The Art of Computational Science

The Maya Open Lab

School Series

Volume 1

Moving Stars Around

Piet Hut and Jun Makino

September 14, 2007

Contents

Preface

We present an introduction to setting up, running and analyzing simulations of stellar system. This description is self-contained: a high-school student should be able to start at page 1, and work her way through the series. We believe that the current book is unique, in providing all the details needed, when starting from scratch.

In many areas of science, computer simulations of complex physical systems cannot be performed with off-the-shelf software packages. Instead, computational scientists have to design and build their own software environment, just as experimental scientists have to design and build their own laboratories, before being able to use them. For a long time, however, the know-how needed to construct computational laboratories has remained only a form of tacit knowledge.

Unlike explicit knowledge that can be found in manuals, this type of implicit knowledge has been alive in conversations among experts, and has been passed down in that way only as a form of oral tradition. This kind of knowledge has not been written down anywhere in sufficient detail to allow it to be passed down without direct personal instructions, or indirect osmosis through personal participation in a joint project.

The problem with the hundreds of introductory text books to science is that they mostly provide summaries, highly distilled collections of knowledge that can only be internalized through a process of hands-on experience that is generally left out. We think there is room for a different approach, one that has not been attempted earlier, as far as we know. We will try to follow a few individual students, getting occasional guidance from a teacher, in the actual process of learning through trial and error. This choice dictates the format as that of a dialogue, in which we can overhear what goes wrong, and how the students sooner or later find out how to correct their errors and misunderstandings.

This book aims at three groups of readers. For scientists, it gives a concrete example for the first steps in setting up a scientific simulation software environment. Whether you are a biologist, physicist, psychologist, or working in another area of science, many of the issues discussed here will come up for you too, when you want to build a new software system, or what is often more challenging, when you want to fully overhaul and modernize an archaic existing system. Because our scientific example has such a simple base, nothing more than Newton's laws of gravity, it is easy to grasp the underlying physics, after which you can focus on the complexity of developing and managing a software laboratory.

The second target group of readers are computer scientists, and in general everyone building complex software systems. While we apply modern concepts such as the use of object-oriented languages and design patterns, and notions such as extreme programming, our main *forte* is that we fill a gap in the market, by providing a complete discussion of the process of constructing a large-scale software system.

Readers in our third group neither work in natural science nor in computer science. They are simply curious how a modern software system is set up. For example, they may have read about the billions of dollars that are lost because of late delivery of software, or worse, delivery of faulty software. Newly built airports have experienced very costly delays because software for baggage transport was delivered way too late. Perfectly functioning rockets have been blown up because of glitches in complex software systems (see the stories about the loss of the *Mars Climate Orbiter*[1] and an *Ariane 5 rocket*[2]). Perhaps you are an average user of the internet, and just curious about what makes writing large software environments so hard. Perhaps you are working in business or finance, and you are wondering whether to invest in a software company. How are you going to judge the soundness of the company's approach? Having a good look in the kitchen will help, to see how software is actually designed and written. But actually serving as an apprentice in the kitchen would be even better. That is exactly what this book offers.

0.1 Acknowledgments

We thank Hans-Peter Bischof, Stan Blank and his high school students, Steve Giess and Peter Teuben for their comments on the manuscript.

Piet Hut and Jun Makino

[1]`http://mars.jpl.nasa.gov/msp98/orbiter`
[2]`http://www.ima.umn.edu/~arnold/disasters/ariane.html`

Chapter 1

The Universe in a Computer

1.1 Gravity

Gravity is the weakest of all fundamental forces in physics, far weaker than electromagnetism or the so-called weak and strong interactions between subatomic particles. However, the other three forces lose out in the competition with gravity over long distances. The weak and strong interactions both have an intrinsically short range. Electromagnetism, while being long-range like gravity, suffers from a cancellation of attraction and repulsion in bulk matter, since there tend to be as almost exactly as many positive as negative charges in any sizable piece of matter. In contrast, gravitational interactions between particles are always attractive. Therefore, the more massive a piece of matter is, the more gravitational force it exerts on its surroundings.

This dominance of gravity at long distances simplifies the job of modeling a chunk of the Universe. To a first approximation, it is often a good idea to neglect the other forces, and to model the objects as if they were interacting only through gravity. In many cases, we can also neglect the intrinsic dimensions of the objects, treating each object as a point in space with a given mass. All this greatly simplifies the mathematical treatment of a system, by leaving out most of the physics and chemistry that would be needed in a more accurate treatment.

The objects we will be studying are stars, and the environment we will focus on are dense stellar systems, star clusters where the stars are so close together that they will occasionally collide and in general have frequent interesting and complex interactions. Some of the stars can take on rather extremely dense forms, like white dwarfs and neutron stars, and some stars may even collapse to form black holes. However, in first approximation we can treat all these different types of objects as point particles, as far as their gravitational interactions are concerned.

We lay the groundwork for modeling a system of stars. We start absolutely from scratch, with a most simple code of less than a page long. In many small steps we then improve that code, pointing out the many pitfalls along the way, on the level of programming as well as astrophysical understanding. We introduce helpful code development facilities and visualization tools and give many hints as to how to balance simplicity, efficiency, clarity, and modularity of the code. Our intention is to introduce the topic from square one, and then to work our way up to a robust set of codes with which one can do actual research. In later volumes in this series, we will continue to develop these codes, adding many useful diagnostic tools, and integrating those in a full production-level software environment.

1.2 Galactic Suburbia

Within the visible part of the universe, there are some hundred billion galaxies. Our galaxy is a rather typical spiral galaxy, one of those many billions, and within our galaxy, our sun is a star like any other among the hundred billion or so stars in our galaxy.

The sun is unremarkable in its properties. Its mass is in the mid range of what is normal for stars: there are others more than ten times more massive, and there are also stars more ten times less massive, but the vast majority of stars have a mass within a factor ten of that of the sun. Our home star is also unremarkable in its location, at a distance of some thirty thousand light years from the center of the galaxy. Again, the number of stars closer to the center and further away from the center are comparable. Our closest neighbor, Proxima Centauri, lies at a distance of a bit more than four light years.

This distance is typical for separations between stars in our neck of the woods. A light year is ten million times larger than the diameter of the sun (a million km, or three light seconds). In a scale model, if we would represent each star as a cherry, an inch across, the separation between the stars would be many hundreds of miles. It is clear from these numbers that collisions between stars in the solar neighborhood must be very rare. Although the stars follow random orbits without any traffic control, they present such tiny targets that we have to wait very long indeed in order to witness two of them crashing into each other. A quick estimate tells us that the sun has a chance of hitting another star of less than 10^{-16} per year. In other words, we would have to wait at least 10^{16} years to have an appreciable chance to witness such a collision. Given that the sun's age is less than five Gigayears, 5.10^9 years, it is no surprise that it does not show any signs of a past collision: the chance that that would have happened was less than one in a million. Life in our galactic suburbs is really quite safe for a star.

There are other places in our galaxy that are far more crowded, and consequently are a lot more dangerous to venture into. We will have a brief look at four types of crowded neighborhoods: globular clusters, galactic nuclei, star forming regions, and open clusters.

1.3 Globular Clusters

In Fig. 1.1 we see a picture of the globular cluster M15, taken with the Hubble Space Telescope. This cluster contains roughly a million stars. In the central region typical distances between neighboring stars are only a few hundredths of a light year, more than a hundred times smaller than those in the solar neighborhood. This implies a stellar density that is more than a million times larger than that near the sun. Since the typical relative velocities of stars in M15 are comparable to that of the sun and its neighbors, a few tens of km/sec, collision times scale with the density, leading to a central time between collisions of around 10^{10} years. With globular clusters having an age of roughly 10^{10} years, a typical star near the center has a significant chance to have undergone a collision in the past. To be a bit more precise, we don't know how long a typical star in the core has remained in its current environment, but even if such a star has been there only for a billion years, the chance of a collision has already been $\sim 10\%$.

In fact, the chances are even higher than this rough estimate indicates. One reason is the stars spend some part of their life time in a much more extended state. A star like the sun increases its diameter by more than a factor of one hundred toward the end of its life, when they become a red giant. By presenting a much larger target to other stars, they increase their chance for a collision during this stage (even though this increase is partly offset by the fact that the red giant stage lasts shorter than the so-called main-sequence life time of a star, during which they have a normal appearance and diameter). The other reason is that many stars are part of a double star system, a type of dynamic spider web that can catch a third star, or another double star, into a temporarily bound three- or four-body system. Once engaged in such a tightly bound dance, the chance for collisions between the stars is greatly increased.

A detailed analysis of all these factors predicts that a significant fraction of stars in the core of a dense globular cluster such as M15 has already undergone at least one collision in its life time. This analysis, however, is quiet complex. To study all of the important channels through which collisions may occur, we have to analyze encounters between a great variety of single and double stars, and occasional bound triples and larger bound multiples of stars. Since each star in a bound subsystem can be a normal main-sequence star, a red giant, a white dwarf, a neutron star or even a black hole, as well as an exotic collision product itself, the combinatorial richness of flavors of double stars and triples

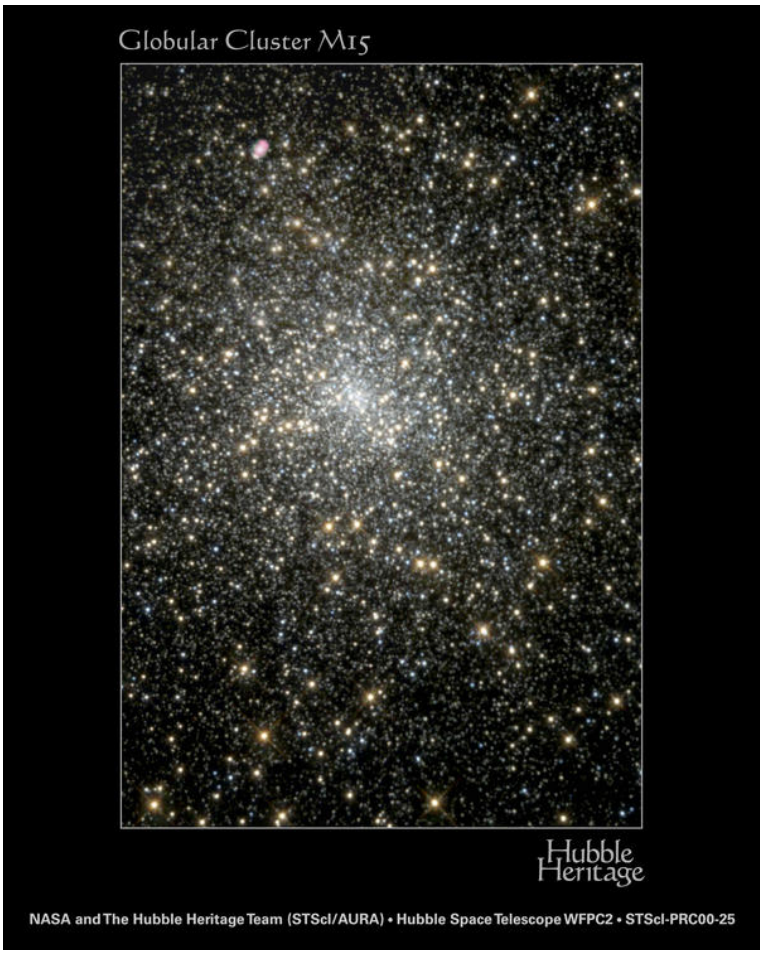
Globular Cluster M15
Hubble Heritage
NASA and The Hubble Heritage Team (STScI/AURA) • Hubble Space Telescope WFPC2 • STScI-PRC00-25

Figure 1.1: A snapshot of the globular cluster M15, taken with the *Hubble Space Telescope*[†].

[†]`http://oposite.stsci.edu/pubinfo/PR/2000/25/content/0025y.jpg`

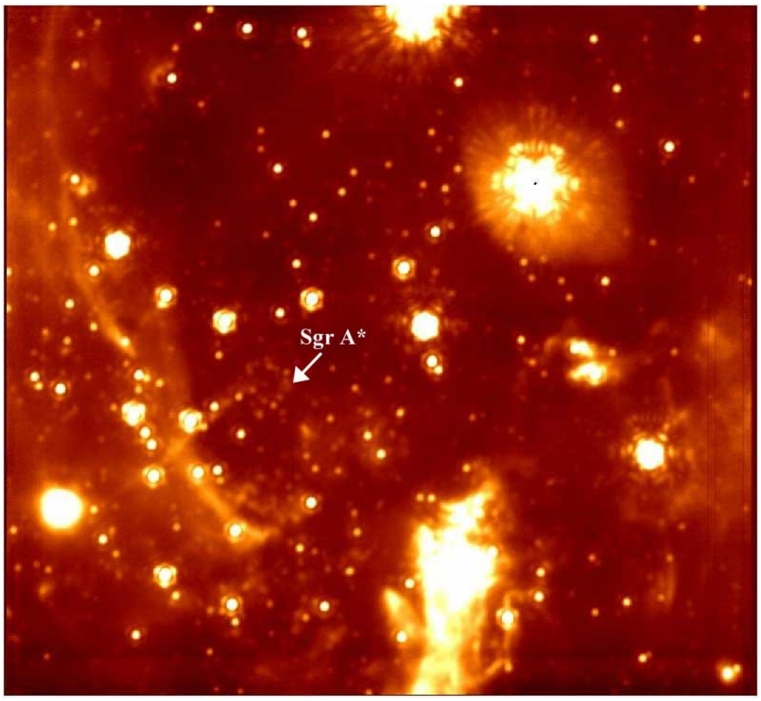

Figure 1.2: An image of the central region of our galaxy, as seen with the *Keck telescope*[†]. The massive black hole is located near the head of the arrow labeled SgrA*. The size of this image is two light years by two light years, and the crowding is enormous: in the neighborhood of the sun, typical distances between stars are several light years, and a snapshot like this most likely would show just one star or none at all.

[†]`http://www.astro.ucla.edu/~jlu/gc/pictures/lgs.shtml`

is enormous. If we want to pick a particular double star, we not only have to choose a star type for each of its members, but in addition we have to specify the mass of each star, and the parameters of its orbit, such as the semi-major axis (a measure for the typical separation of the two stars) as well as the orbital eccentricity.

1.4 Galactic Nuclei

In Fig. 1.2 we see an image of the very center of our galaxy. This picture has been obtained with the Keck telescope, in a near infrared wavelength band.

In the very center of our galaxy, a black hole resides with a mass a few million times larger than the mass of our sun. Although the black hole itself is invisible, we can infer its presence by its strong gravitational field, which in turn is reflected in the speed with which stars pass near the black hole. In normal visible light it is impossible to get a glimpse of the galactic center, because of the obscuring gas clouds that are positioned between us and the center. Infrared light, however, can penetrate deeper in dusty regions.

In the central few light years near the black hole, the total mass of stars is comparable to the mass of the hole. This region is called the galactic nucleus. Here the stellar density is at least as large as that in the center of the densest globular clusters. However, due to the strong attraction of the black hole, the stars zip around at much higher velocities. Whereas a typical star in the core of M15 has a speed of a few tens of km/sec, stars near the black hole in the center of our galaxy move with speeds exceeding a 1000 km/sec. However, gravitational focusing is less by the same factor, and as a consequence, the frequency of stellar collisions is comparable.

Modeling the detailed behavior of stars in this region remains a great challenge, partly because of the complicated environmental features. A globular cluster forms a theorist's dream of a laboratory, with its absence of gas and dust and star forming regions. All we find there are stars that can be modeled well as point particles unless they come close and collide, after which we can apply the point particle approximation once again. In contrast, there are giant molecular clouds containing enormous amounts of gas and dust right close up to the galactic center. In these clouds, new stars are formed, some of which will soon afterwards end their life in brilliant supernova explosions, while spewing much of their debris back into the interstellar medium. Such complications are not present in globular clusters, where supernovae no longer occur since the member stars are too old and small to become a supernova.

Most other galaxies also harbor a massive black hole in their nuclei. Some of those have a mass of hundreds of millions of solar masses, or in extreme cases even more than a billion times the mass of the sun. The holy grail of the study of dense stellar systems is to perform and analyze accurate simulations of the complex ecology of stars and gas in the environment of such enormous holes in space. Much of the research on globular clusters can be seen as providing the initial steps toward a detailed modeling of galactic nuclei.

1.5 Star Forming Regions

There are many other places in the galactic disk where the density of stars is high enough to make collisions likely, at least temporarily. These are the sites where stars are born. Fig. 1.3 taken by the Japanese Subaru telescope in Hawaii shows the Orion Nebula, also known as M42, at a distance of 1500 light years from the sun. This picture, too, is taking in infrared light in order to penetrate the dusty regions surrounding the young stars. These stars all recently formed from the gas and dust that still surrounds them.

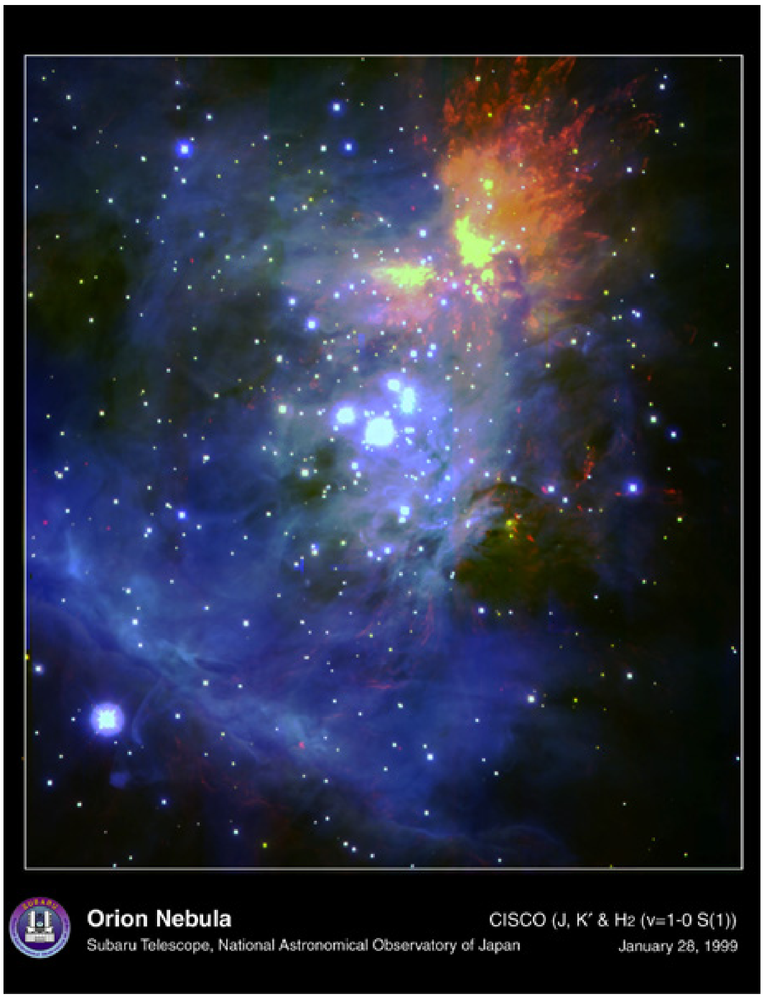
Orion Nebula
Subaru Telescope, National Astronomical Observatory of Japan
CISCO (J, K' & H2 (v=1-0 S(1))
January 28, 1999

Figure 1.3: The Orion Nebula, as seen by the *Subaru telescope*[†].

