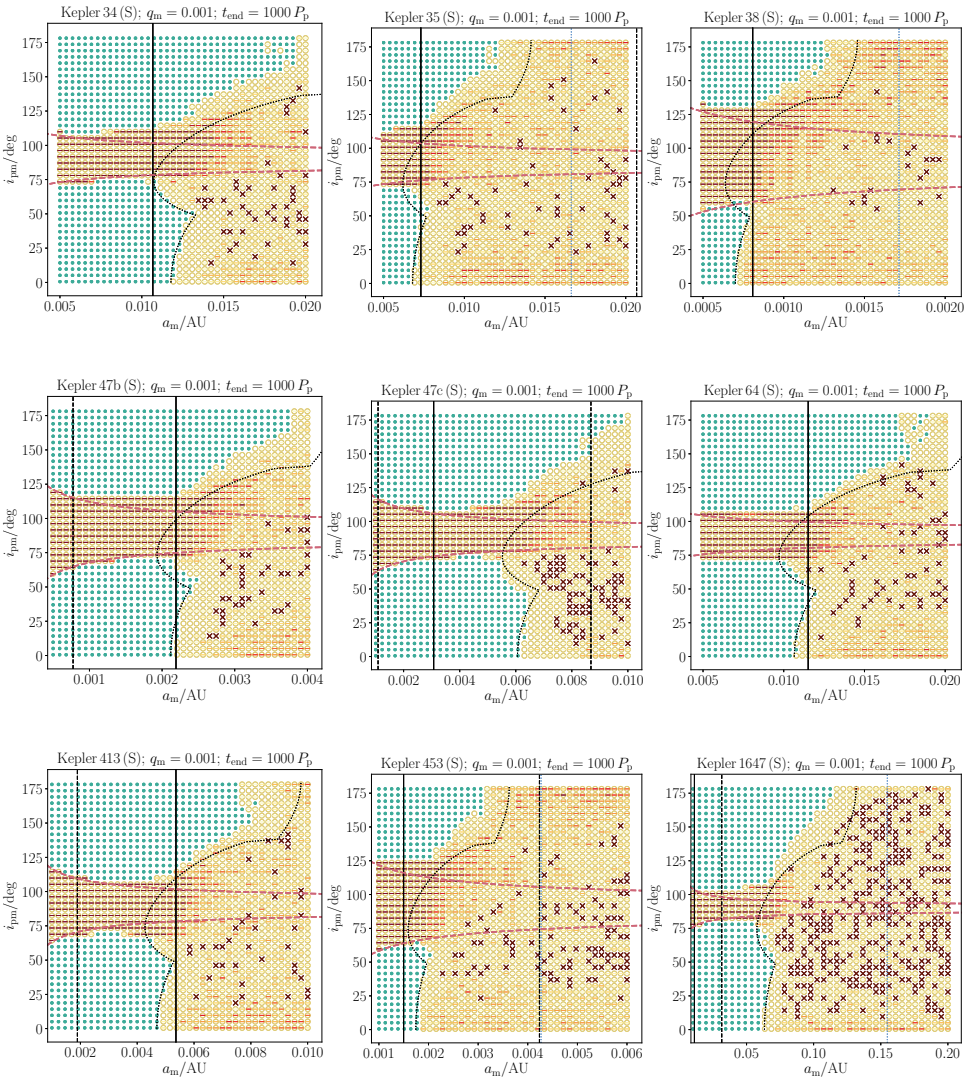


Figure 13.5
Similar to figure 13.4, now for the case when the stellar binary is replaced by a point mass. The black dotted lines show the Hill stability boundary according to Grishin et al. (2017).

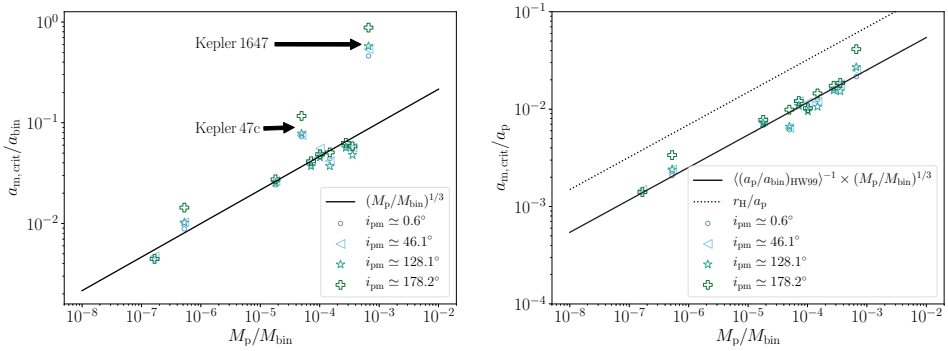


Figure 13.6
Left panel: largest stable value of a_m, $a_{m,crit}$, normalized to a_{bin} and plotted as a function of M_p/M_{bin}, where $M_{bin} = M_1 + M_2$, for each system in the binary case with $q_m = 0.001$ and for a given value of i_{pm} (refer to the legend). The solid black line shows equation (13.8), the location of the 1:1 MMC. Right panel: $a_{m,crit}$ normalized to a_p. The black dotted line shows the Hill radius (equation 12.5), and the black solid line shows the estimate in equation (13.12) assuming the CBP is close to the critical boundary for stability.

Bob: Okay, but due to LK-induced collisions at high inclinations, there are no stable values of a_m in our simulations for inclinations close to 90°, so we will have to exclude those.

Alice: Right. Outside of that regime, the dependence on inclination is weak anyway. So we essentially have just one number, $a_{m,crit}$, for each system. We could make this dimensionless by dividing either by a_{bin} or by a_p. Then, we could plot that dimensionless number as a function of another dimensionless number, the mass ratio $M_p/(M_1 + M_2)$. Equation (13.8) predicts a simple relation between $a_{m,crit}/a_{bin}$ and $M_p/(M_1 + M_2)$. We tested it before visually, but it would be nice to see it quantitatively in a plot.

Bob: I agree. Let me make that plot and get back to you!

Alice leaves to attend a class. A few hours pass before Alice and Bob meet again.

Bob: Okay, I have plotted $a_{m,crit}$ as a function of $M_p/(M_1 + M_2)$ in figure 13.6. In the left-hand panel, I normalized $a_{m,crit}$ to a_{bin}, and in the right-hand panel, I normalized it to a_p.

Alice: Nice!

Bob: Generally, $a_{m,crit}$ shows a power law dependence as a function of $M_p/(M_1 + M_2)$ with slope 1/3. There is indeed little dependence on the initial inclination, i_{pm}. The absolute normalization works out to agree pretty well with equation (13.8) in the left-hand panel. There are two noticeable exceptions, Kepler-47c and Kepler-1647, but we already noted before that the MMC explanation does not apply to them.

Alice: Right, so a deviation from equation (13.8) is not unexpected.

Bob: In the right-hand panel, I also plotted the Hill radius, equation (12.5), with the black dotted line. The scaling of $r_{\rm H}$ with the mass ratio is consistent with the data, but the absolute normalization is not.

Alice: This is consistent with what we noted before: the Hill radius gives an estimate of the stability boundary but is off by a factor of a few.

Bob: I wonder if we can get an analytic expression for $a_{\rm m,crit}$ normalized to $a_{\rm p}$ that gives the correct absolute normalization.

Alice: Maybe we should ask Prof. Starmover?

Alice and Bob enter Prof. Starmover's office and inform her of their progress.

Professor: This is all very interesting! With regard to an analytic expression for $a_{\rm m,crit}$ normalized to $a_{\rm p}$, I can think of the following trick. First, let me write

$$\frac{a_{\rm m}}{a_{\rm p}} = \frac{a_{\rm m}}{a_{\rm bin}}\frac{a_{\rm bin}}{a_{\rm p}} = \left(\frac{M_{\rm p}}{M_{\rm bin}}\right)^{1/3}\frac{a_{\rm bin}}{a_{\rm p}}, \tag{13.9}$$

where I used equation (13.8) with $\alpha = 1$ (the 1:1 MMC) to replace $a_{\rm m}/a_{\rm bin}$. Next, we know that most of the *Kepler* circumbinary planets are close to their dynamical stability boundary with respect to the stellar binary. The latter can be estimated using an analytic fit from Holman and Wiegert (1999), who give the following relation for the critical semimajor axis of the circumbinary planet:

$$\frac{a_{\rm p,crit}}{a_{\rm bin}} = 1.60 + 5.10\,e_{\rm bin} - 2.22\,e_{\rm bin}^2 + 4.12\,\mu - 4.27\,e_{\rm bin}\mu - 5.09\,\mu^2 + 4.61\,e_{\rm bin}^2\mu^2, \tag{13.10}$$

where $\mu = M_2/(M_1 + M_2)$, and I ignored the error bars on the coefficients. To capture this equation in a statistical sense, I will average it assuming a thermal eccentricity distribution of $e_{\rm bin}$ (i.e., ${\rm d}N/{\rm d}e_{\rm bin} \propto e_{\rm bin}$) and a flat mass ratio distribution (i.e., flat in $q = M_2/M_1$).

Prof. Starmover opens a MATHEMATICA notebook on her laptop and performs a few calculations.

Professor: I then get

$$\left\langle\left(\frac{a_{\rm p,crit}}{a_{\rm bin}}\right)_{\rm HW99}\right\rangle^{-1} \simeq 0.252267 \tag{13.11}$$

(alternatively, one could compute $\langle 1/(a_{\rm p}/a_{\rm bin})_{\rm HW99}\rangle \simeq 0.25592$, which gives nearly the same result). Combining equations (13.9) and (13.11) gives us

$$\frac{a_{\rm m}}{a_{\rm p}} \simeq 0.252267\left(\frac{M_{\rm p}}{M_{\rm bin}}\right)^{1/3}. \tag{13.12}$$

Bob: Let me test that equation! I have plotted it in the right-hand panel of figure 13.6 with the solid black line. It seems to agree well with the data!

Alice: That's nice! Now we can use equations (13.8) and (13.12) to predict the stability boundary of moons around circumbinary planets that are close to the stability boundary with respect to the binary, without having to do expensive numerical simulations.

Bob: Sure, but we wouldn't have gotten this far without them!

Professor: You have achieved a lot indeed. In fact, I think your results would be good enough for a publication in a peer-reviewed scientific journal.

Alice: Wow, that would be awesome!

Bob: For sure! We *have* to do it, if it is even a remote possibility. This is so exciting!

Professor: Come back tomorrow, and I will explain to you both how to write a real science paper!

Bob: We're in! I already know how it should start: "A long time ago in a galaxy far, far away…."

Professor: Was that a *Star Wars* reference?

Bob: It sure was!

Alice: Tatooine, the home of Luke Skywalker, was a circumbinary planet in the movie. That's why we chose this project.

Professor: Ah! I get it now.

14 How to Write a Publishable Research Paper

Overview. Prof. Starmover explains to Bob and Alice how to write a research paper worthy of submission to a peer-reviewed astronomy journal. She goes over the different sections that a paper should have, describing their content and sharing some tips on how to write them.

Professor: What came first: the chicken or the egg? An analogy could be drawn here to writing a research paper and conducting scientific research. We're not going to, though.

Alice: Ha ha. Okay, but if we did, what might it sound like?

Bob: I got this one. It is important to take diligent notes while executing a research study, and it is not unprecedented to then use those notes to directly write your paper.

Professor: It occurs to me that this is probably the kind of advice that should have been given to you at the beginning of all this….

Bob: Too late now.

Alice: Don't worry, Professor. We did our duty and took good notes.

Bob: Phew. I'm glad you were on top of that. Alice saves the day. For a second there, all I was thinking is: Uh…. Notes?

Alice: I didn't want to forget anything.

Professor: Glad to hear you took notes, Alice. Bob, be more like Alice.

Bob picks up a pen and begins taking notes on a pad of paper, mumbling to himself.

Bob: Be … more … like … Alice. Got it!

Professor: We are then ready to write our paper. In the fields of astronomy and astrophysics, research papers take on a common general architecture. An abstract, an introduction, followed by the Methods and Results sections, and finally a summary of the main conclusions. Of course, this simple formula is not written in stone, and authors take freedom in deciding the structure of their paper. Let's begin with a brief review of the general

layout of a paper, followed by a more detailed description of how to write and present each section. This will prepare you for writing a paper for submission to a peer-reviewed journal.

Bob: Can we save ourselves some trouble and just talk about *our* results?

Professor: Of course!

14.1 Abstract

Professor: Research papers begin with an abstract of about a paragraph or two in length, which briefly summarizes the motivation, purpose, results, and conclusions of the paper. All in one go. They tend to be around 250 words of text in length.

A successful abstract, I would say, effectively and succinctly conveys the question(s) trying to be answered, their relevance to the field of study, the method being used to address these overarching questions and a justification of their novelty and reliability, and, finally, a clear and concise summary of the primary results and conclusions. Often, a good abstract will finish with a brief statement connecting the new results to some immediately relevant paradigm in the field. For example, if your primary results are providing constraints on the parameter space allowed for exomoons orbiting circumbinary planets, then you might finish the abstract with something like: "Our results inform future observations aimed at detecting exomoons in the sample of circumbinary planets considered here."

Bob: Oh … I think I get it. Leave the reader with some general sense of what they should take away from the paper for future applicability. I suppose it would be inappropriate to simply title our paper: "Do you want to find exomoons? Then read this paper!"

Professor: I would argue that that particular title is not specific enough and would get a lot of criticism from reviewers and editors for various reasons, many of which I am sure I could not predict….

Bob: So that title flopped, I guess. Well, you win some and you lose some.

Alice: Or, in your case, you lose some. Then you lose some more. After several more losses, you lose yet another time. What follows is yet another ruthless series of merciless losses. Eventually, of course, you win … well, almost, but then really you just lose again. And so on and so forth.

Bob: Okay, okay. I think we get it.

14.2 Introduction

Professor: The abstract is followed by an introduction, or "intro," if you prefer. The intro can vary in length but is usually no more than a few pages long. The main purpose here is to review the state of the literature at the time of writing the paper. The authors will

briefly review previous work that has been done, citing directly the studies used to decide what we currently know about the subject matter. This isn't always as easy as it seems. It is important to digest the relevant literature critically, try to understand the views of the authors conducting each study, and arrive at your own conclusions. The intro can and often should mostly just stick to paraphrasing the results and conclusions of the most important papers. However, understanding these studies in detail will go a long way toward guiding your own research.

Alice: So we just review all the previous work that has been done? That's it?

Professor: There's a second, but equally important, purpose for the introduction. This is to motivate the study you are conducting. What overarching question(s) are you trying to answer? Why are you trying to answer it? What do you hope to learn? That kind of thing.

Bob: Review the literature, then motivate the paper. Got it.

Professor: That's right, but the order is up to you. The idea is to grab the readers' attention and provide them with whatever information they will need to understand your paper.

Alice: Anything else?

Professor: The last paragraph usually provides a brief summary of each section of the paper, using about one sentence for each section.

Bob: What about subsections? And sub-subsections? And sub-sub-subsections? And then of course there are sub-sub-sub-subsections to worry about….

Professor: That's a good question.

Bob: It is!?

Alice: It is!?

Professor: Well, almost. Some authors use subsections and even sub-subsections to help them divide the presentation of their paper into a logical structure. It is meant to help the reader digest and understand the main points. The number of times the term "sub" appears before the word "section," however, is usually limited to two. Only sections are usually mentioned in the introduction, but, like I said, the rules aren't written in stone.

Bob: All right, I have a few questions.

Professor: Shoot!

Bob: How do we know in what order to put everything? Like, where do we want to start, and where do we want to end?

Professor: Well, that is largely up to you. But I like to start with a brief skeleton of the overall structure I want to aim for in the intro. So, I try to summarize the content of each paragraph and the main point(s) I want to get across before I start trying to actually write those paragraphs. I usually do this in bullet-point form. Then, I look at their order and might sometimes decide to change or alter it slightly. Once I feel pretty good about knowing what

I want to convey and the order I want to convey it in, I start filling in each paragraph with the relevant points and evidence, including the appropriate citations. Once I have a rough "intro" in place, I then go back to the beginning, read it all over with fresh eyes, and take a little time to think about it. Finally, I return to it, and make whatever final edits I have decided upon. *Then*, I move on to the next section of the paper.

Alice: That is a very good strategy, Prof. Starmover. We will use it!

Bob: Set 'em up, then knock 'em down. I like it.

14.3 Methods

Professor: Next up is the Methods section. In our case, this is where we discuss the gravity integrator we are using to conduct our research and ultimately to answer the overarching science question(s) we have posed.

Alice: Do we need to say anything about the computers we used?

Professor: Some authors choose to describe the computational facilities used to run the integrator, but that usually means that the machines are state-of-the-art or some special setup was used. If you are just using a laptop or a basic desktop computer, it is typically not necessary to mention that in much, if any, detail.

Bob: How long should this section be?

Professor: The Methods section consists of as many paragraphs and subsections as are needed to properly describe all tools and techniques used in the study.

Alice: What tools did we take from the existing literature? What tools and methods did we create ourselves? How do they work, what do they do, and why do they do them? That sort of thing?

Professor: Yes. Exactly. In our case, we are using ABIE and we should review its basic features and why it is a good choice to use for our purposes.

Alice: Gotcha. We can definitely do that. Anything else?

Professor: We should include a table providing all of our chosen initial conditions. How many particles did we use and why? What are the properties of those particles and why? Did we vary our initial conditions at all, or did we stick to just one set? Either way, why did we do that? For example, did we have any computational limitations that contributed to our choices, such as the number of simulations we ran or the number of particles? If we were able to explore a range of initial conditions, why did we choose that range? Was it random, was it computational limitations, or were we motivated by some aspect of the astrophysics driving the overarching question we are trying to answer?

Alice: I think I am starting to get it.

A long pause follows Alice's comment. Alice and the professor both turn to look at Bob. Bob shrugs, wearing a defeated look on his face.

Bob: I know, I know. Be more like Alice.

Alice: What is the "flow" of this section supposed to look like? I mean, where should we begin, and where should we be aiming to end?

Professor: A very good question, Alice. I would recommend starting with the basics about the software and hardware. Clearly describe the code(s) you are using, its intended function(s), the underlying physical assumptions, and so on. Be sure to properly cite all the relevant literature here that contributed to the code, or a specific version of it, that you are using in your own paper. Then, you can explain where you have acquired any data you are using. Actually, you have some freedom here and could break this up into two sections: one for the computational methods and another to present the data.

Bob: Options are good. In our case, it may actually make sense to implement this last suggestion, since we use both computational methods and data obtained from the literature for the known circumbinary planets. I guess we will have to make this decision when the time comes, but I am leaning toward two different sections: one to present the data and one to present the computational methods.

Alice: I'm inclined to agree, Bob. But we'll have to see what works best in practice when we start the actual writing. Then what, Prof. Starmover?

Professor: Then I would start by describing the initial setup or structure of the code, directly connecting those initial conditions as much as possible to the physical problem at hand. Basically, this is where you describe and justify your chosen experiment. You have now presented the tools you will use for your study in the previous paragraph; what is left is for you to describe how you will implement them. What is the initial setup, configuration, and so on of the simulations you intend to perform using your aforementioned code of choice? This is usually a good place to refer to a table of your chosen initial conditions, which you then justify and refer to in the main body of this section.

Typically, a balance between computational expense and statistical significance must be reached. That is, we need enough simulations to properly sample our parameter space of initial conditions, while ideally not going beyond some critical limit that becomes too time-consuming. One nice "trick" you can apply here is to look for "convergence" of your results. That is, once you obtain an answer, does simply increasing the total number of simulations change it? If not, that would argue strongly that your results have converged and that more computationally expensive calculations are not needed.

Bob: That's clever! I like it.

Alice: Me too. It could save us from blindly performing far more simulations than we actually need, which could be a huge time-saver for us. Thanks, Prof. Starmover!

Professor: Happy to help.

14.4 Results

Professor: The results of the paper are presented in the form of figures and tables, followed by a discussion of what they mean and what the reader should take away from them. Aaaaaannd…. That's pretty much the Results section for you.

Bob: Oh come on, Professor! You have to give us more than that! I mean, please and thank you?

Alice: More details could only help us.

Professor: Fair enough. I suppose I made it sound a little more straightforward than it can be in practice. Sometimes, you will have to get creative in order to find the best way to present your results. A common example might be if you have found some sort of correlation between two empirically measured quantities, and you think this correlation has an interesting astrophysical interpretation. In order to demonstrate this effectively, you will need to apply a suitable statistical method to quantify the strength of the correlation and convince the reader that it is a real, statistically significant correlation. This is not always as easy as it sounds.

Alice: Why not?

Professor: Well, for one thing, you need to make sure that you have some way of calculating reliable uncertainties on your measured parameters. Without this crucial step, the results of any statistical tests you do likely won't have too much meaning behind them. Without knowing how large the uncertainties are, it becomes difficult to firmly establish the statistical significance of any result(s) you might find. See what I mean?

Alice: I think so.

Professor: In our case, we first want to show that our integrator is working properly and is giving reliable results for the problem at hand. Are energy and angular momentum conserved? Things like that.

Bob: What else?

Professor: We also need to remind ourselves of the overarching question and think about what information we need to present in order to effectively answer it.

Alice: This is actually starting to sound like fun!

Professor: It is! This, and the Discussion section, are my favorites. This is where you really need to think critically about your results, be honest with yourself and your collaborators about their statistical significance, think about how your new results are informing you about directions that should be taken and additional studies that should be performed in the

future, and think about how best to present any figures or tables, for maximum digestibility, and so on.

Bob: Okay, so let's break this down again. First, we want to decide on the main results we want to convey to the reader in this section, given the analysis we have already performed. This effectively boils down to choosing suitable figures and tables to be included in the paper, which most effectively convey the primary results of our study. Right?

Professor: That's right, Bob. Next, we want to establish the statistical robustness of any key results, correlations, and so on that we have identified via our analysis. This typically involves performing some sort of additional test to quantify the level of agreement (or disagreement) between the data and a given model. To do this, we typically focus on obtaining, for a given input model, reliable estimates for the fitting parameters and their corresponding uncertainties (e.g., bootstrap, Monte Carlo Markov chain, and so on).

Alice: How do you know which statistical method to apply to the problem at hand?

Professor: That can be a bit of an art form really, and there are almost always multiple options to choose from. For example, let's say you have some data and want to quantify the robustness of any possible underlying correlation between two particular parameters in your data set. Here, an often useful strategy is the "bootstrap" method.

Here, you take your data set, and sample from it randomly (but with replacement) to generate new fake data sets of the same total size (i.e., number of data points). So, in practice, what you do is randomly draw a data point from your sample. Then, you replace the missing data point in the original data set with an identical point. You then repeat the procedure, until you have constructed an entirely new but analogous "fake" data set with the same total number of data points. After repeating this procedure many times, you will have generated something like 1,000 fake data sets. You can then apply your fitting technique to each fake data set to obtain the best-fitting parameters for each (often a slope and y-intercept, for example). This allows you to construct histograms for each parameter, which tend to take on an approximately Gaussian form; the peak of the distribution gives the best-fit value and the wings of the distribution can be used to quantify the uncertainties (i.e., the values within one standard deviation of the mean comprise about 68 percent of the total data set, and are often quoted as the 1-σ uncertainty).

Alice: We're with you so far, Prof. Starmover. Keep going!

Professor: We're almost done with the Results section, I think. Once the result is firmly established (in your own mind, at least) as being statistically significant and hence worthy of further consideration, you should move on to further considering it, preferably within the context of the original overarching question(s) that you posed in the Introduction. This transitions us nicely away from the Results section and into the Discussion section.

14.5 Discussion

Bob: Okay, so what is the point of a Discussion section? I'm not sure I completely get it….

Professor: This is where you relate your key results back to the guiding questions you posed in the Introduction. Once this is done, there are a few options for what comes next. Some people might go right into the relevance of these results to the immediate state of the field and/or its future. Have the results informed us of some future study or observational campaign that could be done, knowing now what exactly the data will provide, how constraining they would be (or need to be), and so on? Do the results solve or address some major open question in the field? The Discussion section is often where you discuss any limitations of your results. For example, perhaps you would have liked to better measure some parameter and, having now done the present study, are aware that this would be an important step forward for some reason related to its ability to directly address some outstanding but crucial questions. The Discussion section is a good place to include this, by discussing the limitations of the data that have prohibited the desired analysis, what will be learned by doing better in a future study, and, if possible, what could be improved upon to successfully achieve that goal.

Bob: Okay, so something like this: what did we learn, what weren't we able to learn, what got in our way, and how can we improve on things in the future? That kind of thing?

Professor: Yes, Bob. That is actually a pretty good synopsis of the Discussion section. For example, do you compare your numerical integrations to any predictions from analytic theory? Do they agree, and at what level of significance? If they do agree, you can often argue that the simpler analytic model is actually successfully capturing all of the dominant or relevant physics. If not, this might suggest that the simpler model is too simple and is missing some crucial physics. Or maybe you trust the analytic theory more and are worried something important is missing from your simulations. I would recommend avoiding pure speculation and trying to come up with ways of testing your initial hypotheses to answer these questions, at least tentatively. Even if you cannot yet test them, this is one justification for why the study was necessary and why the results should be published in a peer-reviewed journal. Make sense?

Alice: Yep! I am with you, Prof. Starmover.

Professor: Another example might be if you are comparing data to theory. Do they agree, and at what level of statistical significance? Often, the data come along with unfortunately large uncertainties, which can significantly reduce the probability of reliably using them to constrain or identify any underlying trends. One suggestion to improve upon things in a future study would then simply be to figure out how to reduce those uncertainties, by acquiring new or additional data with the same instrument, using a different instrument

(e.g., with better resolution), proposing some technique to allow for a reliable measurement of some specific feature in the data, and so on.

Alice: Okay, so the basic idea is to put your work in the context of the field and use it to move the field forward, even slightly.

Professor: That's right! You guys are starting to get the hang of this.

14.6 Summary

Professor: Last but not least comes the Summary or Conclusions section. This is every bit as easy as it sounds. Remind the reader of your overarching question(s), the method(s), technique(s) and/or data you used to address your question(s), what results you find and their statistical robustness, and finally a sentence or two linking your work to the "bigger picture."

Alice: Okay, so it's basically a longer version of the abstract?

Professor: Yes, that is actually a good way of looking at it. Take a few paragraphs to do more or less what is done in the abstract, but feel free to go into a little more detail. One suggestion would be to first write an abstract without worrying about length restrictions. Then, move this to the Summary section and try again, this time making the necessary effort to be succinct, and stay within your allotted word limit.

Bob: Who allots the word limit?

Professor: Whichever journal to which you decide to submit your paper.

Alice: Cool! So, which journal will it be? I mean, which journal do you recommend?

Professor: I recommend either *Monthly Notices of the Royal Astronomical Society*, or *MNRAS* for short, or the *Astrophysical Journal*, or *ApJ*. These are the two main peer-reviewed publications in astrophysics. There is also *Astronomy & Astrophysics*, but that tends to attract a more European crowd. For your purposes, I would recommend *MNRAS*, since they do not apply page charges to the authors. So it is free for you to publish your work in their journal. *ApJ*, on the other hand, charges a fee for each page. These days, all journals charge for color images in the printed version of the paper but not in the version online. It really is not terribly necessary for you to have color images in the printed version, so I would argue that publishing in *MNRAS* should be entirely free for you.

Bob: Gotcha! This is exciting…. Wait, why do some journals charge the authors, and some do not?

Professor: Good question. I think it relates mostly to how the journal will be disseminated to its readers and whether or not the page charges get indirectly transferred from the authors to the readers. For example, you would have to pay for a subscription to *MNRAS*, or your home institution would, but *ApJ* makes its articles more freely accessible to the general

public. With the Internet, though, these things are changing and evolving so quickly that new journals are always appearing, some test different models for how they intend to make money, some are purely predatorial, and so on.

Alice: Wow. It is a jungle out there, from the sounds of things. Getting our paper published sounds like it could be a nasty business if we don't know what we are doing.

Professor: Well, I recommend you write the paper before getting too ahead of ourselves with the submission process. Typically, it takes a lot of editing to get a paper ready for submission. You can download a LaTeX template for the journal of your choice from their website. Fill in the text, include any figures or tables you have made, and away you go! Plus, when you submit the paper, you are free to upload it to the arXiv,[1] which provides a daily posting of all astrophysics papers. So your work can get out to the community before it is even done with the peer-review process. I prefer to wait until the paper is accepted before posting it to the arXiv, but some people prefer posting it much earlier (and then replacing the old version on the arXiv once the paper gets accepted). It's up to you entirely.

Bob: We're practically published authors already!

Alice: Wow, Bob. Always the optimist. I'm excited to get started with the writing. We've already made a lot of figures that should be very useful. Mind if I take first stab at the writing?

Bob: Not at all! Once you've made whatever additions you have in mind, just send it over and I'll take a crack at it! In the meantime, I think I'll read a few published astrophysics papers to try to get a sense of how it is done.

Professor: Good idea, Bob. Writing publishable papers takes a lot of work and experience. I am of course more than happy to read your drafts, once you deem them ready for feedback.

Alice: That would be amazing! Thanks, Prof. Starmover. We're definitely going to take you up on that.

Bob: All right, dare I say, we're ready to write our first peer-reviewed publication. Alice, are you with me?

Alice: You had to make it creepy right at the end there, didn't you?

Bob: I … No?

Alice: I'm just messing with you. I'm all in!

Alice gives Bob a big hug. Bob returns the gesture. A friendship forged in fire. All that remains is to tell the tale.

[1] https://arxiv.org

15 Conclusions

Alice was careful enough to keep detailed notes about each part of the project, convinced that they will prove useful for future research projects. Her notes are a collection of key theoretical concepts and practical recommendations taken from Prof. Starmover's explanations and Alice and Bob's own experience.

15.1 Physics of the *N*-Body Problem

1. Newton's law of motion combined with Newton's law of gravitation yield the equations of motion for the N-body problem. They are given by

$$\frac{\mathrm{d}^2\mathbf{R}_i}{\mathrm{d}t^2} = -G\sum_{\substack{j=1\\ j\neq i}}^{N} m_j \frac{\mathbf{R}_i - \mathbf{R}_j}{\|\mathbf{R}_i - \mathbf{R}_j\|^3}, \tag{15.1}$$

   where m_i and $\mathbf{R}_i$ are the mass and position vector of body i, respectively.
2. The two-body problem can be solved analytically in closed form. The relative orbit is a closed ellipse (bound system with negative total energy), a parabola (unbound system with exactly zero total energy), or a hyperbola (unbound system with positive total energy).
3. Technically, the N-body problem has analytical solutions. However, these solutions consist of infinite series of time that converge extremely slowly and are not useful for any practical applications. Useful analytic solutions exist only in specific restricted cases (for example, when strong hierarchies exist within the system). Generally, one has to solve the equations for the N-body problem numerically using an N-body integrator.
4. There are generally ten conserved quantities in an N-body system: two constant vectors describing the position and velocity of the center of mass (six constants), the total angular momentum vector (three constants), and the total energy.

5. Newton's mechanics and Hamilton's equations are equivalent. However, in cases in which the "standard" Cartesian coordinates do not work well such as multiplanet systems, Hamilton's formulation has advantages.
6. Orbital elements are convenient coordinates to describe orbits, in particular in the case of multiplanet systems.
7. Laplace–Lagrange theory describes the secular (orbit-averaged) evolution of multiplanet systems in the limit of small eccentricities and inclinations. It gives a good description of the future evolution of the Solar System for order million-year time scales.

15.2 Programming Languages

1. Choosing a programming language is crucial for the subsequent development and maintenance of complex simulation software. PYTHON is a general-purpose programming language. It features a dynamic type system and an automatic memory management system. The source code is concise and highly readable. It has a large community of scientific users and a large collection of scientific computing libraries. For these reasons, we chose PYTHON to be our main programming language.
2. Due to the dynamic features, the PYTHON interpreter needs to perform checks to ensure that the data types are correct and the memory is properly managed in runtime, which leads to an overhead in performance. However, it is possible to boost the performance by offloading the computationally expensive routines in native libraries (written in C/C++ or FORTRAN, for example).
3. One can structure a complex code into three layers: a presentation layer that interacts with human users, a logic layer that consists of the algorithms of the code, and a data layer that implements the input/output facilities.
4. The fundamental idea of parallel computing is to divide a large computational problem into a set of (independent) small problems and subsequently use multiple processors to simultaneously execute these small problems. The results of the small problems are eventually collected and reduced into the result of the original large problem. Care should be taken to ensure proper communication and synchronization between different subtasks.
5. Traditionally, graphics processing units (GPUs) are dedicated hardware optimized for generating real-time computer graphics. Nowadays, GPUs also play an increasingly important role in general-purpose parallel computing. GPUs are particularly efficient in handling matrix manipulation and computational problems that can be divided into a large number of subtasks.

15.3 Numerical Integrators

15.3.1 Understanding Numerical Errors

1. High-order integrators reduce the truncation error but tend to be slower.
2. Checking how well the constants of motion (energy and angular momentum, for example) are preserved is a good technique for assessing the accuracy of the integration.
3. The size of the time step should be chosen according to the physics of the problem, typically as a fraction of the shortest orbital period.
4. For variable step-size integrators, recommended values of the tolerance range from 10^{-10} to 10^{-13}. Using a tolerance below that level increases the computational time and the round-off error grows rapidly, while a coarser tolerance speeds up the integration at the cost of increasing the truncation error.
5. The error in the conservation of the energy when using symplectic integrators remains bounded, whereas the error in conventional integrators builds over time.

15.3.2 Choosing the Integrator

1. Choosing different integrators can change the results and the runtime dramatically.
2. Symplectic integrators use a fixed time step. They are particularly useful for long-term propagation of conservative systems but may suffer when modeling close encounters.
3. Conventional integrators with automatic step-size control are more efficient than their fixed-step counterparts, and they are ideal for handling close encounters.
4. The Runge–Kutta–Fehlberg 4(5) and 7(8) methods are good choices for integrating generic problems, even outside of orbital mechanics.
5. The Gauss–Radau integrator of the fifteenth order is recommended for highly accurate integrations.
6. The Wisdom–Holman mapping is an attractive option for very long integrations of several millions of years, as long as the underlying assumptions about the dynamics apply to the problem at hand.

15.4 Research Project

1. The *Kepler* spacecraft has found planets orbiting around binary stars using the transit detection method. These circumbinary planets (CBPs) might host moons, and we investigated their dynamical stability in our research project.
2. To zeroth-order approximation, the stability region of moons around a CBP is given by the Hill radius, which quantifies the size of the "sphere of influence" of a small body (the CBP) orbiting a more massive body (the stellar binary, here interpreted as a point mass).

3. Considered more quantitatively, the stability regions for the *Kepler* systems are influenced by a number of effects.
   - If the orbit of the moon around the CBP is inclined relative to the orbit of the CBP around the stellar binary, then Lidov–Kozai (LK) oscillations can be induced in the orbit of the moon around the CBP. Consequently, the orbit of the moon can become highly eccentric, resulting in, for example, the tidal disruption of the moon by the planet. Highly inclined moons (close to 90° inclination) around the CBPs are, therefore, unlikely to exist.
   - In the three-body problem with a test particle orbiting a small body, which in turn is orbiting a more massive body, retrograde orbits of the test particle relative to the small body around the massive body tend to be more stable than prograde orbits. In our four-body problem, however, we found that there is typically no difference in terms of stability between prograde and retrograde orbits. This can be attributed to commensurabilities of the orbit of the moon around the CBP (orbital period of several days) with the orbit of the binary (period of several days).
4. We generated stability maps of moons around the currently known *Kepler* CBPs, thereby indicating in which parts of the parameter space moons could exist and which parts of parameter space are excluded based on dynamical stability arguments.

15.5 Publishing the Results

1. Making careful notes during the research saves a lot of time and effort when writing the research paper at the end.
2. Research papers take on a common general architecture consisting of an abstract, introduction, methods and results sections, and finally a summary of the main conclusions. Of course, this simple formula is not written in stone, and authors can take freedom in deciding the structure of their paper.

The code presented in this book is available on GitHub.[1]

[1] https://github.com/MovingPlanetsAround

Appendices

A Derivation of Kepler's Third Law and the Kepler Equation

On a very rainy day, Alice and Bob are at the university. Sometime shortly after lunch, the power goes out. No buses are operating due to the bad weather. Alice and Bob are stuck at the university. They are unable to use their computers; the Internet is down. To them, this seems like a good opportunity to visit Prof. Starmover's office to ask about the more detailed derivations she had promised earlier.

A.1 The Kepler Equation

Alice: Prof. Starmover, can you tell us how to derive Kepler's third law and the Kepler equation?

Professor: Sure! I could certainly use a break from grading midterm exams. Let us first consider closed orbits ($0 \le e < 1$) and compute the time needed to complete a full revolution, that is, the orbital period P. In polar coordinates, we can write the position vector as

$$\mathbf{r} = r\,\hat{\mathbf{r}}, \tag{A.1}$$

where $\hat{\mathbf{r}}$ points in the direction of $\mathbf{r}$. Differentiating this with respect to time, we get

$$\dot{\mathbf{r}} = \dot{r}\,\hat{\mathbf{r}} + r\dot{\hat{\mathbf{r}}}. \tag{A.2}$$

We can relate $\dot{\hat{\mathbf{r}}}$, the time derivative of the unit vector of $\mathbf{r}$, to the angle θ by writing $\hat{\mathbf{r}}$ in terms of the unit vectors $\hat{\mathbf{x}}$ and $\hat{\mathbf{y}}$ of some coordinate frame, similarly to equation (2.38),

$$\hat{\mathbf{r}} = \cos(\theta)\,\hat{\mathbf{x}} + \sin(\theta)\,\hat{\mathbf{y}}. \tag{A.3}$$

Let us also introduce the unit vector $\hat{\boldsymbol{\theta}}$ that points in the direction of increasing θ. Looking at figure A.1, you can convince yourself that $\hat{\boldsymbol{\theta}} = \hat{\mathbf{z}} \times \hat{\mathbf{r}}$; therefore,

$$\hat{\boldsymbol{\theta}} = \hat{\mathbf{z}} \times \left[\cos(\theta)\,\hat{\mathbf{x}} + \sin(\theta)\,\hat{\mathbf{y}}\right] = -\sin(\theta)\,\hat{\mathbf{x}} + \cos(\theta)\,\hat{\mathbf{y}}. \tag{A.4}$$

Now we can easily compute

$$\dot{\hat{\mathbf{r}}} = -\sin(\theta)\dot{\theta}\,\hat{\mathbf{x}} + \cos(\theta)\dot{\theta}\,\hat{\mathbf{y}} = \dot{\theta}\left[-\sin(\theta)\,\hat{\mathbf{x}} + \cos(\theta)\,\hat{\mathbf{y}}\right] = \dot{\theta}\,\hat{\boldsymbol{\theta}} \tag{A.5}$$

(note that $\hat{\mathbf{x}}$ and $\hat{\mathbf{y}}$ are constant). Substituting this into equation (A.2), we get

$$\dot{\mathbf{r}} = \dot{r}\,\hat{\mathbf{r}} + r\dot{\theta}\,\hat{\boldsymbol{\theta}}. \tag{A.6}$$

Now we can compute the angular momentum vector $\mathbf{k}$ in terms of r and θ:

$$\mathbf{k} \equiv \mathbf{r} \times \dot{\mathbf{r}} = r\,\hat{\mathbf{r}} \times \left(\dot{r}\,\hat{\mathbf{r}} + r\dot{\theta}\,\hat{\boldsymbol{\theta}}\right) = r^2\dot{\theta}\,\hat{\mathbf{z}}. \tag{A.7}$$

Here, I used that $\hat{\mathbf{r}} \times \hat{\mathbf{r}} = \mathbf{0}$, and $\hat{\mathbf{r}} \times \hat{\boldsymbol{\theta}} = \hat{\mathbf{z}}$.

Now, let us look at a small piece of area swept out in the polar plane when increasing θ from θ to $\theta + \mathrm{d}\theta$. The associated area is given by

$$\mathrm{d}A = \frac{1}{2}r^2\,\mathrm{d}\theta. \tag{A.8}$$

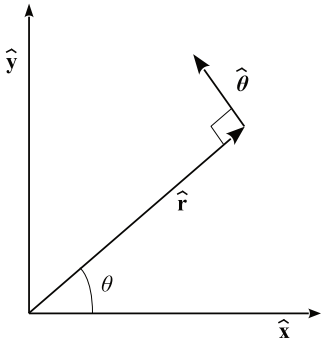


Figure A.1
An illustration of the unit vectors $\hat{\mathbf{r}}$ and $\hat{\boldsymbol{\theta}}$ in polar coordinates.

This means that the rate of change of the area with time is given by

$$\dot{A} = \frac{1}{2} r^2 \dot{\theta} = \frac{1}{2} k \tag{A.9}$$

(see equation A.7). Since $\dot{A} = \mathrm{d}A/\mathrm{d}t$, we can integrate both sides of this equation to get

$$\int_{\text{ellipse}} \mathrm{d}A = \frac{1}{2} k \int_0^P \mathrm{d}t. \tag{A.10}$$

I am integrating over a full orbital revolution, meaning that the left-hand side is just the area of an ellipse, and the right-hand side is the orbital period.

Alice: Ah, but we know the area of an ellipse: it is πab, where a and b are the major and minor axes, respectively.

Professor: Yes. The major axis is simply the orbital semimajor axis a, whereas the minor axis is given by $a\sqrt{1-e^2}$. Therefore,

$$\pi a^2 \sqrt{1-e^2} = \frac{1}{2} kP = \frac{1}{2} \sqrt{GMa(1-e^2)}\, P, \tag{A.11}$$

where I used equation (2.42), and so

$$P = 2\pi \sqrt{\frac{a^3}{GM}}, \tag{A.12}$$

which is Kepler's third law. Note that the orbital period does not depend on the eccentricity. I already mentioned a related quantity to the orbital period is the mean motion n,

$$n \equiv \frac{2\pi}{P} = \sqrt{\frac{GM}{a^3}}. \tag{A.13}$$

In terms of n, we can write $k = na^2\sqrt{1-e^2}$.

Bob: Do you guys remember when Alice said, "Let's get started with our own code!"? We were so close to coding….

Alice: We'll get there, Bob. Please continue, Prof. Starmover.

Professor: To derive the Kepler equation, we need to use geometry. In figure A.2, I show an ellipse. In addition, I draw a circle with radius a centered at point C, where C lies at the middle of the major axis of the ellipse. Note that our origin is at the focal point F. Consider a point P on the ellipse, and draw a vertical line through P. This line intersects the major axis at a point Q and the circle at a point P'. We now introduce the eccentric anomaly, $\mathcal{E}$ (not to be confused with the energy E), defined as the angle between the major axis (or CQ) and CP'. From the figure, we see that

$$|FQ| = r\cos(\theta) = |CQ| - |CF| = a\cos(\mathcal{E}) - ae, \tag{A.14}$$

where I used that $|CF| = ae$; since $|CB| = a$ and $|FB| = a(1-e)$, $|CF| = |CB| - |FB| = ae$. Plugging in equation (2.41) for r as a function of θ, we can solve for the eccentric anomaly in terms of the true anomaly θ or the true anomaly

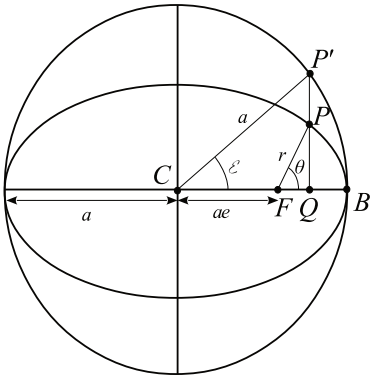


Figure A.2
The relation between the true anomaly θ and the eccentric anomaly $\mathcal{E}$ for a Kepler orbit. The focal point closest to periapsis is F, and P is a point on the ellipse. P' is the vertical extension of P on a greater circle with radius a centered at the (geometric) center of the ellipse, C. Q is the projection of P on the major axis of the ellipse.

in terms of the eccentric anomaly,

$$\cos(\mathcal{E}) = \frac{\cos(\theta)+e}{1+e\cos(\theta)}; \qquad \cos(\theta) = \frac{\cos(\mathcal{E})-e}{1-e\cos(\mathcal{E})}. \tag{A.15}$$

By using similar relations for the sine of θ and $\mathcal{E}$, I will leave it up to you to show that

$$\sin(\mathcal{E}) = \sqrt{1-e^2}\,\frac{\sin(\theta)}{1+e\cos(\theta)}; \qquad \sin(\theta) = \sqrt{1-e^2}\,\frac{\sin(\mathcal{E})}{1-e\cos(\mathcal{E})}. \tag{A.16}$$

Using the expression for $\cos(\theta)$ from equation (A.15) and using equation (2.41) for $r(\theta)$, we can write r in terms of $\mathcal{E}$ only:

$$r = a[1-e\cos(\mathcal{E})]. \tag{A.17}$$

If we now differentiate equation (A.17) with respect to time, we get

$$\dot{r} = ae\sin(\mathcal{E})\,\dot{\mathcal{E}}. \tag{A.18}$$

We can do the same with equation (2.41) to get

$$\dot{r} = e\sin(\theta)\frac{r^2\dot{\theta}}{a(1-e^2)} = e\sin(\theta)\frac{k}{a(1-e^2)} = \frac{e\sin(\theta)an}{\sqrt{1-e^2}}, \tag{A.19}$$

where I used that $k = r^2\dot{\theta} = na^2\sqrt{1-e^2}$. By equating (A.18) and (A.19), we can find an expression for $\dot{\mathcal{E}}$:

$$\dot{\mathcal{E}} = \frac{n}{1-e\cos(\mathcal{E})}, \tag{A.20}$$

and rearranging and integrating this equation gives

$$\int_0^{\mathcal{E}} \mathrm{d}\mathcal{E}'\,[1-e\cos(\mathcal{E}')] = n\int_\tau^t \mathrm{d}t'. \tag{A.21}$$

Here, I am assuming that $t = \tau$ at the periapsis distance $E = 0$. Therefore, we get

$$\mathcal{E} - e\sin(\mathcal{E}) = n(t-\tau), \tag{A.22}$$

which is the Kepler equation. The right-hand side of equation (A.22) is also known as the mean anomaly,

$$\mathcal{M} \equiv n(t-\tau). \tag{A.23}$$

The mean anomaly increases linearly with time. Unfortunately, the Kepler equation cannot be solved analytically.

Bob: How can we solve the Kepler equation, in practice? Please tell me that the answer is computers.

Professor: Yes, we need computers to solve the equation numerically.

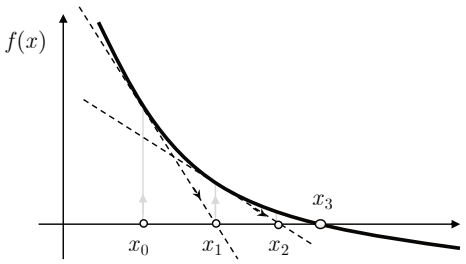


Figure A.3
Newton–Raphson method.

Bob: Yes! Score one for computers!

Professor: There are various ways to solve equation (A.22) numerically for $\mathcal{E}$. One way is to use the Newton–Raphson method: let $f(x)$ be a function of x; we are interested in the value of x for which $f(x) = 0$. Let x_0 be an initial guess of this value. We will advance along the derivative of $f(x)$ at x_0 (the line tangent to $f(x)$ at x_0) until we intersect the horizontal axis. The intersection point defines x_1. We repeat the process by advancing along the derivative at x_1 until we intersect the horizontal axis at x_2, then at x_3, and so on. Figure A.3 sketches how the method works.

Mathematically, the equation of the line tangent to $f(x)$ at a given x_n is written in terms of the derivative $f'(x_n)$ like

$$y(x) = f(x_n) + f'(x_n)(x - x_n). \tag{A.24}$$

Making $y(x_{n+1}) = 0$, we can solve for the next intersection point, x_{n+1}. Then we can iteratively find new values of x that better approximate the value for which $f(x) = 0$ by applying the recursion relation

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}, \tag{A.25}$$

where the prime denotes the derivative with respect to x. If we apply this technique to the function $f(\mathcal{E}) = \mathcal{E} - e\sin(\mathcal{E}) - \mathcal{M}$, we can solve the Kepler equation numerically by iteratively computing

$$\mathcal{E}_{n+1} = \mathcal{E}_n - \frac{\mathcal{E}_n - e\sin(\mathcal{E}_n) - \mathcal{M}}{1 - e\cos(\mathcal{E}_n)}. \tag{A.26}$$

In practice, one can take $\mathcal{E}_0 = \mathcal{M}$ as the initial guess. In fact, if $e = 0$, then the solution is trivial and exact: $\mathcal{E} = \mathcal{M}$. The iteration is stopped if $\mathcal{E}_{n+1}$ does not differ more from $\mathcal{E}_n$ by some specified tolerance (say, 10^{-12}). Typically, only a few iterations are necessary to find $\mathcal{E}$, so the algorithm is very fast.

A.2 Conserved Quantities in the *N*-Body Problem

Alice: Prof. Starmover, you mentioned that there generally are ten conserved quantities in the N-body problem. Can you tell us how these conserved quantities are derived?

Professor: Sure. Let us start by introducing the potential. The potential V is a function such that the force can be computed from the negative spatial derivative of V. Or, more precisely, the negative gradient, $-\nabla V$:

$$\mathbf{F}_i = -\nabla_i V. \tag{A.27}$$

Bob: Wait a moment, I am a bit unsure about this gradient operator, ∇. How is it defined?

Professor: Generally, the gradient in Cartesian coordinates is given by

$$\nabla = \frac{\partial}{\partial x}\hat{\mathbf{x}} + \frac{\partial}{\partial y}\hat{\mathbf{y}} + \frac{\partial}{\partial z}\hat{\mathbf{z}}. \tag{A.28}$$

So it is a vector whose components are given by the spatial derivatives. If we were to take the gradient in non-Cartesian coordinates, we would have to take into account that the expression for ∇ will be more complicated. The subscript i in ∇ just means that we should take the gradient with respect to the coordinates of body i.

Let me assert that V for the N-body problem is given by

$$V = -\frac{1}{2}G\sum_{j=1}^{N}\sum_{\substack{k=1\\k\neq j}}^{N}\frac{m_j m_k}{\|\mathbf{R}_j - \mathbf{R}_k\|}. \tag{A.29}$$

Let us compute the gradient of V to check if our expression is correct:

$$\nabla_i V = -\frac{1}{2}G\sum_{j=1}^{N}\sum_{\substack{k=1\\k\neq j}}^{N} m_j m_k \nabla_i\left(\frac{1}{\|\mathbf{R}_j - \mathbf{R}_k\|}\right). \tag{A.30}$$

Now, in equation (A.30), note that only the terms with $j = i$ or $k = i$ depend on the coordinates of body i. Therefore, only these terms remain after taking the gradient, and so

$$\begin{aligned}\nabla_i V &= -\frac{1}{2}G\sum_{\substack{k=1\\k\neq i}}^{N} m_i m_k \nabla_i\left(\frac{1}{\|\mathbf{R}_i - \mathbf{R}_k\|}\right) - \frac{1}{2}G\sum_{\substack{j=1\\j\neq i}}^{N} m_j m_i \nabla_i\left(\frac{1}{\|\mathbf{R}_j - \mathbf{R}_i\|}\right)\\ &= -G\sum_{\substack{j=1\\j\neq i}}^{N} m_i m_j \nabla_i\left(\frac{1}{\|\mathbf{R}_i - \mathbf{R}_j\|}\right).\end{aligned} \tag{A.31}$$

Notice that the two summations in V are exactly the same if we relabel the summation variables, and therefore we get a single summation without the factor $1/2$. Also, note that the ordering of the indices in the expression $\|\mathbf{R}_i - \mathbf{R}_k\|$ does not matter, since we are interested only in the distance between the vectors $\mathbf{R}_i$ and $\mathbf{R}_k$. What is left is the gradient of $1/r_{ij}$. I will leave it up to you to check that

$$\nabla_i\left(\frac{1}{\|\mathbf{R}_i - \mathbf{R}_j\|}\right) = -\frac{\mathbf{R}_i - \mathbf{R}_j}{\|\mathbf{R}_i - \mathbf{R}_j\|^3}. \tag{A.32}$$

Plugging this into equation (A.31), we find

$$\nabla_i V = m_i G\sum_{\substack{j=1\\j\neq i}}^{N} m_j \frac{\mathbf{R}_i - \mathbf{R}_j}{\|\mathbf{R}_i - \mathbf{R}_j\|^3}. \tag{A.33}$$

If we compare this with equation (2.16) for the acceleration, we conclude that

$$\mathbf{F}_i = m_i\ddot{\mathbf{R}}_i = -Gm_i\sum_{\substack{j=1\\j\neq i}}^{N} m_j \frac{\mathbf{R}_i - \mathbf{R}_j}{\|\mathbf{R}_i - \mathbf{R}_j\|^3} = -\nabla_i V, \tag{A.34}$$

showing that V is indeed the potential corresponding to equation (2.16).

I have already talked about the kinetic energy for the two-body problem. Let us now consider it for the N-body problem. A neat way to show that the total energy E is constant is to compute the vector dot product of the force of body i with its velocity $\dot{\mathbf{R}}_i$ and to sum this quantity over all bodies. In mathematical terms:

$$\sum_{i=1}^{N} m_i\left(\dot{\mathbf{R}}_i \cdot \ddot{\mathbf{R}}_i\right) = -\sum_{i=1}^{N}\left(\dot{\mathbf{R}}_i \cdot \nabla_i V\right). \tag{A.35}$$

Let us write down explicitly what the latter term means in Cartesian coordinates:

$$-\sum_{i=1}^{N}\left(\dot{\mathbf{R}}_i \cdot \nabla_i V\right) = -\sum_{i=1}^{N}\left(\dot{x}_i\frac{\partial V}{\partial x_i} + \dot{y}_i\frac{\partial V}{\partial y_i} + \dot{z}_i\frac{\partial V}{\partial z_i}\right) = -\sum_{i=1}^{N}\left(\frac{\partial x_i}{\partial t}\frac{\partial V}{\partial x_i} + \frac{\partial y_i}{\partial t}\frac{\partial V}{\partial y_i} + \frac{\partial z_i}{\partial t}\frac{\partial V}{\partial z_i}\right). \tag{A.36}$$

Note that V is a function of the positions of all the bodies. You may recognize the last expression as just the total time derivative of V, usually denoted by $\mathrm{d}V/\mathrm{d}t$ (using "d" to indicate the total derivative, rather than "∂" for the partial derivative). So we simply get

$$\sum_{i=1}^{N} m_i\left(\dot{\mathbf{R}}_i \cdot \ddot{\mathbf{R}}_i\right) = -\frac{\mathrm{d}V}{\mathrm{d}t}. \tag{A.37}$$

If we differentiate a related expression on the left-hand side,

$$\frac{\mathrm{d}}{\mathrm{d}t}\left(\dot{\mathbf{R}}_i \cdot \dot{\mathbf{R}}_i\right) = \ddot{\mathbf{R}}_i \cdot \dot{\mathbf{R}}_i + \dot{\mathbf{R}}_i \cdot \ddot{\mathbf{R}}_i = 2\,\dot{\mathbf{R}}_i \cdot \ddot{\mathbf{R}}_i, \tag{A.38}$$

we can use this to write equation (A.37) as

$$\frac{\mathrm{d}}{\mathrm{d}t}\left[\frac{1}{2}\sum_{i=1}^{N} m_i\left(\dot{\mathbf{R}}_i \cdot \dot{\mathbf{R}}_i\right) + V\right] = 0. \tag{A.39}$$

The first terms constitute the kinetic energy T,

$$T \equiv \frac{1}{2}\sum_{i=1}^{N} m_i\left(\dot{\mathbf{R}}_i \cdot \dot{\mathbf{R}}_i\right), \tag{A.40}$$

so we get

$$\frac{\mathrm{d}}{\mathrm{d}t}(T+V) = 0, \tag{A.41}$$

implying that the total energy, which we define as the sum of the kinetic and potential energies, $E = T + V$, is constant.

Alice: What about the total angular momentum?

Professor: The total angular momentum is conserved both in magnitude and direction. We can show the constancy of the total-angular momentum vector explicitly by computing the time derivative:

$$\dot{\mathbf{L}} = \frac{\mathrm{d}}{\mathrm{d}t}\left(\sum_{i=1}^{N} m_i \mathbf{R}_i \times \dot{\mathbf{R}}_i\right) = \sum_{i=1}^{N} m_i\left(\dot{\mathbf{R}}_i \times \dot{\mathbf{R}}_i + \mathbf{R}_i \times \ddot{\mathbf{R}}_i\right) = -G\sum_{i=1}^{N} m_i \sum_{\substack{j=1 \\ j\neq i}}^{N} m_j \frac{\mathbf{R}_i \times \left(\mathbf{R}_i - \mathbf{R}_j\right)}{\|\mathbf{R}_i - \mathbf{R}_j\|^3}$$

$$= -G\sum_{i=1}^{N}\sum_{\substack{j=1 \\ j\neq i}}^{N} m_i m_j \frac{\mathbf{R}_i \times \mathbf{R}_j}{\|\mathbf{R}_i - \mathbf{R}_j\|^3} = \mathbf{0}. \tag{A.42}$$

Here, I substituted equation (2.16) for the acceleration. The last step follows from the fact that terms cancel pairwise: for each term corresponding to $\mathbf{R}_i \times \mathbf{R}_j$, there is a term corresponding to $\mathbf{R}_j \times \mathbf{R}_i = -\mathbf{R}_i \times \mathbf{R}_j$. The denominator is $\|\mathbf{R}_i - \mathbf{R}_j\|^3 = \|\mathbf{R}_j - \mathbf{R}_i\|^3$, and of course, $m_i m_j = m_j m_i$.

Finally, there are two more conserved quantities related to the center of mass. We can obtain them by summing over all forces:

$$\sum_{i=1}^{N} m_i \ddot{\mathbf{R}}_i = -G\sum_{i=1}^{N}\sum_{\substack{j=1 \\ j\neq i}}^{N} m_i m_j \frac{\mathbf{R}_i - \mathbf{R}_j}{\|\mathbf{R}_i - \mathbf{R}_j\|^3} = \mathbf{0}. \tag{A.43}$$

As before, pairwise terms cancel. Another way of looking at this is that the net force on the entire system should be zero. If we integrate equation (A.43) twice with respect to time, we get

$$\sum_{i=1}^{N} m_i \mathbf{R}_i = \mathbf{a}t + \mathbf{b}, \tag{A.44}$$

where $\mathbf{a}$ and $\mathbf{b}$ are constant vectors. The definition of the center of mass is

$$\mathbf{r}_{\mathrm{CM}} \equiv \frac{\sum_{i=1}^{N} m_i \mathbf{R}_i}{M}, \tag{A.45}$$

where

$$M = \sum_{i=1}^{N} m_i \tag{A.46}$$

is the total mass. So equation (A.44) states

$$\mathbf{r}_{\mathrm{CM}} = \frac{\mathbf{a}t + \mathbf{b}}{M}, \tag{A.47}$$

that is, the center of mass position moves at constant velocity.