[†]`http://www.naoj.org/Science/press_release/1999/01/Orion_300.jpg`

In order to study collisions in these star forming regions, we can no longer treat the stars are point masses. Many of the collisions take place while the stars are still in the process of forming, before they settle into their normal equilibrium state. While a protostar is still in the process of contracting from the gas cloud in which it was born, it presents a larger target for collisions with other stars. In addition, a single contracting gas cloud may fission, giving rise to more than one star at the same time. In this way, the correlated appearance of protostars is even more likely to lead to subsequent collisions.

The proper way to model these processes is to combine gas dynamics and stellar dynamics. Much progress has been made recently in this area. One way to use stellar dynamics in an approximate fashion is to begin with the output of the gas dynamics codes, which present the positions and velocities of a group of newly formed stars, and then to follow and analyze the motions of those stars, including their collisions.

1.6 Open Clusters

Although stars are formed in groups, these groups typically do not stay together for very long. Perturbations from other stars and gas clouds in their vicinity are often enough to break up the fragile gravitational hold they initially have over each other. Some of the more massive groups of newly formed stars, however, are tightly bound, enough to survive their environmental harassment. They form the so-called open clusters, where their name indicates that they have central densities that are typically less than what we see in globular clusters.

Fig. 1.4 shows one of the richest and densest open clusters, M67, as observed by the Anglo-Australian Observatory. Since this cluster is old enough to have lost its gas and dust, all stars are visible at normal optical wavelengths, at which this image is taken. In the central regions of this cluster, there are indications that some of the stars have undergone close encounters or even collisions. In particular, some of the so-called blue stragglers may be merger products. Consequently, this star cluster qualifies as a dense stellar system.

Open clusters typically have fewer members than globular clusters. Also, they are younger. Both facts makes it easier to simulate open clusters than globular clusters. On the other hand, the densest globular clusters show a higher frequency and a far richer variety of stellar collisions, making them a more interesting laboratory. In that sense, a dynamical simulation of an open cluster can be seen as providing preparatory steps toward the modeling of globular clusters, just as a study of the latter forms a stepping stone toward the investigation of galactic nuclei.

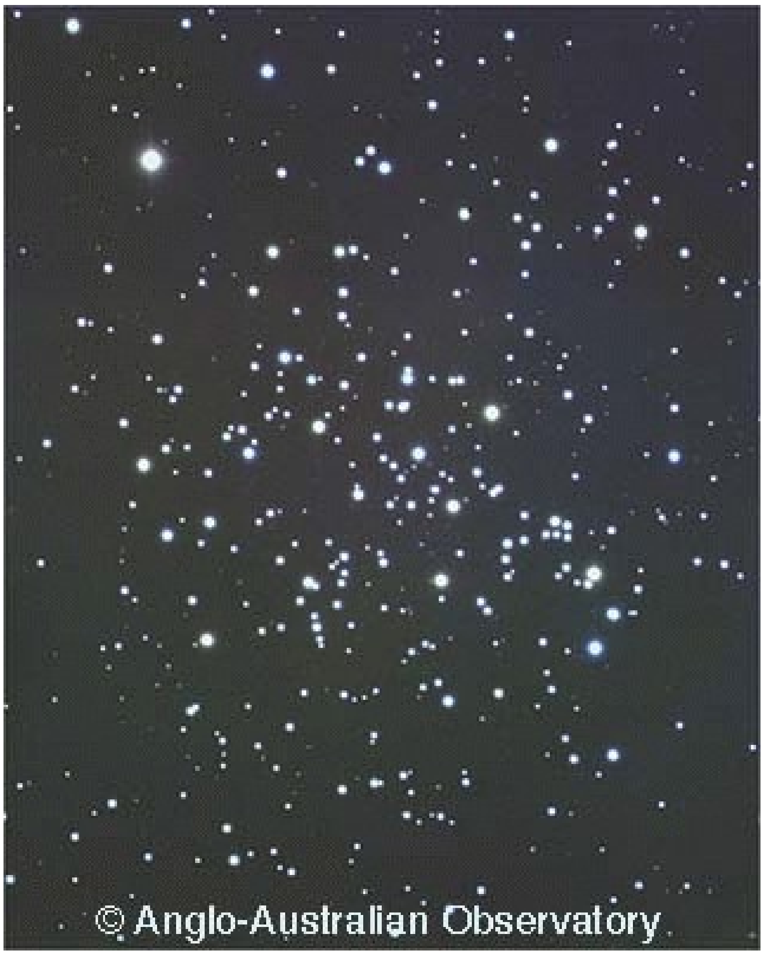

Figure 1.4: The open star cluster M67, in a picture taken at the *Anglo-Australian Observatory*[†].

[†]`http://www.seds.org/messier/Pics/More/m67aat.jpg`

1.7 Writing your own star cluster simulator

Astronomers have half a century of experience in writing computer codes to simulate dense stellar systems. The first published results date back to 1960, and it was in the subsequent decade that it became clear just how tricky it was to simulate a group of interacting stars. The task seems so easy: for each star, just solve Newton's equations of motion (an object's acceleration is given by the applied force divided by the mass of the object) under the influence of the gravitational pairwise interactions of all other stars. Indeed, it is straightforward to write a simple code to do so, which integrates a rather simple differential equation, as we will see below. And as long as all stars remain fairly well separated from each other, even a simple code will do a reasonably good job. For historical reasons, this type of code is called an N-body code.

In practice, though, even a small group of stars will spontaneously form one or more double stars. This was discovered experimentally in the early sixties. One way to understand this result, after the fact, is from an energetic point of view. When a double star, or binary as they are generally called, is formed, energy has to be released. The reason is that the two stars in a binary are bound, which means that the total energy is negative, whereas two stars meeting each other after coming in from far away have a positive net energy. When three stars come together randomly, there is a chance that two of the three are left in a bound state, while the third one escapes, carrying the excess energy. Left by itself, a stellar system will exploit this energy liberation mechanism by spontaneously forming binaries.

As soon as even one binary appears, a simple code with constant time steps will give unacceptably large errors. The first modification needed is the introduction of an adaptive time step. In the simplest case, all particles will still share the same time step size, but that size will change in time, in order to adequately resolve the closest encounters between particles. However, even a single binary can then impose a tiny time step on the whole system, slowing everybody down.

By the end of the sixties, this problem was overcome by the development of codes that employed individual time steps. Stars with close neighbors were stepped forward in time more frequently than stars at large, and in this way the computational power was brought to where it was most needed.

This modification in itself brought gravitational N-body codes already well outside the range of systems that are normally discussed in text books on numerical integration methods. The internal book keeping needed to write a correct and efficient code with individual time steps is surprisingly large, given the simplicity of the task: integrate the effect of pairwise attractive inverse square forces, in order to solve the differential equations that constitute the equations of motion of classical Newtonian gravity.

However, introducing individual time steps was only a first step toward the development of modern N-body codes. The presence of tight binaries produced much more of an obstacle, and throughout the seventies a variety of clever mechanisms were developed in order to deal with them efficiently.

For one thing, there are problems with round-off. Two stars in a tight orbit around each other have almost the same position vector, as seen from the center of a star cluster, where we normally anchor the global coordinate system. And yet it is the separation between the stars that determines their mutual forces. When we compute the separation by subtracting two almost identical spatial vectors, we are asking for (numerical) trouble. The solution is to introduce a local coordinate system whenever two or more stars undergo a close interaction. This does away with the round-off problem, but it introduces a host of administrative complexities, in order to make sure that any arbitrary configuration of stars is locally presented correctly – and that the right thing happens when two or more of such local coordinate patches encounter each other. This may not happen often, but one occurrence in a long run is enough to cause an unacceptably large error if no precautions have been taken to deal properly with such a situation.

We can continue the list of tricks that have been invented to allow every larger and denser systems to be modeled correctly. We will encounter them later on, and explain them then in detail, but just to list a few, here are some of the techniques. Numerical problems with the singularity in the two-body system have been overcome by mapping two or more interacting stars from the three-dimensional Kepler problem to a four-dimensional harmonic oscillator. The total force on particles has been split into different contributions, the first from a near zone of relatively close neighbors and the second from a far zone of all other particles, with each partial force being governed with different integration time steps. Tree codes have been used to group the contributions of a number of more and more distant zones together in ever larger chunks, for efficiency. Triple stars have received their own special treatment, especially the marginally stable triples that are sometimes long-lived, but continuously changed their inner state due to internal perturbations. The list goes on. See Sverre Aarseth's book *Gravitational N-Body Simulations*[1]

In this book, we will introduce a modern integrator, the Hermite scheme, developed in the 1990s, together with a variable time step integration scheme, where all stars share a common time step at any given time. Our emphasis will be on a complete explanation of all the steps involved, together with a discussion of the motivation for those steps. In the last few chapters, we will embark on a research project featuring stellar collisions, in a simple gravity-only approximation.

One of the roles of the current book is to provide an introduction to the Kali code, and the other software tools that are part of the *Maya open lab*[2]. The Maya project will make it possible to simulate an entire star cluster.

[1]`http://www.cambridge.org/catalogue/catalogue.asp?isbn=0521432723`
[2]`http://www.ArtCompSci.org`

Chapter 2

The Gravitational N-Body Problem

2.1 Background

Our goal is build a laboratory to study the interactions between stars. Since stars don't fit in traditional laboratories, we have no choice but to use virtual stars in virtual labs. The computer provides us with the right virtual environment, and it is our task to write the software that will correctly simulate the behavior of the virtual stars and their interactions. Once that software is in place, or at least enough of it to start playing, the user can provide a starting situation, after which our software will evolve the system, perhaps for a few billion years.

In this book we will focus in detail on the whole process of developing the software needed. We will aim at realistic detail, showing the way of thinking that underlies the construction of a complex and ever-growing software environment. We will require a bit of patience, since it will take a while to have a full package in hand for modeling, say, the long-term behavior of a star cluster, and we are presenting in this book only the first few steps. This drawback, we feel, is more than offset by the advantages of our approach:

- you will be fully empowered to customize *any* aspect of the software environment or *any* larger or smaller part of it;
- you will be able to use the package with complete understanding and appreciation of what are and are not reasonable ways to apply the tools;
- you will learn to embark on completely different large-scale software projects, be they in (astro)physics or other areas of science;

- and in addition, we hope that reading these books will be as much fun for the reader as it was for us to write them.

2.2 Our Setting

We want to convey some of the atmosphere in which large software environments are grown, in a dynamic and evolutionary way, often starting from very small beginnings – and always surprising the original authors when they see in what unexpected ways others can and will modify their products in whole new directions. Most of our narrative will follow this process step by step, but occasionally we will turn away from the process to the players: the developers writing the software. We have chosen one representative of each of the three target groups mentioned in our preface, from natural science, business and computer science.

The setting is an undergraduate lounge, where three friends occasionally hang out after dinner, and sometimes tell each other about the projects they are working on. Tonight, Erica talks with great animation to her friends Dan and Carol. Erica is an astrophysics major, Dan is currently studying biology but preparing to go to business school, and Carol majors in computer science.

Erica: Guess what! Today I was asked to choose a student project, for me to work on for half a year. Many of the choices offered seemed to be interesting, but for me the most exciting opportunity was to work on the overhaul of a laboratory for interactions between stars.

Dan: What are the interactions that are so interesting?

Erica: Imagine this, the current software package allows you to create a star cluster and to let it evolve for billions of years, and then you can fly through the whole four-dimensional history in space and time to watch all the collisions and close encounters between normal stars and black holes and white dwarfs and you name it!

Carol: If that package already exists, what then is so exciting about an overhaul?

Erica: Yes, the package exists, but every large software package tends to grow and to become overweight. As you both know, this is true in business-driven software projects, but it is even more true in science settings, where the value of clean software engineering is underrated even more than in profit-oriented areas. As a result, by far the most reasonable and efficient way to extend older packages is first to do a thorough overhaul.

Dan: I see. You mean that rewriting a package is worth the time, presumably because you have already figured out the physics and you have similarly built up extensive experience with hooking everything together in various ways in software.

Erica: Exactly. Rewriting a package takes far less time than writing it in the first place – if we want to keep the same functionality. In practice, it may take longer than we think, since we will for sure find new extensions and more powerful ways to include more physics. As long as we don't get carried away, and keep our science goals in sight, this extra time is well spent and will lead to greater productivity.

Carol: I wonder, though, whether a complete overhaul is desirable. I have just learned about a notion called *refactoring.* The idea is to continuously refine and clean up code while you go along.

Erica: Yes, that would be better. In fact, I already had a brief chat with my supervisor, a professor specializing in stellar dynamics, and he mentioned just that. He said that this was the last really major overhaul he hoped to do for his software environment. The main reason for the planned overhaul is to make it flexible enough that the system from now on can grow more organically.

Dan: The overhaul that will be the end of all overhauls!

Carol: Well, maybe. I've heard a lot of hype about programming, in the few years that I have been exposed to it. But the basic idea sounds good. And even if you will have to overhaul in the future, a cleaner and more modular code will surely be easier to understand and disentangle and hence to overhaul.

2.3 Fun and Profit

Dan: May I ask a critical question? You have half a year to get your feet wet, doing a real piece of scientific research. Would it really be prudent to spend that time overhauling someone else's code?

Erica: I asked that question, too. My supervisor told me that a thorough-going attempt to improve a large software environment in a fundamental way from the bottom up is guaranteed to lead to new science. Instead of overhauling, a better term might be brewing. You will reap the benefits of all the years of experience that have gone into building the software, from working with the science to the figuring out of the architecture of the software environment. Those who wrote the original code have become too engrossed in teaching and administration. But they will have time to share their experience, and they will gladly do so when they see someone seriously working on improvements.

Carol: In other words, during this coming half year you might find yourself engaging in actual research projects, as a form of spin-off of the overhauling, or brewing as you just called it?

Erica: Exactly.

Dan: You know what? Perhaps this is a silly thing to suggest, but I suddenly got an idea. It seems that Erica today has started what amounts to an infinite task. She will have her hands full at it, even if she could clone herself into several people working simultaneously, and she is not expected to reach anywhere near completion in half a year. At the same time, she is expected to start absolutely from start. If she wouldn't do so, it wouldn't be a complete overhaul. Here is my proposal: how about all three of us pitching in, a couple times a week, after dinner, using the time we tend to spend here anyway?

Carol: To keep Erica honest?

Dan: Exactly! Of course, she may well get way ahead of us into all kinds of arcane astrophysics applications, but even so, if we plod behind her, asking her questions about each and every decision she has made from the start, we will probably keep her more honest than any astrophysicist could – simply because we know less about astrophysics than any astrophysicist! And besides, for me too it would be a form of fun and profit. I intend to focus on the software industry when I go to business school, and I might as well get some real sense of what is brewing in the kitchen, when people write non-trivial software systems.

Carol: Hmm, you have a point. Obviously, something similar holds for me too, in that I can hone my freshly learned theoretical knowledge on realistic astrophysics problems. What do you think, Erica, are we rudely intruding upon your new project?

Erica: No, on the contrary! As long as I keep my actual project separated, as Dan stressed, I am more than happy to discuss the basics with you both during after-dinner sessions, as long as you have the stamina and interest to play with orbital dynamics of stars and star systems. And I'm sure we will all three learn from the experience: I would be very surprised if you didn't inject new ideas I hadn't thought about, or notice old ideas of mine that can be improved.

Dan: We have a deal! Let's get started right away, and get back here tomorrow, same time, same place.

Carol: Okay, but let's say almost the same place: next door is the computer center, where we will be able to find a big enough screen so that the three of us can gather around it, to start writing our first star-moving program.

Erica: An N-body code, that is what it is called. I'm game too. See you both tomorrow!

2.4 What is the Problem, and why N and Bodies?

The next day, our three friends have gathered again, ready to go.

Erica: Hi, you're all back, so I guess you were really serious. Well, let's write our first code for solving the gravitational N-body problem.

Dan: I understand that we are dealing with something like gravitational attractions between objects, but why is that a *problem* and not, say, a system?

Carol: And why are you talking about N bodies, and not p bodies or anything else?

Dan: And why *bodies* and not just objects?

Erica: Traditionally, in mathematics and mathematical physics, when we pose a question, we call it a problem, as in a home work problem. And stars and planets just happen to be called celestial bodies. Specifically, the gravitational 2-body problem is defined as the question: given the initial positions and velocities of two stars, together with their masses, describe their orbits.

Dan: What if the stars collide?

Erica: For simplicity, we treat the stars as if they are mass points, without any size. In this case they will not collide, unless they happen to hit each other head-on. Of course, we can set two point masses up such that they will hit each other, and we will have to take such possibilities into account, at some point. However, when the stars in a star cluster are born from a gas cloud, their motions will not be fine tuned to lead to exactly head-on collisions with mathematical precision. Therefore, the chance of a collision between point particles is negligible.

Carol: In mathematical terms: the set of initial conditions for collisions to occur has measure zero in the space of all possible initial conditions. Don't worry, that's just a formal way of saying the same thing. But I have a serious question: real stars are not points, so why can we treat them as such?

Erica: The goal of building a laboratory for star cluster evolution is to introduce real stars with finite sizes, nuclear reactions, loss of radiation and mass, and all that good stuff. But we have to start somewhere, and a convenient starting place is to treat stars as point masses, as a first approximation.

This brings me to Carol's question: why do astrophysicists talk about N-body simulations? This is simply a historical convention. I would prefer the term many-body simulations, but somehow somewhere someone stuck in the variable N as a place-holder for how many bodies where involved, and we seem to be stuck with that notation.

Chapter 3

The Gravitational 2-Body Problem

3.1 Absolute Coordinates

A decision was made to let Carol take the controls, for now. Taking the keyboard in front of a large computer screen, she opens a new file `nbody.rb` in her favorite editor. Expectantly, she looks at Erica, sitting to her left, for instructions, but Dan first raises a hand.

Dan: I'm a big believer in keeping things simple. Why not start by coding up the 2-body problem first, before indulging in more bodies? Also, I seem to remember from an introductory physics class for poets that the 2-body problem was solved, whatever that means.

Erica: Good point. Let's do that. It is after all the simplest case that is nontrivial: a 1-body problem would involve a single particle that is just sitting there, or moving in a straight line with constant velocity, since there would be no other particles to disturb its orbit.

And yes, the 2-body problem can be solved analytically. That means that you can write a mathematical formula for the solution. For higher values of N, whether 3 or 4 or more, no such closed formulas are known, and we have no choice but to do numerical calculations in order to determine the orbits. For $N = 2$, we have the luxury of being able to test the accuracy of our numerical calculations by comparing our results with the formula that Newton discovered for the 2-body problem.

Yet another reason to start with $N = 2$ is that the description can be simplified. Instead of giving the *absolute* positions and velocities for each of the two particles, with respect to a given coordinate system, it is enough to deal with the *relative* positions and velocities. In other words, we can transform the two-body

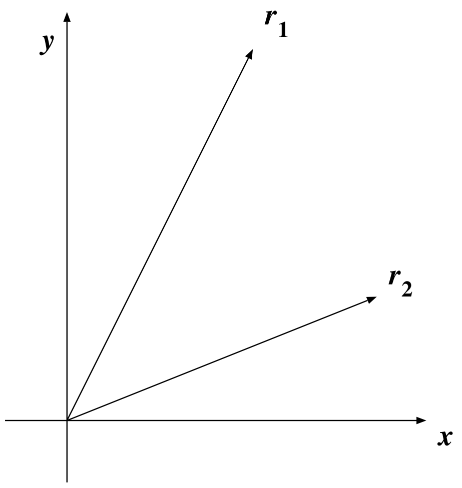

Figure 3.1: Two bodies with absolute coordinates $\mathbf{r}_1, \mathbf{r}_2$.

problem into a one-body problem, effectively.