B Keplerian Propagator in Universal Variables

Professor: While you are here, I would like to explain to you an alternative way to propagate Keplerian orbits. Given the orbital elements, we can use the equations in section 2.5.2.1 to obtain the position and velocity vectors as a function of true anomaly and then solve Kepler's equation (2.43) to find the value of the anomaly at time t.

Instead of the traditional orbital elements, we will now use the Gauss f and g functions, which allow us to write the solution to the Kepler problem like

$$\begin{aligned} \mathbf{r}(t) &= f(t)\mathbf{r}_0 + g(t)\mathbf{v}_0 \\ \mathbf{v}(t) &= \dot{f}(t)\mathbf{r}_0 + \dot{g}(t)\mathbf{v}_0. \end{aligned} \tag{B.1}$$

To know the position and velocity at t_f, we just need to compute the value of $f(t_f)$, $g(t_f)$, and their derivatives. We will define these functions using *universal variables*, which unify all three cases (elliptic, parabolic, and hyperbolic) under the same formulation. We can achieve this by introducing the universal variable s, which is valid for all orbit types. In the literature—for example, the works by Herrick (1965), Everhart and Pitkin (1983), Burkardt and Danby (1983), Danby (1992, pp. 162–178), Vallado (1997, pp. 224–230), or Battin (1999, pp. 174–182)—you might find slightly different derivations, but the idea is essentially the same.

Alice: I like the form of equation (B.1) because it is much more compact than the conversions from orbital elements to Cartesian coordinates and vice versa. The question is, how can we compute the f and g functions?

Professor: The f and g functions can be written in terms of the new independent variable s like

$$\begin{aligned} f &= 1 - \frac{GM}{r_0} s^2 c_2(\alpha s^2), & g &= (t - t_0) - GMs^3 c_3(\alpha s^2) \\ \dot{f} &= -\frac{GM}{r r_0} s c_1(\alpha s^2), & \dot{g} &= 1 - \frac{GM}{r} s^2 c_2(\alpha s^2). \end{aligned} \tag{B.2}$$

I have introduced three special functions, $c_1(\alpha s^2)$, $c_2(\alpha s^2)$, and $c_3(\alpha s^2)$. These are the *Stumpff functions* and the parameter α reads

$$\alpha = \frac{GM}{a}. \tag{B.3}$$

Recall that G is the gravitational constant and $M = m_1 + m_2$ is the total mass.

Bob: Great. First, we had two special functions, f and g, and then we defined them in terms of another three special functions. We didn't advance much!

B.1 Stumpff Functions

Professor: Well, the key improvement is that the Stumpff functions can be computed in a pretty straightforward manner using the following nested expressions:

$$c_3(z) = \frac{1}{6}\Big(1 - \frac{z}{20}\Big(1 - \frac{z}{42}\Big(1 - \frac{z}{72}\Big(1 - \frac{z}{110}\Big(1 - \frac{z}{156}\Big(1 - \frac{z}{210}\Big)\Big)\Big)\Big)\Big)\Big) \tag{B.4a}$$

$$c_2(z) = \frac{1}{2}\Big(1 - \frac{z}{12}\Big(1 - \frac{z}{30}\Big(1 - \frac{z}{56}\Big(1 - \frac{z}{90}\Big(1 - \frac{z}{132}\Big(1 - \frac{z}{182}\Big)\Big)\Big)\Big)\Big)\Big) \tag{B.4b}$$

$$c_1(z) = 1 - zc_3 \tag{B.4c}$$

$$c_0(z) = 1 - zc_2. \tag{B.4d}$$

There's one important detail to remember when dealing with the Stumpff functions. If their argument z is large, then the solution in equation (B.4) becomes inaccurate. The trick is to reduce the value of z by dividing it by four several times, then compute the Stumpff functions for the reduced value of z, and finally recover the actual solution using the half-angle formulae,

$$c_0(4z) = 2c_0^2(z) - 1 \tag{B.5a}$$

$$c_1(4z) = c_0(z)c_1(z) \tag{B.5b}$$

$$c_2(4z) = \frac{1}{2}c_1^2(z) \tag{B.5c}$$

$$c_3(4z) = \frac{1}{4}[c_2(z) + c_0(z)c_3(z)]. \tag{B.5d}$$

Why don't you write a function to compute the value of the Stumpff functions for a given z?

Alice: Let me get into that…. Okay, I think snippet B.1 will do. I took a look at one of the references you provided, in particular Danby (1992, pp. 173), to check the algorithm and to see how much I should reduce the argument at first.

Snippet B.1

Function `stumpff_functions`: Evaluate the first four Stumpff functions.

```
def stumpff_functions( z ):

    # Reduce the argument:
    n = 0
    while ( abs(z) > 0.1 ):
        n += 1
        z /= 4.0

    # Compute c3, c2, c1, c0:
    c3 = (1. - z * (1. - z * (1. - z * (1. - z * (1. - z * (1. - z / 210.)
          /
      156.) / 110.) / 72.) / 42.) / 20.) / 6.
    c2 = (1. - z * (1. - z * (1. - z * (1. - z * (1. - z * (1. - z / 182.)
          /
      132.) / 90.) / 56.) / 30.) / 12.) / 2.
    c1 = 1.0 - z * c3
    c0 = 1.0 - z * c2

    # Half-angle formulae to recover the actual argument:
    while ( n > 0 ):
        n -= 1
        c3 = (c2 + c0 * c3) / 4.
        c2 = c1 * c1 / 2.
        c1 = c0 * c1
```

```
        c0 = 2. * c0 * c0 - 1.

    return c0, c1, c2, c3
```

Bob: Ah, that's cheating!

Professor: I'm actually glad that you checked that reference. That's what books and papers are for! You can save a lot of time using formulae and algorithms developed by previous researchers. We cannot reinvent the wheel every time we face a problem.

Let's check your function before going on. It turns out that the Stumpff functions admit analytic solutions that depend on the sign of z:

$$c_0(z) = \begin{cases} \cos\sqrt{z}, & z > 0 \\ \cosh\sqrt{-z}, & z < 0 \\ 1, & z = 0 \end{cases} \qquad c_1(z) = \begin{cases} \dfrac{\sin\sqrt{z}}{\sqrt{z}}, & z > 0 \\ \dfrac{\sinh\sqrt{-z}}{\sqrt{-z}}, & z < 0 \\ 1, & z = 0 \end{cases}$$

$$c_2(z) = \begin{cases} \dfrac{1-\cos\sqrt{z}}{z}, & z > 0 \\ \dfrac{1-\cosh\sqrt{-z}}{z}, & z < 0 \\ 1/2, & z = 0 \end{cases} \qquad c_3(z) = \begin{cases} \dfrac{\sqrt{z}-\sin\sqrt{z}}{z^{3/2}}, & z > 0 \\ \dfrac{\sqrt{-z}-\sinh\sqrt{z}}{z\sqrt{-z}}, & z < 0 \\ 1/6, & z = 0 \end{cases} \tag{B.6}$$

Bob: Okay, let me run our function first for three different values of z:

```
 >>> print(stumpff_functions(30.0))
(0.6924191115937441, -0.13172645569509198, 0.010252696280208544,
    0.03772421518983646)
>>> print(stumpff_functions(-30.0))
(119.59318692388037, 21.83386540721416, 3.9531062307960405,
    0.6944621802404771)
>>> print(stumpff_functions(0.0))
(1.0, 1.0, 0.5, 0.16666666666666666)
```

And then I will evaluate equation (B.6):

```
 (0.6924191115937478, -0.13172645569509123, 0.010252696280208407,
     0.037724215189836374)
(119.59318692388277, 21.833865407214518, 3.9531062307960925,
    0.6944621802404839)
(1.0, 1.0, 0.5, 0.16666666666666666)
```

Looks good, I got the same results! The differences are of the order of the machine zero.

B.2 Solving the Universal Kepler Equation

Alice: With this function, we can now find c_1, c_2, and c_3. We needed them to evaluate equation (B.2). But I checked the equation and we are still missing the value of r, and we don't know the value of the independent variable s either.

Professor: You are right, Alice. We need to find the value of s from the final time t. That's the same as we did in section 2.3.3 when we figured out how to solve for the eccentric anomaly $\mathcal{E}$ at a given time t. We arrived at Kepler's

equation (2.43). When working with universal variables, there is an equivalent *universal Kepler's equation*:

$$t - t_0 = r_0 s c_1(\alpha s^2) + r_0 \dot{r}_0 s^2 c_2(\alpha s^2) + GM s^3 c_3(\alpha s^2). \tag{B.7}$$

This is equivalent to finding the value of s that makes the following function $F(s)$ equal to zero:

$$F(s) \equiv r_0 s c_1(\alpha s^2) + r_0 \dot{r}_0 s^2 c_2(\alpha s^2) + GM s^3 c_3(\alpha s^2) - (t - t_0) = 0. \tag{B.8}$$

Bob: I know how to do that! We can use the Newton–Raphson method using the expressions in appendix A. In particular, equation (A.25) tells us how to advance step by step until we find the right value of s.

Alice: That means that we need to find the derivative of equation (B.8) with respect to s. How can we compute the derivatives of the Stumpff functions?

Professor: You will need the following two identities that hold for $i \geq 0$ (Danby, 1992, p. 174):

$$\frac{\mathrm{d}}{\mathrm{d}s}[c_0(\alpha s^2)] = -\alpha s c_1(\alpha s^2), \qquad \frac{\mathrm{d}}{\mathrm{d}s}[s^{i+1} c_{i+1}(\alpha s^2)] = s^i c_i(\alpha s^2). \tag{B.9}$$

Alice: Thanks. The derivative of Kepler's equation then takes the form

$$F'(s) = r_0 c_0(\alpha s^2) + r_0 \dot{r}_0 s c_1(\alpha s^2) + GM s^2 c_2(\alpha s^2). \tag{B.10}$$

Professor: Correct. And I have good news for you; the radial distance at s, $r(s)$, coincides exactly with $F'(s)$. So you don't need to compute anything else, just

$$r(s) \equiv F'(s) = r_0 c_0(\alpha s^2) + r_0 \dot{r}_0 s c_1(\alpha s^2) + GM s^2 c_2(\alpha s^2). \tag{B.11}$$

You can now implement the two-body propagator.

Alice: Hmm, and what about the initial guess? What value of s should we use to initialize the method?

Professor: As long as the propagation is short (small Δt), we can estimate s_0 like

$$s_0 = \frac{\Delta t}{r_0}. \tag{B.12}$$

For longer arcs, you can find more accurate initial guesses in the references I provided.

Bob: Every time I see tricks like this I wonder … is this the only way to solve the problem?

Professor: Not at all! To give you an idea of how many methods there are out there, you can take a look at the book by Colwell (1993). He reviews algorithms to solve this equation that go back to the seventeenth century! I recommend that you check the work of Danby and Burkardt (1983), Conway (1986), Mikkola (1987), Fukushima (1999), Mortari and Elipe (2014), and Raposo–Pulido and Peláez (2017) to be aware of more recent developments.

B.3 Two-Body Propagator

Alice: Finally! I'm going to start writing the function `propagate_kepler` in snippet B.2. The inputs will be the initial and final times, `t0` and `tf`; the initial position and velocity vectors, `vr0` and `vv0`; and the gravitational parameter `gm`. First, I will compute some of the parameters required in equations (B.2), (B.3), (B.8), and (B.10).

Let's start with the time step $\Delta t = t_f - t_0$:

```
 # Compute time step:
dt = tf - t0
```

We also need magnitudes of the initial position and velocity vectors, r_0 and v_0. Oh, and the radial velocity too, $\dot{r}_0$. That is the projection of the velocity in the direction of $\mathbf{r}$, isn't it?

Professor: Exactly.

Alice: Okay, then we can compute it like

$$\dot{r}_0 = \mathbf{v}_0 \cdot \mathbf{r}_0 / r_0, \tag{B.13}$$

resulting in the following piece of code:

```
 # Compute the magnitude of the initial position and velocity vectors:
r0 = np.linalg.norm(vr0)
v0 = np.linalg.norm(vv0)
```

```
# Radial velocity:
dr0 = np.dot(vr0, vv0) / r0
```

Bob: We also need to set the tolerance to solve Kepler's equation using the Newton–Raphson method.

```
 # Internal tolerance for solving Kepler's equation:
tol = 1e-12
```

Alice: Oh, thanks, Bob. That's right. The only parameter left is α. Its definition in equation (B.3) can be combined with equations (2.61–2.62),

$$\frac{v_0^2}{2} - \frac{GM}{r_0} = -\frac{GM}{2a} = -\frac{\alpha}{2} \implies \alpha = \frac{2GM}{r_0} - v_0^2. \tag{B.14}$$

```
 # Parameter alpha:
alpha = 2.0 * gm / r0 - v0**2
```

Bob: Now we need to solve Kepler's equation using the Newton–Raphson method. To advance s using equation (A.25), we need to evaluate $F(s)$ and $F'(s)$, and for that we have to implement equations (B.8) and (B.10). After advancing one step from s_i to $s_{i+1} = s_i + \Delta s$, we will check if either $|F(s)|$ or $|\Delta s|$ is smaller than the tolerance.

```
 # Solve Kepler's equation:
s = dt / r0
for j in range(0, 50):

    # Compute Stumpff functions:
    c0, c1, c2, c3 = stumpff_functions(alpha * s**2)

    # Evaluate Kepler's equation:
    F = r0 * s * c1 + r0 * dr0 * s**2 * c2 + gm * s**3 * c3 - dt

    # Convergence, check function value:
    if ( abs(F) < tol ):
        break

    # Compute derivative:
    dF = r0 * c0 + r0 * dr0 * s * c1 + gm * s**2 * c2

    # Find step size:
    ds = -F / dF

    # Convergence, check size of next step:
    if ( abs(ds) < tol ):
        break

    # Advance step:
    s += ds
```

Alice: I like it! Now that we know s, we can evaluate the f and g functions in equation (B.2) using the definition of r from equation (B.11). The function is complete! Take a look at snippet B.2.

Bob: Let me add a neat trick at the very beginning: if we call the function with $t_f = t_0$, we will return immediately the initial conditions without any computation.

Snippet B.2
Function `propagate_kepler`: Analytic propagation of Kepler's problem in universal variables.

```
def propagate_kepler( t0, tf, vr0, vv0, gm ):

    # Check for trivial propagation:
    if ( t0 == tf ):
        vrf = vr0
        vvf = vv0
        return

    # Compute time step:
    dt = tf - t0

    # Internal tolerance for solving Kepler's equation:
    tol = 1e-12

    # Compute the magnitude of the initial position and velocity vectors:
    r0 = np.linalg.norm(vr0)
    v0 = np.linalg.norm(vv0)

    # Parameter alpha:
    alpha = -(v0**2 - 2.0 * gm / r0)

    # Radial velocity:
    dr0 = np.dot(vr0, vv0) / r0

    # Solve Kepler's equation:
    s = dt / r0
    for j in range(0, 50):

        # Compute Stumpff functions:
        c0, c1, c2, c3 = stumpff_functions(alpha * s**2)

        # Evaluate Kepler's equation:
        F = r0 * s * c1 + r0 * dr0 * s**2 * c2 + gm * s**3 * c3 - dt

        # Convergence, check function value:
        if ( abs(F) < tol ):
            break

        # Compute derivative:
        dF = r0 * c0 + r0 * dr0 * s * c1 + gm * s**2 * c2

        # Find step size:
        ds = -F / dF

        # Convergence, check size of next step:
        if ( abs(ds) < tol ):
            break

        # Advance step:
        s += ds

    # The radial distance is equal to the derivative of F:
```

```
    r = dF

    # Evaluate f and g functions:
    f = 1.0 - gm * s**2 * c2 / r0
    g = dt - gm * s**3 * c3
    df = -gm / (r * r0) * s * c1
    dg = 1.0 - gm / r * s**2 * c2

    # Compute position and velocity vectors:
    vr = f * vr0 + g * vv0
    vv = df * vr0 + dg * vv0

    return vr, vv
```

C Introduction to Matrices

Bob: Prof. Starmover, could you tell us a bit more about matrices?

Professor: Sure! There is a lot to say about this subject, and I will just give a brief introduction to matrices. Matrices have a larger number of applications in mathematics and science in general, especially in the area of linear algebra (i.e., the study of linear equations). If you want to know more, you should take a look at the many excellent textbooks that discuss matrices and linear algebra in particular (e.g., Strang 2016).

In a nutshell, matrices are mathematical objects consisting of arrays of numbers. They can operate on vectors and return new vectors or operate on themselves (and return new matrices). For example, a rotation matrix will rotate a vector around a certain axis by a certain angle. Generally, a matrix $\mathbf{A}$ with dimension (m,n) is written as

$$\mathbf{A} = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix}. \tag{C.1}$$

The a_{ij} are called (matrix) components and can be any number. The dots just mean that there can be many more components. In the above matrix, there are m rows and n columns, and we usually refer to such a matrix as an $m \times n$ matrix. So a 3×2 matrix looks like

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{bmatrix}. \tag{C.2}$$

A "special" type of matrices is row and column vectors. These are matrices consisting of just a single row and single column, respectively. So a column vector can be represented as an $m \times 1$ matrix and has the form

$$\mathbf{A} = \begin{bmatrix} a_1 \\ \vdots \\ a_m \end{bmatrix}, \tag{C.3}$$

and a row vector is a $1 \times m$ matrix with the form

$$\mathbf{A} = \begin{bmatrix} a_1 & \cdots & a_m \end{bmatrix}, \tag{C.4}$$

where m is the dimension. If a matrix has an equal number of rows and columns ($n = m$), we also call it a **square** matrix. In practice, you will encounter these most of the time.

We define a number of matrix operations. Let me list them below:

- **Scalar multiplication.** Let c be a scalar and $\mathbf{A}$ an $m \times n$ matrix. Then, $\mathbf{B} = c\mathbf{A}$ is a new matrix with components $b_{ij} = ca_{ij}$.

- **Matrix addition/subtraction.** Let $\mathbf{A}$ and $\mathbf{B}$ both be $m \times n$ matrices. Then, $\mathbf{C} = \mathbf{A} \pm \mathbf{B}$ is a new $m \times n$ matrix with components $c_{ij} = a_{ij} \pm b_{ij}$.

- **Multiplication.** Let $\mathbf{A}$ be an $m \times n$ matrix and $\mathbf{B}$ be an $n \times s$ matrix. The product, $\mathbf{C} = \mathbf{AB}$, is an $m \times s$ matrix with components c_{ik} given by

$$c_{ik} = \sum_{j=1}^{n} a_{ij} b_{jk} = a_{i1} b_{1k} + a_{i2} b_{2k} + \cdots + a_{in} b_{nk}. \tag{C.5}$$

- **Transpose.** Let $\mathbf{A}$ be an $m \times n$ matrix. The transpose of $\mathbf{A}$, $\mathbf{B} = \mathbf{A}^\top$, is an $n \times m$ matrix with components $b_{ij} = a_{ji}$. In other words, the rows and columns in the transpose matrix are reversed.

- **Inverse.** Let $\mathbf{A}$ be an $m \times n$ matrix. The inverse matrix $\mathbf{B} = \mathbf{A}^{-1}$ is defined such that the matrix multiplication of $\mathbf{A}$ and $\mathbf{A}^{-1}$ yields the **identity** matrix $\mathbf{I}$, in both orders, that is, both $\mathbf{A}\mathbf{A}^{-1} = \mathbf{I}$, and $\mathbf{A}^{-1}\mathbf{A} = \mathbf{I}$. The identity matrix $\mathbf{I}$ is an $n \times n$ matrix (i.e., square matrix) consisting of zeros everywhere, except for the diagonal entries, which are unity. In other words,

$$\mathbf{I} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix}. \tag{C.6}$$

Let me also list a number of general matrix properties. They can be easily derived from the above definitions.

- **Distributive law.** Let $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ be matrices for which $\mathbf{A}$ and $\mathbf{B}$ can be multiplied, $\mathbf{A}$ and $\mathbf{C}$ can be multiplied, and $\mathbf{B}$ and $\mathbf{C}$ can be added. Then, $\mathbf{A}$ can be multiplied with $\mathbf{B} + \mathbf{C}$, and $\mathbf{A}(\mathbf{B} + \mathbf{C}) = \mathbf{AB} + \mathbf{AC}$. Also, for any scalar x, we have $\mathbf{A}(x\mathbf{B}) = x(\mathbf{AB})$.

- **Associative law.** Let $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ be matrices for which $\mathbf{A}$ and $\mathbf{B}$ can be multiplied, and $\mathbf{B}$ and $\mathbf{C}$ can be multiplied. Then, we can multiply $\mathbf{A}$ with $\mathbf{BC}$, and we can multiply $\mathbf{AB}$ with $\mathbf{C}$. Also, $(\mathbf{AB})\mathbf{C} = \mathbf{A}(\mathbf{BC})$, that is, the order of association in matrix multiplication does not matter.

- **Transpose of products.** Let $\mathbf{A}$ and $\mathbf{B}$ be matrices that can be multiplied. Then, we have $(\mathbf{AB})^\top = \mathbf{B}^\top \mathbf{A}^\top$.

- **Inverse of products.** Let $\mathbf{A}$ and $\mathbf{B}$ be matrices that can be multiplied. Then, we have $(\mathbf{AB})^{-1} = \mathbf{B}^{-1}\mathbf{A}^{-1}$.

Alice: Scalar multiplication and matrix addition seem pretty obvious to me. However, matrix multiplication seems a bit abstract. Maybe you could give an example?

Professor: Sure, let's multiply a 3×2 matrix with a 2×3 matrix:

$$\mathbf{AB} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \end{bmatrix} = \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} & a_{11}b_{13} + a_{12}b_{23} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} & a_{21}b_{13} + a_{22}b_{23} \\ a_{31}b_{11} + a_{32}b_{21} & a_{31}b_{12} + a_{32}b_{22} & a_{31}b_{13} + a_{32}b_{23} \end{bmatrix}. \tag{C.7}$$

Be aware that matrix multiplication is only well defined if the number of columns in the first matrix matches with the number of rows in the second matrix. Also, be careful about the order in matrix multiplication! For example, if we take the same matrices $\mathbf{A}$ and $\mathbf{B}$ as above and compute $\mathbf{BA}$, we get

$$\mathbf{BA} = \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{bmatrix} = \begin{bmatrix} a_{11}b_{11} + a_{21}b_{12} + a_{31}b_{13} & a_{12}b_{11} + a_{22}b_{12} + a_{32}b_{13} \\ a_{11}b_{21} + a_{21}b_{22} + a_{31}b_{23} & a_{12}b_{21} + a_{22}b_{22} + a_{32}b_{23} \end{bmatrix}. \tag{C.8}$$

Apart from the obvious different dimension of $\mathbf{BA}$ compared to $\mathbf{AB}$, the elements are also very different. In other words, matrix multiplication does not generally commute. For some matrices, however, it can be true that $\mathbf{AB} = \mathbf{BA}$. Also, note that matrix division is not defined. The closest analogy to division is taking the inverse of a matrix, $\mathbf{A}^{-1}$.

Alice: How does one find the inverse?

Professor: For small matrices, this is easy. There is a well-known expression for the inverse of a 2×2 matrix $\mathbf{A}$:

$$\mathbf{A}^{-1} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}^{-1} = \frac{1}{a_{11}a_{22} - a_{12}a_{21}} \begin{bmatrix} a_{22} & -a_{12} \\ -a_{21} & a_{11} \end{bmatrix}. \tag{C.9}$$

By working out the matrix product, you can easily see that $\mathbf{A}^{-1}\mathbf{A} = \mathbf{I}$. Note that the expression for the inverse matrix involves the quantity $a_{11}a_{22} - a_{12}a_{21}$. This is also known as the determinant of $\mathbf{A}$, that is, $\det \mathbf{A} = a_{11}a_{22} - a_{12}a_{21}$. If the determinant is zero, then the inverse of $\mathbf{A}$ is not defined, and in this case, $\mathbf{A}$ is also called a **singular** matrix.

Alice: I guess it is not so simple for larger matrices.

Professor: That's right. There exist general expressions for the inverses of matrices with given dimension, but they quickly become very complicated with increasing dimension. There are two standard techniques to find the inverse of a general matrix, known as elementary row operations, and the minors, cofactors, and adjugate method. I will not discuss both techniques here, so I suggest you look at a textbook such as Strang (2016).

Inverse matrices have many applications. For example, suppose you have a set of linear equations for n variables x_i, of the form

$$a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1; \tag{C.10}$$

$$a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2; \tag{C.11}$$

$$\vdots \tag{C.12}$$

$$a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n = b_m, \tag{C.13}$$

where a_{ij} and b_i are given coefficients. If the number of equations (m) is equal to the number of unknowns (n), then a unique solution $(x_1, \cdots, x_n)$ exists. We could find it by, for example, eliminating x_1 in the second equation using the first equation, then eliminating x_2 in the third equation using the second equation, and so on. This can be a tedious task. Instead, let us write the equations in the compact form

$$\mathbf{AX} = \mathbf{B}, \tag{C.14}$$

with $\mathbf{A}$ given by equation (C.1), and

$$\mathbf{X} = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}; \quad \mathbf{B} = \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix}. \tag{C.15}$$

By applying the inverse of $\mathbf{A}$ to the left side of equation (C.14), we get (for the left-hand side)

$$\mathbf{A}^{-1}(\mathbf{AX}) = (\mathbf{A}^{-1}\mathbf{A})\mathbf{X} = \mathbf{IX} = \mathbf{X}, \tag{C.16}$$

whereas the right-hand side is $\mathbf{A}^{-1}\mathbf{B}$. Therefore,

$$\mathbf{X} = \mathbf{A}^{-1}\mathbf{B}. \tag{C.17}$$

In other words, we get the solution by computing the inverse matrix of $\mathbf{A}$ and multiplying the result with $\mathbf{B}$.

Alice: That's neat! For good measure, could you illustrate the transpose of a matrix?

Professor: Sure. Let's transpose this 3×2 matrix:

$$\mathbf{A}^{\top} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{bmatrix}^{\top} = \begin{bmatrix} a_{11} & a_{21} & a_{31} \\ a_{12} & a_{22} & a_{32} \end{bmatrix}. \tag{C.18}$$

Bob: I am starting to like these matrices. I am sure there are more applications.

Professor: Indeed. A nice example is the rotation matrix. If you have a vector $\mathbf{v}$, you might want to be able to express $\mathbf{v}$ after rotating it about an axis with some angle θ. Let's keep it simple and consider a rotation of $\mathbf{v}$ around the z-axis. Then, all we need to do is let the corresponding rotation matrix $\mathbf{R}_z(\theta)$ act on $\mathbf{v}$:

$$\mathbf{R}_z(\theta)\mathbf{v} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} = \begin{bmatrix} v_1\cos\theta - v_2\sin\theta \\ v_1\sin\theta + v_2\cos\theta \\ v_3 \end{bmatrix}. \tag{C.19}$$

If you work the geometry out yourself, you can probably convince yourself that $v_1\cos\theta - v_2\sin\theta$ and $v_1\sin\theta + v_2\cos\theta$ are the x- and y-coordinates of the rotated vector, whereas the z-component of course remains unaffected. In this simple example, you could have worked out the result geometrically. However, in more complicated situations (rotations about an arbitrary axis), rotation matrices are much more convenient. In particular, any

rotation can be achieved by three successive rotations around the x-, y-, and z-axes with different angles. If we choose to first rotate around the x-axis with angle γ, then around the y-axis with angle β, and finally around the z-axis with angle α, then a general rotation can be described with the composite matrix

$$\mathbf{R}_z(\alpha)\mathbf{R}_y(\beta)\mathbf{R}_x(\gamma). \tag{C.20}$$

Note that, generally, the order of matrix multiplication matters. This makes sense in this context, since rotations do not generally commute with each other. For example, rotating a unit vector along the x-axis around the z-axis by 90° and then around the x-axis by 90° gives you a unit vector in the direction of the z-axis, whereas rotating the same unit vector along the x-axis around the x-axis by 90° (which does nothing) and then around the z-axis by 90° gives you a unit vector in the direction of the y-axis. For completeness, I will give the explicit expressions for the individual rotation matrices:

$$\mathbf{R}_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix}; \quad \mathbf{R}_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}; \quad \mathbf{R}_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{C.21}$$

Since they are relevant for the Laplace–Lagrange solutions (remember section 2.5.3 in chapter 2), I will close with a brief note on eigenvalues and eigenvectors. Let $\mathbf{T}$ be a matrix and $\mathbf{v}$ be a vector. Then, we say that $\mathbf{v}$ is an **eigenvector** of $\mathbf{T}$ if

$$\mathbf{T}\mathbf{v} = \lambda\mathbf{v}, \tag{C.22}$$

where λ is a scalar quantity. In other words, $\mathbf{T}$ acting on $\mathbf{v}$ returns the same vector modulo a constant. This constant, λ, is known as an **eigenvalue**. Note that a matrix $\mathbf{T}$ may have more than one eigenvector with associated eigenvalues. Eigenvectors and eigenvalues have important applications in mathematics and physics. For example, in the case of Laplace–Lagrange theory (multiplanet systems), the properties of the system are encoded into two matrices, $\mathbf{A}$ and $\mathbf{B}$. The eigenvalues of these matrices are the fundamental frequencies, which are the building blocks of the solutions of the eccentricities and inclinations.

D Derivations in the Lidov–Kozai Problem

Alice: Lidov–Kozai cycles play such an important role in our moons-around-circumbinary-planets project. Wouldn't it be nice to know more about them?

Bob: Yes, I suppose so.

Alice: Prof. Starmover, you had offered to tell us more about Lidov–Kozai (LK) cycles. In particular, I would like to know where the equations that you gave for the maximum eccentricity and the LK time scale come from. Would now be a good time?

Professor: Sure! I will tell you about the following. First, I will derive the Hamiltonian of the hierarchical three-body problem. Then I will show how to get to the secular equations of motion and discuss analytic solutions that can be obtained with a number of further approximations. From those solutions, we can gain more insight into the problem and also get the relations that you mention.

Alice: Sounds good!

D.1 The Three-Body Hamiltonian

Professor: As you might remember from chapter 2, Hamilton's formalism of mechanics is equivalent to Newtonian mechanics, but the former can be advantageous when using the right coordinates. It turns out that the three-body problem lends itself well to Hamilton's formalism, so I will stick to that. Be aware, however, that it is equally valid to approach the problem using Newton's formulation.

The classical Hamiltonian $\mathcal{H}$ is defined as $\mathcal{H} = T + V$, where T is the total kinetic energy, and V is the potential energy. Consider a system of three point masses with masses m_1, m_2, and m_3 and with position vectors $\mathbf{R}_1$, $\mathbf{R}_2$, and $\mathbf{R}_3$. As usual, these vectors are defined with respect to some arbitrary fixed origin. I will introduce a coordinate transformation that is very convenient for hierarchical three-body systems. Later, I will assume that $\mathbf{R}_3$ is the third, more distant body, orbiting the other two bodies around their barycenter, that is, their center of mass. However, for now, I will not yet make specific assumptions about the architecture of the system.

First, let us define the position vector $\mathbf{r}_1 = \mathbf{R}_2 - \mathbf{R}_1$ pointing from body 1 toward body 2, the center of mass of bodies 1 and 2, $\mathbf{r}_{\rm bin} = (\mathbf{R}_1 m_1 + \mathbf{R}_2 m_2)/(m_1 + m_2)$, and the vector pointing to the third body relative to $\mathbf{r}_{\rm bin}$, $\mathbf{r}_2 = \mathbf{R}_3 - \mathbf{r}_{\rm bin}$ (see figure D.1). By simple manipulation of these definitions, we get the following two useful relations:

$$\mathbf{r}_{\rm bin} = \frac{1}{m_1+m_2}\left[m_1\mathbf{R}_1 + m_2\mathbf{R}_2\right] = \frac{1}{m_1+m_2}\left[m_1\mathbf{R}_1 + m_2\left(\mathbf{r}_1+\mathbf{R}_1\right)\right] = \mathbf{R}_1 + \frac{m_2}{m_1+m_2}\mathbf{r}_1 \tag{D.1a}$$

$$= \frac{1}{m_1+m_2}\left[m_1\left(\mathbf{R}_2-\mathbf{r}_1\right) + m_2\mathbf{R}_2\right] = \mathbf{R}_2 - \frac{m_1}{m_1+m_2}\mathbf{r}_1. \tag{D.1b}$$

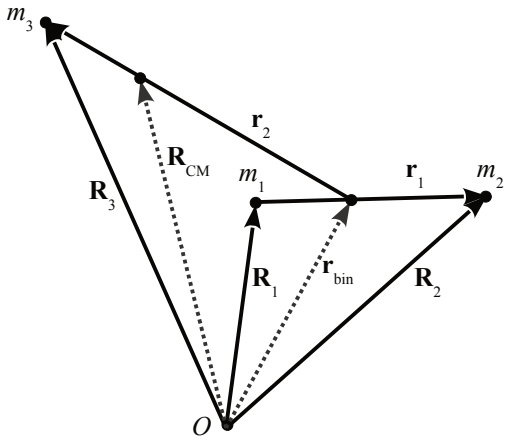


Figure D.1
Schematic depiction of a three-body system.

Using the definition of $\mathbf{r}_2$ and equations (D.1a) and (D.1b), the distances between masses 3 and 1 and masses 3 and 2, respectively, are given by the lengths of the difference vectors,

$$\mathbf{R}_3 - \mathbf{R}_1 = \mathbf{r}_2 + \mathbf{r}_{\rm bin} - \mathbf{R}_1 = \mathbf{r}_2 + \frac{m_2}{m_1+m_2}\mathbf{r}_1; \tag{D.2}$$

$$\mathbf{R}_3 - \mathbf{R}_2 = \mathbf{r}_2 + \mathbf{r}_{\rm bin} - \mathbf{R}_2 = \mathbf{r}_2 - \frac{m_1}{m_1+m_2}\mathbf{r}_1. \tag{D.3}$$

Bob: I guess you are computing these differences because we need them for the potential energy of the system.

Professor: That's right. We had defined the potential energy before in appendix A.2, specifically, equation (A.29). In this case, we only have three bodies, so there are three pairs to consider,

$$\begin{aligned} V &= -\frac{1}{2}\sum_{\substack{i,j=1\\ i\neq j}}^{3}\frac{Gm_im_j}{\left\|\mathbf{R}_i-\mathbf{R}_j\right\|} = -\frac{Gm_2m_1}{\|\mathbf{R}_2-\mathbf{R}_1\|} - \frac{Gm_3m_1}{\|\mathbf{R}_3-\mathbf{R}_1\|} - \frac{Gm_3m_2}{\|\mathbf{R}_3-\mathbf{R}_2\|} \\ &= -\frac{Gm_2m_1}{r_1} - \frac{Gm_3m_1}{\left\|\mathbf{r}_2+\frac{m_2}{m_1+m_2}\mathbf{r}_1\right\|} - \frac{Gm_3m_2}{\left\|\mathbf{r}_2-\frac{m_1}{m_1+m_2}\mathbf{r}_1\right\|}. \end{aligned} \tag{D.4}$$

So far, I have not assumed anything about the masses and/or the position vectors of the bodies. However, to proceed, I will now make the assumption that $r_1 \ll r_2$, that is, the inner binary is much more compact than the orbit of the third body around it. If we make this assumption, then we can expand the terms that depend on both $\mathbf{r}_1$ and $\mathbf{r}_2$.

Let me consider the general expression $1/\|\mathbf{r}-\alpha\mathbf{r}'\|$, where $\mathbf{r}$ and $\mathbf{r}'$ are two arbitrary vectors, and α is a constant. Write the angle between $\mathbf{r}$ and $\mathbf{r}'$ as γ, that is, $\hat{\mathbf{r}}\cdot\hat{\mathbf{r}}' = \cos(\gamma)$. We then have

$$\begin{aligned} \frac{1}{\|\mathbf{r}-\alpha\mathbf{r}'\|} &= \left[(\mathbf{r}-\alpha\mathbf{r}')^2\right]^{-1/2} = \left[r^2 - 2\alpha\,\mathbf{r}\cdot\mathbf{r}' + \alpha^2 r'^2\right]^{-1/2} \\ &= \frac{1}{r}\left[1 - 2\alpha\left(\frac{r'}{r}\right)\cos(\gamma) + \alpha^2\left(\frac{r'}{r}\right)^2\right]^{-1/2} = \frac{1}{r}\sum_{n=0}^{\infty}\alpha^n\left(\frac{r'}{r}\right)^n \widetilde{P}_n(\cos(\gamma)), \end{aligned} \tag{D.5}$$

where $\widetilde{P}_n$ is the nth Legendre polynomial (I am using a tilde to avoid any confusion with the inner and outer orbital periods, P_j). Applying equation (D.5) with $\mathbf{r} = \mathbf{r}_2$ and $\mathbf{r}' = \mathbf{r}_1$ twice to the last two terms in equation (D.4) and

with $\alpha = -m_2/(m_1+m_2)$ and $\alpha = m_1/(m_1+m_2)$, respectively, we get

$$\begin{aligned}
V &= -\frac{Gm_2m_1}{r_1} - \frac{Gm_3m_1}{r_2}\sum_{n=0}^{\infty}\left(\frac{-m_2}{m_1+m_2}\right)^n\left(\frac{r_1}{r_2}\right)^n\widetilde{P}_n(\cos(\Phi)) \\
&\quad - \frac{Gm_3m_2}{r_2}\sum_{n=0}^{\infty}\left(\frac{m_1}{m_1+m_2}\right)^n\left(\frac{r_1}{r_2}\right)^n\widetilde{P}_n(\cos(\Phi)) \\
&= -\frac{Gm_2m_1}{r_1} - \frac{Gm_3m_2m_1}{r_2}\sum_{n=0}^{\infty}\frac{-(-m_2)^{n-1}}{(m_1+m_2)^n}\left(\frac{r_1}{r_2}\right)^n\widetilde{P}_n(\cos(\Phi)) \\
&\quad - \frac{Gm_3m_2m_1}{r_2}\sum_{n=0}^{\infty}\frac{m_1^{n-1}}{(m_1+m_2)^n}\left(\frac{r_1}{r_2}\right)^n\widetilde{P}_n(\cos(\Phi)) \\
&= -\frac{Gm_2m_1}{r_1} - \frac{G}{r_2}\sum_{n=0}^{\infty}\left[m_1m_2m_3\frac{m_1^{n-1}-(-m_2)^{n-1}}{(m_1+m_2)^n}\right]\left(\frac{r_1}{r_2}\right)^n\widetilde{P}_n(\cos(\Phi)),
\end{aligned} \tag{D.6}$$

where Φ is the angle between $\mathbf{r}_1$ and $\mathbf{r}_2$, that is, $\cos(\Phi) = \hat{\mathbf{r}}_1\cdot\hat{\mathbf{r}}_2$.

Next, we explicitly write out the first two terms in the summation. The term with $n = 1$ vanishes because for $n = 1$, the mass term $m_1^{n-1} - (-m_2)^{n-1} = 0$. For the $n = 0$ term, we use that $\widetilde{P}_0(x) = 1$, so

$$\begin{aligned}
V &= -\frac{Gm_2m_1}{r_1} - \frac{G}{r_2}m_1m_2m_3\left(\frac{1}{m_1}+\frac{1}{m_2}\right) - \frac{G}{r_2}\sum_{n=2}^{\infty}\left[m_1m_2m_3\frac{m_1^{n-1}-(-m_2)^{n-1}}{(m_1+m_2)^n}\right]\left(\frac{r_1}{r_2}\right)^n\widetilde{P}_n(\cos(\Phi)) \\
&= -\frac{Gm_2m_1}{r_1} - \frac{Gm_3(m_1+m_2)}{r_2} - \frac{Gm_3}{r_2}\sum_{n=2}^{\infty}\mathcal{M}_n\left(\frac{r_1}{r_2}\right)^n\widetilde{P}_n(\cos(\Phi)),
\end{aligned} \tag{D.7}$$

where I defined a dimensionless mass parameter

$$\mathcal{M}_n \equiv \frac{m_1m_2}{m_1+m_2}\frac{m_1^{n-1}-(-m_2)^{n-1}}{(m_1+m_2)^{n-1}}. \tag{D.8}$$

Alice: Why did you define $\mathcal{M}_n$ in that particular way?

Professor: It turns out that if you write it this way, $\mathcal{M}_n$ generalizes nicely to hierarchical systems with an arbitrary number of bodies and structure. This is beyond the scope here, but if you are interested, you can find the details in Hamers and Portegies Zwart (2016).

Bob: And how many terms do you need for the expansion to be accurate?

Professor: That entirely depends on the value of the quantity $x \equiv r_1/r_2$. It should at least be $x < 1$ for the series to converge. If you add an infinite number of terms, the expansion is exact. Of course, that is not very practical—there comes a point when calculating all these expansion terms is more work than simply solving the three-body system with an N-body integrator like ABIE. In practice, for many hierarchical triple systems occurring in nature, it suffices to include just the lowest-order terms, $n = 2$, also known as the quadrupole-order terms. However, there are situations in which one needs to add the octupole-order terms $n = 3$ as well, and it turns out that the behavior can be dramatically different if the octupole-order terms are included: the evolution becomes much less regular, possibly even chaotic, and much higher maximum eccentricities can be reached. Unfortunately, however, the analytic solutions that I alluded to only apply to the simplest possible approximation, with only the quadrupole-order terms included and with an additional restriction, as we shall see.

Before discussing that, we still need to finish writing the expression for the Hamiltonian. We need to compute the kinetic energy T, which we defined in appendix A.2, specifically, equation (A.40), and is given in the three-body case by

$$T = \frac{1}{2}\sum_{i=1}^{3}m_i\dot{\mathbf{R}}_i^2 = \frac{1}{2}m_1\dot{\mathbf{R}}_1^2 + \frac{1}{2}m_2\dot{\mathbf{R}}_2^2 + \frac{1}{2}m_3\dot{\mathbf{R}}_3^2. \tag{D.9}$$

The aim is to write T in terms of the relative velocities, $\dot{\mathbf{r}}_1$ and $\dot{\mathbf{r}}_2$, as measured in a frame comoving with the center of mass, that is, in a frame in which the time derivative of the center of mass

$$\mathbf{R}_{\rm CM} = \frac{1}{m_1+m_2+m_3}\left(m_1\mathbf{R}_1 + m_2\mathbf{R}_2 + m_3\mathbf{R}_3\right), \tag{D.10}$$

vanishes: $\dot{\mathbf{R}}_{\rm CM} = \mathbf{0}$. Such a frame clearly exists if there are no external forces acting on the three-body system. First, let's write $\dot{\mathbf{R}}_1$ and $\dot{\mathbf{R}}_2$ in equation (D.9) in terms of $\dot{\mathbf{r}}_1$ and $\dot{\mathbf{r}}_{\rm bin}$ using equations (D.1a) and (D.1b),

$$\begin{aligned} T &= \frac{1}{2}m_1\left(\dot{\mathbf{r}}_{\rm bin} - \frac{m_2}{m_1+m_2}\dot{\mathbf{r}}_1\right)^2 + \frac{1}{2}m_2\left(\dot{\mathbf{r}}_{\rm bin} + \frac{m_1}{m_1+m_2}\dot{\mathbf{r}}_1\right)^2 + \frac{1}{2}m_3\dot{\mathbf{R}}_3^2 = \frac{1}{2}(m_1+m_2)\dot{\mathbf{r}}_{\rm bin}^2 \\ &\quad - \frac{m_1m_2}{m_1+m_2}\dot{\mathbf{r}}_{\rm bin}\cdot\dot{\mathbf{r}}_1 + \frac{m_1m_2}{m_1+m_2}\dot{\mathbf{r}}_{\rm bin}\cdot\dot{\mathbf{r}}_1 + \frac{1}{2}\frac{m_1m_2^2}{(m_1+m_2)^2}\dot{\mathbf{r}}_1^2 + \frac{1}{2}\frac{m_2m_1^2}{(m_1+m_2)^2}\dot{\mathbf{r}}_1^2 + \frac{1}{2}m_3\dot{\mathbf{R}}_3^2 \\ &= \frac{1}{2}\frac{m_1m_2}{m_1+m_2}\dot{\mathbf{r}}_1^2 + \frac{1}{2}(m_1+m_2)\dot{\mathbf{r}}_{\rm bin}^2 + \frac{1}{2}m_3\dot{\mathbf{R}}_3^2. \end{aligned} \tag{D.11}$$

To express $\mathbf{r}_{\rm bin}$ and $\mathbf{R}_3$ in terms of $\mathbf{R}_{\rm CM}$ and $\mathbf{r}_2$, first note that, by virtue of equations (D.1a) and (D.1b),

$$\begin{aligned} \mathbf{R}_{\rm CM} &= \frac{1}{M}\left(m_1\mathbf{R}_1 + m_2\mathbf{R}_2 + m_3\mathbf{R}_3\right) \\ &= \frac{1}{M}\left(m_1\mathbf{r}_{\rm bin} - \frac{m_1m_2}{m_1+m_2}\mathbf{r}_1 + m_2\mathbf{r}_{\rm bin} + \frac{m_1m_2}{m_1+m_2}\mathbf{r}_1 + m_3\mathbf{R}_3\right) \\ &= \frac{1}{M}\left((m_1+m_2)\mathbf{r}_{\rm bin} + m_3\mathbf{R}_3\right), \end{aligned} \tag{D.12}$$

where $M \equiv m_1 + m_2 + m_3$. Thus, we have the following two equations,

$$\begin{cases} \mathbf{r}_2 &= \mathbf{R}_3 - \mathbf{r}_{\rm bin}; \\ M\mathbf{R}_{\rm CM} &= (m_1+m_2)\mathbf{r}_{\rm bin} + m_3\mathbf{R}_3, \end{cases} \tag{D.13}$$

which can be solved for $\mathbf{R}_3$ and $\mathbf{r}_{\rm bin}$ in terms of $\mathbf{R}_{\rm CM}$ and $\mathbf{r}_2$. Multiplying the first equation in equation (D.13) by (m_1+m_2) and adding the result to the second equation yields

$$\mathbf{R}_3 = \mathbf{R}_{\rm CM} + \frac{m_1+m_2}{M}\mathbf{r}_2, \tag{D.14}$$

while multiplying the first equation in equation (D.13) by m_3 and subtracting from the result the second equation yields

$$\mathbf{r}_{\rm bin} = \mathbf{R}_{\rm CM} - \frac{m_3}{M}\mathbf{r}_2. \tag{D.15}$$

Alice: It looks like equations (D.14) and (D.15) are natural extensions of equations (D.1a) and (D.1b), which apply to the inner binary system, to the outer binary.

Professor: That's right. Differentiating equations (D.14) and (D.15) with respect to time, substituting the results into equation (D.11), and using that $\dot{\mathbf{R}}_{\rm CM} = \mathbf{0}$ then gives

$$\begin{aligned} T &= \frac{1}{2}\frac{m_1m_2}{m_1+m_2}\dot{\mathbf{r}}_1^2 + \frac{1}{2}(m_1+m_2)\frac{m_3^2}{M^2}\dot{\mathbf{r}}_2^2 + \frac{1}{2}m_3\frac{(m_1+m_2)^2}{M^2}\dot{\mathbf{r}}_2^2 \\ &= \frac{1}{2}\frac{m_1m_2}{m_1+m_2}\dot{\mathbf{r}}_1^2 + \frac{1}{2}\frac{m_3(m_1+m_2)}{M}\underbrace{\left(\frac{m_3}{M} + \frac{m_1+m_2}{M}\right)}_{=1}\dot{\mathbf{r}}_2^2 = \frac{1}{2}\frac{m_1m_2}{m_1+m_2}\dot{\mathbf{r}}_1^2 + \frac{1}{2}\frac{(m_1+m_2)m_3}{M}\dot{\mathbf{r}}_2^2. \end{aligned} \tag{D.16}$$

Bob: Ah, so the total kinetic energy, as measured in any frame comoving with the frame of the center of mass, consists of the relative kinetic energies of the inner and outer binary orbits.

Alice: Yes, this sounds very similar to what we established for the two-body problem in chapter 2.

Professor: Indeed. And it even generalizes to hierarchical systems with an arbitrary number of bodies and structure.

We have now arrived at the complete expression for the expanded Hamiltonian (combining equations D.7 and D.16),

$$\begin{aligned} \mathcal{H} = T + V &= \left(\frac{1}{2}\frac{m_1m_2}{m_1+m_2}\dot{\mathbf{r}}_1^2 - \frac{Gm_1m_2}{r_1}\right) + \left(\frac{1}{2}\frac{(m_1+m_2)m_3}{m_1+m_2+m_3}\dot{\mathbf{r}}_2^2 - \frac{G(m_1+m_2)m_3}{r_2}\right) \\ &\quad - \frac{Gm_3}{r_2}\sum_{n=2}^{\infty}\mathcal{M}_n\left(\frac{r_1}{r_2}\right)^n\widetilde{P}_n(\cos(\Phi)). \end{aligned} \tag{D.17}$$

The two terms in large brackets in the first line of equation (D.17) can be interpreted as the Hamiltonians (or, equivalently, the total energies) of two isolated binary systems. The inner binary consists of point masses m_1 and m_2 and has relative position and velocity vectors $\mathbf{r}_1$ and $\dot{\mathbf{r}}_1$, respectively, while the outer binary consists of point masses with masses (m_1+m_2) and m_3 and has relative position and velocity vectors $\mathbf{r}_2$ and $\dot{\mathbf{r}}_2$. The last term in equation (D.17) is a coupling term. If it were absent, then the Hamiltonian equation (D.17) would simply describe the motion of two orbits that are perfectly Keplerian and remain so ad infinitum. With the coupling term present, the two Keplerian orbits are perturbed. So the coupling term describes the effect of the LK oscillations.

Since the total energy in a binary system comprising point masses m_1 and m_2 reads $\mathcal{H}_{\rm bin} = -Gm_1m_2/(2a)$, where a is the semimajor axis, it is natural to introduce in a similar fashion the semimajor axes a_1 and a_2 of the inner and outer binary systems, respectively, such that the Hamiltonian of the three-body system can be written, somewhat suggestively, as

$$\mathcal{H} = -\frac{Gm_1m_2}{2a_1} - \frac{G(m_1+m_2)m_3}{2a_2} - \frac{Gm_3}{r_2}\sum_{n=2}^{\infty}\mathcal{M}_n\left(\frac{r_1}{r_2}\right)^n \widetilde{P}_n(\cos(\Phi)). \tag{D.18}$$

Note that in equation (D.18), the semimajor axes a_1 and a_2 could, in principle, vary as a function of time as a result of interactions between the orbits. However, if $r_2 \gg r_1$ at all times, then the coupling term in equation (D.18), also known as the disturbing function, is small.

Bob: Ha ha, it indeed looks disturbingly complicated.

Alice: Just wait, I have a feeling that the equations are going to become even more complicated.

Professor: Somewhat, yes, but still manageable! Anyway, if $r_2 \gg r_1$, then the coupling term is relatively small and both orbits remain almost unaffected by the perturbing term, that is, the orbits remain Keplerian to very good approximation, and the semimajor axes a_1 and a_2 remain constant (although they could slightly oscillate, especially near periapsis, but there will be no net change). However, the shapes and orientations of the orbits can change over time scales that are (much) longer than the orbital periods. In this situation, the triple system can be considered hierarchical. Because $a_2 \gg a_1$ in such systems (or, at least, $a_2 > a_1$), it is convenient to write the coupling term in terms of the semimajor axis ratio, $\alpha \equiv a_1/a_2$, as follows:

$$\mathcal{H} = -\frac{Gm_1m_2}{2a_1} - \frac{G(m_1+m_2)m_3}{2a_2} - \frac{Gm_3}{a_2}\sum_{n=2}^{\infty}\alpha^n\mathcal{M}_n\left(\frac{r_1}{a_1}\right)^n\left(\frac{a_2}{r_2}\right)^{n+1} \widetilde{P}_n(\cos(\Phi)). \tag{D.19}$$

D.2 Averaging the Hamiltonian

D.2.1 Approximations

To summarize, we have derived the Hamiltonian of the three-body system, equation (D.19). If r_1 is not much smaller than r_2, then the system is likely not dynamically stable; that is, it will fly apart on short time scales (comparable to the orbital time scales). So, for stability, we require that the system is sufficiently hierarchical. In that case, the semimajor axes remain approximately constant; that is, the orbits do not exchange energy. There can be an exchange of angular momentum between the orbits (of course, conserving the total angular momentum), and this is what leads to LK oscillations.

The next step is to average the Hamiltonian over the inner and outer orbits, assuming that they are perfectly Keplerian. A note of caution is advised here. Until now, equation (D.19), the Hamiltonian, is exact if $r_1 < r_2$ and if an infinite number of terms are included in the expansion. Although one can (formally) compute the orbit-averaged Hamiltonian to *any* order, in which case one could think the Hamiltonian is still exact, the process of orbit averaging itself is not a canonical transformation. Mathematically, this means that the orbital elements before and after averaging differ by terms of the order of $(a_1/a_2)^2$ and higher. However, one can show that the orbital elements are the same when one restricts to the quadrupole-order terms.

Physically, the fact that the transformation is not canonical arises because, as it turns out, there can be situations in which the exchange of angular momentum between the orbits can occur on time scales that are *shorter* than the orbital periods. Clearly, in these situations, it is not justified to average over both orbits. In these cases, the evolution can still be quasi-LK-like, with the system remaining dynamically stable and the eccentricity of the

inner orbit changing periodically, but potentially much higher eccentricities can be reached. This regime in which the secular averaging approximation breaks down is known as the semisecular regime.

Alice: How can one check if the semisecular regime is important or not?

Professor: One can compare the time scale for the inner orbit eccentricity to change, which is comparable to $t_{\rm LK}$ (equation 13.1), to the inner or outer orbital periods.

Another regime in which the orbit averaging can break down is when there exists a resonance between the precession of the line of apsides in the inner orbit and the motion of the third body. This resonance is known as the evection resonance and is missed when averaging over *both* orbits. In any case, since I am focusing on the "regular" and "simple" LK regime, I will not discuss these complications further, but you should be aware that orbit averaging is not always a good approximation.

D.2.2 Orbital Angles and the Choice of Reference Frame

There are multiple ways to carry out the averaging of the Hamiltonian in practice. One can use vectors to describe the orbit, that is, equations like equation (2.58a) from chapter 2. Alternatively, one can use orbital elements. Both approaches are equally valid. In practice, it turns out that orbital elements greatly simplify the equations of motion at the lowest order (quadrupole order). However, at higher orders and when considering higher-order systems such as hierarchical quadruple systems, it is more convenient to use orbital vectors. Since we are focusing on just the lowest-order expansion and on triples, I will stick to orbital elements.

Alice: I guess that expressing the terms r_1/a_1 and r_2/a_2 in terms of the orbital elements is straightforward. However, the expression $\cos(\Phi)$ is probably more complicated.

Professor: Indeed. One can get it by taking the vector expression of $\mathbf{r}$ in equation (2.58a) from chapter 2 and combine it with the expressions of $\hat{\mathbf{e}}$ and $\hat{\mathbf{q}}$. With some trigonometric identities, this will give

$$\begin{aligned}\hat{\mathbf{r}} = &[\cos(\theta+\omega)\cos(\Omega) - \sin(\theta+\omega)\cos(i)\sin(\Omega)]\,\hat{\mathbf{x}}\\ &+[\sin(\theta+\omega)\cos(i)\cos(\Omega) + \cos(\theta+\omega)\sin(\Omega)]\,\hat{\mathbf{y}} + \sin(\theta+\omega)\sin(i)\,\hat{\mathbf{z}}.\end{aligned} \tag{D.20}$$

We then get $\cos(\Phi)$ by computing the dot product between two independent vectors, $\hat{\mathbf{r}}_1$ and $\hat{\mathbf{r}}_2$. Again, after using some trigonometric identities, this gives

$$\begin{aligned}\cos(\Phi) = \hat{\mathbf{r}}_1\cdot\hat{\mathbf{r}}_2 = &\cos(\omega_2+f_2)\left[\cos(\omega_1+f_1)\cos(\Delta\Omega) - \cos(i_1)\sin(\omega_1+f_1)\sin(\Delta\Omega)\right]\\ &+\sin(\omega_2+f_2)\left[\cos(i_2)\cos(\omega_1+f_1)\sin(\Delta\Omega) + \sin(\omega_1+f_1)\{\cos(i_1)\cos(i_2)\cos(\Delta\Omega)\right.\\ &\left.+\sin(i_1)\sin(i_2)\}\right],\end{aligned} \tag{D.21}$$

where $\Delta\Omega \equiv \Omega_1 - \Omega_2$. As you might remember from section 2.5.2 in chapter 2, when defining the orbital elements, one should specify a reference plane. We have not explicitly chosen a reference plane yet in equation (D.21). When we do and choose the so-called *invariable* plane for the reference plane, then it turns out that $\Delta\Omega = \pi$, in which case equation (D.21) simplifies.

The invariable plane is the plane perpendicular to the total angular momentum vector of the system. The latter is obviously conserved, so the invariable plane is a legitimate reference plane. To show that $\Delta\Omega = \pi$ if we choose the reference plane to be the invariable plane, first remember the angular-momentum vector per unit mass in chapter 2, $\mathbf{k} = \mathbf{r}\times\dot{\mathbf{r}}$, with $k = \sqrt{GMa(1-e^2)}$, and

$$\hat{\mathbf{k}} = \hat{\mathbf{e}}\times\hat{\mathbf{q}} = \sin(\Omega)\sin(i)\,\hat{\mathbf{x}} - \cos(\Omega)\sin(i)\,\hat{\mathbf{y}} + \cos(i)\,\hat{\mathbf{z}}. \tag{D.22}$$

The "full" angular-momentum vector (i.e., not per unit mass) is $\tilde{\mathbf{G}} = \mu\mathbf{k}$, where $\mu = m_1m_2/(m_1+m_2)$ is the reduced mass. This applies to any orbit with two components with masses m_1 and m_2. So we have

$$\tilde{\mathbf{G}}_1 = \mu_1\mathbf{k}_1 = \tilde{G}_1\hat{\mathbf{k}}_1; \qquad \tilde{\mathbf{G}}_2 = \mu_2\mathbf{k}_2 = \tilde{G}_2\hat{\mathbf{k}}_2 \tag{D.23}$$

for the inner and outer orbits, respectively, where $\mu_1 \equiv m_1m_2/(m_1+m_2)$, $\mu_2 \equiv (m_1+m_2)m_3/(m_1+m_2+m_3)$, and $\tilde{G}_1$ and $\tilde{G}_2$ are the magnitudes of their corresponding vectors. Note that I am here using a different notation for the angular momentum vector: $\tilde{\mathbf{G}}$ instead of $\mathbf{L}$. This is because of a different notation used in the context of hierarchical three-body systems and so-called Delaunay elements, which will become clear in a moment (section D.3).