Dan: A one-body problem? You mean that one body is fixed in space, and the other moves around it?

Carol: I guess that Erica is talking about a situation where one body is much more massive, so that it doesn't seem to move much, while the other body moves around it. When the Earth moves around the Sun, the Sun can almost be considered to be fixed. Similarly, when you look at the motion of the Moon around the Earth: the Earth has a much larger mass than the Moon, and the mass of the Sun is again much larger than that of the Earth.

Erica: Yes, a large mass ratio makes things simpler. In that case we can make the approximation that the heavy mass sits still in the center, and the lighter mass moves around it. But that is only an approximation, and not completely accurate. In reality, both masses move around each other, even though the heavier mass moves only a little bit.

Even for the case where the masses are comparable, we can still talk about the relative motion between the two particles. And we don't have to make any approximation. Here, let me draw a few pictures, to make it clear.

Since my notepad is two-dimensional, let us start by discussing the two-dimensional two-body problem. Later, we can easily go to 3D. Here, in figure 3.1 I show the positions of our two particles by drawing the position vectors $\mathbf{r}_1, \mathbf{r}_2$.

Carol: But in order to define those vectors, you first have to choose a coordinate system, right?

Erica: Yes. We have to choose an x axis and a y axis, and they together make up a coordinate system. The point of intersection of the two axes is called the origin of the coordinate system. With respect to the origin, the positions of the two bodies are pointed out by the tips of the arrows that stand for the two vectors $\mathbf{r}_1, \mathbf{r}_2$.

Dan: I see the vectors, but I don't see the bodies.

Erica: You have to imagine the bodies to reside at the very end of each arrow.

Carol: They are point masses, remember, so they are too small to see!

Erica: Well, yes, but of course I could still have drawn them as small points. However, I wanted to keep the figures simple.

3.2 Coordinate Systems

Dan: What would happen if you had chosen a different coordinate system?

Erica: In that case, the tips of the arrows would stay at the positions of the particles, since they would not change. However, the arrows themselves would change, because they would be rooted in the new origin.

Dan: But could you really have chosen *any* coordinate system? And could you then let it change or rotate or let it jump up and down? That seems rather unlikely.

Erica: Ah, no, certainly not. Or more precisely, preferably not. If you choose a coordinate system that moves in a strange way, you then have to correct for those strange motions of the coordinate axes, which would be reflected in the description of the motions of the particles. And you would then have to correct for that, in the equations of motion.

Carol: I'm not sure I follow all that, but I guess the upshot is that you want to keep things simple, and therefore you prefer to use coordinates where the origin stays at rest in space.

Erica: Yes, either at rest, or otherwise the origin should move at a constant speed in one fixed direction. A coordinate system that moves in such a way is called an inertial coordinate system. By the way, we use the term *uniform motion* for the type of motion that occurs at constant speed in the same direction.

Dan: Ah, that is a term I remember from my physics class. This is related to the fact that Newton discovered that an object which does not feel any forces will keep moving in a straight line with constant speed. Because of the inertia of an object, you have to exert a force in order to change that type of motion.

Erica: Indeed. And this means that any object that feels no force will move in a straight line at constant speed in any inertial coordinate system. But if you start with, for example, a rotating coordinate system, then even an object at rest seems to rotate in the opposite direction, when you describe its motion with respect to the rotating frame.

Figure 3.2: Two bodies at rest.

Dan: And all of this is related to Newton's concept of absolute space, right? And Einstein showed that there is no such thing as absolute space.

Erica: The idea of an absolute space was a brilliant invention, very useful to describe the phenomena on the level of classical mechanics. But when you deal with high velocities, approaching the speed of light, or even with lower velocities and very high accuracies, you have to take into account the fact that special relativity gives a more accurate description of the world.

Carol: And general relativity is even more accurate, I presume?

Erica: Yes, but in general relativity space and time are curved, in a way that is influenced by the distribution of masses and energy in the world. This means that it is extremely difficult to make a computer simulation of the motion of objects such as black holes when they approach each other.

Dan: I'm happy to keep things simple, for now at least, and to stick to Newton's classical mechanics.

Carol: So am I.

3.3 A Fourth Point

Dan: Okay, I got the picture. Now where does the one-body representation come in?

Erica: It comes in through a clever coordinate transformation. The trick is to add a fourth point to the picture given in fig. 3.1, besides the origin and the positions of the two particles. The extra point is what is called the center of mass of the system, often abbreviated to c.o.m. for simplicity.

Dan: You mean the point right in the middle of the two masses?

Erica: No, not if the particles have different masses. Here is the basic idea. Let me draw two masses at rest, in fig. 3.2. Let particle number 1, at the left, be twice as massive as particle 2, at the right: $M_1 = 2M_2$. As soon as we let the particles move freely, from rest, they will start falling toward each other.

Carol: Can you write down the equations? Sooner or later we'll have to program them, after all.

Dan: I hope it will be sooner!

Erica: In this very simple example, let us define the x axis of our coordinate

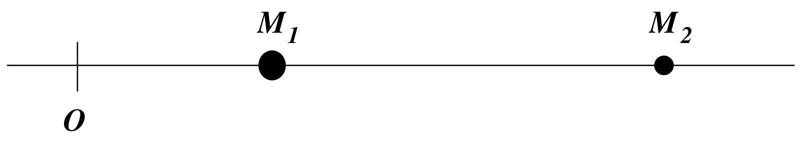

Figure 3.3: Two bodies at rest, with the origin O of a one-dimensional coordinate system shown at the left.

system to go through both particles, with the positive side at the right, as usual.

Carol: Where is the origin?

Erica: You can put the origin anywhere you want. For example, if you like to put it to the left of M_1, as in fig. 3.3, then both particles are located on the positive x axis, and therefore both particles have positive values for their positions r_1 and r_2 with respect to the origin. Moreover, $r_2 > r_1$, so the distance between the two particles is given as $r = r_2 - r_1$.

But really, the position of the origin is not important; if you shift the origin to the left or to the right, the value of r remains the same, and that's the only thing that counts.

Dan: Yes, I can see that as long as O stays to the left of both particles. But how about the other cases? Let's check. If we take O to be at the right, then the two particle positions have negative values, with M_1 further away with a more negative value, say $r_1 = -5$ and $r_2 = -3$. In that particular case, $r = r_2 - r_1 = -3 - (-5) = 5 - 3 = 2$. Okay, that gives the right answer.

The last possibility is to have O located right in between the two particles. In that case, r_1 is negative and r_2 is positive, and clearly $r = r_2 - r_1$ is positive as well. Good! I am convinced now that $r = r_2 - r_1$ is the right expression for the physical distance between the two particles, with a value that is always greater than zero, if the particles are not at exactly the same place.

Erica: In any of the tree cases that you just explored, the force that particle 1 feels from particle 2 is:

$$F_1 = G\frac{M_1 M_2}{r^2} \tag{3.1}$$

where r is the distance between the two particles and G is Newton's universal constant of gravity. Particle 1 is being pulled in the direction of the positive x axis. This means that the velocity changes from zero to a positive value, which in turn means that the acceleration is positive.

In contrast, particle 2 will be pulled to the left, and it will start to move with a negative velocity, so its acceleration is negative. Other than this important minus sign, we can simply reverse subscripts 1 and 2, which gives:

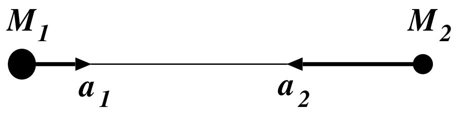

Figure 3.4: Two bodies falling toward each other.

$$F_2 = -G\frac{M_1 M_2}{r^2} \tag{3.2}$$

Dan: So the two forces are equal, but opposite in direction.

Carol: Action equals reaction, that's one of Newton's laws, right?

Erica: Right indeed, that is Newton's third law. Now let us use Newton's second law, to see how the particles will actually start moving. His second law is the famous

$$F_1 = M_1 a_1 \tag{3.3}$$

and of course similarly

$$F_2 = M_2 a_2 \tag{3.4}$$

where a_i is the acceleration that particle i undergoes, due to the force F_i.

If we use these relations with Eqs. (3.1, 3.2), we find

$$a_1 = G\frac{M_2}{r^2} \tag{3.5}$$

and

$$a_2 = -G\frac{M_1}{r^2} \tag{3.6}$$

Carol: So even though the forces are equal, the accelerations are not. In the case you mentioned, particle 1 is twice as massive as particle 2. That means that the acceleration a_2 of particle 2 is twice as large as the acceleration a_1 of particle 1.

Dan: That makes sense. It takes twice as much force to move a particle that is twice as massive.

3.4 Center of Mass

Erica: Let's draw the accelerations, as arrows in the picture, fig. 3.2 that I sketched earlier. Here they are, in figure 3.4.

Carol: And this means that the two particles start falling toward each other at different rates, and therefore they will not meet in the middle.

Dan: They will meet closer to particle 1. In fact, because particle 2 will always be accelerated twice as fast as particle 1, I bet they will meet at a point that is one third of the way over, going from 1 to 2.

Carol: Could that be the fourth point that you talked about, Erica, what you called the center of mass?

Erica: Yes, indeed! The center of mass, c.o.m., in the case of the two-body problem, is the point of symmetry, around which each of the two bodies moves in a way that is inversely proportional to their mass. As I remember it, the position of the c.o.m. is given by

$$r_{com} = \frac{M_1 r_1 + M_2 r_2}{M_1 + M_2} \tag{3.7}$$

Let me show you explicitly how this all works, using the equations of motion, Eqs. (3.5, 3.6), where I will remind us that, for each particle, the acceleration is the second derivative of the position:

$$\frac{d^2}{dt^2} r_1 = G\frac{M_2}{r^2} \tag{3.8}$$

and

$$\frac{d^2}{dt^2} r_2 = -G\frac{M_1}{r^2} \tag{3.9}$$

We are looking for the point around which everything else turns, a point that does not get pushed, in other words, a point that does not undergo an acceleration. Perhaps you can see how to find a point with zero acceleration, when you add those two equations?

Dan: No, I don't see that. Adding the right hand sides does not give zero.

Carol: Ah, but multiplying the first equation by M_1 and the second equation by M_2 gives

$$M_1 \frac{d^2}{dt^2} r_1 = G\frac{M_1 M_2}{r^2} \tag{3.10}$$

and

$$M_2\frac{d^2}{dt^2}r_2 = -G\frac{M_1M_2}{r^2} \tag{3.11}$$

so when we add the two equations we get zero at the right hand side:

$$M_1\frac{d^2}{dt^2}r_1 + M_2\frac{d^2}{dt^2}r_2 = 0 \tag{3.12}$$

Dan: Okay, that I can see. But what does it mean?

Carol: Erica, correct me if I'm wrong, but I think what it means is that the attractive gravitational forces that the two particles exert on each other are balanced: equal but opposite in direction. For my own peace of mind, let me summarize what I think has happened so far.

Eq. (3.1) is Newton's gravitational equation, and so is Eq. (3.2), while (3.3) and (3.4) express Newton's law connecting force and acceleration. Combining both of these Newtonian laws allows us to write (3.5) and (3.6) as Newton's gravitational equations in terms of acceleration rather than force. After that, (3.8) and (3.9) are just rewriting (3.5) and (3.6) in a different notation. Then (3.12) adds (3.8) and (3.9), showing thereby in a more formal way that the attractions between the two stars cancel each other. We could have drawn that conclusion directly from (3.1) and (3.2) as well, but there we would not yet have had the information about the second derivatives of the positions.

Erica: Yes, that sums it up nicely. And we can write Eq. (3.12) as:

$$\frac{d^2}{dt^2}\left\{M_1r_1 + M_2r_2\right\} = 0 \tag{3.13}$$

Notice that this expression looks a lot like the definition of the center of mass, Eq. (3.7). Indeed, it implies that:

$$\frac{d^2}{dt^2}\left\{\frac{M_1r_1 + M_2r_2}{M_1 + M_2}\right\} = 0 \tag{3.14}$$

and with Eq. (3.7) it can be written as:

$$\frac{d^2}{dt^2}r_{com} = 0 \tag{3.15}$$

This means that the c.o.m. position will either be at rest, or move at a uniform velocity, with zero acceleration.

Carol: Let's check whether that makes sense in our case. We have $M_1 = 2M_2$, so:

$$r_{com} = \frac{2M_2r_1 + M_2r_2}{2M_2 + M_2} = \frac{2r_1 + r_2}{2+1} = \tfrac{2}{3}r_1 + \tfrac{1}{3}r_2 \tag{3.16}$$

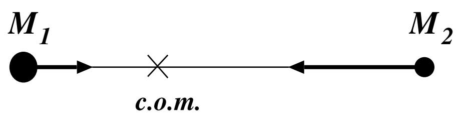

Figure 3.5: The center of mass is the point at rest, toward which the bodies fall.

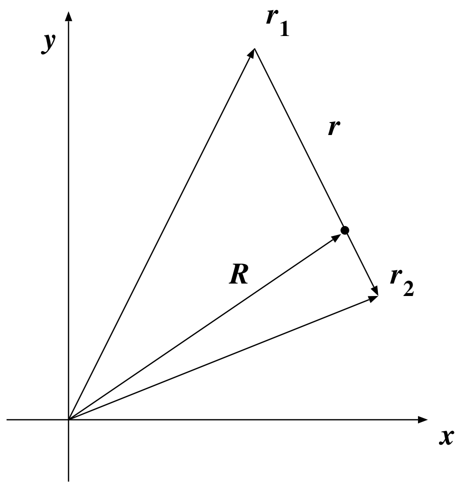

Figure 3.6: Construction of alternative coordinates **R** and **r** for the center of mass position and the relative position of the two bodies, respectively.

Indeed, this is the point that is twice as close to particle 1 as to particle 2.

Dan: Not so fast, how can you see that?

Carol: You can choose the origin of our coordinates where you like. Take particle 1 to be the origin, for example, and you will get $r_1 = 0$ and then Eq. (3.16) gives you $r_{com} = \frac{1}{3}r_2$. Or take particle 2 as the origin, which means $r_2 = 0$ and Eq. (3.16) gives you $r_{com} = \frac{2}{3}r_1$, which means at a distance from particle 2 that is two thirds on the way to particle 1.

Dan: Well, let me try your game too, at a somewhat more complicated point. Let me put the origin of the coordinates exactly half-way between the particles. In that case, $r_1 = -r_2$. Now let's see what Eq. (3.16) gives me. Aha: $r_{com} = -\frac{2}{3}r_2 + \frac{1}{3}r_2 = -\frac{1}{3}r_2$. What do you know! One third of the way from the origin in the direction toward particle 1. Exactly the point that is twice as close to particle 1 as to particle 2. Okay, I'm convinced!

3.5 Relative Coordinates

Erica: In the last few pictures we have dealt with a one-dimensional configuration, two mass points on a line. This meant that we only had one coordinate the worry about, the x value of the position. In the more general case of two or three dimensions, we have to go back to vector notation. Eq. (3.7) in vector form becomes:

$$\mathbf{R} = \frac{M_1\mathbf{r}_1 + M_2\mathbf{r}_2}{M_1 + M_2} \tag{3.17}$$

It is also convenient to define $\mathbf{r}$ as the relative position of the second particle with respect to the first particle:

$$\mathbf{r} = \mathbf{r}_2 - \mathbf{r}_1 \tag{3.18}$$

I have drawn them in fig. 3.6 as they are constructed from $\mathbf{r}_1, \mathbf{r}_2$.

Dan: But now you have changed your mind, assuming that particle 2 is heavier than particle 1!

Erica: Why not? These figures should be correct for any choice of masses. I just didn't want to get stuck with the same simple choice for all the figures we will draw.

Carol: So far, you have told us *what* the center of mass is, but you haven't told us *why* it is called that way. Looking again at Eq. (3.17), it seems that the c.o.m. is the mass weighted position of an extended object. We use that notion in computer graphics for computer games all the time. One way to look at it is to ask yourself where you should support a rod with two weights attached at the end. If one of the weights is twice as heavy as the other, it should be twice as close to the support point as the other is, in order to balance the rod.

Dan: I see, that helps. So with 'mass weighted' you mean that I multiply the position of each particle with its mass, so that more massive particles have a larger vote, so to speak, in where the center of mass will be.

Carol: Yes, and then you have to divide by the total mass. Already on dimensional grounds it is clear that you have to divide by a mass, as Erica told us, to wind up with a result that has the dimension of length. And in the equal mass case, it is clear that the c.o.m. should be the average of the two positions, the point right in between them.

Dan: Okay, I am happy now with Eq. (3.17), which leads to fig. 3.6. But now I'm confused about something else. Let us draw a figure that contains only the two new vectors, $\mathbf{R}, \mathbf{r}$, that define the alternative coordinate system. Here it is, fig. 3.7.

Now here is what seems odd. In fig. 3.1 the two vectors $\mathbf{r}_1, \mathbf{r}_2$ start at the same

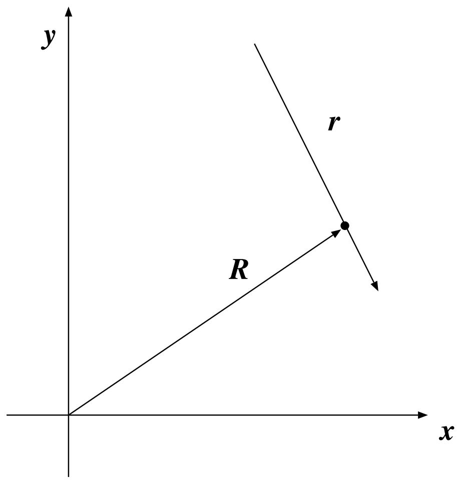

Figure 3.7: Alternative coordinates **R** and **r**.

point, namely the origin. But in fig. 3.7 only the c.o.m. vector **R** starts in the origin, while the relative position vector **r** connecting the two particles does not.

In other words, the two coordinate systems do not seem to be compatible.

Erica: Good point. It is true that the information contained in the pair of vectors $\mathbf{r}_1, \mathbf{r}_2$ is the same as the information contained in the vector pair $\mathbf{R}, \mathbf{r}$. But in the first case, both vectors appear in the same inertial coordinate system, while in the second case, only **R** is a vector in an inertial coordinate system, while **r** points to particle 2 within a coordinate system anchored to particle 1, which is certainly not inertial.

Carol: It is not inertial because particle 1 is not moving in a straight line at constant speed. Is that what you mean?

Erica: Yes. Any coordinate system anchored to a particle that deflected by forces acting upon it cannot be an inertial system.

Dan: So the vectors $\mathbf{r}_1, \mathbf{r}_2, \mathbf{R}$ are all defined in the same original inertial coordinate system, while **r** is defined in a different coordinate system, which is not inertial. Let me make that clear, and draw two new figures. Figure 3.8 shows the c.o.m. vector in the original coordinate frame, while figure 3.9 shows the relative separation vector in a different frame, anchored on particle 1, as you just explained.