Bob: Okay. But why are you putting a tilde above the $\mathbf{G}$s?

Professor: Oh, that's just to distinguish the angular-momentum vector from Newton's gravitational constant G!

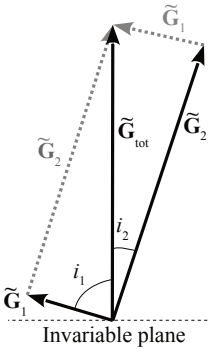


Figure D.2
A depiction of the vectors $\tilde{\mathbf{G}}_1$ and $\tilde{\mathbf{G}}_2$ and their sum $\tilde{\mathbf{G}}_{\rm tot} = \tilde{\mathbf{G}}_1 + \tilde{\mathbf{G}}_2$. All the vectors lie in the same plane, the plane of the drawing. The invariable plane is perpendicular to this plane, that is, perpendicular to $\tilde{\mathbf{G}}_{\rm tot}$, and to the plane of the drawing.

So, after a bit of rewriting, this means that the total angular momentum vector is given by

$$\begin{aligned}\tilde{\mathbf{G}}_{\rm tot} = \mu_1\mathbf{k}_1 + \mu_2\mathbf{k}_2 = &\Big[\tilde{G}_1\sin(i_1)\sin(\Omega_1) + \tilde{G}_2\sin(i_2)\sin(\Omega_2)\Big]\hat{\mathbf{x}}\\ &-\Big[\tilde{G}_1\sin(i_1)\cos(\Omega_1) + \tilde{G}_2\sin(i_2)\cos(\Omega_2)\Big]\hat{\mathbf{y}} + \Big[\tilde{G}_1\cos(i_1) + \tilde{G}_2\cos(i_2)\Big]\hat{\mathbf{z}}.\end{aligned} \tag{D.24}$$

If we choose the reference plane (i.e., the directions of $\hat{\mathbf{x}}$, $\hat{\mathbf{y}}$, and $\hat{\mathbf{z}}$) to be the invariable plane, then the total angular-momentum vector must be parallel to the z-axis of the invariant coordinate system. This implies that the x- and y-components of $\tilde{\mathbf{G}}_{\rm tot}$ must be zero, that is,

$$\begin{cases} \tilde{G}_1\sin(i_1)\sin(\Omega_1) &= -\tilde{G}_2\sin(i_2)\sin(\Omega_2);\\ \tilde{G}_1\sin(i_1)\cos(\Omega_1) &= -\tilde{G}_2\sin(i_2)\cos(\Omega_2);\\ \tilde{G}_{\rm tot} &= \tilde{G}_1\cos(i_1) + \tilde{G}_2\cos(i_2).\end{cases} \tag{D.25}$$

If we divide the first two equations in equation (D.25), we get $\tan(\Omega_1) = \tan(\Omega_2)$, or $\Delta\Omega \equiv \Omega_1 - \Omega_2 = \pi$. Note that if no external forces act on the triple system, then this relation must hold at all times because the total angular momentum vector $\tilde{\mathbf{G}}_{\rm tot}$ is conserved in that case.

It is useful to draw a picture of the vectors $\tilde{\mathbf{G}}_1$, $\tilde{\mathbf{G}}_2$, and $\tilde{\mathbf{G}}_{\rm tot}$, which I have done in figure D.2. All the vectors lie in the same plane, the plane of the drawing. The invariable plane is perpendicular to this plane, that is, perpendicular to $\tilde{\mathbf{G}}_{\rm tot}$, and to the plane of the drawing. If we apply the cosine rule to this figure to the two triangles spanned by $\tilde{\mathbf{G}}_1$ and $\tilde{\mathbf{G}}_{\rm tot}$, as well as by $\tilde{\mathbf{G}}_2$ and $\tilde{\mathbf{G}}_{\rm tot}$, we get the useful relations

$$\cos(i_1) = \frac{\tilde{G}_{\rm tot}^2 + \tilde{G}_1^2 - \tilde{G}_2^2}{2\tilde{G}_{\rm tot}\tilde{G}_1}; \quad \cos(i_2) = \frac{\tilde{G}_{\rm tot}^2 + \tilde{G}_2^2 - \tilde{G}_1^2}{2\tilde{G}_{\rm tot}\tilde{G}_2}. \tag{D.26}$$

We can obtain another useful relation by computing the dot product between $\tilde{\mathbf{G}}_1$ and $\tilde{\mathbf{G}}_2$, that is,

$$\tilde{G}_{\rm tot}^2 = (\tilde{\mathbf{G}}_1 + \tilde{\mathbf{G}}_2)^2 = \tilde{G}_1^2 + \tilde{G}_2^2 + 2\tilde{G}_1\tilde{G}_2\cos(i_{\rm tot}), \tag{D.27}$$

where $i_{\rm tot} = i_1 + i_2$ is the mutual inclination. Note that $i_{\rm tot} = i_1 + i_2$ is also consistent with the equation for the mutual inclination, equation (2.60), from chapter 2.

Before continuing with the orbit averaging, let me talk for a moment about a subtlety that has confused some researchers in the past.

Bob: If even researchers get confused, I guess it is good to know about it!

Professor: Indeed. The confusion concerns the fact that by the choice of the coordinate system, $\Delta\Omega = \pi$; that is, $\Delta\Omega$ is constant. This means that the Hamiltonian is independent of $\Delta\Omega$. However, this fact has led some people to conclude that the Hamiltonian is independent of *both* Ω_1 and Ω_2 and that, therefore, the associated conjugate

momenta, $\tilde{H}_1$ and $\tilde{H}_2$, are constant (I will talk about this later when discussing the equations of motion). This is, however, not generally true; the Hamiltonian does still depend on Ω_1 and Ω_2; there just happens to be a fixed relation between them. It will turn out that the relation does imply that the equations of motion for the eccentricities and the inclinations are decoupled from the longitudes of the ascending node; that is, $\dot{e}_1$, $\dot{e}_2$, $\dot{\omega}_1$, and $\dot{\omega}_2$ do not depend on Ω_1 and Ω_2; therefore, one does not have to include the equations of motion for Ω_1 and Ω_2 when solving the equations of motion. For clarity, in computing the orbit-averaged Hamiltonian, I will for now *not* make the substitution $\Delta\Omega = \pi$.

D.2.3 The Averaging Calculation

Bob: To compute the orbit-averaged Hamiltonian, I guess we just have to calculate

$$\frac{1}{2\pi}\frac{1}{2\pi}\int_0^{2\pi}\int_0^{2\pi}\mathcal{H}\,\mathrm{d}\theta_1\,\mathrm{d}\theta_2, \tag{D.28}$$

where we set all orbital elements except θ_1 and θ_2 to be constant.

Professor: Close, but no cigar! When averaging over an orbit, we actually want to average over *time*. Generally, the *true* anomaly does not increase linearly with time (except for a circular orbit), so it is not a good quantity to use to define the orbit average. The *mean* anomaly does increase linearly with time; in fact, $\mathcal{M} = n(t-T)$, where $n = 2\pi/P$ is the mean motion (not to be confused with the expansion order of the Hamiltonian, n), and T is the time of periapsis passage. So I will define the orbit average of a quantity as

$$\langle ... \rangle_j = \frac{1}{2\pi}\int_0^{2\pi} ...\,\mathrm{d}\mathcal{M}_j, \tag{D.29}$$

where the dots denote the quantity to be averaged, and j can be 1 or 2 for the inner or outer orbits.

To simplify the integration, it can, nonetheless, be more convenient to *transform* the integration over mean anomaly to an integration over the true or eccentric anomalies. Therefore, let me briefly mention some relations between the mean, eccentric, and true anomalies, denoted by $\mathcal{M}$, $\mathcal{E}$, and θ, respectively. Recalling from chapter 2, the binary separation r is given in terms of $\mathcal{E}$ and θ by

$$r/a = 1 - e\cos(\mathcal{E}) = \frac{1-e^2}{1+e\cos(\theta)}. \tag{D.30}$$

The true anomaly and eccentric anomalies are related according to

$$\sin(\mathcal{E}) = \sqrt{1-e^2}\,\frac{\sin(\theta)}{1+e\cos(\theta)}; \qquad \cos(\mathcal{E}) = \frac{\cos(\theta)+e}{1+e\cos(\theta)}; \tag{D.31}$$

these equations may also written in inverted form as

$$\sin(\theta) = \sqrt{1-e^2}\,\frac{\sin(\mathcal{E})}{1-e\cos(\mathcal{E})}; \qquad \cos(\theta) = \frac{\cos(\mathcal{E})-e}{1-e\cos(\mathcal{E})}. \tag{D.32}$$

Lastly, we have the Kepler equation,

$$\mathcal{M} = \mathcal{E} - e\sin(\mathcal{E}). \tag{D.33}$$

From equation (D.33), it immediately follows that

$$\mathrm{d}\mathcal{M} = (1-e\cos(\mathcal{E}))\,\mathrm{d}\mathcal{E} = (r/a)\,\mathrm{d}\mathcal{E}, \tag{D.34}$$

which states the differential transformation between the mean and eccentric anomalies. Furthermore, differentiating both sides of the second equation of equation (D.31), we find

$$-\sin(\mathcal{E})\,\mathrm{d}\mathcal{E} = \frac{-\sin(\theta)(1+e\cos(\theta))+e\sin(\theta)(\cos(\theta)+e)}{(1+e\cos(\theta))^2}\,\mathrm{d}\theta = -\sin(\theta)\frac{1-e^2}{(1+e\cos(\theta))^2}\,\mathrm{d}\theta, \tag{D.35}$$

which gives, after writing $\sin(\mathcal{E})$ in terms of θ with the first equation of equation (D.31),

$$\begin{aligned} -\sqrt{1-e^2}\,\frac{\sin(\theta)}{1+e\cos(\theta)}\,\mathrm{d}\mathcal{E} &= -\sin(\theta)\frac{1-e^2}{(1+e\cos(\theta))^2}\,\mathrm{d}\theta \\ \Rightarrow \mathrm{d}\mathcal{E} &= \frac{\sqrt{1-e^2}}{1+e\cos(\theta)}\,\mathrm{d}\theta = \frac{1}{\sqrt{1-e^2}}\,\frac{r}{a}\,\mathrm{d}\theta. \end{aligned} \tag{D.36}$$

Lastly, combining equations (D.34) and (D.36), we get

$$\mathrm{d}\mathcal{M} = \frac{1}{\sqrt{1-e^2}}\left(\frac{r}{a}\right)^2 \mathrm{d}\theta, \tag{D.37}$$

which states the differential transformation between the mean and true anomalies.

Now, let us finally start doing the averaging calculation, that is, averaging equation (D.19) over both orbits. In principle, the calculation can be done for any arbitrary expansion order in equation (D.19), but as I mentioned before, for many hierarchical triple systems, the lowest-order term ($n = 2$, the quadrupole-order term) is already quite adequate. For a smaller (but nonnegligible) fraction of systems, also the next-order term ($n = 3$, the octupole-order term) is required for an adequate description. In perhaps a few cases, the next-order terms could be important. However, one should bear in mind that in these cases, the system is likely marginally hierarchical and so is close to dynamical instability.

Here, I will derive the orbit-averaged Hamiltonian to the quadrupole and octupole orders. First, let us write equation (D.19) as

$$\mathcal{H} = -\frac{Gm_1m_2}{2a_1} - \frac{G(m_1+m_2)m_3}{2a_2} - R_{\rm quad} - R_{\rm oct} - \frac{Gm_3}{a_2}\sum_{n=4}^{\infty}\alpha^n\mathcal{M}_n\left(\frac{r_1}{a_1}\right)^n\left(\frac{a_2}{r_2}\right)^{n+1}\widetilde{P}_n(\cos(\Phi)), \tag{D.38}$$

where

$$\begin{aligned} R_{\rm quad} &= \frac{Gm_3}{a_2}\left(\frac{a_1}{a_2}\right)^2\mathcal{M}_2\left(\frac{r_1}{a_1}\right)^2\left(\frac{a_2}{r_2}\right)^3\widetilde{P}_2(\cos(\Phi)) = \frac{Gm_1m_2m_3}{(m_1+m_2)a_2}\left(\frac{a_1}{a_2}\right)^2\left(\frac{r_1}{a_1}\right)^2\left(\frac{a_2}{r_2}\right)^3\widetilde{P}_2(\cos(\Phi)) \\ &= 16C_{\rm quad}\left(\frac{r_1}{a_1}\right)^2\left(\frac{a_2}{r_2}\right)^3\left(1-e_2^2\right)^{3/2}\widetilde{P}_2(\cos(\Phi)), \end{aligned} \tag{D.39}$$

with the quantity (with dimensions of energy) $C_{\rm quad}$ given by

$$C_{\rm quad} \equiv \frac{1}{16}\frac{Gm_1m_2m_3}{(m_1+m_2)a_2}\left(\frac{a_1}{a_2}\right)^2\left(1-e_2^2\right)^{-3/2}, \tag{D.40}$$

and

$$\begin{aligned} R_{\rm oct} &= \frac{Gm_3}{a_2}\left(\frac{a_1}{a_2}\right)^3\mathcal{M}_3\left(\frac{r_1}{a_1}\right)^3\left(\frac{a_2}{r_2}\right)^4\widetilde{P}_3(\cos(\Phi)) \\ &= \frac{G}{a_2}\left(\frac{a_1}{a_2}\right)^3\left(\frac{r_1}{a_1}\right)^3\left(\frac{a_2}{r_2}\right)^4\frac{m_1m_2m_3}{(m_1+m_2)^3}\left(m_1^2-m_2^2\right)\widetilde{P}_3(\cos(\Phi)) \\ &= \frac{G}{a_2}\left(\frac{a_1}{a_2}\right)^3\left(\frac{r_1}{a_1}\right)^3\left(\frac{a_2}{r_2}\right)^4\frac{m_1m_2m_3}{(m_1+m_2)^2}(m_1-m_2)\widetilde{P}_3(\cos(\Phi)) \\ &= -\frac{16}{15}4C_{\rm oct}\left(1-e_2^2\right)^{5/2}\left(\frac{r_1}{a_1}\right)^3\left(\frac{a_2}{r_2}\right)^4\widetilde{P}_3(\cos(\Phi)), \end{aligned} \tag{D.41}$$

with the quantity (again, with dimensions of energy) $C_{\rm oct}$ given by

$$C_{\rm oct} \equiv -\frac{15}{16}\frac{G}{4}\frac{m_1m_2m_3}{(m_1+m_2)^2}(m_1-m_2)\left(\frac{a_1}{a_2}\right)^3\frac{1}{a_2}\left(1-e_2^2\right)^{-5/2}. \tag{D.42}$$

Alice: Why are you introducing all those extra integer factors and the factors depending on e_2?

Professor: These factors make sure that the averaged Hamiltonian will not have overall multiplicative factors (i.e., it will look nice).

D.2.3.1 Quadrupole order

Now, let us first look at the quadrupole-order term, $R_{\rm quad}$. We can average first over the inner orbit and then over the outer orbit, or vice versa. Mathematically, there is no difference. I will choose to first average over the outer

orbital period and use the true anomaly as the integration variable.

$$\begin{aligned}\langle R_{\rm quad}\rangle_2 &= \frac{1}{2\pi}\int_0^{2\pi} R_{\rm quad}\,\mathrm{d}\mathcal{M}_2\\ &= 16\,C_{\rm quad}\left(1-e_2^2\right)^{3/2}\left(\frac{r_1}{a_1}\right)^2\frac{1}{2\pi}\int_0^{2\pi}\left(\frac{a_2}{r_2}\right)^3\underbrace{\frac{1}{2}\left(3\cos^2(\Phi)-1\right)}_{=\widetilde{P}_2(\cos(\Phi))}\underbrace{\left(\frac{r_2}{a_2}\right)^2\frac{1}{\sqrt{1-e_2^2}}\,\mathrm{d}\theta_2}_{=\mathrm{d}\mathcal{M}_2}\\ &= 8\,C_{\rm quad}\left(1-e_2^2\right)^{3/2}\left(\frac{r_1}{a_1}\right)^2\frac{1}{2\pi}\int_0^{2\pi}\frac{1+e_2\cos(\theta_2)}{1-e_2^2}\frac{1}{\sqrt{1-e_2^2}}\left(3\cos^2(\Phi)-1\right)\mathrm{d}\theta_2\\ &= 8\,C_{\rm quad}\left(\frac{r_1}{a_1}\right)^2\frac{1}{2\pi}\int_0^{2\pi}\left[1+e_2\cos(\theta_2)\right]\left[3\cos^2(\Phi)-1\right]\mathrm{d}\theta_2. \end{aligned}\tag{D.43}$$

In the last line, you can see why the factor $(1-e_2^2)^{-3/2}$ was included in the definition of $C_{\rm quad}$. Let us make our lives a bit easier by first isolating all the terms that depend on θ_1 and θ_2. All other terms are constant when doing the integrations. With some trigonometric identities, $\cos(\Phi)$, equation (D.21), can be written as

$$\cos(\Phi) = \tilde{A}_a\cos(\theta_2)+\tilde{A}_b\sin(\theta_2),\tag{D.44}$$

where

$$\begin{cases}\tilde{A}_a &= Z_a\cos(\theta_1)+Z_b\sin(\theta_1);\\ \tilde{A}_b &= Z_c\cos(\theta_1)+Z_d\sin(\theta_1),\end{cases}\tag{D.45}$$

and

$$\begin{cases} Z_a &\equiv D_a\cos(\omega_1)\cos(\omega_2)-D_b\sin(\omega_1)\cos(\omega_2)+D_c\cos(\omega_1)\sin(\omega_2)+D_d\sin(\omega_1)\sin(\omega_2);\\ Z_b &\equiv -D_a\sin(\omega_1)\cos(\omega_2)-D_b\cos(\omega_1)\cos(\omega_2)-D_c\sin(\omega_1)\sin(\omega_2)+D_d\cos(\omega_1)\sin(\omega_2);\\ Z_c &\equiv -D_a\cos(\omega_1)\sin(\omega_2)+D_b\sin(\omega_1)\sin(\omega_2)+D_c\cos(\omega_1)\cos(\omega_2)+D_d\sin(\omega_1)\cos(\omega_2);\\ Z_d &\equiv D_a\sin(\omega_1)\sin(\omega_2)+D_b\cos(\omega_1)\sin(\omega_2)-D_c\sin(\omega_1)\cos(\omega_2)+D_d\cos(\omega_1)\cos(\omega_2);\end{cases}$$

$$\begin{cases} D_a &\equiv \cos(\Delta\Omega);\\ D_b &\equiv \cos(i_1)\sin(\Delta\Omega);\\ D_c &\equiv \cos(i_2)\sin(\Delta\Omega);\\ D_d &\equiv \cos(i_1)\cos(i_2)\cos(\Delta\Omega)+\sin(i_1)\sin(i_2).\end{cases}\tag{D.46}$$

This looks daunting, but note that the Zs and Ds are all just constant in the integrals. We then get

$$\begin{aligned}\langle R_{\rm quad}\rangle_2 = 8\,C_{\rm quad}\left(\frac{r_1}{a_1}\right)^2\frac{1}{2\pi}\int_0^{2\pi}\left[1+e_2\cos(\theta_2)\right]\Big[&3\tilde{A}_a^2\cos^2(\theta_2)+6\tilde{A}_a\tilde{A}_b\cos(\theta_2)\sin(\theta_2)\\ &+3\tilde{A}_b^2\sin^2(\theta_2)-1\Big]\mathrm{d}\theta_2 = 8\,C_{\rm quad}\left(\frac{r_1}{a_1}\right)^2\left[\frac{3}{2}\tilde{A}_a^2+\frac{3}{2}\tilde{A}_b^2-1\right].\end{aligned}\tag{D.47}$$

For the integration over the inner orbit, it is convenient to transform to the eccentric anomaly. So we use equations (D.32) to replace $\sin(\theta_1)$ and $\cos(\theta_1)$ with expressions depending on θ, and we use equation (D.34) to transform from the mean anomaly to the eccentric anomaly.

$$\begin{aligned}\langle\langle R_{\rm quad}\rangle_2\rangle_1 = \frac{1}{2\pi}\int_0^{2\pi}\langle R_{\rm quad}\rangle_2\,\mathrm{d}\mathcal{M}_1 = 8\,C_{\rm quad}\frac{1}{2\pi}\int_0^{2\pi}\left(\frac{r_1}{a_1}\right)^2\Big[&\frac{3}{2}\left(Z_a^2+Z_c^2\right)\cos^2(\theta_1)\\ &+3(Z_aZ_b+Z_cZ_d)\cos(\theta_1)\sin(\theta_1)+\frac{3}{2}\left(Z_b^2+Z_d^2\right)\sin^2(\theta_1)-1\Big]\underbrace{\left(\frac{r_1}{a_1}\right)\mathrm{d}\mathcal{E}_1}_{=\mathrm{d}\mathcal{M}_1}\end{aligned}$$

$$
\begin{aligned}
&= 8\,C_{\rm quad}\frac{1}{2\pi}\int_0^{2\pi}[1-e_1\cos(\mathcal{E}_1)]^3\left[\frac{3}{2}\left(Z_a^2+Z_c^2\right)\left(\frac{\cos(\mathcal{E}_1)-e_1}{1-e_1\cos(\mathcal{E}_1)}\right)^2\right.\\
&\quad\left.+3(Z_aZ_b+Z_cZ_d)\frac{\sqrt{1-e_1^2}\sin(\mathcal{E}_1)(\cos(\mathcal{E}_1)-e_1)}{(1-e_1\cos(\mathcal{E}_1))^2}+\frac{3}{2}\left(Z_b^2+Z_d^2\right)\left(\frac{\sqrt{1-e_1^2}\sin(\mathcal{E}_1)}{1-e_1\cos(\mathcal{E}_1)}\right)^2-1\right]\mathrm{d}\mathcal{E}_1\\
&= 8\,C_{\rm quad}\frac{1}{2\pi}\int_0^{2\pi}[1-e_1\cos(\mathcal{E}_1)]\left[\frac{3}{2}\left(Z_a^2+Z_c^2\right)(\cos(\mathcal{E}_1)-e_1)^2\right.\\
&\quad+3(Z_aZ_b+Z_cZ_d)\sqrt{1-e_1^2}\sin(\mathcal{E}_1)(\cos(\mathcal{E}_1)-e_1)\\
&\quad\left.+\frac{3}{2}\left(Z_b^2+Z_d^2\right)\left(1-e_1^2\right)\sin^2(\mathcal{E}_1)-(1-e_1\cos(\mathcal{E}_1))^2\right]\mathrm{d}\mathcal{E}_1 = 8\,C_{\rm quad}\left[\frac{3}{2}\left(Z_a^2+Z_c^2\right)\frac{1}{2}\left(1+4e_1^2\right)\right.\\
&\quad\left.+\frac{3}{2}\left(Z_b^2+Z_d^2\right)\left(1-e_1^2\right)\cdot\frac{1}{2}-\frac{1}{2}\left(2+3e_1^2\right)\right]\\
&= C_{\rm quad}\left[6\left(Z_a^2+Z_c^2\right)\left(1+4e_1^2\right)+6\left(Z_b^2+Z_d^2\right)\left(1-e_1^2\right)-4\left(2+3e_1^2\right)\right]. \qquad \text{(D.48)}
\end{aligned}
$$

Now let us look at how equation (D.48) simplifies when we make the substitution $\Delta\Omega = \pi$. With $\Delta\Omega = \pi$, equation (D.46) gives $D_a = -1$, $D_b = D_c = 0$, and $D_d = -\cos(i_{\rm tot})$, where $i_{\rm tot} \equiv i_1 + i_2$ is the mutual inclination between the inner and outer binary orbits. Therefore,

$$
\begin{aligned}
\left(Z_a^2+Z_c^2\right)\Big|_{\Delta h=\pi} &= \cos^2(\omega_1)+\cos^2(i_{\rm tot})\sin^2(\omega_1)\\
&= \frac{1}{2}\left(1+\cos(2\omega_1)\right)+\frac{1}{2}\cos^2(i_{\rm tot})\left(1-\cos(2\omega_1)\right); \qquad \text{(D.49a)}\\
\left(Z_b^2+Z_d^2\right)\Big|_{\Delta h=\pi} &= \sin^2(\omega_1)+\cos^2(i_{\rm tot})\cos^2(\omega_1)\\
&= \frac{1}{2}\left(1-\cos(2\omega_1)\right)+\frac{1}{2}\cos^2(i_{\rm tot})\left(1+\cos(2\omega_1)\right), \qquad \text{(D.49b)}
\end{aligned}
$$

which means, after a bit of work,

$$
\begin{aligned}
\langle\langle R_{\rm quad}\rangle_2\rangle_1\big|_{\Delta h=\pi} &= C_{\rm quad}\left[\left\{(3(1+\cos(2\omega_1))+3\cos^2(i_{\rm tot})(1-\cos(2\omega_1))\right\}\left\{1+4e_1^2\right\}\right.\\
&\quad\left.+\left\{3(1-\cos(2\omega_1))+3\cos^2(i_{\rm tot})(1+\cos(2\omega_1))\right\}\left\{1-e_1^2\right\}-4\left(2+3e_1^2\right)\right]\\
&= C_{\rm quad}\left[3+3\cos(2\omega_1)+3\cos^2(i_{\rm tot})-3\cos^2(i_{\rm tot})\cos(2\omega_1)\right.\\
&\quad+12e_1^2+12e_1^2\cos(2\omega_1)+12e_1^2\cos^2(i_{\rm tot})-12e_1^2\cos^2(i_{\rm tot})\cos(2\omega_1)+3\\
&\quad-3\cos(2\omega_1)+3\cos^2(i_{\rm tot})+3\cos^2(i_{\rm tot})\cos(2\omega_1)-3e_1^2+3e_1^2\cos(2\omega_1)\\
&\quad\left.-3e_1^2\cos^2(i_{\rm tot})-3e_1^2\cos^2(i_{\rm tot})\cos(2\omega_1)-8-12e_1^2\right]\\
&= C_{\rm quad}\left[-2+6\cos^2(i_{\rm tot})-3e_1^2+15e_1^2\cos(2\omega_1)\right.\\
&\quad\left.-15e_1^2\cos^2(i_{\rm tot})\cos(2\omega_1)+9e_1^2\cos^2(i_{\rm tot})\right]\\
&= C_{\rm quad}\left[\left(2+3e_1^2\right)\left(3\cos^2(i_{\rm tot})-1\right)+15e_1^2\sin^2(i_{\rm tot})\cos(2\omega_1)\right]. \qquad \text{(D.50)}
\end{aligned}
$$

D.2.3.2 Octupole order

Next, let us continue with the octupole-order term. I will first average $R_{\rm oct}$, equation (D.41), over the outer orbit employing the true anomaly.

$$
\begin{aligned}
\langle R_{\rm oct}\rangle_2 &= \frac{1}{2\pi}\int_0^{2\pi}R_{\rm oct}\,\mathrm{d}\mathcal{M}_2\\
&= -\frac{32}{15}2\,C_{\rm oct}\left(1-e_2^2\right)^{5/2}\left(\frac{r_1}{a_1}\right)^3\frac{1}{2\pi}\int_0^{2\pi}\left(\frac{a_2}{r_2}\right)^4\underbrace{\frac{1}{2}\left(5\cos^3(\Phi)-3\cos(\Phi)\right)}_{=\bar{P}_3(\cos(\Phi))}\underbrace{\left(\frac{r_2}{a_2}\right)^2\frac{1}{\sqrt{1-e_2^2}}\,\mathrm{d}\theta_2}_{=\mathrm{d}\mathcal{M}_2}
\end{aligned}
$$

$$= -\frac{32}{15} C_{\rm oct} \left(1-e_2^2\right)^{5/2} \left(\frac{r_1}{a_1}\right)^3 \frac{1}{2\pi} \int_0^{2\pi} \left(\frac{1+e_2\cos(\theta_2)}{1-e_2^2}\right)^2 \frac{1}{\sqrt{1-e_2^2}} \left(5\cos^3(\Phi) - 3\cos(\Phi)\right) {\rm d}\theta_2$$

$$= \frac{32}{15} C_{\rm oct} \left(\frac{r_1}{a_1}\right)^3 \frac{1}{2\pi} \int_0^{2\pi} [1+e_2\cos(\theta_2)]^2 \left[5\cos^3(\Phi) - \cos(\Phi)\right] {\rm d}\theta_2. \tag{D.51}$$

Applying equation (D.45) and using the identity $(a+b)^3 = a^3 + 3a^2b + 3ab^2 + b^3$, we get

$$\begin{aligned}
\langle R_{\rm oct}\rangle_2 &= -\frac{32}{15} C_{\rm oct} \left(\frac{r_1}{a_1}\right)^3 \frac{1}{2\pi} \int_0^{2\pi} [1+e_2\cos(\theta_2)]^2 \left[5\tilde{A}_a^3\cos^3(\theta_2) + 15\tilde{A}_a^2\tilde{A}_b\cos^3(\theta_2)\sin(\theta_2)\right. \\
&\quad \left. + 15\tilde{A}_a\tilde{A}_b^2\cos(\theta_2)\sin^2(\theta_2) + 5\tilde{A}_b^3\sin^3(\theta_2) - 3\tilde{A}_a\cos(\theta_2) - 3\tilde{A}_b\sin(\theta_2)\right] {\rm d}\theta_2 \\
&= -\frac{32}{15} C_{\rm oct} \left(\frac{r_1}{a_1}\right)^3 \left[\frac{3}{4} e_2 \cdot 5\tilde{A}_a^3 + \frac{1}{4} e_2 \cdot 15\tilde{A}_a\tilde{A}_b^2 - 3e_2\tilde{A}_a\right] \\
&= -8\, C_{\rm oct} e_2 \left(\frac{r_1}{a_1}\right)^3 \left[\tilde{A}_a^3 + \tilde{A}_a\tilde{A}_b^2 - \frac{4}{5}\tilde{A}_a\right].
\end{aligned} \tag{D.52}$$