Erica: Yes, that is a nice way to split it out.

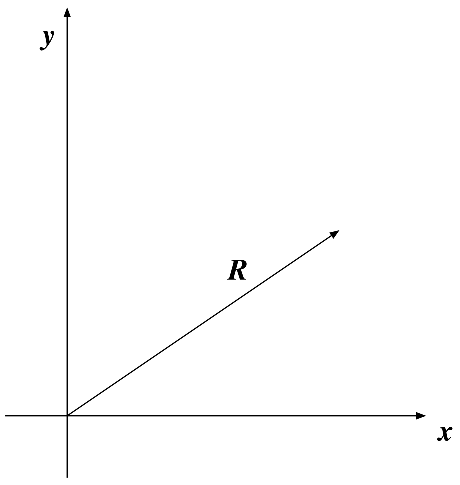

Figure 3.8: The vector **R** in the inertial coordinate system

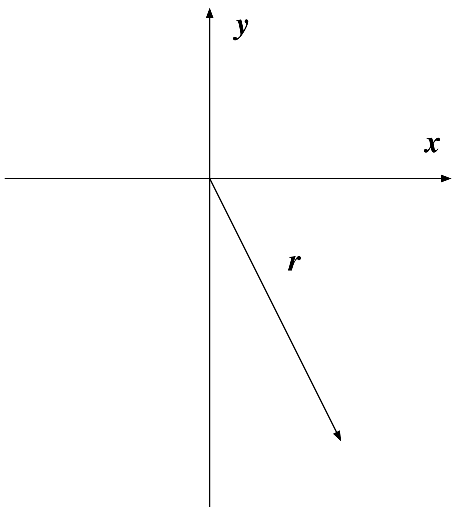

Figure 3.9: The vector **r** in a non-inertial coordinate system

Chapter 4

A Gravitational 1-Body Problem

4.1 Coordinates

Dan: But I'm still confused. You started talking about a change from absolute to relative coordinates. But in fact we seem to switch from two vectors in an absolute inertial coordinate system to two other vectors, one in the old coordinate system and one in a new, non-inertial coordinate system. What is going on?

Erica: I guess the notation and terminology in physics is rather sloppy. We generally don't try to be logically precise, the way mathematicians are. But I'm glad you're forcing us to be clear about our terms. It helps me, too, to build things up again from scratch.

It seems that there are two ways to use the notion of coordinates. But let me first summarize what we have learned.

We have, so far, dealt with two coordinate frames. We have the inertial one, centered on an arbitrary point, a point that is either at rest or in uniform motion with respect to absolute space. The coordinate axes point in fixed directions with respect to absolute space. And we also have the non-inertial frame, centered on particle 1. And although this frame is non-inertial, because particle 1 feels the force of particle 2 and therefore does not move in a straight line at constant speed, the coordinate axes of the non-inertial coordinate system remain parallel to the coordinate axes of the inertial coordinate system.

This last point is important. When the two particles complete one orbit around each other, the relative vector **r** will also complete one orbit in the non-inertial coordinate system given in fig. 3.9.

Now here are the two ways that physicists use ‘coordinates’, at least as I understand it. The first way is to give the coordinates of vectors with respect to the coordinate frame in which they are defined. In that case, we have already dealt with the coordinates of the four vectors $\mathbf{r}_1, \mathbf{r}_2, \mathbf{R}, \mathbf{r}$. In each case, a single vector, or equivalently the values of the components of that vector, describe the position of one particle with respect to a point in space. For example, $\mathbf{r}_2$ describes the position of particle 2 with respect to the origin, and $\mathbf{r}$ describes the position of particle 2 with respect to the point in space temporarily occupied by particle 1.

The second way to talk about coordinates is to capture the information about a whole system, and not just a single particle with respect to a point in space. We can say that we have described a two-body configuration completely when we give the information contained in the pair of vectors $\mathbf{r}_1, \mathbf{r}_2$, or equivalently, in the values of their four components in two dimensions, or their six components in three dimensions. But we can also describe the same two-body configuration completely by giving the information contained in the pair of vectors $\mathbf{R}, \mathbf{r}$.

So in the second way of speaking, a coordinate transformation means a change between describing the two-body system through specifying $\mathbf{r}_1, \mathbf{r}_2$ and describing the same system through specifying $\mathbf{R}, \mathbf{r}$.

In the beginning of our session, I used the second way of speaking. But then, when we started talking about coordinate systems, I guess I slipped into the first way of speaking.

Dan: Thanks for separating those two ways of speaking. I'll have to go over it a few times more, to get fluent in this way of thinking, but I'm beginning to see the light.

Carol: Now the beautiful thing is: it *is* possible to use the same language for both ways of speaking. This is something else I learned in our ‘computer game’ class as we called it, although it was really titled as a ‘geometrical representation’ class. If a single vector lives in a d-dimensional space, then a system of two such vectors can also be represented as a single vector in a $2d$-dimensional space. And then your second way of speaking with respect to the d-dimensional space boils down to your first way of speaking with respect to the $2d$-dimensional space.

Mathematically speaking, each choice of a set of two vectors, whether it is $\{\mathbf{r}_1, \mathbf{r}_2\}$ or $\{\mathbf{R}, \mathbf{r}\}$, determines a single point in the direct product of two copies of the base space in which the single vectors live. In our case, we have started from two-dimensional vectors, so the space for pairs of vectors is four-dimensional. And the coordinate transformation that Erica has introduced is really a bijective mapping between two four-dimensional spaces.

Dan: Just when I thought I understood something, Carol manages to make it sound all gobbledygook again. I'll just stick with Erica's explanation.

4.2 Equivalent coordinates

Carol: Well, I have one question left. I mentioned that the transformation between the $\{\mathbf{r}_1, \mathbf{r}_2\}$ coordinate system and the $\{\mathbf{R}, \mathbf{r}\}$ was bijective. What that means is that any pair $\{\mathbf{r}_1, \mathbf{r}_2\}$ corresponds to a unique pair $\{\mathbf{R}, \mathbf{r}\}$, and vice versa.

I think that is true, but I would like to prove it, to make sure, and to see explicitly which pair of $\{\mathbf{r}_1, \mathbf{r}_2\}$ corresponds to which pair of $\{\mathbf{R}, \mathbf{r}\}$ vectors.

Dan: Good questions! Erica has shown how to derive $\mathbf{R}$ and $\mathbf{r}$ from $\mathbf{r}_1$ and $\mathbf{r}_2$, with the definitions above. But when we are given only $\mathbf{R}$ and $\mathbf{r}$, can we then really recover the original $\mathbf{r}_1$ and $\mathbf{r}_2$?

Erica: Yes, we can, or at least I'm pretty sure we can. But we'll have to scratch our heads a bit to write it down. Let us start with figure 3.6. We can use the same figure, but now we should consider $\mathbf{R}$ and $\mathbf{r}$ as given, and the question is how to derive the values for $\mathbf{r}_1$ and $\mathbf{r}_2$.

When I took my classical mechanics course, many homework questions were of this type, and generally they involved a clever form of coordinate transformation. Hmmm. Let's see. Right now the origin is at an arbitrary point in space. We could shift to a coordinate frame that is centered on one of our three special points; the position of particle 1, the position of particle 2, or the position of the c.o.m.

Well, why not start with particle 1, and see what happens. Let me draw the new coordinate frame, using primed symbols: x' and y' instead of x and y for the coordinate axes.

Carol: Ah, I see, that is a great move, really! In this new coordinate system, the c.o.m. position is given by the vector $\mathbf{R}'$, pointing from the position of particle 1 to the c.o.m. This means that we can reconstruct $\mathbf{r}_1$ as the sum of $\mathbf{R}$ and $-\mathbf{R}'$!

Dan: Wait a minute, not so quick. I don't see that yet. Let me try to take smaller steps. The vector $-\mathbf{R}'$ must point, by definition, in the opposite direction of $\mathbf{R}'$. So you can go from the old origin to $\mathbf{r}_1$ by first following the vector $\mathbf{R}$ from start till tip, and then following the vector $-\mathbf{R}'$, which conveniently starts at the tip of $\mathbf{R}$. And the tip of $-\mathbf{R}'$ lands on particle 1.

So far so good. And this means that we have $\mathbf{r}_1 = \mathbf{R} + (-\mathbf{R}')$, or in simpler terms

$$\mathbf{r}_1 = \mathbf{R} - \mathbf{R}' \qquad (4.1)$$

Fine! But how do we compute this vector $-\mathbf{R}'$?

Carol: Elementary, my dear Watson. In the old coordinate frame, we had Eq. (3.17) which gave us the position vector of the c.o.m. in that frame. Let's write

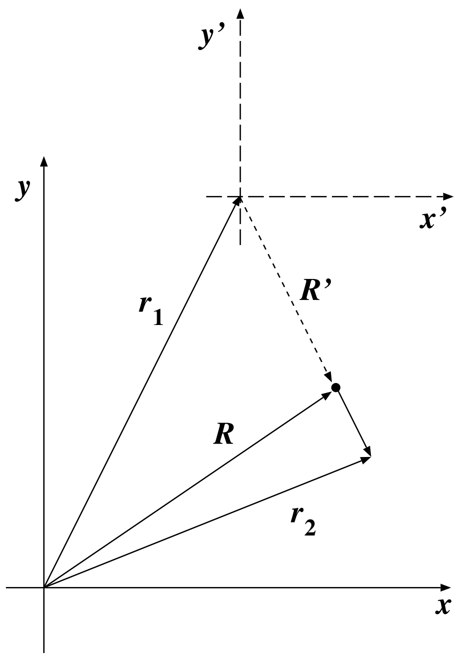

Figure 4.1: A shift of coordinate frame brings the origin to the position of particle 1. In this new frame, the c.o.m. vector is $\mathbf{R}'$.

it again:

$$\mathbf{R} = \frac{M_1\mathbf{r}_1 + M_2\mathbf{r}_2}{M_1 + M_2} \tag{4.2}$$

This expression is valid in any coordinate frame, so we can use it for the new primed coordinate frame as well. In the new frame, $\mathbf{r}'_1 = 0$ because the new origin lies smack on the first particle, so that particle has distance zero to the new origin. And the position vector of the second particle is given as $\mathbf{r}'_2 = \mathbf{r}_2 - \mathbf{r}_1 = \mathbf{r}$. Erica, your choice of coordinate frame shifting was brilliant! We have in the new frame:

$$\mathbf{R}' = \frac{M_1\mathbf{r}'_1 + M_2\mathbf{r}'_2}{M_1 + M_2} = \frac{M_1 0 + M_2\mathbf{r}}{M_1 + M_2} = \frac{M_2}{M_1 + M_2}\mathbf{r} \tag{4.3}$$

If we now use Eq.(4.1) that Dan just derived, and substitute the value of $\mathbf{R}'$ that we found in Eq.(4.3), we get:

$$\mathbf{r}_1 = \mathbf{R} - \frac{M_2}{M_1 + M_2}\mathbf{r} \tag{4.4}$$

4.3 Closing the Circle

Dan: And the second particle's position is obtained simply by interchanging the subscripts 1 and 2 everywhere, right?

Erica: Wrong, but almost right: there is an additional sign change. We can show that in the same way as Carol just did, but putting the origin now in the position of particle 2. Or, even simpler, we can look at the picture, which tells us that:

$$\mathbf{r}_2 = \mathbf{r}_1 + r = \mathbf{R} - \frac{M_2}{M_1 + M_2}\mathbf{r} + \frac{M_1 + M_2}{M_1 + M_2}\mathbf{r} \tag{4.5}$$

So you see, this boils down to an expression with a plus sign, rather than the minus sign in Eq.(4.4):

$$\mathbf{r}_2 = \mathbf{R} + \frac{M_1}{M_1 + M_2}\mathbf{r} \tag{4.6}$$

Dan: Okay, okay, I grant you your positive sign. But I'm still not fully satisfied.

Carol: First, starting with $\{\mathbf{r}_1, \mathbf{r}_2\}$ we have derived expressions for $\{\mathbf{R}, \mathbf{r}\}$. Then, starting with $\{\mathbf{R}, \mathbf{r}\}$ we have derived expressions for $\{\mathbf{r}_1, \mathbf{r}_2\}$. What more could you possibly want?!?

Dan: Let me play the devil's advocate. We are doing science, so I want to have hard evidence! As you already saw, it is all too easy to replace a plus sign with a minus sign and stuff like that, so we'd better make sure we really get things right. Let me try to prove it my way.

Carol: Prove what?

Dan: Prove that we are consistent, and that we can close the circle of transformations, from $\{\mathbf{r}_1, \mathbf{r}_2\}$ to $\{\mathbf{R}, \mathbf{r}\}$ and then back again to $\{\mathbf{r}_1, \mathbf{r}_2\}$.

I will take Eq. (4.4) and then use the original definitions Eqs. (3.17) and (3.18):

$$\begin{aligned}
\mathbf{r}_1 &= \mathbf{R} - \frac{M_2}{M_1+M_2}\mathbf{r} \\
&= \frac{M_1\mathbf{r}_1 + M_2\mathbf{r}_2}{M_1+M_2} - \frac{M_2}{M_1+M_2}(\mathbf{r}_2 - \mathbf{r}_1) \\
&= \frac{M_1+M_2}{M_1+M_2}\mathbf{r}_1 + \frac{M_2-M_2}{M_1+M_2}\mathbf{r}_2 \\
&= \mathbf{r}_1 \qquad (4.7)
\end{aligned}$$

Yes, now I am convinced. And I can already see more or less how it works for $\mathbf{r}_2$. But, to be really sure, I'd like to finish the job:

$$\begin{aligned}
\mathbf{r}_2 &= \mathbf{R} + \frac{M_1}{M_1+M_2}\mathbf{r} \\
&= \frac{M_1\mathbf{r}_1 + M_2\mathbf{r}_2}{M_1+M_2} + \frac{M_1}{M_1+M_2}(\mathbf{r}_2 - \mathbf{r}_1) \\
&= \frac{M_1-M_1}{M_1+M_2}\mathbf{r}_1 + \frac{M_2+M_1}{M_1+M_2}\mathbf{r}_2 \\
&= \mathbf{r}_2 \qquad (4.8)
\end{aligned}$$

Carol: That *is* nice, I must admit, to see the truth in front of us so clearly.

Erica: I agree. Okay, all three of us are happy now. Let's move on!

4.4 Newton's Equations of Motion

Carol: Well, we got a new system of relative coordinates. I presume we're going to use it for something, right?

Erica: Yes, time to rewrite Newton's equations of motion into the new system. For the one-dimensional case above, we used Eqs. (4.9) in scalar form. Let me write it in vector form. So this is the equation for the acceleration of the first particle, due to the gravitational force that the second particle exerts on it:

$$\frac{d^2}{dt^2}\mathbf{r}_1 = G\frac{M_2}{r^3}\mathbf{r} \tag{4.9}$$

Here I have used the abbreviation

$$r = |\mathbf{r}| \tag{4.10}$$

for the absolute value of the vector **r**, which in our two-dimensional case can be written as

$$r = \sqrt{x^2 + y^2} \tag{4.11}$$

while in general, in 3D, it will be

$$r = \sqrt{x^2 + y^2 + z^2} \tag{4.12}$$

I'm glad you both have at least some familiarity with differential equations. It may not be a bad idea to brush up your knowledge, if you want to know more about the background of Newtonian gravity. There are certainly plenty of good introductory books. At this point it is not necessary, though, to go deeply into all that. I can just provide the few equations we need to get started, and for quite a while our main challenge will be to figure out how to solve these equations.

Dan: Glad to hear that! But I'm puzzled about one thing. Why is there a third power in the denominator? I thought that gravity shows an inverse square attraction, not an inverse cube! And after all, that was what we wrote in Eq. (4.9).

Erica: Yes, the magnitude of the acceleration is indeed proportional to the minus second power of the separation. However, we also need to indicate the direction of the acceleration. We can define a unit vector $\hat{\mathbf{r}}$ pointing in the direction of the second particle, as seen from the position of the first particle:

$$\hat{\mathbf{r}} = \frac{\mathbf{r}}{r} \tag{4.13}$$

Using this unit vector, we can rewrite Eq. (4.9) as:

$$\frac{d^2}{dt^2}\mathbf{r}_1 = G\frac{M_2}{r^2}\hat{\mathbf{r}} \tag{4.14}$$

Dan: I see. So the magnitude is indeed inverse square, but the direction is given by the unit vector, which has length one, and therefor does not influence the length of the acceleration vector. I like this way of writing better, since it brings out the physics more clearly.

Carol: I guess that Erica wrote it in the form of Eq. (4.9) because it will be easier to program that way.

Erica: Indeed. Once you get used to this way of writing the equations of motion, there is no need to introduce the new quantity $\hat{\mathbf{r}}$, since it is not used anywhere separately.

Let me also write the acceleration for the second particle:

$$\frac{d^2}{dt^2}\mathbf{r}_2 = -G\frac{M_1}{r^3}\mathbf{r} \tag{4.15}$$

Carol: Look Dan, another sign change: the force of attraction exerted by the first particle on the second points toward the first particle.

Dan: Hmm, I'm still a bit confused about these signs. When the force points to the first particle, why does that imply a minus sign?

Carol: The easiest way to see this is to take a particular case. Imagine that the second particle is positioned in the origin of the coordinates. Since gravity pulls particle 2 in the direction of particle 1, the acceleration that particle 2 experiences points in the direction of $\mathbf{r}_1$. Notice that in this particular case $\mathbf{r} = \mathbf{r}_2 - \mathbf{r}_1 = -\mathbf{r}_1$. Therefore, the direction $\mathbf{r}_1$ is the opposite of the direction of $\mathbf{r}$, hence the minus sign.

Dan: Ah, that is neat. Instead of trying to figure things out in full generality, you take a particular limiting case, and check the sign. Sort of like what Erica did, in constructing here a primed coordinate system. Since you already know the magnitude and the line along which the acceleration is directed, once you know the sign in one case, you know the sign in all cases.

Carol: Yes. Technically we call this invariance under continuous deformation. If you bring the second particle a little bit out of the origin, by small continuous changes, the acceleration between the particles must change continuously as well; it cannot suddenly flip to the opposite direction.

Erica: Neat indeed: this means that understanding the sign in one place let you know the sign in all places. I'll remember using that trick.

Okay, onward with the equations of motion. Given Eqs.(4.9) and (4.15), we can calculate the accelerations for the alternative coordinates by using the defining equations (3.17) and (3.18), as follows.

$$\begin{aligned}
\frac{d^2}{dt^2}\mathbf{R} &= \frac{M_1}{M_1+M_2}\frac{d^2}{dt^2}\mathbf{r}_1 + \frac{M_2}{M_1+M_2}\frac{d^2}{dt^2}\mathbf{r}_2 \\
&= \frac{M_1}{M_1+M_2}\left(G\frac{M_2}{r^3}\mathbf{r}\right) + \frac{M_2}{M_1+M_2}\left(-G\frac{M_1}{r^3}\mathbf{r}\right) \\
&= G\frac{M_1M_2}{M_1+M_2}\left(\frac{1}{r^3}\mathbf{r} - \frac{1}{r^3}\mathbf{r}\right)
\end{aligned}$$

$$= \quad 0 \tag{4.16}$$

and

$$\begin{aligned}\frac{d^2}{dt^2}\mathbf{r} &= \frac{d^2}{dt^2}\mathbf{r}_2 - \frac{d^2}{dt^2}\mathbf{r}_1 \\ &= \left(-G\frac{M_2}{r^3}\mathbf{r}\right) - \left(G\frac{M_1}{r^3}\mathbf{r}\right) \\ &= -G\frac{M_1+M_2}{r^3}\mathbf{r}\end{aligned} \tag{4.17}$$

4.5 An Equivalent 1-Body Problem

Dan: The second equation looks like a form of Newton's law of gravity, but what does it mean that the first equation gives just zero as an answer?

Carol: Well, it seems that there is zero acceleration for the position of the center of mass.

Dan: Ah, so the c.o.m. moves with constant velocity? But of course, that is what we found in the one-dimensional case too.

Erica: Yes, and this means that we can choose an inertial coordinate frame that moves along with the c.o.m., and in that coordinate frame the center of mass does not move at all. And to make it really simple, we can choose a coordinate frame where the c.o.m. is located at the origin of the coordinates.

Dan: That does make life simple. In this coordinate frame, all the information about the motions in the two-body problem is now bundled in Eq. (4.17). It almost looks as if we are dealing with a 1-body problem, instead of a 2-body problem!

Erica: Yes, this is what I meant when I announced that we could map the two-body problem into an equivalent one-body problem, for any choice of the masses. The original equations (4.9) and (4.15) are coupled: both $\mathbf{r}_1$ and $\mathbf{r}_2$ occur in both equations, indirectly through the fact that $\mathbf{r} = \mathbf{r}_2 - \mathbf{r}_1$. In contrast, the equations (4.16) and (4.17) are decoupled.

A clear way to show this is to draw two separate figures, Figs. 3.8 and 3.9. The c.o.m. vector in Fig. 3.8 moves in a way that is totally independent of the way the relative vector in 3.9 moves. The c.o.m. vector moves at a constant speed, while the relative vector moves as if it follows an abstract particle in a gravitational field.

To see this, notice that Eq. (4.17) is *exactly* Newton's equation for the gravitational acceleration of a small body, a test particle, that feels the attraction of a hypothetical body of mass $M_1 + M_2$. To check this, look at Eq. (4.4), and replace $\mathbf{r}_1$ by $\mathbf{r}$ and replace M_2 by $M_1 + M_2$.

Dan: Indeed. So the relative motion between two bodies can be described as if it was the motion of just one body under the gravitational attraction of another body, that happens to have a mass equal to the sum of the masses of the two original bodies.

Erica: Exactly. And to complete this particular picture, we have to make sure that the other body stays in the origin, at the place of the center of mass. We can do this by giving our alternative body a mass zero. In other words, we consider the motion of a massless test particle under the influence of the gravitational field of a body with mass $M_1 + M_2$, that is located in the center of the coordinate system.

Carol: I prefer to give it a non-zero mass. No material body can have really zero mass. Instead, we can consider it to have just a very very small mass. We could call it ϵ, as mathematicians do when they talk about something so small as to be almost negligible.

Dan: If you like. I'm happy with the physical limit that Erica mentioned, rather than the type of mathematical nicety that you introduced. Zero I can understand. Because the test particle has zero mass, it exerts zero gravitational pull on the central body. Therefore, the central body does not move at all, and the only task we are left with is to determine the motion of the test particle around the center.

And that is a one-body problem. Okay, I now see the whole picture.

4.6 Wrapping Up

Carol: Let us gather the formulas we have obtained so far, for our coordinate transformation, from absolute to relative coordinates.

$$\left\{\begin{aligned} \mathbf{r}_1 &= \mathbf{R} - \frac{M_2}{M_1+M_2}\mathbf{r} \\ \mathbf{r}_2 &= \mathbf{R} + \frac{M_1}{M_1+M_2}\mathbf{r} \end{aligned}\right. \tag{4.18}$$

$$\left\{\begin{aligned} \mathbf{R} &= \frac{M_1\mathbf{r}_1 + M_2\mathbf{r}_2}{M_1+M_2} \\ \mathbf{r} &= \mathbf{r}_2 - \mathbf{r}_1 \end{aligned}\right. \tag{4.19}$$

$$\left\{\begin{aligned} \frac{d^2}{dt^2}\mathbf{r}_1 &= G\frac{M_2}{r^3}\mathbf{r} \\ \frac{d^2}{dt^2}\mathbf{r}_2 &= -G\frac{M_1}{r^3}\mathbf{r} \end{aligned}\right. \tag{4.20}$$

$$\begin{cases} \frac{d^2}{dt^2}\mathbf{R} = 0 \\ \\ \frac{d^2}{dt^2}\mathbf{r} = -G\frac{M_1 + M_2}{r^3}\mathbf{r} \end{cases} \tag{4.21}$$

Dan: So from all these equations, what we are going to solve with our computer program is the last one above, right?

Erica: Yes, and let me write the equation once again here:

$$\frac{d^2}{dt^2}\mathbf{r} = -G\frac{M_1 + M_2}{r^3}\mathbf{r} \tag{4.22}$$

And there is one more thing: let's make life as simple as we can, by choosing a system of physical units in which the gravitational constant and the total mass of the 2-body system are both unity:

$$\begin{array}{rcl} G & = & 1 \\ M_1 + M_2 & = & 1 \end{array} \tag{4.23}$$

Our original equation of motion now becomes simply:

$$\frac{d^2}{dt^2}\mathbf{r} = -\frac{\mathbf{r}}{r^3} \tag{4.24}$$

Dan: I can't imagine anything simpler than that! Let's start coding.

Chapter 5

Writing the Code

5.1 Choosing a Computer Language

Carol: Let's start coding! Which language shall we use to write a computer code? I bet you physics types insist on using Fortran.

Erica: Believe it or not, most of the astrophysics code I'm familiar with has been written in C++. It may not be exactly my favorite, but it is at least widely available, well supported, and likely to stay with us for decades.

Dan: What is C++, and why the obscure name? Makes the notion of an N-body problem seem like clarity itself!

Carol: Long story. I don't know whether there was ever a language A, but there certainly was a language B, which was followed alphabetically by a newer language C, which became quite popular. Then C was extended to a new language for object-oriented programming, something we'll talk about later. In a nerdy pun, the increment operation "++" from the C language was used to indicate that C++ was the successor language to C.

Dan: But everybody I know seems to be coding in Java.

Carol: Java has a clear advantage over C++ in being almost platform independent, but frankly, I don't like either C++ or Java. Recently, I took a course in which the instructor used a scripting language, Ruby. And I was surprised at the flexible way in which I could quickly express rather complex ideas in Ruby.

Erica: Does Ruby have something like STL?

Carol: You mean the Standard Template Library in C++? Ruby doesn't need any such complications because it is already dynamically typed from the start!

Dan: I have no idea what the two of you are talking about, but I agree with Carol, let's start coding, in whatever language!

5.2 Choosing an Algorithm

Carol: We want to write a simulation code, to enable us to simulate the behavior of stars that move under the influence of gravity. So far we have derived the equations of motion for the relative position of one particle with respect to the other. What we need now is an algorithm to integrate these equations.

Dan: What does it mean to integrate an equation?

Carol: We are dealing with differential equations. In calculus, differentiation is the opposite of integration. If you differentiate an expression, and then integrate it again, you get the same expression back, apart from a constant. Our differential equations describe the time derivatives of position and velocity. In order to obtain the actual values for the position and velocity as a function of time, we have to integrate the differential equation.

Erica: And to do so, we need an integration algorithm. And yes, there is a large choice! If you pick up any book on numerical methods, you will see that you can select from a variety of lower-order and higher-order integrators, and for each one there are additional choices as to the precise structure of the algorithm.

Dan: What is the order of an algorithm?

Erica: It signifies the rate of convergence. Since no algorithm with a finite time step size is perfect, they all make numerical errors. In a fourth-order algorithm, for example, this error scales as the fourth power of the time step – hence the name fourth order.

Carol: If that is the case, why not take a tenth order or even a twentieth order algorithm. By only slightly reducing the time step, we would read machine accuracy, of order 10^{-15} for the usual double precision (8 byte, i.e. 64 bit) representation of floating point numbers.

Erica: The drawback of using high-order integrators is two-fold: first, they are far more complex to code; and secondly, they do not allow arbitrarily large time steps, since their region of convergence is limited. As a consequence, there is an optimal order for each type of problem. When you want to integrate a relatively well-behaved system, such as the motion of the planets in the solar system, a twelfth-order integrator may well be optimal. Since all planets follow well-separated orbits, there will be no sudden surprises there. But when you integrate a star cluster, where some of the stars can come arbitrarily close to each other, experience shows that very high order integrators lose their edge. In practice, fourth-order integrators have been used most often for the job.

Dan: How about starting with the lowest-order integrator we can think of? A zeroth-order integrator would make no sense, since the error would remain constant, independent of the time step size. So the simplest one must be a first-order integrator.

Erica: Indeed. And the simplest version of a first-order integrator is called the *forward Euler* integrator.

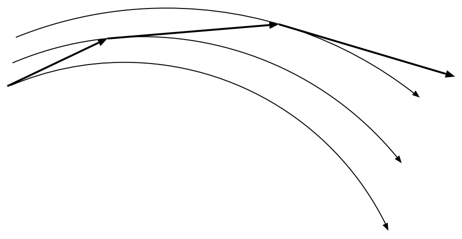

Figure 5.1: The forward Euler approximation is indicated by the straight arrows, while the curved lines show the true solutions to the differential equation.

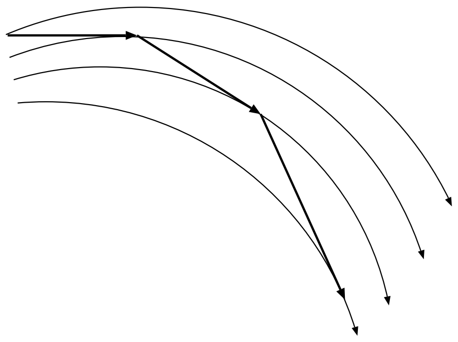

Figure 5.2: As figure 5.1, but now for the backward Euler approximation.

Dan: Was Euler so forward-looking, or is there also a *backward Euler* algorithm?

Erica: There is indeed. In the forward version, at each time step you simply take a step tangential to the orbit you are on. After that, at the next step, the new value of the acceleration forces you to slightly change direction, and again you move for a time step *dt* in a straight line in that direction. Your approximate orbit is thus constructed out of a number of straight line segments, where each one has the proper direction at the beginning of the segment, but the wrong one at the end.

Dan: And the *backward Euler* algorithm must have the right direction at the end of a time step, and the wrong one at the beginning. Let's see. That seems much harder to construct. How do you know at the beginning of a time step in what direction to move so that you come out with the right direction tangential to a correct orbit at that point?

Erica: You do that through iteration. You guess a direction, and then you correct for the mistake you find yourself making, so that your second iteration is much more accurate, in fact first-order accurate. Given this extra complexity, I suggest that we start with the forward Euler algorithm.

Carol: Can't we do both, *ie* make half the mistakes of each of the two, while trying to strike the right balance between forward and backward Euler?

Erica: Aha! That is a good way to construct better algorithms, which then become second-order accurate, because you have canceled the first-order errors. Examples are second-order Runge Kutta, and leapfrog. We'll soon come to that, but for now let's keep it simple, and stay with first order. Here is the mathematical notation:

$$\begin{aligned} \mathbf{r}_{i+1} &= \mathbf{r}_i + \mathbf{v}_i dt \\ \mathbf{v}_{i+1} &= \mathbf{v}_i + \mathbf{a}_i dt \end{aligned} \qquad (5.1)$$

for the position **r** and velocity **v** of an individual particle, where the index i indicates the values for time t_i and $i+1$ for the time t_{i+1} after one more time step has been taken: $dt = t_{i+1} - t_i$. The acceleration induced on a particle by the gravitational forces of all other particles is indicated by **a**. So, all we have to do now is to code it up. By the way, let's rename the file. Rather than a generic name `nbody.rb`, let's call it `euler.rb`, or even better `euler_try.rb`. After all, most likely we'll make a mistake, or two, or more, before we're finished!

5.3 Specifying Initial Conditions

Carol: I have learned that in order to solve a differential equation, you have to provide initial conditions.

Erica: Yes. It is like using a map: if you don't know where you are, you can't use it. You start with the spot marked "you are here", and then you can start walking, using the knowledge given by the map.

In our case, a differential equation tells you how a system evolves, from one point to the next. Once you know where you are at time `0`, the equation tells you where you will move to, and how, in subsequent times.

So we have to specify the initial separation between the particles. How about a simple choice like this?

$$\begin{aligned} \mathbf{r} &= \{x, y, z\} = \{1, 0, 0\} \\ \mathbf{v} &= \{v_x, v_y, v_z\} = \{0, 0.5, 0\} \end{aligned} \qquad (5.2)$$

Dan: Let me put this into words. The relative position vector is chosen along the x axis at a distance of one from the origin. The origin is the place where the other particle resides, and it is the origin of the relative coordinate system that we use. And the relative velocity is chosen to be 0.5 in the direction of the positive y axis. This means that the initial motion is at right angles to the initial separation.

Carol: Would it not be easier to use a position of $\mathbf{r} = \{1, 0\}$ and a velocity of $\mathbf{v} = \{0, 0.5\}$, in other words, to work in two dimensions?

Erica: Well, as soon as we will do anything connected with the real universe, we will have to go to 3D, so why not just start there, even though the two-body problem is essential a 2D problem.

Dan: I don't care, either way, let's just move on. We have to translate what Erica has just written into computer code. If it were Fortran, I would start writing the first line as

```
x = 1
```

Erica: That is how you would do it in C or C++ as well, although you first would have to declare the type of `x`, by specifying that it is a number, in our case a floating point number, even though we initialize it here with an integer. The meaning of the equal sign, `=`, can be interpreted as follows: the value of the right-hand side of the equation gets assigned to the variable that appears at the left-hand side. In this case, the value of the right-hand side is already clear, it is just 1, and after execution of this statement, the variable `x` has acquired the value 1.

Carol: In Ruby you do the same as in Fortran or C or C++, or Java for that matter. In general, Ruby is designed around the 'principle of least surprise.' If you have some experience with computer languages, you will find that whenever you encounter something new in Ruby, it is not too far from what you might have guessed.

Erica: So assignment is exactly the same as in C?

Carol: The assignment itself is exactly the same, but there is no need to declare anything.

Erica: Really? How does Ruby now that the variable `x` can hold a floating point number and not, say, a character string or an array or whatever?

Carol: In Ruby there is no need for variable declaration, simply because there is nothing to declare: variables have no intrinsic type. The type of a variable is whatever you assign to it. If you write `x = 3.14`, `x` becomes a floating point number; and if you then write `x = "abs"`, it becomes a string. This freedom and flexibility is expressed by saying that Ruby is a dynamically typed language.

Erica: Isn't that dangerous?

Carol: I expected it would be, but in my experience, I in fact made fewer mistakes using Ruby than using so-called strongly typed languages, such as C and C++. Or stated more precisely: what mistakes I made, I could catch far more quickly, since it would be rather obvious if you assign `pi = "abc"` and then try to compute `2*pi*r` and the like: you would get an error message telling you that a string cannot be forced into a number.

Erica: so this means that we can just list the six assignments for $\mathbf{r} = \{1, 0, 0\}$ and $\mathbf{v} = \{0, 0.5, 0\}$? Like one line for each assignment?

Carol: You can write several assignments on one line, separated by semicolons, but I prefer to keep it simple and do it one per line. Here they are:

```
x = 1
y = 0
z = 0
vx = 0
vy = 0.5
vz = 0
```

By the way, does this specific choice of initial conditions mean that the two particles will move around each other in a circle?

Erica: Probably not, unless we give exactly the right velocity needed for circular motion. In general, the orbit will take the shape of an ellipse, if the two particles are bound. If the initial speed is too high, the particles escape from each other, in a parabolic or hyperbolic orbit.

Dan: Starting from the initial conditions, we have to step forward in time. I have no idea how large the time step step `dt` should be.

Carol: But at least we can argue that it should not be too large. The distance `dr` over which the particles travel during a time step `dt` must be very small compared to the separation between the two particles:

$$dr = v\ dt \ll r \tag{5.3}$$

With $v = 1$ and $r = 1$, this means $dt \ll 1$.

Dan: In that case, we could take 'much less than 1' to mean 0.1, for starters.

Carol: I would prefer an even smaller value. Looking at fig. (5.1) we see how quickly the forward Euler scheme flies off the tracks, so to speak. How about letting 'much less than 1' be 0.01? We can always make it larger later:

```
dt = 0.01
```

5.4 Looping in Ruby

Erica: We now know where we start in space, and with what velocity. We also know the size of each time step. All we have to do is start taking steps.

Dan: With a tiny time step of $dt = 0.01$, we'll have to take at least a hundred steps to see much happening, I guess. And to go a bit further, say from time $t = 0$ to time $t = 10$, we will need a thousand steps.

Erica: That means we have to construct a loop. Something like 'for $i = 1$ till $i = 1000$ do something.' At least that is how most computer languages express it. I wonder how ruby does it.

Carol: Let's have a look at the Ruby book. How do you repeat the same thing `k` times? Ah, here it is. That looks funny! You write `k.times`! So to traverse a piece of code 1000 times, you enclose that piece of code within the following loop wrapper:

```
1000.times{
}
```

Dan: Surely you are joking! That is carrying the principle of least surprise a bit too far to believe. How can that work? Can a computer language really stay so close to English?

Carol: The key seems to be that Ruby is an object-oriented language. Each 'thing' in ruby is an object, which can have attributes such as perhaps internal data or internal methods, which may or may not be visible from the outside.

Dan: What is a method?

Carol: In Ruby, the word *method* is used for a piece of code that can be called from another place in a longer code. In Fortran, you call that a *subroutine*, while in C and C++ you call it a *function.* In Ruby, it is called a *method.*

Erica: I have heard the term 'object-oriented programming.' I really should browse through the Ruby book, to get a bit more familiar with that approach.

Dan: We all should. But for now, Carol, how does your key work? Is the number 1000 also an object?

Carol: You guessed it! And every number has by default various methods associated with it. One method happens to be called `times`.

Erica: And what `times` does is repeat the content of its argument, whatever is within the curly brackets, `k` times, if the number is `k`.

Carol: Precisely. A bunch of expressions between curly brackets is called a block in Ruby, and this block is executed `k` times. We will have to get used to the main concepts of Ruby as we go along, but if you want to read about Ruby in more systematic way, here is a *a good place to start*[1], and here is a web site specifically aimed at *scientific applications*[2].

[1]`http://www.rubycentral.com/`
[2]`http://sciruby.codeforpeople.com/`

5.5 Interactive Ruby: `irb`

Dan: Amazing. Well, I would be even more amazed to see this work.

Carol: Let's test it out, using `irb`. This is an interactive program that allows you to test little snippets of Ruby code. Let us explore what it can do for us. You can invoke it simply by typing its name:

```
|gravity> irb
quit
```

and you can get out at any time by typing `quit`.

Dan: I like your prompt!

Carol: Well, I called my computer 'gravity', and I set up my shell to echo the name of my computer, so that's why it shows up here.

Dan: Quite appropriate. Now how do we interact with `irb`?