Let's do the integration over the inner orbit using the eccentric anomaly,

$$\langle\langle R_{\rm oct}\rangle_2\rangle_1 = \frac{1}{2\pi}\int_0^{2\pi} \langle R_{\rm oct}\rangle_2 \, {\rm d}\mathcal{M}_1 = -8\, C_{\rm oct} e_2 \frac{1}{2\pi} \int_0^{2\pi} \left(\frac{r_1}{a_1}\right)^3 \left[\tilde{A}_a^3 + \tilde{A}_a\tilde{A}_b^2 - \frac{4}{5}\tilde{A}_a\right] \underbrace{\left(\frac{r_1}{a_1}\right) {\rm d}\mathcal{E}_1}_{={\rm d}\mathcal{M}_1}. \tag{D.53}$$

Since all terms in $\tilde{A}_a^3$ and $\tilde{A}_a\tilde{A}_b^2$ involve products of $\cos(\theta_1)$ and $\sin(\theta_1)$ to the third order (see equation D.45), these terms all contain a denominator, which is third order in $1 - e_1\cos(\mathcal{E}_1)$ (see equation D.32), that is, $\tilde{A}_a^3 \propto 1/(1-e_1\cos(\mathcal{E}_1))^3$ and $\tilde{A}_a\tilde{A}_b^2 \propto 1/(1-e_1\cos(\mathcal{E}_1))^3$. Similarly, $\tilde{A}_a \propto 1/(1-e_1\cos(\mathcal{E}_1))$. We therefore need to compute integrals of the form

$$\begin{aligned}
\frac{1}{2\pi}\int_0^{2\pi} (1-e_1\cos(\mathcal{E}_1))\cdot \begin{cases} (\cos(\mathcal{E}_1)-e_1)^3\,{\rm d}\mathcal{E}_1 & = -\frac{5}{8}e_1\left(3+4e_1^2\right); \\ \sin^3(\mathcal{E}_1)\,{\rm d}\mathcal{E}_1 & = 0; \\ (\cos(\mathcal{E}_1)-e_1)^2\sin(\mathcal{E}_1)\,{\rm d}\mathcal{E}_1 & = 0; \\ (\cos(\mathcal{E}_1)-e_1)\sin^2(\mathcal{E}_1)\,{\rm d}\mathcal{E}_1 & = -\frac{5}{8}e_1; \end{cases} \\
\frac{1}{2\pi}\int_0^{2\pi} (1-e_1\cos(\mathcal{E}_1))^3\cdot \begin{cases} (\cos(\mathcal{E}_1)-e_1)\,{\rm d}\mathcal{E}_1 & = -\frac{5}{8}e_1\left(3e_1^2+4\right); \\ \sin(\mathcal{E}_1)\,{\rm d}\mathcal{E}_1 & = 0. \end{cases}
\end{aligned} \tag{D.54}$$

This means that only terms proportional to $\cos^3(\theta_1)$, $\cos(\theta_1)\sin^2(\theta_1)$, and $\cos(\theta_1)$ in equation (D.53) remain after averaging over the inner orbit. Using equation (D.45), the relevant (i.e., nonvanishing) terms are

$$\begin{aligned}
\tilde{A}_a^3 &= Z_a^3\cos^3(\theta_1) + 3Z_aZ_b^2\cos(\theta_1)\sin^2(\theta_1) + ...; \\
\tilde{A}_a\tilde{A}_b^2 &= [Z_a\cos(\theta_1) + Z_b\sin(\theta_1)]\left[Z_c^2\cos^2(\theta_1) + 2Z_cZ_d\cos(\theta_1)\sin(\theta_1) + Z_d^2\sin^2(\theta_1)\right] \\
&= Z_aZ_c^2\cos^3(\theta_1) + \left(Z_aZ_d^2 + 2Z_bZ_cZ_d\right)\cos(\theta_1)\sin^2(\theta_1) + ...,
\end{aligned} \tag{D.55}$$

where the dots denote terms that vanish after integration. Therefore,

$$\begin{aligned}
\langle\langle R_{\rm oct}\rangle_2\rangle_1 &= -8\, C_{\rm oct} e_2 \frac{1}{2\pi}\int_0^{2\pi} \left\{[1-e_1\cos(\mathcal{E}_1)]\left[(\cos(\mathcal{E}_1)-e_1)^3\left(Z_a^3 + Z_aZ_c^2\right)\right.\right. \\
&\quad \left. + \left(1-e_1^2\right)\sin^2(\mathcal{E}_1)(\cos(\mathcal{E}_1)-e_1)\left(3Z_aZ_b^2 + Z_aZ_d^2 + 2Z_bZ_cZ_d\right)\right] \\
&\quad \left. + [1-e_1\cos(\mathcal{E}_1)]^3\left[-\frac{4}{5}Z_a(\cos(\mathcal{E}_1)-e_1)\right]\right\} {\rm d}\mathcal{E}_1 \\
&= -8\, C_{\rm oct} e_2 \left[-\frac{5}{8}e_1\left(3+4e_1^2\right)Z_a\left(Z_a^2+Z_c^2\right) - \frac{5}{8}e_1\left(1-e_1^2\right)\left(3Z_aZ_b + Z_aZ_d^2 + 2Z_bZ_cZ_d\right)\right. \\
&\quad \left. + \frac{4}{5}\frac{5}{8}e_1\left(3e_1^2+4\right)Z_a\right] \\
&= C_{\rm oct} e_1 e_2 \left[Z_a\left\{5\left(3+4e_1^2\right)\left(Z_a^2+Z_c^2\right) - 4\left(3e_1^2+4\right)\right\} + 5\left(1-e_1^2\right)\left(3Z_aZ_b^2 + Z_aZ_d^2 + 2Z_bZ_cZ_d\right)\right].
\end{aligned} \tag{D.56}$$

I will not show it explicitly here, but it can be proven that if one makes the substitution $\Delta\Omega = \pi$, then equation (D.56) reduces to

$$\langle\langle R_{\rm oct}\rangle_2\rangle_1|_{\Delta\Omega=\pi} = C_{\rm oct} e_1 e_2 \left[A\cos(\phi) + 10\cos(i_{\rm tot})\sin^2(i_{\rm tot})\left(1-e_1^2\right)\sin(\omega_1)\sin(\omega_2)\right], \tag{D.57}$$

where $i_{\rm tot} = i_1 + i_2$, and

$$\begin{cases} \cos(\phi) & = Z_a|_{\Delta\Omega=\pi} = -\cos(\omega_1)\cos(\omega_2) - \cos(i_{\rm tot})\sin(\omega_1)\sin(\omega_2); \\ A & = 4 + 3e_1^2 - \dfrac{5}{2}\sin^2(i_{\rm tot})B; \\ B & = 2 + 5e_1^2 - 7e_1^2\cos(2\omega_1). \end{cases} \tag{D.58}$$

Bob: Pfff…. Are we there yet?

Professor: We are getting close. We have derived the orbit-averaged Hamiltonian to the quadrupole and octupole orders, but we still need to derive the equations of motion.

D.3 The Equations of Motion

Professor: Part of our motivation for the Hamiltonian formalism was that the equations of motion can be easily derived. I have mentioned Hamilton's equations before in chapter 2. In this case, we need to ask what are the canonical coordinates and momenta in our problem. This is actually a very technical question but with a simple answer. If you want to know the details, you can refer to chapter 4 of Valtonen and Karttunen (2006). It turns out that, for a particular orbit, the coordinates are the mean anomaly $\mathcal{M}$, the argument of periapsis ω, and the longitude of the ascending node Ω; the associated conjugate momenta are $\tilde{L}$, $\tilde{G}$, and $\tilde{H}$, given by

$$\begin{cases} \tilde{L} & = \mu\sqrt{Ga}; \\ \tilde{G} & = L\sqrt{1-e^2}; \\ \tilde{H} & = G\cos(i). \end{cases} \tag{D.59}$$

Here, μ is the reduced mass. The quantity $\tilde{G}$ is just the magnitude of the angular momentum, and $\tilde{L}$ is $\tilde{G}$ in the case of a circular orbit. Furthermore, $\tilde{H}$ is the z-component of the angular moment. In our case, with two orbits, we have

$$\begin{cases} \tilde{L}_j & = \mu_j\sqrt{Ga_j}; \\ \tilde{G}_j & = L_j\sqrt{1-e_j^2}; \\ \tilde{H}_j & = G_j\cos(i_j), \end{cases} \tag{D.60}$$

where

$$\mu_1 = \frac{m_1 m_2}{m_1+m_2}; \qquad \mu_2 = \frac{(m_1+m_2)m_3}{m_1+m_2+m_3}. \tag{D.61}$$

In terms of these canonical coordinates, which are also known as the Delaunay elements, the Hamiltonian equations of motion are given by

$$\begin{cases} \dot{\tilde{L}}_j & = \dfrac{\partial\mathcal{H}}{\partial\mathcal{M}_j}; & \dot{\mathcal{M}}_j = -\dfrac{\partial\mathcal{H}}{\partial\tilde{L}_j}; \\ \dot{\tilde{G}}_j & = \dfrac{\partial\mathcal{H}}{\partial\omega_j}; & \dot{\omega}_j = -\dfrac{\partial\mathcal{H}}{\partial\tilde{G}_j}; \\ \dot{\tilde{H}}_j & = \dfrac{\partial\mathcal{H}}{\partial\Omega_j}; & \dot{\Omega}_j = -\dfrac{\partial\mathcal{H}}{\partial\tilde{H}_j}. \end{cases} \tag{D.62}$$

There is one thing we can immediately conclude from these equations. As a consequence of the averaging procedure, $\mathcal{H}$ is no longer dependent on $\mathcal{M}_j$. Therefore, the $\tilde{L}_j$ are constant, and thus so are a_j. Some people like to call this consequence of averaging "adiabatic invariance."

Also, let me reiterate that the constraint $\Delta\Omega = \pi$ does not imply that the Hamiltonian is independent of Ω_1 and Ω_2. Therefore, the $\dot{\tilde{H}}_j$ are not necessarily constant. Strictly speaking, it is only correct to make the substitution $\Delta\Omega = \pi$ *after* deriving the equations of motion.

However, having said that, the derivatives in equations (D.62) commute with setting $\Delta\Omega = \pi$. Therefore, it is actually okay to use the Hamiltonian in which we have already made the substitution $\Delta\Omega = \pi$. Nonetheless, at

the quadrupole order, it is not very complicated to do the substitution after deriving the equations of motion, so I will be doing that here, just for illustration. For the octupole-order term, it does simplify matters a lot to make the substitution in the Hamiltonian, so I will be taking that approach.

You might notice that the inclinations of both orbits, i_j, are not part of the canonical set of coordinates. This is essentially because these are constrained by the angular momenta through equation (D.26).

I should also mention that, regarding the differentiations with respect to $\tilde{G}_j$, we should be aware that the quantities $C_{\rm quad}$ and $C_{\rm oct}$ depend on $\tilde{G}_2$ through their dependence on e_2. With a little algebra, we can express these quantities in terms of $\tilde{L}_j$ and $\tilde{G}_j$ using equation (D.60), that is,

$$C_{\rm quad} = \frac{G^2}{16}\frac{(m_1+m_2)^7}{(m_1+m_2+m_3)^3}\frac{m_3^7}{(m_1m_2)^3}\frac{\tilde{L}_1^4}{\tilde{L}_2^3\tilde{G}_2^3}; \tag{D.63}$$

$$C_{\rm oct} = -\frac{15}{16}\frac{G^2}{4}\frac{m_1+m_2)^9}{(m_1+m_2+m_3)^4}\frac{m_3^8(m_1-m_2)}{(m_1m_2)^5}\frac{\tilde{L}_1^6}{\tilde{L}_2^3\tilde{G}_2^5}. \tag{D.64}$$

Bob: It is a good thing that you put a tilde on the G (the angular momentum) to distinguish it from the gravitational constant!

Professor: Indeed. Next, I will explicitly derive the equations of motion to the quadrupole and octupole orders. However, these derivations are rather technical and not particularly insightful. So I will not hold it against you if you skip this and return for the analytic solutions in section D.4.

Bob: Okay, good to know.

D.3.1 Quadrupole Order

Professor: I will start with the equations of motion at the quadrupole order. From equations (D.40) and (D.48), we can see that $\langle\langle R_{\rm quad}\rangle_2\rangle_1$ depends on $e_1, e_2, \omega_1, i_1, i_2$, and $\Delta\Omega$. There is no dependence on ω_2, which, through Hamilton's equations of motion, implies that $\tilde{G}_2 = \tilde{L}_2\sqrt{1-e_2^2}$ is constant. Therefore, e_2 is constant at the quadrupole order. Furthermore, note that i_1 and i_2 depend on $\tilde{G}_1$ and $\tilde{G}_2$ through equation (D.26) and that e_1 is a function of $\tilde{G}_1$. Therefore, applying Hamilton's equation of motion and using the chain rule for differentiation,

$$\dot{\omega}_{1,\rm quad} = -\left.\frac{\partial\langle\langle R_{\rm quad}\rangle_2\rangle_1}{\partial\tilde{G}_1}\right|_{\Delta\Omega=\pi} = -\left.\frac{\partial\langle\langle R_{\rm quad}\rangle_2\rangle_1}{\partial e_1}\frac{\partial e_1}{\tilde{G}_1}\right|_{\Delta\Omega=\pi} - \left.\frac{\partial\langle\langle R_{\rm quad}\rangle_2\rangle_1}{\partial i_1}\frac{\partial i_1}{\partial\tilde{G}_1}\right|_{\Delta\Omega=\pi}. \tag{D.65}$$

From equation (D.60), it follows that

$$\frac{\partial e_1}{\partial\tilde{G}_1} = \frac{\partial}{\partial\tilde{G}_1}\left(\sqrt{1-\left(\frac{\tilde{G}_1}{\tilde{L}_1}\right)^2}\right) = \frac{-1}{\sqrt{1-(\tilde{G}_1/\tilde{L}_1)^2}}\frac{\tilde{G}_1}{\tilde{L}_1^2} = \frac{-1}{e_1}\frac{\tilde{G}_1}{\tilde{G}_1^2}(1-e_1^2) = -\frac{1-e_1^2}{e_1\tilde{G}_1}. \tag{D.66}$$

From equation (D.48), we have

$$\begin{aligned}\frac{\partial\langle\langle R_{\rm quad}\rangle_2\rangle_1}{\partial e_1} &= C_{\rm quad}\left[6\cdot 8\left(Z_a^2+Z_c^2\right)e_1 - 6\cdot 2\left(Z_b^2+Z_d^2\right)e_1 - 4\cdot 6e_1\right]\\ &= 12C_{\rm quad}e_1\left[4\left(Z_a^2+Z_c^2\right)-\left(Z_b^2+Z_d^2\right)-2\right].\end{aligned} \tag{D.67}$$

Therefore, with equation (D.49),

$$\begin{aligned}\left.\frac{\partial\langle\langle R_{\rm quad}\rangle_2\rangle_1}{\partial e_1}\right|_{\Delta\Omega=\pi} &= 12C_{\rm quad}e_1\Big[2(1+\cos(2\omega_1))+2(1-\cos(2\omega_1))\cos^2(i_{\rm tot})\\ &\quad -\frac{1}{2}(1-\cos(2\omega_1))-\frac{1}{2}(1+\cos(2\omega_1))\times\cos^2(i_{\rm tot})-2\Big]\\ &= 12C_{\rm quad}e_1\Big[2+2\cos(2\omega_1)+2\cos^2(i_{\rm tot})-2\cos(2\omega_1)\cos^2(i_{\rm tot})-\frac{1}{2}\\ &\quad +\frac{1}{2}\cos(2\omega_1)-\frac{1}{2}\cos^2(i_{\rm tot})-\frac{1}{2}\cos(2\omega_1)\cos^2(i_{\rm tot})-2\Big]\\ &= 6C_{\rm quad}e_1\left[5\cos(2\omega_1)+3\cos^2(i_{\rm tot})-5\cos(2\omega_1)\cos^2(i_{\rm tot})-1\right].\end{aligned} \tag{D.68}$$

From the definitions in equation (D.46), it follows that

$$\begin{cases} Z_a^2 + Z_c^2 &= (D_a\cos(\omega_1) - D_b\sin(\omega_1))^2 + (D_c\cos(\omega_1) + D_d\sin(\omega_1))^2; \\ Z_b^2 + Z_d^2 &\equiv (D_a\sin(\omega_1) + D_b\cos(\omega_1))^2 + (D_c\sin(\omega_1) - D_d\cos(\omega_1))^2, \end{cases} \tag{D.69}$$

which means that

$$\begin{aligned} \frac{\partial\left(Z_a^2 + Z_c^2\right)}{\partial i_1} &= 2\left(D_a\cos(\omega_1) - D_b\sin(\omega_1)\right)\left(\frac{\partial D_a}{\partial i_1}\cos(\omega_1) - \frac{\partial D_b}{\partial i_1}\sin(\omega_1)\right) \\ &\quad + 2\left(D_c\cos(\omega_1) + D_d\sin(\omega_1)\right)\left(\frac{\partial D_c}{\partial i_1}\cos(\omega_1) + \frac{\partial D_d}{\partial i_1}\sin(\omega_1)\right) \\ &= 2\left(\cos(\Delta\Omega)\cos(\omega_1) - \cos(i_1)\sin(\Delta h)\sin(\omega_1)\right)\sin(i_1)\sin(\Delta\Omega)\sin(\omega_1) \\ &\quad + 2\left\{\cos(i_2)\sin(\Delta\Omega)\cos(\omega_1) + \sin(\omega_1)\left[\cos(i_1)\cos(i_2)\cos(\Delta\Omega)\right.\right. \\ &\quad \left.\left. + \sin(i_1)\sin(i_2)\right]\right\}\sin(\omega_1)\left(-\sin(i_1)\cos(i_2)\cos(\Delta h) + \cos(i_1)\sin(i_2)\right), \end{aligned} \tag{D.70}$$

and so

$$\begin{aligned} \left.\frac{\partial\left(Z_a^2 + Z_c^2\right)}{\partial i_1}\right|_{\Delta\Omega=\pi} &= 2\sin(\omega_1)\left[-\cos(i_1)\cos(i_2) + \sin(i_1)\sin(i_2)\right]\sin(\omega_1)\left[\sin(i_1)\cos(i_2)\right. \\ &\quad \left. + \cos(i_1)\sin(i_2)\right] = -2\sin^2(\omega_1)\cos(i_{\text{tot}})\sin(i_{\text{tot}}). \end{aligned} \tag{D.71}$$

Similarly, it may be shown that

$$\left.\frac{\partial\left(Z_b^2 + Z_d^2\right)}{\partial i_1}\right|_{\Delta\Omega=\pi} = -2\cos^2(\omega_1)\cos(i_{\text{tot}})\sin(i_{\text{tot}}). \tag{D.72}$$

With these results, it follows from equation (D.48) that

$$\begin{aligned} \left.\frac{\partial\langle\langle R_{\text{quad}}\rangle_2\rangle_1}{\partial i_1}\right|_{\Delta\Omega=\pi} &= 6C_{\text{quad}}\left[-2\sin^2(\omega_1)\cos(i_{\text{tot}})\sin(i_{\text{tot}})\left(1 + 4e_1^2\right)\right. \\ &\quad \left. -2\cos^2(\omega_1)\cos(i_{\text{tot}})\sin(i_{\text{tot}})\left(1 - e_1^2\right)\right] \\ &= 6C_{\text{quad}}\cos(i_{\text{tot}})\sin(i_{\text{tot}})\left[-2 - 8e_1^2\sin^2(\omega_1) + 2e_1^2\cos^2(\omega_1)\right] \\ &= 6C_{\text{quad}}\cos(i_{\text{tot}})\sin(i_{\text{tot}})\left[-2 - 4e_1^2 + 4e_1^2\cos(2\omega_1) + e_1^2 + e_1^2\cos(2\omega_1)\right] \\ &= -6C_{\text{quad}}\cos(i_{\text{tot}})\sin(i_{\text{tot}})\left[2 + e_1^2(3 - 5\cos(2\omega_1))\right]. \end{aligned} \tag{D.73}$$

Furthermore, differentiating $\tilde{G}_{\text{tot}}^2 = \tilde{G}_1^2 + \tilde{G}_2^2 + 2\tilde{G}_1\tilde{G}_2\cos(i_{\text{tot}})$ with $\tilde{G}_2$, i_2, and $\tilde{G}_{\text{tot}}$ constant gives

$$\begin{aligned} 0 &= 2\tilde{G}_1\,\mathrm{d}\tilde{G}_1 + 2\tilde{G}_2\cos(i_{\text{tot}})\,\mathrm{d}\tilde{G}_1 - 2\tilde{G}_1\tilde{G}_2\sin(i_{\text{tot}})\,\mathrm{d}i_1 = 2(\tilde{G}_1 + \tilde{G}_2\cos(i_{\text{tot}}))\,\mathrm{d}\tilde{G}_1 \\ &\quad - 2\tilde{G}_1\tilde{G}_2\sin(i_{\text{tot}})\,\mathrm{d}i_1 \Rightarrow \frac{\partial i_1}{\partial\tilde{G}_1} = \frac{1}{\sin(i_{\text{tot}})}\left[\frac{\cos(i_{\text{tot}})}{\tilde{G}_1} + \frac{1}{\tilde{G}_2}\right]. \end{aligned} \tag{D.74}$$

Combining equations (D.65), (D.66), (D.68), (D.73), and (D.74) yields

$$\begin{aligned} \dot{\omega}_{1,\text{quad}} &= \frac{1 - e_1^2}{e_1\tilde{G}_1}6C_{\text{quad}}\,e_1\left[5\cos(2\omega_1) + 3\cos^2(i_{\text{tot}}) - 5\cos(2\omega_1)\cos^2(i_{\text{tot}}) - 1\right] \\ &\quad + \frac{6C_{\text{quad}}}{\sin(i_{\text{tot}})}\left[\frac{\cos(i_{\text{tot}})}{\tilde{G}_1} + \frac{1}{G_2}\right]\cos(i_{\text{tot}})\sin(i_{\text{tot}})\left[2 + e_1^2(3 - 5\cos(2\omega_1))\right] \\ &= 6C_{\text{quad}}\left\{\frac{1}{\tilde{G}_1}\left[5\cos(2\omega_1) + 3\cos^2(i_{\text{tot}}) - 5\cos(2\omega_1)\cos^2(i_{\text{tot}}) - 1 - 5e_1^2\cos(2\omega_1)\right.\right. \\ &\quad -3e_1^2\cos^2(i_{\text{tot}}) + 5e_1^2\cos(2\omega_1)\cos^2(i_{\text{tot}}) + e_1^2 + 2\cos^2(i_{\text{tot}}) + 3e_1^2\cos^2(i_{\text{tot}}) \\ &\quad \left.\left. -5e_1^2\cos^2(i_{\text{tot}})\cos(2\omega_1)\right] + \frac{\cos(i_{\text{tot}})}{\tilde{G}_2}\left[2 + e_1^2(3 - 5\cos(2\omega_1))\right]\right\} \end{aligned}$$

$$= 6\,C_{\rm quad}\left\{\frac{1}{\tilde{G}_1}\left[4\cos^2(i_{\rm tot})+(5\cos(2\omega_1)-1)\left(1-e_1^2-\cos^2(i_{\rm tot})\right)\right]\right.$$
$$\left.+\frac{\cos(i_{\rm tot})}{\tilde{G}_2}\left[2+e_1^2(3-5\cos(2\omega_1))\right]\right\}. \tag{D.75}$$

To compute $\dot{\omega}_{2,\rm quad}$, note that $\langle\langle R_{\rm quad}\rangle_2\rangle_1$ has the form $\langle\langle R_{\rm quad}\rangle_2\rangle_1 = C_{\rm quad}(\tilde{G}_2)\times f_{\rm quad}(i_2(\tilde{G}_2))$, where the function $f_{\rm quad}$ is defined by equation (D.48), that is, $f_{\rm quad} = 6\left(Z_a^2+Z_c^2\right)\left(1+4e_1^2\right)+6\left(Z_b^2+Z_d^2\right)\left(1-e_1^2\right)-4\left(2+3e_1^2\right)$. Therefore,

$$\dot{\omega}_{2,\rm quad} = -\left.\frac{\partial\langle\langle R_{\rm quad}\rangle_2\rangle_1}{\partial\tilde{G}_2}\right|_{\Delta\Omega=\pi} = -\left.\frac{\partial C_{\rm quad}}{\partial\tilde{G}_2}f_{\rm quad}\right|_{\Delta\Omega=\pi} - C_{\rm quad}\left.\frac{\partial f}{\partial i_2}\frac{\partial i_2}{\partial\tilde{G}_2}\right|_{\Delta\Omega=\pi}. \tag{D.76}$$

From equation (D.63), it follows that $\partial C_{\rm quad}/\partial\tilde{G}_2 = -3\,C_{\rm quad}/\tilde{G}_2$. Furthermore, with similar calculations as above, the following relations follow from equation (D.69):

$$\left.\frac{\partial\left(Z_a^2+Z_c^2\right)}{\partial i_2}\right|_{\Delta\Omega=\pi} = -2\sin^2(\omega_1)\cos(i_{\rm tot})\sin(i_{\rm tot});$$
$$\left.\frac{\partial\left(Z_b^2+Z_d^2\right)}{\partial i_2}\right|_{\Delta\Omega=\pi} = -2\cos^2(\omega_1)\cos(i_{\rm tot})\sin(i_{\rm tot}), \tag{D.77}$$

from which it follows that (analogously to the derivation in equation D.73)

$$\left.\frac{\partial f_{\rm quad}}{\partial i_2}\right|_{\Delta\Omega=\pi} = 6\left.\frac{\partial\left(Z_a^2+Z_c^2\right)}{\partial i_2}\right|_{\Delta\Omega=\pi}\times\left(1+4e_1^2\right)+6\left.\frac{\partial\left(Z_b^2+Z_d^2\right)}{\partial i_2}\right|_{\Delta\Omega=\pi}\times\left(1-e_1^2\right)$$
$$= -6\cos(i_{\rm tot})\sin(i_{\rm tot})\left[2+e_1^2(3-5\cos(2\omega_1))\right]. \tag{D.78}$$

Similarly to equation (D.74), but now with $\tilde{G}_2$ and i_2 varying, and $\tilde{G}_1$, i_1, and $\tilde{G}_{\rm tot}$ constant, one finds

$$\frac{\partial i_2}{\partial\tilde{G}_2} = \frac{1}{\sin(i_{\rm tot})}\left[\frac{\cos(i_{\rm tot})}{\tilde{G}_2}+\frac{1}{\tilde{G}_1}\right]. \tag{D.79}$$

Combining equations (D.76), (D.78), and (D.79), and using equation (D.50) for $f_{\rm quad}\big|_{\Delta\Omega=\pi}$, gives

$$\dot{\omega}_2 = \frac{3\,C_{\rm quad}}{\tilde{G}_2}\left[\left(2+3e_1^2\right)\left(3\cos^2(i_{\rm tot})-1\right)+15e_1^2\left(1-\cos^2(i_{\rm tot})\right)\cos(2\omega_1)\right]$$
$$+\frac{6\,C_{\rm quad}}{\sin(i_{\rm tot})}\left[\frac{\cos(i_{\rm tot})}{\tilde{G}_2}+\frac{1}{\tilde{G}_1}\right]\sin(i_{\rm tot})\cos(i_{\rm tot})\left[2+e_1^2(3-5\cos(2\omega_1))\right]$$
$$= \frac{6\,C_{\rm quad}}{\tilde{G}_1}\cos(i_{\rm tot})\left[2+e_1^2(3-5\cos(2\omega_1))\right]+\frac{3\,C_{\rm quad}}{\tilde{G}_2}\left[6\cos^2(i_{\rm tot})-2+9e_1^2\cos^2(i_{\rm tot})\right.$$
$$-3e_1^2+15e_1^2\cos(2\omega_1)-15e_1^2\cos^2(i_{\rm tot})\cos(2\omega_1)+4\cos^2(i_{\rm tot})+6e_1^2\cos^2(i_{\rm tot})$$
$$\left.-10e_1^2\cos(2\omega_1)\cos^2(i_{\rm tot})\right]$$
$$= 3\,C_{\rm quad}\left\{\frac{2\cos(i_{\rm tot})}{\tilde{G}_1}\left[2+e_1^2(3-5\cos(2\omega_1))\right]\right.$$
$$\left.+\frac{1}{\tilde{G}_2}\left[4+6e_1^2+\left(5\cos^2(i_{\rm tot})-3\right)\left(2+e_1^2(3-5\cos(2\omega_1))\right)\right]\right\}. \tag{D.80}$$

Lastly, the change of the inner orbit eccentricity with time valid in the quadrupole order follows from equation (D.62) as follows.

$$\dot{e}_{1,\rm quad} = \left.\frac{\partial e_1}{\partial\tilde{G}_1}\dot{\tilde{G}}_1\right|_{\Delta\Omega=\pi} = \left.\frac{\partial e_1}{\partial\tilde{G}_1}\frac{\partial\langle\langle R_{\rm quad}\rangle_2\rangle_1}{\partial\omega_1}\right|_{\Delta\Omega=\pi}. \tag{D.81}$$

The derivative $\partial e_1/\partial \tilde{G}_1$ is given by equation (D.66), whereas $\partial\langle\langle R_{\rm quad}\rangle_2\rangle_1/\partial\omega_1$ follows from equations (D.48) and (D.69), that is,

$$\begin{aligned}
\frac{\partial\left(Z_a^2+Z_c^2\right)}{\partial\omega_1} &= 2\left[D_a\cos(\omega_1)-D_b\sin(\omega_1)\right]\left[-D_a\sin(\omega_1)-D_b\cos(\omega_1)\right] \\
&\quad +2\left[D_c\cos(\omega_1)+D_d\sin(\omega_1)\right]\left[-D_c\sin(\omega_1)+D_d\cos(\omega_1)\right]\Rightarrow \\
\left.\frac{\partial\left(Z_a^2+Z_c^2\right)}{\partial\omega_1}\right|_{\Delta\Omega=\pi} &= -2\cos(\omega_1)\sin(\omega_1)+2\sin(\omega_1)\cos(\omega_1)\cos^2(i_{\rm tot}) \\
&= -\sin(2\omega_1)\sin^2(i_{\rm tot}); \qquad \text{(D.82)} \\
\frac{\partial\left(Z_b^2+Z_d^2\right)}{\partial\omega_1} &= 2\left[D_a\sin(\omega_1)+D_b\cos(\omega_1)\right]\left[D_a\cos(\omega_1)-D_b\sin(\omega_1)\right] \\
&\quad +2\left[D_c\sin(\omega_1)-D_d\cos(\omega_1)\right]\left[D_c\cos(\omega_1)+D_d\sin(\omega_1)\right]\Rightarrow \\
\left.\frac{\partial\left(Z_b^2+Z_d^2\right)}{\partial\omega_1}\right|_{\Delta\Omega=\pi} &= 2\sin(\omega_1)\cos(\omega_1)-2\sin(\omega_1)\cos(\omega_1)\cos^2(i_{\rm tot}) \\
&= \sin(2\omega_1)\sin^2(i_{\rm tot}), \qquad \text{(D.83)}
\end{aligned}$$

such that

$$\begin{aligned}
\left.\frac{\partial\langle\langle R_{\rm quad}\rangle_2\rangle_1}{\partial\omega_1}\right|_{\Delta\Omega=\pi} &= 6C_{\rm quad}\left[\left.\frac{\partial\left(Z_a^2+Z_c^2\right)}{\partial\omega_1}\right|_{\Delta\Omega=\pi}\times\left(1+4e_1^2\right)+\left.\frac{\partial\left(Z_b^2+Z_d^2\right)}{\partial\omega_1}\right|_{\Delta\Omega=\pi}\times\left(1-e_1^2\right)\right] \\
&= 6C_{\rm quad}\sin(2\omega_1)\sin^2(i_{\rm tot})\left[-\left(1+4e_1^2\right)+\left(1-e_1^2\right)\right] \\
&= -30C_{\rm quad}e_1^2\sin(2\omega_1)\sin^2(i_{\rm tot}). \qquad \text{(D.84)}
\end{aligned}$$

Combining equations (D.81), (D.66), and (D.84) gives

$$\dot{e}_{1,\rm quad} = C_{\rm quad}\frac{1-e_1^2}{\tilde{G}_1}30\,e_1\sin^2(i_{\rm tot})\sin(2\omega_1). \qquad \text{(D.85)}$$

As I mentioned before, since $\langle\langle R_{\rm quad}\rangle_2\rangle_1$ does not depend on ω_2, we have

$$\dot{e}_{2,\rm quad} = 0. \qquad \text{(D.86)}$$

D.3.2 Octupole Order

In contrast to the quadrupole order, the octupole-order term $\langle\langle R_{\rm oct}\rangle_2\rangle_1$ does depend on ω_2, meaning that e_2 is no longer constant in this limit. As I mentioned before, it is allowed to substitute $\Delta\Omega=\pi$ in the perturbing term without affecting the equations for $\dot{\omega}_j$ and $\dot{e}_j$. Therefore, I here use equation (D.57), in which the substitution has already been made.