It seems that we can now type any Ruby expression, which will then be evaluated right away. Let me try something simple:

```
|gravity> irb
2 + 3
5
quit
```

Erica: How about going from arithmetic to algebra, by using some variables?

```
|gravity> irb
a = 4
4
b = 5
5
c = a * b
20
quit
```

I see. At the end of each line, the result of the operation is echoed. And the value 20 is now assigned to the variable `c`.

Carol: Indeed. Time to test Ruby's looping construct:

```
|gravity> irb
c = 20
20
3.times{ c += 1 }
3
c
23
quit
```

Perfect! We started with 20 and three times we added 1.

Dan: ah, so `c += 1` is the same as `c = c + 1`?

Carol: Yes. This is a construction used in C, and since taken over by various other languages. It applies for many operators, not only addition. For example, `c *= d` is the same as `c = c * d`.

5.6 Compiled vs. Interpreted vs. Interactive

Erica: This `irb` program is quite useful clearly, but I'm puzzled about the various ways in which we can use Ruby. We are now writing a Ruby program, by adding lines to a file, in just the same way we would be writing a C or Fortran program, yes?

Carol: Yes and no. Yes, it looks the same, but no, it really is a quite different approach. Ruby is an interpreted language, while C and Fortran, and C++ as well, are all compiled languages. In Ruby, there is no need to compile a source code file; you can just run it directly, as we will see soon.

This, by the way, is why Ruby is called a scripting language, like Perl and Python. In all three cases, whatever you write can be run right away, just like a shell script. As soon as we have finished writing our program, we will run it with the command `ruby` in front of the file name, in the same way as you would run a cshell script by typing `csh filename`. In our case we will type `ruby euler_try.rb`.

Erica: So the difference is that in C you first compile each piece of source code into object modules, and then you link those modules into a single executable file, and then you run that file – whereas in Ruby the script itself is executable.

Carol: Exactly.

Erica: But what is the difference between typing `ruby` and typing `irb`? If the `ruby` command interprets each line as it goes along, what does `irb` add?

Carol: The difference is the `i` in `irb`, which stands for *interactive*. In the case of `irb`, each line is not only interpreted, it is also evaluated and the result is printed on the screen. In this way, you can look into the mind of the interpreter, so to speak, and you can follow step by step what is going on.

Dan: It sounds a bit like going into a debug mode.

Carol: I guess you could say that, yes. However, if you run a Ruby script using the command `ruby`, you only get results on the screen when you give a specific print command, such as `print`, as we will see.

And just to give full disclosure, there is another hitch. If you are starting to write a loop, say, you may have included an open parenthesis, but not yet a closing parenthesis. In `irb` that is no problem; in fact, the prompt will change, to indicate that you are one or more levels deep inside nested expressions. But if you try to run such an incomplete file with the `ruby` command, you will get an error message, even before the Ruby interpreter starts.

Here is what happens. Upon typing `ruby some-file.rb`, a syntax check is being carried out on the file `some-file.rb`. If the parentheses are not balanced, a syntax error is produced, and the real interpreter part of Ruby is not even started up.

Dan: Just like what happens in Fortran, when you get a compile error!

Carol: Yes, in a way. And to make things more confusing, many people tend to call such an error a 'compile error', even when working with Ruby, even though in Ruby the code is not really compiled, strictly speaking. The problem is that so-called compile errors in compiled languages are really syntax errors; and interpreted languages can of course have syntax errors as well. So when you hear someone telling you 'my Ruby (or Perl and Python) program didn't compile,' they mean 'my script had syntax errors.' However, strictly speaking, Ruby initially parses the input program and transform it to a tree structure, and then the interpreter actually traces this tree structure, not the text string itself. So it is not entirely incorrect to say that ruby first "compiles" a program. But this is probably more than what you would want to know at this point.

5.7 One Step at a Time

Erica: All that is left for us to do is to write the content of the loop. That means we have to describe how to take a single step forward in time.

Specifically, at the start we have to tell the particles how to move to their next relative position, from our starting point of time $t = 0$ to $t = 0.01$, in our case. Or for a general `dt` value, using the forward Euler approximation (5.1), we obtain the position $\mathbf{r} = \{r_x, r_y, r_z\}$ at the end of the first step:

```
  x += vx*dt
  y += vy*dt
  z += vz*dt
```

In addition, we have to tell the particles what their new relative velocity $\mathbf{v} = \{v_x, v_y, v_z\}$ should be. Using the same forward Euler construction, we can write:

```
    vx += ax*dt
    vy += ax*dt
    vz += az*dt
```

using the acceleration vector $\mathbf{a} = \{a_x, a_y, a_z\}$.

Dan: But we haven't calculated the acceleration `a` yet!

Carol: This is an important example of code writing, something called 'wishful thinking' or 'wishful programming'. You start writing a code as if you already have the main ingredients in hand, and then you go back and fill in the commands needed to compute those ingredients.

Dan: That sounds like a top-down programming approach, which makes sense: I like to start with an overview of what needs to be done. It is all too easy to start from the bottom up, only to get lost while trying to put all the pieces together.

Erica: To compute the acceleration, we have to solve the differential equation for the Newtonian two-body problem, Eq. (4.24). I will copy it here again:

$$\mathbf{a} = -\frac{\mathbf{r}}{r^3} \tag{5.4}$$

Dan: Can you write it out in component notation?

Erica: Sure:

$$\begin{aligned} a_x &= -\frac{x}{r^3} \\ a_y &= -\frac{y}{r^3} \\ a_z &= -\frac{z}{r^3} \end{aligned} \tag{5.5}$$

where the magnitude of the separation r is defined as

$$r = \sqrt{x^2 + y^2 + z^2} \tag{5.6}$$

Let me start with the last line. Since we will often need the square of the scalar distance between the particles, we may as well first compute $r^2 = x^2 + y^2 + z^2$:

```
    r2 = x*x + y*y + z*z
```

Carol: Let's see whether I remember my vector analysis class. The quantity $\mathbf{r}$ is called a vector, and the quantity r is called a scalar, right?

Erica: Indeed. The last quantity is a scalar because it is independent of your choice of coordinate system. If we rotate out coordinates, the values of x and of y and of z may all change, and therefore $\mathbf{r}$ will change. However, r will stay the same, and that is a good thing: it denotes the physical distance between the particles, something that you can measure. When two people use two different coordinate systems, and both measure r, the value they find had better be the same.

Dan: My Ruby book tells me that you must add the line

```
include Math
```

in order to use the square root method `sqrt`, where the term method is used in the same way the word function is used in C and the word subroutine is used in Fortran. The `include` statement is needed to gain access to the `Math` module in Ruby, where many of the mathematical methods reside.

Erica: Thanks! Now the rest is straightforward. To code up Eq. (5.5), we first need to determine r^3, and a simple way to do that is to write it as a product of two express we have already found: $r^3 = r^2 r$:

```
  r3 = r2 * sqrt(r2)
  ax = - x / r3
  ay = - y / r3
  az = - z / r3
```

5.8 Printing the Result

Dan: Shall we see whether the program works, so far? Let's run it!

Erica: Small point, but . . . perhaps we should add a print statement, to get the result on the screen?

Carol: I guess that doesn't hurt! The Ruby syntax for printing is very intuitive, following the Ruby notion of the 'principle of least surprise':

```
  print(x, "  ", y, "  ", z, "  ")
  print(vx, "  ", vy, "  ", vz, "\n")
```

Erica: I like that principle! And indeed, this couldn't be simpler!

Dan: Apart from this mysterious `\n` at the end. What does that do?

Carol: It prints a new line. This notation is borrowed from the C language. By the way, I'd like to see a printout of the position and velocity at the start of the run as well, before we enter the loop, so that we get all the points, from start to finish.

Erica: Fine! Here it is, our first program, `euler_try.rb`, which is supposed to evolve our two-body problem for ten time units, from `t = 0` till `t = 10`:

```
include Math

x = 1
y = 0
z = 0
vx = 0
vy = 0.5
vz = 0
dt = 0.01

print(x, "  ", y, "  ", z, "  ")
print(vx, "  ", vy, "  ", vz, "\n")

1000.times{
  r2 = x*x + y*y + z*z
  r3 = r2 * sqrt(r2)
  ax = - x / r3
  ay = - y / r3
  az = - z / r3
  x += vx*dt
  y += vy*dt
  z += vz*dt
  vx += ax*dt
  vy += ax*dt
  vz += az*dt
  print(x, "  ", y, "  ", z, "  ")
  print(vx, "  ", vy, "  ", vz, "\n")
}
```

Chapter 6

Running the Code

6.1 A Surprise

Carol: Well, let's see what happens. I don't want to look at a thousand lines of output. I will first run the code, redirecting the results into an output file, called `euler_try.out`:

```
|gravity> ruby euler_try.rb > euler_try.out
```

In that way, we can look at our leisure at the beginning and at the end of the output file, while skipping the 991 lines in between times 0, 0.01, 0.02, 0.03, 0.04 . . . 9.96, 9.97, 9.98, 9.99, 10.

```
|gravity> head -5 euler_try.out
1  0  0  0  0.5  0
1.0  0.005  0.0  -0.01  0.49  0.0
0.9999  0.0099  0.0  -0.0199996250117184  0.480000374988282  0.0
0.999700003749883  0.0147000037498828  0.0  -0.0300001547537506  0.469999845246249  0.0
0.999400002202345  0.0194000022023453  0.0  -0.0400029130125063  0.459997086987494  0.0
|gravity> tail -5 euler_try.out
-19.9935403671885  -16.0135403671885  0.0  -2.24436240147761  -1.74436240147762  0.0
-20.0159839912033  -16.0309839912033  0.0  -2.24435050659793  -1.74435050659793  0.0
-20.0384274962693  -16.0484274962693  0.0  -2.24433863791641  -1.74433863791641  0.0
-20.0608708826484  -16.0658708826485  0.0  -2.24432679534644  -1.74432679534644  0.0
-20.0833141506019  -16.0833141506019  0.0  -2.2443149788018  -1.74431497880181  0.0
```

Dan: A lot of numbers. Now what? We'd better make a picture of the results, to see whether these numbers make sense or not. Let's plot the orbit.

Erica: I agree, we should do that soon. But hey, the numbers do tell us something already, they tell us that there is something seriously wrong!

Carol: How can you tell?

Erica: At the end of the run, the distance between the two particles is more than 25 in our units, as you can see by applying Pythagoras to the last numbers in the first two columns: $\sqrt{20^2 + 16^2} = 25$.

Dan: So what?

Erica: A bit large already for my taste, but what clinches it is the velocity difference between the particles, which is more than $\sqrt{2.2^2 + 1.7^2} \approx 2.8$.

Dan: So what?

Erica: We started out with a velocity difference of only 0.5, so we have increased the velocity by more than a factor of more than 5, while increasing the distance by a factor of more than 25. When two particles move away from each other, they should slow down, not speed up, because gravity is an attractive force.

Carol: I see, yes, that is strange.

Dan: Even more reason to make a plot!

Carol: How about using `gnuplot`? That one is present on any system running Linux, and something that can be easily installed on many other systems as well. The style is not particularly pretty, but at least it will give us something to look at.

Dan: How do you invoke `gnuplot`?

Carol: To use it is quite simple, with only one command needed to plot a graph. In our case, however, I'll start with the command `set size ratio -1`. A positive value for the size ratio scales the aspect ratio of the vertical and horizontal edge of the box in which a figure appears. But in our case we want to set the scales so that the unit has the same length on both the x and y axes. Gnuplot can be instructed to do so by specifying the ratio to be `-1`. In fact, you can write the line `set size ratio -1` in a file called `.gnuplot` in your home directory, if you want to avoid repeating yourself each time you use gnuplot. But for starters, I'll give the command specifically.

The next command we need to use is `plot <filename>` which by default will plot the data from the first two columns from the file `filename`. And of course, you can specify other columns to be used, if you prefer. However, in our case, the first two columns just happen to contain the `x` and `y` values of the positions, so there is no need to give any further specifications.

Now let's have our picture, in fig 6.1:

```
|gravity> gnuplot
gnuplot> set size ratio -1
gnuplot> plot "euler_try.out"
```

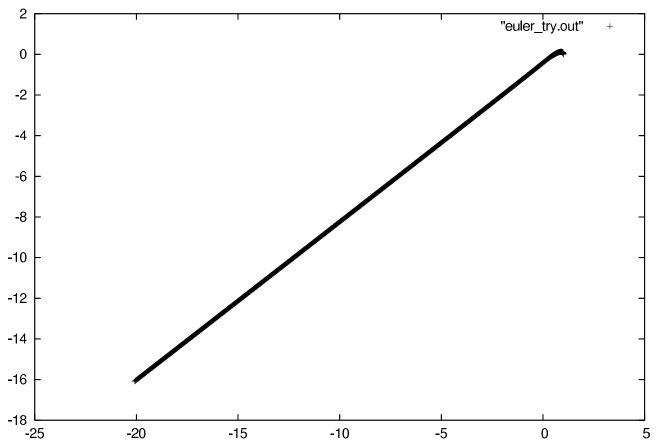

Figure 6.1: First attempt at integrating the two-body problem: failure.

```
gnuplot> quit
```

6.2 Too Much, Too Soon

Dan: Hmmm, that is not what I expected to see. What a disappointment!

Erica: Well, research is like that – the first time you do something, it almost never works.

Carol: Good thing you called the program `euler_try.rb`!

Dan: It seems as if the system exploded. Why would the two particles fly apart like that?

Erica: That's what we have to find out. And we'd better be systematic.

Dan: How will we ever find out what is the case? Shall we look at the code, line by line, to see whether we made a mistake? It is such a short code, there are not that many ways to do something wrong!

Carol: That's not the right approach. If you are starting from the wrong assumptions, just looking at the code will not help you to realize what was wrong with your thinking, no matter how long you stare at it.

Dan: Research *is* difficult! If this would be an exercise out of a book, at least the answer would be in the back, or we could ask a teaching assistant . . .

Erica: Yes, research is difficult, but it also is a lot more fun than chewing on home work assignments. You know, when you start playing in your own way, very soon you start doing things that in that exact form nobody else has ever done before. Isn't that a thrill?!

Dan: It would be a thrill if we could make progress. Frankly, I'm lost.

Carol: I must admit, I don't see a clear way ahead either, but at least I remember that one of my teachers told us to 'divide and conquer' while troubleshooting. In other words, if something goes wrong in a complex situation, try to simplify everything by dividing whatever procedure you have applied in smaller, more modular steps. That way, you can try to see in which step something goes wrong.

Erica: That makes sense. And I remember hearing a graduate student tell us, while we were struggling with a computer program: 'simplify, simplify.' The idea was to first look at the simplest possible parameter choice, because in simpler cases it is often easier to see what goes wrong.

Dan: You mean that we have done too much, too soon, by taking a rather arbitrary choice of initial conditions, and a thousand steps?

Erica: Exactly. The notion of 'divide and conquer' tells us that we'd better do one integration step at a time, instead of a thousand. And the idea of 'simplify, simplify' suggests that we start with a circular orbit, rather than the more general case of an elliptic orbit.

6.3 A Circular Orbit

Carol: So we have to find out what the correct velocity is, for two particles at a distance of 1, in order to move in a circle. It seems to be larger than 0.5, but how much larger?

Dan: Large enough that the particles don't fall toward each other but not so large that they start moving away from each other. Hmm. How can we picture that? Imagine that we would move the two particles in a circular orbit around each other, and measure how much force we have to use to keep them in the circular orbit. We could then require gravity to do the work for us, and insist that the gravitational force would be just equal to the force that we would have to apply by hand.

Erica: Or rather that letting gravity provide the right force, it is easier to compare accelerations, rather than forces. Let us insist on gravity providing the right acceleration.

For the equivalent one-body problem, in our choice of units, the the gravitational acceleration is given in Eq. (4.24). Since we are only interested in the magnitude, we can write it as:

$$a_{grav} = \frac{1}{r^2} \tag{6.1}$$

The acceleration that a particle feels, when being forced to move exactly in a circular orbit is simply given by:

$$a_{circ} = \frac{v^2}{r} \tag{6.2}$$

Dan: What do you mean 'simply given', how do you know?

Erica: Oh, I just remember, it is one of the standard equations I learned in classical mechanics.

Dan: Well, I don't remember, and while I'm sort-of happy to take your word for it, I would be much happier to see whether we can derive it, so that we know for sure we have the right expression.

Carol: Me too, I'm with Dan here.

Erica: Well, hmmm, I suppose we can go to the library and look it up in a text book on classical mechanics. Any textbook should tell you how to derive that expression. Frankly, I don't remember now how we did it.

Dan: It would be much faster to look it up on Google. But of course, then you have to wonder whether it was done correctly or not.

Carol: Come on, it can't be *that* hard. And it is much more fun to derive it ourselves rather than look it up. No Dan, I don't even want to look at Google. Here, let's take a piece of paper, and derive both the first and the second derivatives of the scalar distance r between the two particles. When we force both derivatives to be zero, we now that r will remain constant forever, since equations of motion are second-order differential equations.

Dan: Well, before I ask what you mean, let me first see what you do.

6.4 Radial Acceleration

Carol: We start with the definition of r as the absolute value or, if you like, the length of the vector $\mathbf{r}$:

$$r = (\mathbf{r}\cdot\mathbf{r})^{1/2} \tag{6.3}$$

I will now determine its first time derivative:

$$\frac{d}{dt}r = \tfrac{1}{2}(\mathbf{r}\cdot\mathbf{r})^{-1/2}\ \frac{d}{dt}(\mathbf{r}\cdot\mathbf{r}) = \tfrac{1}{2}\ \frac{1}{r}\ 2\mathbf{r}\cdot\frac{d}{dt}\mathbf{r} = \frac{\mathbf{r}\cdot\mathbf{v}}{r} \tag{6.4}$$

On a circular orbit, the distance between the particles is supposed to remain constant, which means $dr/dt = 0$, and the only way to guarantee this, according to the equation I just derived, is to insist that the vectors $\mathbf{r}$ and $\mathbf{v}$ are perpendicular, so that $\mathbf{r}\cdot\mathbf{v} = 0$.

Erica: That makes sense: on a circular orbit the velocity has no component in the direction toward the other particle, so it is indeed perpendicular.

Dan: There is something I don't understand. In the equation above, you start with the expression dr/dt. But isn't that the velocity? If you insist that $dr/dt = 0$, aren't you telling us that the velocity is zero? But in that case the two particles would start falling toward each other, the next moment!

Carol: Which they don't. You are confused with the expression $v = |\mathbf{v}| = |d\mathbf{r}/dt|$ which is the absolute value of the velocity, and it is a very different beast than what I just wrote down. So it is important to realize that, yes, in a one-dimensional situation you can write

$$\frac{dr}{dt} = v \qquad [\ 1D\ ] \tag{6.5}$$

but in a two-dimensional or three-dimensional situation this is no longer true in general; in a typical situation we have

$$\frac{dr}{dt} \neq v \qquad [\ kD\ ,\ k > 1\ ] \tag{6.6}$$

Dan: Hmmm. Vector analysis is tricky.

Carol: Until you get used to it.

Dan: Well, that's true for everything.

Carol: Fair enough. Okay, onward to the second derivative of the separation between the two particles:

$$\begin{aligned}
\frac{d^2}{dt^2}r &= \frac{1}{r}\frac{d}{dt}(\mathbf{r}\cdot\mathbf{v}) + (\mathbf{r}\cdot\mathbf{v})\frac{d}{dt}\frac{1}{r} \\
&= \frac{1}{r}\left(\frac{d\mathbf{r}}{dt}\cdot\mathbf{v}\right) + \frac{1}{r}\left(\mathbf{r}\cdot\frac{d\mathbf{v}}{dt}\right) + (\mathbf{r}\cdot\mathbf{v})\left(-\frac{1}{r^2}\right)\frac{d}{dt}r \\
&= \frac{1}{r}\mathbf{v}\cdot\mathbf{v} + \frac{1}{r}\mathbf{r}\cdot\mathbf{a} - \frac{(\mathbf{r}\cdot\mathbf{v})^2}{r^3} \\
&= \frac{v^2}{r} - a
\end{aligned} \tag{6.7}$$

At the end of the second line I substituted the result of Eq. (6.4), and at the end of the third line, I used the fact that the position and velocity vector are perpendicular to each other, as we had just derived above. I also used the fact that the acceleration vector **a** points in the opposite direction of the separation vector **r**, which means that $\mathbf{r} \cdot \mathbf{a} = -ra$.

For a circular orbit, we must insist that the separation r between the particles remains constant. This means that the time derivative $dr/dt = 0$, and of course the same holds for the second derivative in time, $d^2r/dt^2 = 0$. And there we are, Eq. (6.7) then gives us:

$$a = \frac{v^2}{r} \tag{6.8}$$

Dan: Wow, that is exactly the acceleration that Erica remembered, needed to sustain a circular motion.

Erica: Neat! Satisfied, Dan?

Dan: Sure thing!

Carol: Let's see why we did all this. Ah, we wanted to balance the gravitational acceleration provided and the acceleration needed to keep a motion being nicely circular. We already found that $a_{grav} = 1/r^2$, so this means:

$$\frac{v^2}{r} = \frac{1}{r^2} \tag{6.9}$$

or simply:

$$v^2 = \frac{1}{r} \tag{6.10}$$

or equivalently

$$v = r^{-1/2} \tag{6.11}$$

In our first attempt at orbit integration, we started with an initial condition $\mathbf{r} = \{1, 0, 0\}$ which implies $r = 1$, but we used an initial velocity of $\mathbf{v} = \{0, 0.5, 0\}$ which means that $v = 1/4$, much too small a value for a circular orbit! It should have been $v = 1$, according to what we just derived.

Dan: Ah, so we should have used $\mathbf{v} = \{0, 1, 0\}$ for the initial velocity. Great! Good to know.

6.5 Virial Theorem

Erica: You know, while Carol was doing her virtuoso derivation act, I suddenly remembered that there is a much quicker way to derive the same result from scratch.

Carol: Show me! I find that hard to believe.

Erica: It just occurred to me that I could use the virial theorem, which tells us that for any bound system, on average the potential energy is equal to minus twice the kinetic energy in the c.o.m. frame. For a circular orbit, both the potential and kinetic energy remain constant, so we don't even have to do any averaging.

In our case, we can use Eqs. (4.4) and (4.6) to write the kinetic energy as:

$$\begin{aligned} E_{kin} &= \tfrac{1}{2}M_1 v_1^2 + \tfrac{1}{2}M_2 v_2^2 \\ &= \tfrac{1}{2}M_1 \left(\frac{M_2}{M_1+M_2}\,\mathbf{v}\right)^2 + \tfrac{1}{2}M_2 \left(\frac{M_1}{M_1+M_2}\,\mathbf{v}\right)^2 \\ &= \tfrac{1}{2}\,\frac{M_1M_2}{(M_1+M_2)^2}\left[M_2+M_1\right]v^2 \\ &= \tfrac{1}{2}\,\frac{M_1M_2}{M_1+M_2}\,v^2 \end{aligned} \tag{6.12}$$

The potential energy is simply:

$$E_{pot} = -\,G\,\frac{M_1M_2}{r} \tag{6.13}$$

The virial theorem tells us that $E_{pot} = -2E_{kin}$, which gives us:

$$\frac{M_1M_2}{M_1+M_2}\,v^2 = G\,\frac{M_1M_2}{r} \tag{6.14}$$

or:

$$v^2 = G\,\frac{M_1+M_2}{r} \tag{6.15}$$

In our units, $G = M_1 + M_2 = 1$ and therefore we have:

$$v = \sqrt{\frac{1}{r}} \tag{6.16}$$

So here you are: for an initial separation of 1, we need an initial velocity of 1.

Carol: I must admit, you got the right answer and your derivation is a bit simpler than the one I just gave. But I have never heard of the virial theorem. What does it mean?

Erica: Weeeeellll, that's quite a long story. I'm not sure whether we should go into that right now. If you really want to know, you can look at a text book, but . . .

Dan: . . . Google gives me a whole bunch of sites. Let's look at a few. Hmmmm. A bit too much math, this one. . . Ah, this one looks easier, with more words and simple examples . . .

Carol: So we know we can look it up when we have to. I agree with Erica, I'd rather move on.

6.6 Circular Motion

Dan: Wait a minute, each of you have just given a detailed derivation, and now you're suddenly in a hurry. You know what? I bet that I can give an even simpler derivation, and that without using complicated vector calculus or the vitrial theorem.

Erica: *virial* theorem.

Dan: Whatever. Here is my suggestion. Why not just write down the circular orbit itself, as if we had already derived it? I don't remember much from my introductory physics class, but I do remember how neat it was that you could write down a simple circular motion in two dimensions in the following way, for the position:

$$\begin{cases} x = A\cos(\omega t) \\ \\ y = A\sin(\omega t) \end{cases} \tag{6.17}$$

Now this is easy to differentiate. No vector notation, just simple coordinate operations. By differentiation with respect to time, the velocity vectors become:

$$\begin{cases} v_x = -A\omega\sin(\omega t) \\ \\ v_y = A\omega\cos(\omega t) \end{cases} \tag{6.18}$$

One more differentiation, and we get the acceleration components:

$$\begin{cases} a_x = -A\omega^2\cos(\omega t) \\ \\ a_y = -A\omega^2\sin(\omega t) \end{cases} \tag{6.19}$$

Comparing Eqs. (6.17) and (6.19), we find

$$\mathbf{a} = -\omega^2\mathbf{r} \tag{6.20}$$

We have seen in Eq.(6.1) that for our initial condition $r = 1$ we have $a = 1$, so this means that $\omega = 1$. Well, Eq. (6.18) now tells us that $v = 1$. Isn't that simple?

Erica: Yes, it is *very* simple, I'm surprised!

Dan: I must admit that I'm a bit surprised too, that it came out so easily. And, frankly, I'm surprised that I came out correctly!

Carol: But working in coordinates like that is not very elegant.

Erica: Oh, come on, Carol, give the guy a break! What counts is to get the right answer, and you must admit that his solution is simpler than either of our ways of deriving the same answer. Let's just be glad that all three methods gave the same answer!

Carol: Ah, you physicists, you're so pragmatic! I'd prefer a bit more style.

Dan: Well, each her own style. I'm happy now, and ready to move on!

6.7 One Step at a Time

Erica: Which means that we've answered the 'simplify, simplify' part of our task of trouble shooting: we now know how to launch the two-body problem on the simplest possible orbit, that of a circle.

The other task was 'divide and conquer', and we had already decided to start with just one step.

Dan: That's a simple change in our program: we can just take out the loop.

Carol: Okay, here is the new code. Let me call it `euler_try_circular_step.rb`.

Dan: You sure like long names! I would have called it `euler_trycs.rb`.

Carol: Right. And three days later you will be wondering why there is a program floating around in your directory that seems to tell you that it uses Euler's algorithm for trying out cool stuff, or for experimenting with communist socialism or for engaging in some casual sin. No, I'm a big believer in looooong names.

Erica: I used to be like Dan, but I've been bitten too often by the problem you just mentioned, that I could for the life of me not remember what the acronym was supposed to mean that I had introduced. So yes, I'm with you.

Dan: Fine, two against one, I lose again! But I'll be `gs`, oops, I mean a `good_sport`.

Carol: What do you think of this version of `euler_try_circular_step.rb`?

```
include Math

x = 1
y = 0
z = 0
vx = 0
```

```
vy = 1
vz = 0
dt = 0.01

print(x, "  ", y, "  ", z, "  ")
print(vx, "  ", vy, "  ", vz, "\n")

r2 = x*x + y*y + z*z
r3 = r2 * sqrt(r2)
ax = - x / r3
ay = - y / r3
az = - z / r3
x += vx*dt
y += vy*dt
z += vz*dt
vx += ax*dt
vy += ax*dt
vz += az*dt

print(x, "  ", y, "  ", z, "  ")
print(vx, "  ", vy, "  ", vz, "\n")
```

Erica: Let's see: you got the circular velocity correct, a value of unity as it should be. And instead of looping, you print, take one step, and print again. Hard to argue with!

Dan: Let's see whether it gives a reasonable result.

Chapter 7

Debugging the Code

7.1 One Integration Step: Verification

Carol: This is about the simplest thing we could possibly do, for the one-body problem: starting with a circular orbit, and then taking only one small step. Here we go . . .

```
|gravity> ruby euler_try_circular_step.rb
1  0  0  0  1  0
1.0  0.01  0.0  -0.01  0.99  0.0
```

Dan: . . . and getting just one short new line of output, after the initial conditions are echoed. Nice and simple!

Erica: Simple, yes, but correct? Let's compute the numbers by hand, to see whether our program gave the right answers. We start with

$$\mathbf{r}(0) = \{x, y, z\} = \{1, 0, 0\} \tag{7.1}$$

and

$$\mathbf{v}(0) = \{v_x, v_y, v_z\} = \{0, 1, 0\} \tag{7.2}$$

so the new position must be:

$$\mathbf{r}(dt) = \mathbf{r} + \mathbf{v}dt = \{1, dt, 0\} \tag{7.3}$$

and since we are using $dt = 0.01$, we expect:

$$\mathbf{r}(dt) = \mathbf{r} + \mathbf{v}dt = \{1, 0.01, 0\} \tag{7.4}$$

Carol: And this is indeed what we see in the first half of the second output line.

Dan: That is encouraging! Now what about the second half of the second output lines?

Erica: To compute the new velocity, we have to first compute the acceleration vector. We can use Eq. (4.24), which I'll copy here once again:

$$\mathbf{a} = -\frac{\mathbf{r}}{r^3} \tag{7.5}$$

In our case this gives

$$\mathbf{a} = \{a_x, a_y, a_z\} = \{-1, 0, 0\} \tag{7.6}$$

And this in turn means that

$$\mathbf{v}(dt) = \mathbf{v} + \mathbf{a}dt = \{-dt, 1, 0\} = \{-0.01, 1, 0\} \tag{7.7}$$

Carol: And this is *not* what is printed in the last half of the last output line, in our one-step program.

Dan: Spot on! We should have gotten $\{-0.01, 1, 0\}$, but we somehow wound up with $\{-0.01, 0.99, 0\}$. So that's were the bug is, in the y component of the new velocity, which should be $v_y(dt) = v_y + a_y dt$, but somehow isn't.

Erica: Easy to check: here is where we compute the new value of v_y, which we call `vy`:

```
vx += ax*dt
vy += ax*dt
vz += az*dt
```

Ooops!! A typo. How silly. No wonder we got the wrong answer for a_y. Let me correct it right away, and write it out as a new file, `euler_circular_step.rb`:

```
vx += ax*dt
vy += ay*dt
vz += az*dt
```

I'm curious to see whether now everything will be all right:

```
|gravity> ruby euler_circular_step.rb
1  0  0  0  1  0
1.0  0.01  0.0  -0.01  1.0  0.0
```

Dan: Wonderful! Now both the position and the velocity components are correct, after the first step. We are winning!

Carol: Yes, we have now verified that we got the right result after one step.

7.2 A Different Surprise

Dan: Great! Let's go back to our original code, correct the bug, and we'll be in business.

Carol: Only if the first bug we caught will be the last bug. Don't be so sure! We may well have made another mistake somewhere else.

Dan: Oh, Carol, you're too pessimistic. I bet everything will be fine now!

Erica: We'll see. Here is the same typo in `euler_try.rb`:

```
   vx += ax*dt
   vy += ax*dt
   vz += az*dt
```

and here is the corrected program, which I will call `euler.rb` in the spirit of Dan's optimism:

```
   vx += ax*dt
   vy += ay*dt
   vz += az*dt
```

I'll run the new code:

```
|gravity> ruby euler.rb > euler.out
```

And here is the plot, in fig 7.1:

```
|gravity> gnuplot
gnuplot> set size ratio -1
gnuplot> plot "euler.out"
gnuplot> quit
```

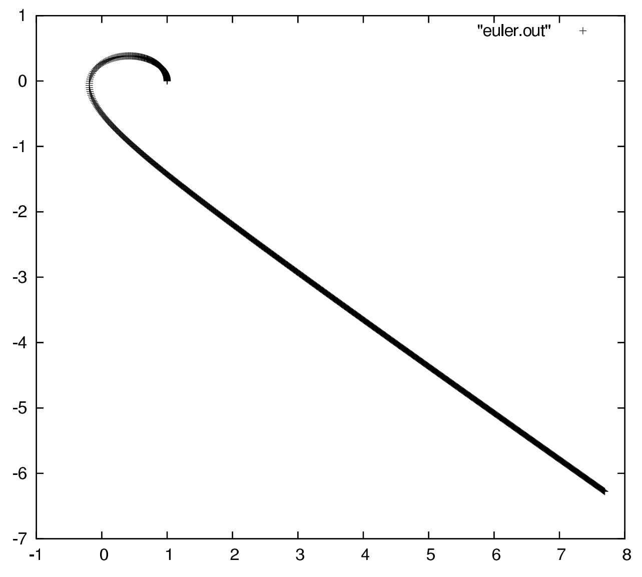

Figure 7.1: Second attempt at integrating the two-body problem: a different failure.

Carol: Well, Dan, what do you say?

Dan: No comment.

7.3 One Integration Step: Validation

Erica: Maybe we should go back to the circular orbit. We tried to take a single step there, and we found our typo. Perhaps we should take a few more steps, before returning to the more general problem.

Carol: I agree. Let us sum up what we've done with our one-step code. We have verified that our program does what the algorithm intended, and that is certainly nice! But it is only half of the work. We now have to check whether our particular algorithm does indeed give a reasonable result, which corresponds to the behavior of gravitating particles in the real world. This is called validation. In the computer science literature these two checks are often called V&V, for Verification and Validation.

In other words, so far in our one-step program we have passed the verification test. The computer code does exactly what we wanted it to do, at least for that one step. But now we have to do a validation test.

Dan: What does that mean, concretely?

Carol: For example, we can ask whether the first step keeps the two particles on a circular orbit. We can answer that question with pure thought. After the first step, the new separation is:

$$r(0.1) = \sqrt{x^2 + y^2 + z^2} = \sqrt{1 + 0.01^2} = \sqrt{1.0001} \approx 1.00005 \tag{7.8}$$

Dan: Instead of the correct value of $r(0.01) = 1$, we are half a one hundredth of one percent off. Not bad, I would say.

Carol: Not bad for one step, perhaps, but our orbit has a radius of unity, which means a circumference of $2\pi \approx 6.3$. With a velocity of unity, it will take 630 steps to go around the circle, at the rate we are going. And if every step introduces 'only' an error of 0.00005, and if the errors built up linearly, we wind up with a total error of $1 + 630 * 0.00005 \approx 1.03$. That is already a 3% error, even during the first revolution! And after just a few dozen revolutions, if not earlier, the results will be meaningless.

Dan: Good point. Of course, we don't know whether the errors build up linearly, but for lack of a better idea, that would be the first guess. Perhaps we should take an even smaller time step. What would happen if we would use $dt = 0.001$? Let's repeat your analysis. After one step, we would have

$$r(0.001) = \sqrt{x^2 + y^2 + z^2} = \sqrt{1 + 0.001^2} = \sqrt{1.000001} \approx 1.0000005 \tag{7.9}$$

We now need roughly 6300 steps to go around the circle, If the errors build up linearly, the radial separation will grow to something like $1+6300*0.0000005 \approx 1.003$. Aha! Only a 0.3% error, instead of 3%.

Erica: Bravo! You have just proved that the forward Euler scheme is a first-order scheme! Remember our discussion at the start? For a first-order scheme, the errors scale like the first power of the time step. You just showed that taking a time step that is ten times smaller leads to a ten times smaller error after completing one revolution.

Dan: Great! I thought that numerical analysis was a lot harder.

Carol: Believe me, it *is* a lot harder for any numerical integration scheme that is more complex than first-order. You'll see!

Dan: I can wait. For now I'm happy to work with a scheme which I can completely understand.

7.4 More Integration Steps

Carol: So were are we. To sum up: we have verified that our simple one-step code `euler_circular_step.rb` does exactly what we want it to do. And we have validated that what we want it to do is reasonable: for smaller and smaller time steps the orbit should stay closer and closer to the true circular orbit.

Dan: That's the good news. But at the same time, that's also the bad news! When we tried to integrate an arbitrary elliptical orbit, we got a nonsense picture. How come?

Erica: We'll have to work our way up to the more complicated situation. Let us stick to the circular orbit for now. We have a basis to start from: the first step was correct, that we know for sure. Let's do a few more steps.

Dan: Let's try a thousand steps again.

Carol: Better to do ten steps. Each time we tried to jump forward too quickly we've run into problems!

Erica: How about a hundred steps, as a compromise? Let's go back to the very first code we wrote, but now for a circular orbit, and for an integration of one time unit, which will give us a hundred steps.

Carol: Let's keep the old code, for comparison. Here is the new one. I will call it `euler_circular_100_steps.rb`:

```
include Math

x = 1
y = 0
z = 0
```

```
vx = 0
vy = 1
vz = 0
dt = 0.01

print(x, "  ", y, "  ", z, "  ")
print(vx, "  ", vy, "  ", vz, "\n")

100.times{
  r2 = x*x + y*y + z*z
  r3 = r2 * sqrt(r2)
  ax = - x / r3
  ay = - y / r3
  az = - z / r3
  x += vx*dt
  y += vy*dt
  z += vz*dt
  vx += ax*dt
  vy += ay*dt
  vz += az*dt
  print(x, "  ", y, "  ", z, "  ")
  print(vx, "  ", vy, "  ", vz, "\n")
}
```

Dan: If you keep making the names longer and longer, they won't fit on a single line anymore!

Carol: You do what you want and I do what I want; I just happen to sit behind the keyboard.

Erica: Peace, peace! Let's not fight about names; we can later make copies with shorter names as much as we like.

```
|gravity> ruby euler_circular_100_steps.rb > euler_circular_100_steps.out
```

Carol: Figure 7.2 may or may not be part of a good circle; hard to see when you only have a small slice.

Dan: Told you so!

Carol: No, you didn't! You didn't give any reason for returning to the old value.

Erica: Hey guys, don't get cranky. Let's just go back to our original choice of 1,000 steps.