The equations for $\dot{\omega}_{j,\rm oct}$ follow from

$$\dot{\omega}_{j,\rm oct} = -\frac{\partial\left(\langle\langle R_{\rm oct}\rangle_2\rangle_1|_{\Delta\Omega=\pi}\right)}{\partial\tilde{G}_j} = -\frac{\partial\left(\langle\langle R_{\rm oct}\rangle_2\rangle_1|_{\Delta\Omega=\pi}\right)}{\partial e_j}\frac{\partial e_j}{\partial\tilde{G}_j}-\frac{\partial\left(\langle\langle R_{\rm oct}\rangle_2\rangle_1|_{\Delta\Omega=\pi}\right)}{\partial i_j}\frac{\partial i_j}{\partial\tilde{G}_j}. \qquad \text{(D.87)}$$

We have already computed the quantities $\partial e_j/\tilde{G}_j$ and $i_j/\tilde{G}_j$ in the above derivation for the quadrupole order (see equations D.66, D.74, and D.79), that is,

$$\frac{\partial e_j}{\partial\tilde{G}_j} = -\frac{1-e_j^2}{e_j\tilde{G}_j}; \quad \frac{\partial i_j}{\tilde{G}_j} = \frac{1}{\sin(i_{\rm tot})}\left[\frac{\cos(i_{\rm tot})}{\tilde{G}_j}+\frac{1}{\tilde{G}_{3-j}}\right]. \qquad \text{(D.88)}$$

The other required quantities are obtained from the octupole-order term, equations (D.57) and (D.58),

$$\begin{aligned}
\frac{\partial(\langle\langle R_{\rm oct}\rangle_2\rangle_1|_{\Delta\Omega=\pi})}{\partial e_1} &= C_{\rm oct}e_2\left[A\cos(\phi)+10\cos(i_{\rm tot})\sin^2(i_{\rm tot})\left(1-e_1^2\right)\sin(\omega_1)\sin(\omega_2)\right]\\
&\quad + C_{\rm oct}e_2\left[e_1\frac{\partial A}{\partial e_1}\cos(\phi)-20e_1^2\cos(i_{\rm tot})\sin^2(i_{\rm tot})\sin(\omega_1)\sin(\omega_2)\right];\\
e_1\frac{\partial A}{\partial e_1} &= 2\left[3e_1^2-\frac{5}{2}\sin^2(i_{\rm tot})\left(5e_1^2-7e_1^2\cos(2\omega_1)\right)\right]\\
&= 2\left[4+3e_1^2-\frac{5}{2}\sin^2(i_{\rm tot})\left(2+5e_1^2-7e_1^2\cos(2\omega_1)\right)\right]-8+10\sin^2(i_{\rm tot})\\
&= 2A-10\cos^2(i_{\rm tot})+2\Rightarrow\\
\frac{\partial(\langle\langle R_{\rm oct}\rangle_2\rangle_1|_{\Delta\Omega=\pi})}{\partial e_1} &= C_{\rm oct}e_2\left[\left(3A-10\cos^2(i_{\rm tot})+2\right)\cos(\phi)\right.\\
&\quad \left.+10\cos(i_{\rm tot})\sin^2(i_{\rm tot})\left(1-3e_1^2\right)\sin(\omega_1)\sin(\omega_2)\right]; \qquad \text{(D.89)}\\
\frac{\partial(\langle\langle R_{\rm oct}\rangle_2\rangle_1|_{\Delta\Omega=\pi})}{\partial i_1} &= C_{\rm oct}e_1e_2\left[\frac{\partial A}{\partial i_1}\cos(\phi)+A\frac{\partial\cos(\phi)}{\partial i_1}\right.\\
&\quad \left.+10\frac{\partial}{\partial i_1}\left\{\cos(i_{\rm tot})\left(1-\cos^2(i_{\rm tot})\right)\right\}\left(1-e_1^2\right)\sin(\omega_1)\sin(\omega_2)\right]\\
&= C_{\rm oct}e_1e_2\left[-5B\sin(i_{\rm tot})\cos(i_{\rm tot})\cos(\phi)+A\sin(i_{\rm tot})\sin(\omega_1)\sin(\omega_2)\right.\\
&\quad \left.+10\sin(i_{\rm tot})\left(3\cos^2(i_{\rm tot})-1\right)\left(1-e_1^2\right)\sin(\omega_1)\sin(\omega_2)\right]\\
&= C_{\rm oct}e_1e_2\sin(i_{\rm tot})\left[\sin(\omega_1)\sin(\omega_2)\left\{A+10\left(3\cos^2(i_{\rm tot})-1\right)\left(1-e_1^2\right)\right\}\right.\\
&\quad \left.-5B\cos(i_{\rm tot})\cos(\phi)\right]. \qquad \text{(D.90)}
\end{aligned}$$

Combining equations (D.87), (D.88), (D.89), and (D.90) gives

$$\begin{aligned}
\dot\omega_{1,\rm oct} &= \frac{1-e_1^2}{e_1\tilde G_1}C_{\rm oct}e_2\left[\cos(\phi)\left(3A-10\cos^2(i_{\rm tot})+2\right)+10\cos(i_{\rm tot})\sin^2(i_{\rm tot})\left(1-3e_1^2\right)\right.\\
&\quad \left.\times\sin(\omega_1)\sin(\omega_2)\right]-C_{\rm oct}e_1e_2\left(\frac{1}{\tilde G_2}+\frac{\cos(i_{\rm tot})}{\tilde G_1}\right)\{\sin(\omega_1)\sin(\omega_2)\\
&\quad \times\left[A+10\left(3\cos^2(i_{\rm tot})-1\right)\left(1-e_1^2\right)\right]-5B\cos(i_{\rm tot})\cos(\phi)\}. \qquad \text{(D.91)}
\end{aligned}$$

For $\dot\omega_{2,\rm oct}$, note that $\langle\langle R_{\rm oct}\rangle_2\rangle_1|_{\Delta\Omega=\pi}=C_{\rm oct}(\tilde G_2)e_1e_2f_{\rm oct}(e_1,i_1,i_2,\omega_1,\omega_2)$, where the function $f_{\rm oct}$ is defined by equation (D.57). Therefore,

$$\dot\omega_{2,\rm oct}=-\frac{\partial(\langle\langle R_{\rm oct}\rangle_2\rangle_1|_{\Delta\Omega=\pi})}{\partial\tilde G_2}=-\frac{\partial C_{\rm oct}}{\partial\tilde G_2}e_1e_2f_{\rm oct}-C_{\rm oct}e_1\frac{\partial e_2}{\partial\tilde G_2}f_{\rm oct}-C_{\rm oct}e_1e_2\frac{\partial f_{\rm oct}}{\partial i_2}\frac{\partial i_2}{\partial\tilde G_2}. \qquad \text{(D.92)}$$

From equation (D.64), it follows that $\partial C_{\rm oct}/\partial\tilde G_2=-5C_{\rm oct}/\tilde G_2$. Furthermore, $\partial f_{\rm oct}/\partial i_2=\partial f_{\rm oct}/\partial i_1$, and the latter quantity is given by equation (D.90). Therefore, equation (D.92) gives

$$\begin{aligned}
\dot\omega_{2,\rm oct} &= 5C_{\rm oct}\frac{e_1e_2}{\tilde G_2}\left[A\cos(\phi)+10\cos(i_{\rm tot})\sin^2(i_{\rm tot})\left(1-e_1^2\right)\sin(\omega_1)\sin(\omega_2)\right]\\
&\quad + C_{\rm oct}e_1\frac{1-e_2^2}{e_2\tilde G_2}\left[A\cos(\phi)+10\cos(i_{\rm tot})\sin^2(i_{\rm tot})\left(1-e_1^2\right)\sin(\omega_1)\sin(\omega_2)\right]\\
&\quad - C_{\rm oct}e_1e_2\frac{1}{\sin(i_{\rm tot})}\left(\frac{1}{\tilde G_1}+\frac{\cos(i_{\rm tot})}{\tilde G_2}\right)\sin(i_{\rm tot})\{\sin(\omega_1)\sin(\omega_2)[A+10\\
&\quad \times\left(3\cos^2(i_{\rm tot})-1\right)\left(1-e_1^2\right)]-5B\cos(i_{\rm tot})\cos(\phi)\}
\end{aligned}$$

$$= C_{\rm oct} e_1 \left\{ \sin(\omega_1)\sin(\omega_2) \left[\frac{4e_2^2+1}{e_2\tilde{G}_2} 10\cos(i_{\rm tot})\sin^2(i_{\rm tot})\left(1-e_1^2\right) - e_2\left(\frac{1}{\tilde{G}_1}+\frac{\cos(i_{\rm tot})}{\tilde{G}_2}\right) \right.\right.$$
$$\left. \times\left(A+10\left(3\cos^2(i_{\rm tot})-1\right)\left(1-e_1^2\right)\right)\right]$$
$$\left. +\cos(\phi)\left[5B\cos(i_{\rm tot})e_2\left(\frac{1}{\tilde{G}_1}+\frac{\cos(i_{\rm tot})}{\tilde{G}_2}\right)+\frac{4e_2^2+1}{e_2\tilde{G}_2}A\right]\right\}. \tag{D.93}$$

Lastly, the $\dot{e}_{j,\rm oct}$ follow from

$$\dot{e}_{j,\rm oct} = \left.\frac{\partial e_j}{\partial \tilde{G}_j}\dot{\tilde{G}}_j\right|_{\Delta\Omega=\pi} = \frac{\partial e_j}{\partial \tilde{G}_j}\frac{\partial\left(\langle\langle R_{\rm oct}\rangle_2\rangle_1|_{\Delta\Omega=\pi}\right)}{\partial\omega_j}. \tag{D.94}$$

The required quantities follow from equations (D.57) and (D.58),

$$\frac{\partial\left(\langle\langle R_{\rm oct}\rangle_2\rangle_1|_{\Delta\Omega=\pi}\right)}{\partial\omega_1} = C_{\rm oct}e_1e_2\left[\frac{\partial A}{\partial\omega_1}\cos(\phi)+A\frac{\partial\cos(\phi)}{\partial\omega_1}+10\cos(i_{\rm tot})\sin^2(i_{\rm tot})\left(1-e_1^2\right)\right.$$
$$\left.\times\cos(\omega_1)\sin(\omega_2)\right]; \tag{D.95}$$

$$\frac{\partial\left(\langle\langle R_{\rm oct}\rangle_2\rangle_1|_{\Delta\Omega=\pi}\right)}{\partial\omega_2} = C_{\rm oct}e_1e_2\left[A\frac{\partial\cos(\phi)}{\partial\omega_2}+10\cos(i_{\rm tot})\sin^2(i_{\rm tot})\left(1-e_1^2\right)\sin(\omega_1)\cos(\omega_2)\right]; \tag{D.96}$$

$$\frac{\partial A}{\partial\omega_1} = -35\sin^2(i_{\rm tot})e_1^2\sin(2\omega_1); \tag{D.97}$$

$$\frac{\partial\cos(\phi)}{\partial\omega_1} = \sin(\omega_1)\cos(\omega_2)-\cos(i_{\rm tot})\cos(\omega_1)\sin(\omega_2); \tag{D.98}$$

$$\frac{\partial\cos(\phi)}{\partial\omega_2} = \cos(\omega_1)\sin(\omega_2)-\cos(i_{\rm tot})\sin(\omega_1)\cos(\omega_2). \tag{D.99}$$

Therefore,

$$\dot{e}_{1,\rm oct} = C_{\rm oct}e_2\frac{1-e_1^2}{\tilde{G}_1}\left[35\cos(\phi)\sin^2(i_{\rm tot})e_1^2\sin(2\omega_1)-10\cos(i_{\rm tot})\sin^2(i_{\rm tot})\cos(\omega_1)\sin(\omega_2)\right.$$
$$\left.\times\left(1-e_1^2\right)-A\left\{\sin(\omega_1)\cos(\omega_2)-\cos(i_{\rm tot})\cos(\omega_1)\sin(\omega_2)\right\}\right]; \tag{D.100}$$

$$\dot{e}_{2,\rm oct} = -C_{\rm oct}e_1\frac{1-e_2^2}{\tilde{G}_2}\left[10\cos(i_{\rm tot})\sin^2(i_{\rm tot})\left(1-e_1^2\right)\sin(\omega_1)\cos(\omega_2)+A\left\{\cos(\omega_1)\sin(\omega_2)\right.\right.$$
$$\left.\left.-\cos(i_{\rm tot})\sin(\omega_1)\cos(\omega_2)\right\}\right]. \tag{D.101}$$

D.3.3 Summary

If we combine all these results, we get the following set of ordinary differential equations (ODEs), valid up to and including the octupole order,

$$\dot{\omega}_1 = 6C_{\rm quad}\left\{\frac{1}{\tilde{G}_1}\left[4\cos^2(i_{\rm tot})+(5\cos(2\omega_1)-1)\left(1-e_1^2-\cos^2(i_{\rm tot})\right)\right]\right.$$
$$\left.+\frac{\cos(i_{\rm tot})}{\tilde{G}_2}\left[2+e_1^2(3-5\cos(2\omega_1))\right]\right\}$$
$$-C_{\rm oct}e_2\left\{e_1\left(\frac{1}{\tilde{G}_2}+\frac{\cos(i_{\rm tot})}{\tilde{G}_1}\right)\left[\sin(\omega_1)\sin(\omega_2)\left[A+10\left(3\cos^2(i_{\rm tot})-1\right)\left(1-e_1^2\right)\right]\right.\right.$$
$$-5B\cos(i_{\rm tot})\cos(\phi)\left]-\frac{1-e_1^2}{e_1\tilde{G}_1}\right[\cos(\phi)\left(3A-10\cos^2(i_{\rm tot})+2\right)+10\cos(i_{\rm tot})\sin^2(i_{\rm tot})$$
$$\left.\left.\times\left(1-3e_1^2\right)\sin(\omega_1)\sin(\omega_2)\right]\right\}; \tag{D.102}$$

$$\dot{\omega}_2 = 3C_{\rm quad}\left\{\frac{2\cos(i_{\rm tot})}{\tilde{G}_1}\left[2+e_1^2(3-5\cos(2\omega_1))\right]+\frac{1}{\tilde{G}_2}\left[4+6e_1^2+\left(5\cos^2(i_{\rm tot})-3\right)\right.\right.$$
$$\left.\times\left(2+e_1^2(3-5\cos(2\omega_1))\right)\right]\Big\}$$
$$+C_{\rm oct}e_1\left\{\sin(\omega_1)\sin(\omega_2)\left[\frac{4e_2^2+1}{e_2\tilde{G}_2}10\cos(i_{\rm tot})\sin^2(i_{\rm tot})\left(1-e_1^2\right)-e_2\left(\frac{1}{\tilde{G}_1}+\frac{\cos(i_{\rm tot})}{\tilde{G}_2}\right)\right.\right.$$
$$\left.\times\left(A+10\left(3\cos^2(i_{\rm tot})-1\right)\left(1-e_1^2\right)\right)\right]$$
$$\left.+\cos(\phi)\left[5B\cos(i_{\rm tot})e_2\left(\frac{1}{\tilde{G}_1}+\frac{\cos(i_{\rm tot})}{\tilde{G}_2}\right)+\frac{4e_2^2+1}{e_2\tilde{G}_2}A\right]\right\}; \tag{D.103}$$

$$\dot{e}_1 = C_{\rm quad}\frac{1-e_1^2}{\tilde{G}_1}30e_1\sin^2(i_{\rm tot})\sin(2\omega_1)+C_{\rm oct}e_2\frac{1-e_1^2}{\tilde{G}_1}\left\{35\cos(\phi)\sin^2(i_{\rm tot})e_1^2\sin(2\omega_1)\right.$$
$$-10\cos(i_{\rm tot})\sin^2(i_{\rm tot})\cos(\omega_1)\sin(\omega_2)\left(1-e_1^2\right)$$
$$\left.-A\left(\sin(\omega_1)\cos(\omega_2)-\cos(i_{\rm tot})\cos(\omega_1)\sin(\omega_2)\right)\right\}; \tag{D.104}$$

$$\dot{e}_2 = -C_{\rm oct}e_1\frac{1-e_2^2}{\tilde{G}_2}\left\{10\cos(i_{\rm tot})\sin^2(i_{\rm tot})\left(1-e_1^2\right)\sin(\omega_1)\cos(\omega_2)+A\left(\cos(\omega_1)\sin(\omega_2)\right.\right.$$
$$\left.\left.-\cos(i_{\rm tot})\sin(\omega_1)\cos(\omega_2)\right)\right\}. \tag{D.105}$$

In these equations, $\tilde{G}_1$ and $\tilde{G}_2$ are determined by e_1 and e_2, as well as the constant $\tilde{L}_1$ and $\tilde{L}_2$ (see equations D.60). The mutual inclination is determined from the constant $\tilde{G}_{\rm tot}$ and the instantaneous values of $\tilde{G}_1$ and $\tilde{G}_2$ through conservation of total angular momentum, that is, equation (D.27).

Bob: These equations look damn complicated. I guess there are no analytic solutions.

Professor: There are, indeed, no general analytic solutions, so it is necessary to employ numerical integration techniques in order to solve the equations of motion. However, for a limiting case in the quadrupole-order approximation, analytic solutions exist, and I will discuss these next.

D.4 Analytic Solutions in the Test Particle Quadrupole-Order Limit

Professor: It turns out that we can obtain analytic solutions if we restrict ourselves to the quadrupole-order approximation, together with an additional assumption, $\tilde{L}_2 \gg \tilde{L}_1$, that is, the outer orbital angular momentum completely dominates the inner orbital angular momentum. If the masses in the system are comparable, this implies that $a_2 \gg a_1$. Alternatively, if one of the particles in the inner binary is a test particle, that is, has zero mass, then $\tilde{L}_1 = 0$, and so evidently $\tilde{L}_2 \gg \tilde{L}_1$. For this reason, the approximation made in this section is also known as the quadrupole-order test particle limit.

D.4.1 Conservation Laws

More quantitatively, consider the three-body system at the initial state, denoted with a subscript 0 and at some later time. Due to conservation of total angular momentum (equation D.27), we have

$$\tilde{G}_{1,0}^2+\tilde{G}_{2,0}^2+2\tilde{G}_{1,0}\tilde{G}_{2,0}\theta_0 = \tilde{G}_1^2+\tilde{G}_2^2+2\tilde{G}_1\tilde{G}_2\theta, \tag{D.106}$$

where, for notational convenience, I defined $\theta \equiv \cos(i_{\rm tot})$. As we saw before, e_2 is constant in the quadrupole approximation; thus equation (D.106), using equation (D.60), becomes

$$\begin{aligned}
&\tilde{L}_1^2\left(1-e_{1,0}^2\right)+\tilde{L}_2^2\left(1-e_{2,0}^2\right)+2\tilde{L}_1\tilde{L}_2\sqrt{1-e_{1,0}^2}\sqrt{1-e_{2,0}^2}\theta_0=\tilde{L}_1^2\left(1-e_1^2\right)+\tilde{L}_2^2\left(1-e_{2,0}^2\right)\\
&\quad+2\tilde{L}_1\tilde{L}_2\sqrt{1-e_1^2}\sqrt{1-e_{2,0}^2}\theta\\
&\Longleftrightarrow \frac{\tilde{L}_1}{\tilde{L}_2}\left(1-e_{1,0}^2\right)+2\sqrt{1-e_{1,0}^2}\sqrt{1-e_{2,0}^2}\theta_0=\frac{\tilde{L}_1}{\tilde{L}_2}\left(1-e_1^2\right)+2\sqrt{1-e_1^2}\sqrt{1-e_{2,0}^2}\theta\\
&\Longleftrightarrow \theta=\frac{1}{2\sqrt{1-e_1^2}\sqrt{1-e_{2,0}^2}}\left[2\sqrt{1-e_{1,0}^2}\sqrt{1-e_{2,0}^2}\theta_0+\frac{\tilde{L}_1}{\tilde{L}_2}\left(e_1^2-e_{1,0}^2\right)\right] \qquad \text{(D.107)}\\
&\xrightarrow{\tilde{L}_2\gg\tilde{L}_1}\frac{\sqrt{1-e_{1,0}^2}}{\sqrt{1-e_1^2}}\theta_0\equiv\sqrt{\frac{x_0}{x}}\theta_0, \qquad \text{(D.108)}
\end{aligned}$$

where, for notational convenience, I defined $x \equiv 1-e_1^2$ and $x_0 \equiv 1-e_{1,0}^2$. Equation (D.108) implies that $\sqrt{1-e_1^2}\cos(i_{\rm tot})$ is constant in the limit $\tilde{L}_2 \gg \tilde{L}_1$. This is a well-known property of the LK resonance: during eccentricity maxima, $\cos(i_{\rm tot})$ is at its maximum value (so $i_{\rm tot}$ is at its minimum value) and vice versa.

Another conserved quantity besides total angular momentum is total energy, that is, the Hamiltonian. Since the a_j (and m_i) are constant, equation (D.19) implies (after averaging) that, to quadrupole order, equation (D.48) is conserved. Therefore, equating the initial energy to the energy at another point in time,

$$\begin{aligned}
C_{\rm quad}r_0 &\equiv C_{\rm quad}\left[\left(2+3e_{1,0}^2\right)\left(3\theta_0^2-1\right)+15e_{1,0}^2\left(1-\theta_0^2\right)\cos(2\omega_{1,0})\right]\\
&= C_{\rm quad}\left[\left(2+3e_1^2\right)\left(3\theta^2-1\right)+15e_1^2\left(1-\theta^2\right)\cos(2\omega_1)\right], \qquad \text{(D.109)}
\end{aligned}$$

where r_0 is defined uniquely by the initial conditions, that is, $e_{1,0}$, θ_0, and $\omega_{1,0}$. In equation (D.109), θ can be eliminated in favor of e_1 using equation (D.108), and the resulting equation can be solved for $\cos(2\omega_1)$ in terms of e_1 and the initial parameters. With this expression for $\cos(2g_1)$ and equation (D.108), the quadrupole-order term of $\dot{e}_1$, given by equation (D.104), can be expressed solely in terms of e_1, or, equivalently, a differential equation is found for $\dot{x}$ in terms of x and known constants.

D.4.2 The Equations of Motion Written in Terms of Eccentricity Alone

Specifically, from equation (D.109), substituting $\theta^2 = (x_0/x)\theta_0^2$ with $x = 1-e_1^2$ and $x_0 = 1-e_{1,0}^2$, $\cos(2\omega_1)$ follows as

$$\cos(2\omega_1)=\frac{r_0-(5-3x)\left(3\theta_0^2\frac{x_0}{x}-1\right)}{15(1-x)\left(1-\theta_0^2\frac{x_0}{x}\right)}=\frac{r_0x-(5-3x)\left(3\theta_0^2x_0-x\right)}{15(1-x)\left(x-\theta_0^2x_0\right)}. \qquad \text{(D.110)}$$

As equation (D.84) shows, the quantity $\sin(2\omega_1) = 2\sin(\omega_1)\cos(\omega_1)$ is required for $\dot{e}_{1,\rm quad}$. To this end, we compute

$$\begin{aligned}
\cos^2(\omega_1)&=\frac{1}{2}(1+\cos(2\omega_1))\\
&=\frac{1}{2}\frac{1}{15(1-x)\left(x-\theta_0^2x_0\right)}\left[15(1-x)\left(x-\theta_0^2x_0\right)+r_0x-15\theta_0^2x_0+5x+9\theta_0^2x_0x-3x^2\right]\\
&=\frac{1}{30(1-x)\left(x-\theta_0^2x_0\right)}\left[-18x^2+x\left(20+r_0+24\theta_0^2x_0\right)-30\theta_0^2x_0\right]; \qquad \text{(D.111)}\\
\sin^2(\omega_1)&=\frac{1}{2}(1-\cos(2\omega_1))\\
&=\frac{1}{2}\frac{1}{15(1-x)\left(x-\theta_0^2x_0\right)}\left[15(1-x)\left(x-\theta_0^2x_0\right)-r_0x+15\theta_0^2x_0-5x-9\theta_0^2x_0x+3x^2\right]\\
&=\frac{x}{30(1-x)\left(x-\theta_0^2x_0\right)}\left[-12x+6\theta_0^2x_0+10-r_0\right]. \qquad \text{(D.112)}
\end{aligned}$$

To quadrupole order, equation (D.104) for $\dot{e}_1$ is thus formulated in terms of x as

$$\begin{aligned}\frac{\mathrm{d}x}{\mathrm{d}t} &= -2e_1\frac{\mathrm{d}e_1}{\mathrm{d}t} = -2e_1 C_{\rm quad}\frac{1-e_1^2}{\tilde{L}_1\sqrt{1-e_1^2}}\,30\,e_1\underbrace{\frac{x-\theta_0^2 x_0}{x}}_{=\sin^2(i_{\rm tot})}\frac{2}{30(1-x)\left(x-\theta_0^2x_0\right)}\\ &\quad\times\sqrt{x\left(-12x+6\theta_0^2x_0+10-r_0\right)}\sqrt{-18x^2+x\left(20+r_0+24\theta_0^2x_0\right)-30\theta_0^2x_0}\\ &= -4\,\frac{C_{\rm quad}}{\tilde{L}_1}(1-x)\frac{\sqrt{x}}{x}\left(x-\theta_0^2x_0\right)\frac{1}{(1-x)\left(x-\theta_0^2x_0\right)}\sqrt{x}\\ &\quad\times\sqrt{12\left[x-\left(\frac{1}{2}\theta_0^2x_0+\frac{5}{6}-\frac{1}{12}r_0\right)\right]}\sqrt{18\left[x^2-x\left(\frac{10}{9}+\frac{1}{18}r_0+\frac{4}{3}\theta_0^2x_0\right)+\frac{5}{3}\theta_0^2x_0\right]}\\ &= -\frac{24\sqrt{6}}{\tau}\sqrt{(x-x_{\rm A})(x-x_{\rm B})(x-x_{\rm C})},\end{aligned}\tag{D.113}$$

where $\tau \equiv \tilde{L}_1/C_{\rm quad}$ is a quantity with dimensions of time and which sets a characteristic time scale; with Kepler's equation to convert a_j to P_j and with equations (D.40) and (D.60), it can be written as

$$\begin{aligned}\tau \equiv \frac{\tilde{L}_1}{C_{\rm quad}} &= \frac{m_1m_2\sqrt{G_N a_1}}{\sqrt{m_1+m_2}}\,16\,\frac{(m_1+m_2)a_2}{G_N m_1m_2m_3}\left(\frac{a_2}{a_1}\right)^2\left(1-e_2^2\right)^{3/2}\\ &= \sqrt{\frac{G}{m_1+m_2}}\left(\frac{2\pi}{P_1}\right)\frac{1}{\sqrt{G(m_1+m_2)}}\,\frac{16}{G}\,\frac{m_1+m_2}{m_3}\\ &\quad\times\left(\frac{P_2}{2\pi}\right)^2 G(m_1+m_2+m_3)\left(1-e_2^2\right)^{3/2}\\ &= \frac{8}{\pi}\left(\frac{P_2}{P_1}\right)P_2\,\frac{m_1+m_2+m_3}{m_3}\left(1-e_2^2\right)^{3/2}.\end{aligned}\tag{D.114}$$

Furthermore, $x_{\rm A} \equiv \frac{1}{2}\theta_0^2x_0 + \frac{5}{6} - \frac{1}{12}r_0$ may be written explicitly in terms of x_0, θ_0, and $\omega_{1,0}$ using equation (D.109), giving

$$\begin{aligned}x_{\rm A} &= \frac{1}{2}\theta_0^2x_0+\frac{5}{6}-\frac{1}{2}\theta_0^2+\frac{1}{6}-\frac{3}{4}(1-x_0)\theta_0^2+\frac{1}{4}(1-x_0)-\frac{5}{4}(1-x_0)\left(1-\theta_0^2\right)\cos(2\omega_{1,0})\\ &= x_0+\frac{5}{4}\left[\left(1-\theta_0^2\right)-x_0\left(1-\theta_0^2\right)-(1-x_0)\left(1-\theta_0^2\right)\cos(2\omega_{1,0})\right]\\ &= x_0+\frac{5}{2}(1-x_0)\left(1-\theta_0^2\right)\sin^2(\omega_{1,0}).\end{aligned}\tag{D.115}$$

The quantities $x_{\rm B}$ and $x_{\rm C}$ are defined through $x^2 - x\left(\frac{10}{9}+\frac{1}{18}r_0+\frac{4}{3}\theta_0^2x_0\right)+\frac{5}{3}\theta_0^2x_0 \equiv (x-x_{\rm B})(x-x_{\rm C})$. Standard algebra then implies that $x_{\rm B} = b/2 - (1/2)\sqrt{b^2-4c}$ and $x_{\rm C} = b/2 + (1/2)\sqrt{b^2-4c}$, where $c \equiv \frac{5}{3}\theta_0^2x_0$, and b can be written with equation (D.109) in terms of x_0, θ_0, and $\omega_{1,0}$ as

$$\begin{aligned}b &\equiv \frac{10}{9}+\frac{1}{18}r_0+\frac{4}{3}\theta_0^2x_0 = x_0+\frac{5}{6}+\frac{5}{6}\theta_0^2+\frac{5}{6}\theta_0^2x_0-\frac{5}{6}x_0+\frac{5}{6}(1-x_0)\left(1-\theta_0^2\right)\underbrace{\cos(2\omega_{1,0})}_{=2\cos^2(\omega_{1,0})-1}\\ &= x_0+\frac{5}{3}\left[\theta_0^2+(1-x_0)\left(1-\theta_0^2\right)\cos^2(\omega_{1,0})\right].\end{aligned}\tag{D.116}$$

D.4.3 Solving the Equation for $\dot{x}$

To recap, we have arrived at a differential equation for the inner orbit eccentricity,

$$\frac{\mathrm{d}x}{\mathrm{d}t} = -\frac{24\sqrt{6}}{\tau}\sqrt{(x-x_{\rm A})(x-x_{\rm B})(x-x_{\rm C})},\tag{D.117}$$

which is a function of x only. The quantities x_A, x_B, and x_C are constants and depend on x_0, θ_0, and $\omega_{1,0}$ through the relations

$$\begin{cases} x_A &= x_0 + \frac{5}{2}(1-x_0)\left(1-\theta_0^2\right)\sin^2(\omega_{1,0}); \\ x_B &= \frac{1}{2}b - \frac{1}{2}\sqrt{b^2-4c}; \\ x_C &= \frac{1}{2}b + \frac{1}{2}\sqrt{b^2-4c}; \end{cases} \tag{D.118}$$

$$\begin{cases} b &= x_0 + \frac{5}{3}\left[\theta_0^2 + (1-x_0)\left(1-\theta_0^2\right)\cos^2(\omega_{1,0})\right]; \\ c &= \frac{5}{3}\theta_0^2 x_0. \end{cases}$$

These quantities satisfy $x_A - x_B > 0$ and $x_C - x_B > 0$. We can actually solve equation (D.117) analytically! Let $\tilde{x} = x - x_B$. As will be clear from the solution, x_B corresponds to the maximum inner orbit eccentricity; therefore, $x_B < x$, and $\tilde{x} > 0$. Equation (D.117) in terms of $\tilde{x}$ reads

$$\frac{\mathrm{d}\tilde{x}}{\mathrm{d}t} = -\frac{24\sqrt{6}}{\tau}\sqrt{\tilde{x}(x_A - x_B - \tilde{x})(x_C - x_B - \tilde{x})}. \tag{D.119}$$

Since $x_C - x_B > 0$ and $x_A - x_B > 0$, the quantity in the square root in equation (D.119) is always positive. With the substitution $y^2 = \tilde{x}$, equation (D.119) is transformed to

$$\begin{aligned} 2y\frac{\mathrm{d}y}{\mathrm{d}t} &= -\frac{24\sqrt{6}}{\tau}y\sqrt{(x_A - x_B - y^2)(x_C - x_B - y^2)} \\ \Rightarrow \left(\frac{\mathrm{d}y}{\mathrm{d}t}\right)^2 &= \left(\frac{12\sqrt{6}}{\tau}\right)^2 (x_A - x_B)\left(1 - \frac{y^2}{x_A - x_B}\right)(x_C - x_B)\left(1 - \frac{y^2}{x_C - x_B}\right). \end{aligned} \tag{D.120}$$

With a rescaled $\tilde{y} = y/\sqrt{x_A - x_B}$, this becomes

$$\left(\frac{\mathrm{d}\tilde{y}}{\mathrm{d}t}\right)^2 (x_A - x_B) = \left(\frac{12\sqrt{6}}{\tau}\right)^2 (x_A - x_B)\left(1-\tilde{y}^2\right)(x_C - x_B)\left(1 - \tilde{y}^2\frac{x_A - x_B}{x_C - x_B}\right), \tag{D.121}$$

or, in terms of a dimensionless time $\tilde{t} = 12\sqrt{6}(t/\tau)\sqrt{x_C - x_B}$,

$$\left(\frac{\mathrm{d}\tilde{y}}{\mathrm{d}\tilde{t}}\right)^2 = \left(1-\tilde{y}^2\right)\left(1-k^2\tilde{y}^2\right), \tag{D.122}$$

where

$$k^2 = \frac{x_A - x_B}{x_C - x_B}. \tag{D.123}$$

A solution to equation (D.122) can be found by noting that the Jacobian elliptic function $\mathrm{sn}(t|k)$ satisfies (Gradshteyn et al. 2007, 8.158)

$$\frac{\mathrm{d}\,\mathrm{sn}(t|k)}{\mathrm{d}t} = \mathrm{cn}(t|k)\,\mathrm{dn}(t|k) \Rightarrow \left(\frac{\mathrm{d}\,\mathrm{sn}(t|k)}{\mathrm{d}t}\right)^2 = \mathrm{cn}^2(t|k)\,\mathrm{dn}^2(t|k) = \left[1 - \mathrm{sn}^2(t|k)\right]\left[1 - k^2\mathrm{sn}^2(t|k)\right], \tag{D.124}$$

where I used the properties $\mathrm{sn}^2(t|k) + \mathrm{cn}^2(t|k) = 1$ and $k^2\,\mathrm{sn}^2(t|k) + \mathrm{dn}^2(t|k) = 1$ (Gradshteyn et al. 2007, 8.154). A comparison of equations (D.122) and (D.124) reveals that

$$\begin{aligned} &\tilde{y}(\tilde{t}) = \mathrm{sn}\left(\tilde{t} - \tilde{t}_0\,\middle|\,k\right) \Longleftrightarrow \frac{1}{x_A - x_B}(x - x_B) = 1 - \mathrm{cn}^2\left(\tilde{t} - \tilde{t}_0\,\middle|\,k\right) \Longleftrightarrow \\ &x(\tilde{t}) = x_A + (x_B - x_A)\,\mathrm{cn}^2\left(\tilde{t} - \tilde{t}_0\,\middle|\,k\right), \end{aligned} \tag{D.125}$$

where $\tilde{t}_0$ is such that $x(0) = x_0$, that is,

$$\tilde{t}_0 = -\mathrm{arccn}\left(\sqrt{\frac{x_0 - x_A}{x_B - x_A}}\right). \tag{D.126}$$

Reverting back to the unscaled (physical) time, we have the solution

$$x(t) = x_A + (x_B - x_A)\,\mathrm{cn}^2\left(12\sqrt{6}\,\frac{t}{\tau}\sqrt{x_C - x_B} - \tilde{t}_0\,\middle|\,k\right). \tag{D.127}$$

D.4.4 Properties of the Analytic Solution

We can gain some insight from the analytical solution, equation (D.127). First, note that the function $\mathrm{cn}(t|k)$ is $4K$-periodic, where K is the complete elliptic integral of the first kind,

$$K = \int_0^{\pi/2} \frac{\mathrm{d}\theta}{\sqrt{1-k^2\sin^2(\theta)}}. \tag{D.128}$$

Also, $\mathrm{cn}(t|k)$ is symmetric around $t = 2K$; hence, $\mathrm{cn}^2(t|k)$ is $2K$-periodic. This implies that a full cycle of x, and hence e_1, is completed after an "LK period" $P_{\rm LK}$, where $P_{\rm LK}$ is given by

$$12\sqrt{6}\,\frac{P_{\rm LK}}{\tau}\,\sqrt{x_{\rm C}-x_{\rm B}} = 2K, \tag{D.129}$$

such that

$$P_{\rm LK} = \frac{\tau K}{6\sqrt{6}}\frac{1}{\sqrt{x_{\rm C}-x_{\rm B}}} = \frac{4K}{3\pi\sqrt{6}}\frac{1}{\sqrt{x_{\rm C}-x_{\rm B}}}\left(\frac{P_2}{P_1}\right)P_2\frac{m_1+m_2+m_3}{m_3}\left(1-e_2^2\right)^{3/2}. \tag{D.130}$$

The factor $4K/\left(3\pi\sqrt{6}\sqrt{x_{\rm C}-x_{\rm B}}\right)$ in equation (D.130) is on the order of unity. So, we have derived a more accurate expression for the LK time scale, that is, more accurate than equation (13.1). Note, however, that equation (D.130) is only valid within the quadrupole approximation. Also, we assumed that $\tilde{L}_2 \gg \tilde{L}_1$.

During the LK cycle, x varies between $x_{\rm B}$ and $x_{\rm A}$, implying that the minimum and maximum values of e_1 are given by

$$e_{1,\min} = \sqrt{1-x_{\rm A}}; \qquad e_{1,\max} = \sqrt{1-x_{\rm B}}. \tag{D.131}$$

Figure D.3 (top) shows $e_{1,\max}$ as a function of $\theta_0 \equiv \cos(i_{\rm tot,0})$, where $i_{\rm tot,0}$ is the initial mutual inclination angle, for several values of $e_{1,0}$ (0.01, 0.1, 0.2, and 0.5) and $\omega_{1,0}$ (0°, 45°, and 90°). If $\omega_{1,0} = 90°$, then $e_{1,\max} = e_{1,0}$ for $\theta_0^2 > \frac{3}{5}\left(1-e_{1,0}^2\right)$. However, this does not mean that there are no LK cycles (see below). In the other two cases shown in figure D.3 (top), $\omega_{1,0} = 0°$ and $\omega_{1,0} = 45°$, $e_{1,\max} > e_{1,0}$ for all values of $\theta_0 < 1$.

Regardless of the value of $\omega_{1,0}$, the maximum eccentricity can become very high as θ_0 approaches 0 (i.e., as $i_{\rm tot,0}$ approaches 90°). In particular, if $e_{1,0} = 0$ and if $\theta_0^2 < \frac{3}{5}$, then it follows from $e_{1,\max} = \sqrt{1-x_{\rm B}}$ and equation (D.118) that

$$e_{1,\max} = \left(1-\frac{5}{3}\theta_0^2\right)^{1/2}, \tag{D.132}$$

which is the relation we have seen before and independent of the value of $\omega_{1,0}$. If we assume that $\omega_{1,0} = 90°$ but $e_{1,0} \neq 0$, then we find the same result, equation (D.132), independent of $e_{1,0}$.

To better understand the different behavior of $e_{1,\max}$ as a function of θ_0 for different $\omega_{1,0}$, I show in figure D.3 (bottom) the $(\cos(\omega_1), e_1)$ space for $e_{1,0} = 0.5$. We can obtain these curves from conservation of energy and angular momentum, that is, using equation (D.109) with θ given by equation (D.108). Solid blue curves correspond to $\omega_{1,0} = 0°$ and red dashed curves to $\omega_{1,0} = 90°$. From top to bottom, both sets of curves correspond to $i_{\rm tot,0} = 70°$, $i_{\rm tot,0} = 46°$, $i_{\rm tot,0} = 40°$, and $i_{\rm tot,0} = 39°$ (the dashed curve with $-1 \le \cos(\omega_1) \le 1$ corresponds to $i_{\rm tot,0} = 39°$).

If $\omega_{1,0} = 0°$, then $\cos(\omega_{1,0}) = 1$, and ω_1 increases monotonically ($-1 \le \cos(\omega_1) \le 1$), independent of θ_0. In this case, ω_1 is said to circulate. Whenever $\omega_{1,0} = 0°$, $e_{1,\max} > e_{1,0}$, which is also shown in figure D.3 (top).

If $\omega_{1,0} = 90°$, then the behavior is more complicated and depends on θ_0. If $\theta_0^2 > 3/5$ (see the curve corresponding to $i_{\rm tot,0} = 39°$ in the bottom panel of figure D.3), then ω_1 again circulates. However, since the cycle starts with $\cos(\omega_1) = 0$, this implies that $e_{1,\max} = e_{1,0}$; that is, the maximum eccentricity is the initial eccentricity, and all other eccentricities during the LK cycles are lower than $e_{1,0}$. As θ_0 decreases and passes the point where $\theta_0^2 = 3/5$ (see the dashed curve corresponding to $i_{\rm tot,0} = 40°$ in the bottom panel of figure D.3), ω_1 no longer circulates but oscillates between two fixed values (approximately $-0.4 < \cos(g_1) < 0.4$ in the bottom panel of figure D.3). In the latter case, ω_1 is said to librate. As shown in the figure, still $e_{1,\max} = e_{1,0}$ for this value of θ_0. As θ_0 decreases further, the size of the "libration island" decreases (see the curve corresponding to $i_{\rm tot,0} = 46°$), until at some critical value of θ_0, this island is reduced to a single point in the $(\cos(\omega_1), e_1)$ space at $(\cos(\omega_1), e_1) = (0, e_{1,0})$.

This critical value of θ_0 at the fixed point, for which there are essentially no oscillations in e_1 and ω_1, can be obtained by setting $\dot{\omega}_{1,\rm quad} = 0$ with $(\cos(\omega_1), e_1) = (0, e_{1,0})$, where $\dot{\omega}_{1,\rm quad}$ is given by the quadrupole-order term in equation (D.102), neglecting the term $\propto \tilde{G}_2^{-1}$ as appropriate for the limit in which our solutions are valid. We

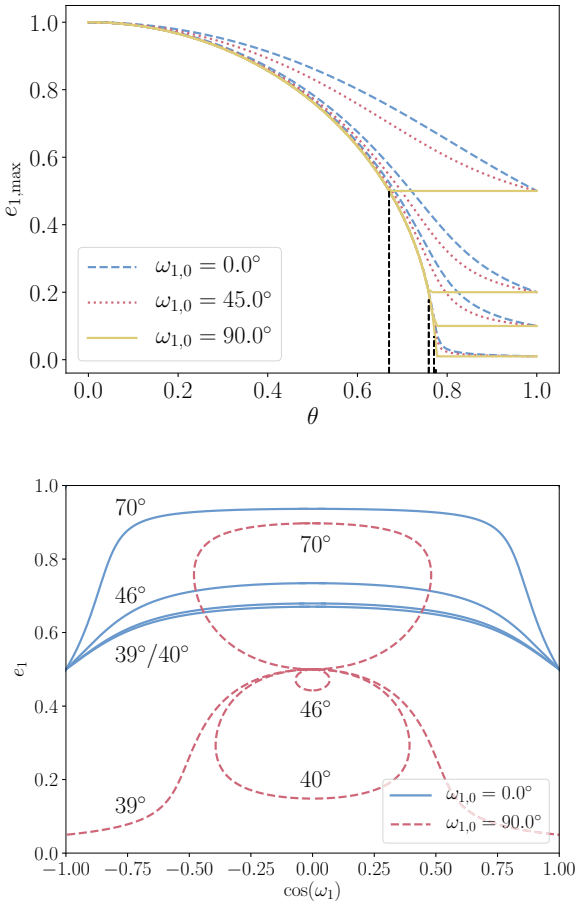


Figure D.3

Top: the maximum inner orbit eccentricity, $e_{1,\max}$, as a function of $\theta_0 \equiv \cos(i_{\rm tot,0})$, where $i_{\rm tot,0}$ is the initial mutual inclination angle, for several values of $e_{1,0}$ (0.01, 0.1, 0.2, and 0.5) and $\omega_{1,0}$ (0°, 45°, and 90°). The black dashed lines show $\theta_0 = \sqrt{(3/5)(1-e_{1,0}^2)}$, the maximum value of θ_0 for eccentricity excitation in the limit that $\omega_{1,0} = 90°$. Bottom: the $(\cos(\omega_1), e_1)$ space for $e_{1,0} = 0.5$. These curves are obtained from conservation of energy and angular momentum, that is, equation (D.109) with θ given by equation (D.108). Solid blue curves correspond to $\omega_{1,0} = 0°$ and red dashed curves to $\omega_{1,0} = 90°$.

can then find that the critical value is given by

$$\theta_{0,\mathrm{crit}} = \left[\frac{3}{5}\left(1 - e_{1,0}^2\right)\right]^{1/2}. \tag{D.133}$$

For $e_{1,0} = 0.5$, this expression yields a critical mutual inclination angle of $i_{\mathrm{tot},0} \approx 47.9°$, consistent with the bottom panel of figure D.3. If θ_0 is less than the critical value, then $e_{1,\mathrm{min}} = e_{1,0}$ and $e_{1,\mathrm{max}} > e_{1,0}$; that is, the libration island has "flipped" from the region $e_1 \leq e_{1,0}$ to $e_1 \geq e_{1,0}$ (note that always $e_1 \geq e_{1,0}$ for $\omega_{1,0} = 0°$). This explains why in the case that $\omega_{1,0} = 90°$, $e_{1,\mathrm{max}} > e_{1,0}$ only if $\theta_0 < \left[\frac{3}{5}\left(1 - e_{1,0}^2\right)\right]^{1/2}$, as we saw in the top panel of figure D.3. If $\theta_0 > \left[\frac{3}{5}\left(1 - e_{1,0}^2\right)\right]^{1/2}$, then there are still LK cycles; that is, e_1 and ω_1 still change periodically. However, the maximum eccentricity does not exceed the initial eccentricity.

Alice: Well, this is more complicated than we expected, but thank you so much for going through all the details!

Professor: Sure thing!

Alice and Bob leave Prof. Starmover's office.

Bob: Well, we sure learned a lot of theory today!

Alice: You can say that again! But it will be very useful for our coding endeavors.

Bob: I am sure of it!

Alice: Look! It has finally stopped raining.

Bob: Let's call it a day. We can resume tomorrow.

Alice: Yes, let's do that!

Bibliography

Abramowitz, M., and I. A. Stegun. 1964. *Handbook of mathematical functions: With formulas, graphs, and mathematical tables*. Vol. 55. Dover.

Barnes, J., and P. Hut. 1986. A hierarchical O(N log N) force-calculation algorithm. *Nature* 324: 446–449.

Battin, R. H. 1999. *An introduction to the mathematics and methods of astrodynamics*. AIAA.

Brouwer, D., and G. M. Clemence. 1961. *Methods of celestial mechanics*. Academic Press.

Burkardt, T. M., and J. M. A. Danby. 1983. The solution of Kepler's equation, II. *Celestial Mechanics and Dynamical Astronomy* 31 (3): 317–328.

Burrau, C. 1913. Numerische Berechnung eines Spezialfalles des Dreikörperproblems. *Astronomische Nachrichten* 195 (6): 113–118.

Butcher, J. C. 1964. Implicit Runge–Kutta processes. *Mathematics of Computation* 18 (85): 50–64.

Chambers, J. E. 1999. A hybrid symplectic integrator that permits close encounters between massive bodies. *Monthly Notices of the Royal Astronomical Society* 304 (4): 793–799.

Chenciner, A., and R. Montgomery. 2000. A remarkable periodic solution of the three-body problem in the case of equal masses. *Annals of Mathematics–Second Series* 152 (3): 881–902.

Colwell, Peter. 1993. *Solving Kepler's equation over three centuries*. Willmann-Bell.

Conway, B. A. 1986. An improved algorithm due to Laguerre for the solution of Kepler's equation. *Celestial Mechanics* 39 (2): 199–211.

Danby, J. M. A. 1992. *Fundamentals of celestial mechanics*. Willmann-Bell.

Danby, J. M. A., and T. M. Burkardt. 1983. The solution of Kepler's equation, I. *Celestial Mechanics* 31 (2): 95–107.

Delaunay, C. 1860. *Théorie du mouvement de la Lune*. Vol. 1. Didot.

Doyle, L. R., J. A. Carter, D. C. Fabrycky, R. W. Slawson, S. B. Howell, J. N. Winn, J. A. Orosz, A. Psa, W. F. Welsh, S. N. Quinn, D. Latham, G. Torres, L. A. Buchhave, G. W. Marcy, J. J. Fortney, A. Shporer, E. B. Ford, J. J. Lissauer, D. Ragozzine, M. Rucker, N. Batalha, J. M. Jenkins, W. J. Borucki, D. Koch, C. K. Middour, J. R. Hall, S. McCauliff, M. N. Fanelli, E. V. Quintana, M. J. Holman, D. A. Caldwell, M. Still, R. P. Stefanik, W. R. Brown, G. A. Esquerdo, S. Tang, G. Furesz, J. C. Geary, P. Berlind, M. L. Calkins, D. R. Short, J. H. Steffen, D. Sasselov, E. W. Dunham, W. D. Cochran, A. Boss, M. R. Haas, D. Buzasi, and D. Fischer. 2011. Kepler-16: A transiting circumbinary planet. *Science* 333: 1602. doi:10.1126/science.1210923.

Everhart, E. 1974. Implicit single-sequence methods for integrating orbits. *Celestial Mechanics and Dynamical Astronomy* 10 (1): 35–55.

Everhart, E. 1985. An efficient integrator that uses Gauss–Radau spacings. In *International astronomical union colloquium*, eds. A. Carusi and G. B. Valsecchi. Vol. 83, 185–202. Cambridge University Press.

Everhart, E., and E. T. Pitkin. 1983. Universal variables in the two-body problem. *American Journal of Physics* 51 (8): 712–717. doi:10.1119/1.13152.

Fukushima, T. 1999. Fast procedure solving universal Kepler's equation. *Celestial Mechanics and Dynamical Astronomy* 75 (3): 201–226.

Goldstein, H., C. P. Poole, and J. L. Safko. 2001. *Classical mechanics*. 3rd ed. Addison-Wesley.

Gradshteyn, I. S., I. M. Ryzhik, A. Jeffrey, and D. Zwillinger. 2007. *Table of integrals, series, and products*. Elsevier.

Grishin, E., H. B. Perets, Y. Zenati, and E. Michaely. 2017. Generalized Hill-stability criteria for hierarchical three-body systems at arbitrary inclinations. *Monthly Notices of the Royal Astronomical Society* 466: 276–285. doi:10.1093/mnras/stw3096.

Gurfil, P., and P. K. Seidelmann. 2016. *Celestial mechanics and astrodynamics: Theory and practice*. Vol. 436. Springer.

Hairer, E., S. P. Nørsett, and G. Wanner. 1991. *Solving ordinary differential equations I*. Springer.

Hamers, A. S., M. X. Cai, J. Roa, and N. Leigh. 2018. Stability of exomoons around the Kepler transiting circumbinary planets. *Monthly Notices of the Royal Astronomical Society* 480 (3): 3800–3811.

Hamers, A. S., and S. F. Portegies Zwart. 2016. Secular dynamics of hierarchical multiple systems composed of nested binaries, with an arbitrary number of bodies and arbitrary hierarchical structure: First applications to multiplanet and multistar systems. *Monthly Notices of the Royal Astronomical Society* 459 (3): 2827–2874.

Herrick, S. 1965. Universal variables. *Astronomical Journal* 70: 309.

Holman, M. J., and P. A. Wiegert. 1999. Long-term stability of planets in binary systems. *Astronomical* 117: 621–628.

Laskar, J., and M. Gastineau. 2009. Existence of collisional trajectories of Mercury, Mars and Venus with the Earth. *Nature* 459 (7248): 817–819.

Levison, H. F., and M. J. Duncan. 1994. The long-term dynamical behavior of short-period comets. *Icarus* 108 (1): 18–36.

Levison, H. F., and M. J. Duncan. 2013. SWIFT: A solar system integration software package. *Astrophysics Source Code Library*.

Lithwick, Y., and Y. Wu. 2011. Theory of secular chaos and Mercury's orbit. *Astronomical Journal* 739 (1): 31.

Makino, J., and S. J. Aarseth. 1992. On a Hermite integrator with Ahmad–Cohen scheme for gravitational many-body problems. *Publications of the Astronomical Society of Japan* 44: 141–151.

Mikkola, S. 1987. A cubic approximation for Kepler's equation. *Celestial Mechanics* 40 (3–4): 329–334.

Mikkola, S., and K. Innanen. 2002. Individual accuracy checks for massive bodies and particles in symplectic integration. *Astronomical Journal* 124 (6): 3445.

Moore, C. 1993. Braids in classical dynamics. *Physical Review Letters* 70 (24): 3675.

Morbidelli, A. 2002. *Modern celestial mechanics: Aspects of solar system dynamics*. Taylor & Francis.

Mortari, D., and A. Elipe. 2014. Solving Kepler's equation using implicit functions. *Celestial Mechanics and Dynamical Astronomy* 118 (1): 1–11.

Murray, C. D., and S. F. Dermott. 1999. *Solar system dynamics*. Cambridge University Press.

Orosz, J. A., W. F. Welsh, N. Haghighipour, B. Quarles, D. R. Short, S. M. Mills, S. Satyal, G. Torres, E. Agol, D. C. Fabrycky, D. Jontof-Hutter, G. Windmiller, T. W. A. Müller, T. C. Hinse, W. D. Cochran, M. Endl, E. B. Ford, T. Mazeh, and J. J. Lissauer. 2019. Discovery of a third transiting planet in the Kepler-47 circumbinary system. *Astronomical Journal* 157 (5): 174. doi:10.3847/1538-3881/ab0ca0.

Plummer, H. C. K. 1918. *An introductory treatise on dynamical astronomy*. Cambridge University Press.

Raposo–Pulido, V., and J. Peláez. 2017. An efficient code to solve the Kepler equation. Elliptic case. *Monthly Notices of the Royal Astronomical Society* 467 (2): 1702–1713. doi:10.1093/mnras/stx138.

Rein, H., and S. F. Liu. 2012. REBOUND: An open-source multi-purpose N-body code for collisional dynamics. *Astronomy & Astrophysics* 537: 128.

Rein, H., and D. S. Spiegel. 2014. 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): 1424–1437.

Rein, H., and D. Tamayo. 2015. WHFAST: A fast and unbiased implementation of a symplectic Wisdom–Holman integrator for long-term gravitational simulations. *Monthly Notices of the Royal Astronomical Society* 452 (1): 376–388.

Simó, C. 2001. New families of solutions in N-body problems. In *European Congress of Mathematics*, 101–115. Springer.

Strang, G. 2016. *Introduction to linear algebra*. Wellesley-Cambridge Press.

Szebehely, V., and C. F. Peters. 1967. Complete solution of a general problem of three bodies. *Astron. J.* 72: 876. doi:10.1086/110355.

Vallado, D. A. 1997. *Fundamentals of astrodynamics and applications*. Vol. 12. McGraw-Hill.

Valtonen, M., and H. Karttunen. 2006. *The three-body problem*. Cambridge University Press.

Wisdom, J., and M. Holman. 1991. Symplectic maps for the N-body problem. *Astronomical Journal* 102: 1528–1538.

Index