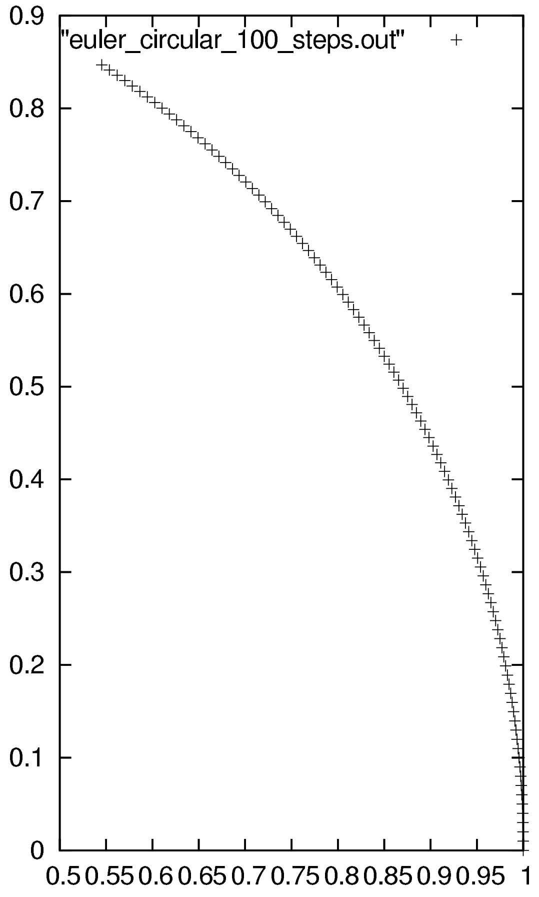

Figure 7.2: Third attempt at integrating the two-body problem: part of a circle?

7.5 Even More Integration Steps

Carol: In that case, let's make yet another new file . . . Dan, close your eyes, I'm adding one more character to the file name . . . `euler_circular_1000_steps.rb`:

```
include Math

x = 1
y = 0
z = 0
vx = 0
vy = 1
vz = 0
dt = 0.01

print(x, "  ", y, "  ", z, "  ")
print(vx, "  ", vy, "  ", vz, "\n")

1000.times{
  r2 = x*x + y*y + z*z
  r3 = r2 * sqrt(r2)
  ax = - x / r3
  ay = - y / r3
  az = - z / r3
  x += vx*dt
  y += vy*dt
  z += vz*dt
  vx += ax*dt
  vy += ay*dt
  vz += az*dt
  print(x, "  ", y, "  ", z, "  ")
  print(vx, "  ", vy, "  ", vz, "\n")
}
```

And here is our new result, which I'm calling figure 7.3:

```
|gravity> ruby euler_circular_1000_steps.rb > euler_circular_1000_steps.out
```

Much better!

Erica: Indeed, we're really make progress.

Dan: We've come around full circle – almost! At least the particles are *trying* to orbit each other in a circle, it seems. They're just making many small errors, that are piling up.

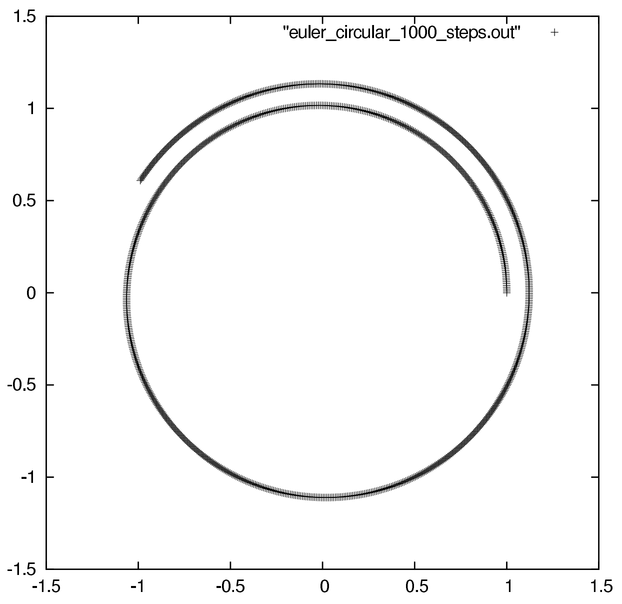

Figure 7.3: Fourth attempt at integrating the two-body problem: looking much better.

7.6 Printing Plots

Erica: Now that we're getting somewhat believable results, I would like to make some printouts of our best pictures. Carol, how do you get gnuplot to make some prints?

Carol: That's easy, once you know how to do it, but it is rather non-intuitive. The easiest way to find out how to do this is to go into gnuplot and then to type help, and to work your way down the information about options. To give you a hint, `set terminal` and `set output`. Of course, if you use gnuplot for the first time, you would have no way of guessing that *those* are the keywords you have to ask help for.

Erica: That's a general problem with software. Having a help facility is a good start, but I often find that I need a meta-help facility to find out how to find out how to ask the right questions to the help facility. In any case, I'm happy to explore gnuplot more, some day, but just for now, why don't you make a printout of the last figure we have just produced.

Carol: Okay, here is how it goes. I'm using abbreviations such as `post` for `postscript` and `q` for `quit`:

```
|gravity> gnuplot
gnuplot> plot "euler_circular_1000_steps.out"
gnuplot> set term post eps
Terminal type set to 'postscript'
Options are 'eps noenhanced monochrome dashed defaultplex "Helvetica" 14'
gnuplot> set output "euler_circular_1000_steps.ps"
gnuplot> replot
gnuplot> q
```

Let's print out this plot:

```
|gravity> lpr euler_circular_1000_steps.ps
```

Erica: Great, same figure. But hey, wait a minute, the symbol used for the points in the printed figure is very different from the symbol that appeared on the screen!

Carol: Welcome to the wonderful world of gnuplot. This is a strange quirk which is one of the things I really don't like about it. But as long as we use gnuplot, we have to live with it.

Chapter 8

Convergence for a Circular Orbit

8.1 Better Numbers

Carol: Yes, the orbit is looking recognizably circular, but that's about it. The errors are still quite large. I would like to know exactly how large they are. Let me have a peak at the numbers at the beginning and at the end of the run, just like we did in the elliptic case.

```
|gravity> ruby euler_circular_1000_steps.rb > euler_circular_1000_steps.out
|gravity> head -5 euler_circular_1000_steps.out
1  0  0  0  1  0
1.0  0.01  0.0  -0.01  1.0  0.0
0.9999  0.02  0.0  -0.0199985001874781  0.999900014998125  0.0
0.999700014998125  0.0299990001499813  0.0  -0.0299945010873182  0.999700074986127  0.0
0.999400069987252  0.0399960008998425  0.0  -0.0399870033746211  0.9994002199565  0.0
|gravity> tail -5 euler_circular_1000_steps.out
-0.967912821753728  0.639773430828751  0.0  -0.534369558098563  -0.764471400394899  0.0
-0.973256517334714  0.632128716824802  0.0  -0.528172455755398  -0.768567576556818  0.0
-0.978538241892268  0.624443041059234  0.0  -0.521945648518888  -0.772611878956261  0.0
-0.983757698377457  0.616716922269671  0.0  -0.515689587420237  -0.776604113138075  0.0
-0.988914594251659  0.608950881138291  0.0  -0.509404724445418  -0.780544088761249  0.0
```

Erica: The first few numbers give a distance of about 1 for the separation of the two particles, as it should be, but the last few numbers are too large. The separation along the x axis is about 1.0, and the separation along the y axis is about 0.6, and with Pythagoras that gives us a distance of $\sqrt{1 \cdot 1 + 0.6 \cdot 0.6} \approx$ 1.17. Not wildly off, but not very good either.

Carol: Remember, just after Eq.(7.8), how I estimated the error after one orbit to be 3%? After one and a third orbit it should then have been 4%, and we got 17%. Perhaps nonlinear effects contributed, but at least my original guess of several percent was not way off! And I remember that Dan showed what should happen for a time step that is ten times smaller: the error should shrink by a factor ten as well. I'd like to test that, by changing the line:

```
dt = 0.01
```

in `euler_circular_1000_steps.rb` to:

```
dt = 0.001
```

and put that code in `euler_circular_10000_steps.rb`, to indicate that we are now taking 1,000 steps per time unit, 10,000 time steps in total. Time to run it!

```
|gravity> ruby euler_circular_10000_steps.rb > euler_circular_10000_steps.out
|gravity> head -5 euler_circular_10000_steps.out
1  0  0  0  1  0
1.0  0.001  0.0  -0.001  1.0  0.0
0.999999  0.002  0.0  -0.00199999850000188  0.9999990000015  0.0
0.9999970000015  0.0029999990000015  0.0  -0.00299999450001088  0.9999970000075  0
0.999994000007  0.003999996000009  0.0  -0.00399998700003375  0.999994000022  0.0
|gravity> tail -5 euler_circular_10000_steps.out
0.544161847963306  0.839852844602723  0.0  -0.839087845804416  0.544480032616632
0.543322760117501  0.840397324635339  0.0  -0.839630814109985  0.54364202187099  0
0.542483129303391  0.84094096665721  0.0  -0.840172943245668  0.542803470813169  0
0.541642956360146  0.841483770128023  0.0  -0.840714232673602  0.541964380286331
0.540802242127472  0.84202573450831  0.0  -0.841254681856773  0.541124751134179  0
```

Dan: Those numbers are *very* different from the earlier ones . . .

8.2 Even Better Numbers

Erica: Ah, but of course! To come back to the same place, after making the time step ten times smaller, we have to take ten times as many steps!

Carol: Of course indeed! Okay, I'll change the line

```
1000.times{
```

in `euler_circular_10000_steps.rb` to:

```
10000.times{
```

and call that file `euler_circular_10000_steps_ok.rb`.

Dan: Your file names are growing again without bounds, and I don't like having so very many different files lying around.

Carol: As long as we're debugging, I'd prefer to have many files, so that we can always backtrack to earlier versions. We can clean up the mess later.

Dan: Okay, try again:

```
|gravity> ruby euler_circular_10000_steps_ok.rb > euler_circular_10000_steps_ok.out
|gravity> head -5 euler_circular_10000_steps_ok.out
1  0  0  0  1  0
1.0  0.001  0.0  -0.001  1.0  0.0
0.999999  0.002  0.0  -0.00199999850000188  0.9999990000015  0.0
0.9999970000015  0.0029999990000015  0.0  -0.00299999450001088  0.9999970000075  0.0
0.999994000007  0.003999996000009  0.0  -0.00399998700003375  0.999994000022  0.0
|gravity> tail -5 euler_circular_10000_steps_ok.out
-0.926888921537776  -0.42643167863865  0.0  0.411096610555956  -0.900278962470303  0.0
-0.92647782492722  -0.42733195760112  0.0  0.411969325107482  -0.899877454670898  0.0
-0.926065855602113  -0.428231835055791  0.0  0.412841644152271  -0.899475103103453  0.0
-0.925653013957961  -0.429131310158894  0.0  0.413713566877832  -0.899071908161608  0.0
-0.925239300391083  -0.430030382067056  0.0  0.414585092472074  -0.898667870239779  0.0
```

Dan: Great! Time for Pythagoras again: $\sqrt{0.925 \cdot 0.925 + 0.430 \cdot 0.430} \approx 1.020$.

Carol: A 2% error, about a factor ten smaller than the 17% error we had before. We're getting there!

8.3 An Even Better Orbit

Erica: And we should get a much better picture now.

Carol: Here it is, fig. 8.1.

Erica: Wonderful! You can hardly see the deviation from a circle.

Dan: Yes, the particles almost cover their own tracks, the second time around.

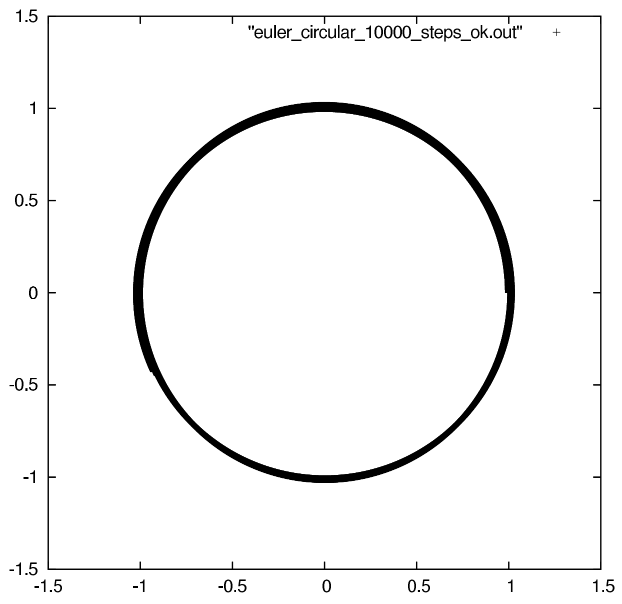

Figure 8.1: Fifth attempt at integrating the two-body problem: looking even better.

Carol: You mean the particle: we're integrating a one-body problem.

Dan: Well, the distance between the two particles is what is plotted, so I feel I can talk about particles.

Erica: And I think you're both right. Stop arguing, you guys! Let's go back to the elliptic case, the one we started with, remember?

8.4 Reasons for Failure

Carol: Sure, I remember that. All we have to do is to take the file `euler_circular.rb` and make the initial velocity half as large, by changing the line:

```
  vy = 1
```

into:

```
  vy = 0.5
```

Dan: But that will be the same file as we started with, `euler.rb`.

Carol: Ah, yes, of course, I had forgotten already. And in that case, the orbit exploded.

Dan: Let's see it again. Now that we seem to understand the circular case, perhaps we can figure out what went wrong in the elliptic case.

Carol: Okay, here we go again:

```
 |gravity> ruby euler.rb > euler.out
```

And here is the plot once more, in fig 8.2:

Erica: Let us take a moment to evaluate what we have learned. We know now how small the steps have to be to get a reasonable convergence for a circular orbit. And we can see in figure 8.2 that the steps get far larger toward the left of the figure.

In fact, even when the steps start off with a reasonably small size at the right hand side, by the time we have reached the left, the steps are so large that even a circular orbit would not reach convergence if we would everywhere use such large steps!

Dan: Why do the steps get so large, all of a sudden?

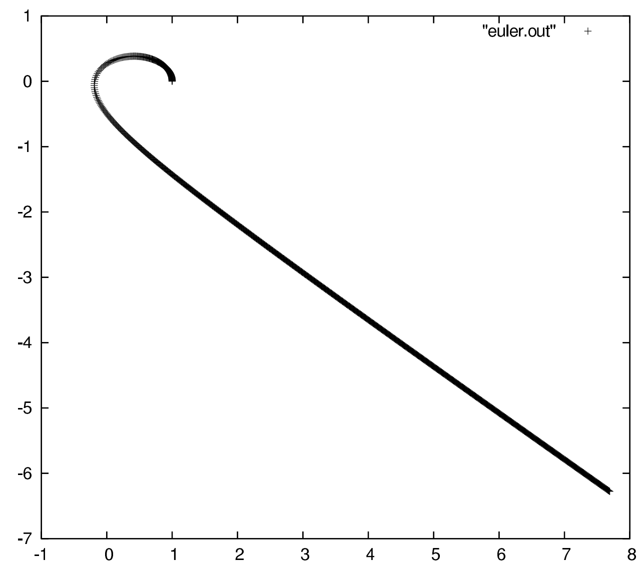

Figure 8.2: A rerun of the second attempt at integrating the two-body problem.

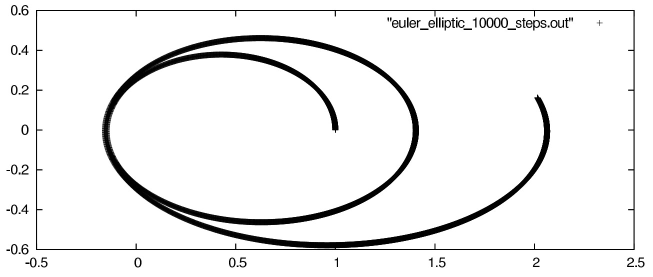

Figure 8.3: Sixth attempt at integrating the two-body problem: signs of hope.

Erica: Because the particles get very close together. Notice that the left-most part of the orbit is also the point in the orbit that is closest to the origin, the place where $x = y = 0$. This is called the *pericenter* of an elliptic orbit. This word is derived from the Greek $\pi\epsilon\rho\iota$ *(peri)* meaning 'around' or 'near'.

You see, we started off with a speed smaller than the speed required for a circular orbit, in fact, we had only have of that speed. So the particles started to fall toward each other right away, and IF we would have computed the orbit correctly, the two particles would have returned to the exact same spot after one revolution, just as we finally managed to see in the circular case when we took very small steps.

Carol: So the initial position is then the place in the orbit where the particles are furthest away from each other?

Erica: Yes, indeed! And that point is called the apocenter, from the Greek $\alpha\pi o$ *(apo)* meaning 'far (away) from'. Well, I am willing to bet that a smaller time step will cure all of our problems.

Dan: Seeing is believing. Can you show us, Carol?

8.5 Signs of Hope

Carol: Here we go, a ten times smaller step size in `euler_elliptic_10000_steps.rb`; I'll plot the result in fig. 8.3.

```
|gravity> ruby euler_elliptic_10000_steps.rb > euler_elliptic_10000_steps.out
```

Dan: I must admit, you may both have been right: at least now the particles are completing a couple orbits that sort-of look elliptical, even though the errors are still large. But at least they don't fly off to infinity like in a slingshot.

Erica: Yes, I think we're getting to the bottom of all this, finally.

Dan: But we'd better make sure, and use even smaller steps.

Carol: Will do!

Chapter 9

Convergence for an Elliptic Orbit

9.1 Adding a Counter

Carol: Yes, let's go to smaller steps, but I'm worried about one thing, though. Each time we make the steps ten times as small, we are generating ten times more output. This means a ten times larger output file, and ten times more points to load into our graphics figure. Something tells me that we may have to make the steps a hundred times smaller yet, to get reasonable convergence, and at some point we will be running into trouble when we start saving millions of points.

Let's check the file size so far:

```
|gravity> ls -l euler_elliptic_10000_steps.out
-rw-r--r--    1 makino   makino     854128 Sep 14 08:21 euler_elliptic_10000_steps.out
```

Dan: I see, almost a Megabyte. This means that a thousand times smaller step size would generate a file of almost a Gigabyte. That would be overkill and probably take quite a while to plot. I guess we'll have to prune the output, and only keep some of the points.

Erica: Good idea. A natural approach would be to keep the same number of points as we got in our first attempt, namely one thousand. In our next-to-last plot, figure 8.2 you could still see how the individual points were separated further from each other at the left hand side, while in our last plot, figure 8.3, everything is so crowded that you can't see what is going on.

Dan: What do you mean with 'going on'?

Erica: In figure 8.2, on the left hand side, you can see that the individual points are separated most when the particles come close together. This means that the particles are moving at the highest speed, which makes sense: when two particles fall toward each other, they speed up. As long as we stick to only a few hundred points per orbit, we will be able to see that effect nicely also when we reach convergence in more accurate calculations.

Carol: I see. That makes sense. I'd like to aks you more about that, but before doing so, let's first get the pruning job done, in order to produce more sparse output. I will take our last code, from `euler_elliptic_10000_steps.rb`, and call it `euler_elliptic_10000_steps_sparse.rb` instead. Yes, Dan, you can later copy it into `ee1s.rb`, if you like. How to prune things? We have a time step of `dt = 0.001` that is ten times smaller than our original choice, and therefore it produces ten times too many points.

The solution is to plot only one out of ten points. The simplest way I can think of is to introduce a counter in our loop, which keeps track of how many times we have traversed the loop. I will call the counter `i`:

```
10000.times{|i|
```

Erica: what do the vertical bars mean?

Carol: That is how Ruby allows you to use a counter. In most languages, you start with a counter, and then you define the looping mechanism explicitly by using the counter. For example, in C you write

```
for (i = 0; i < imax; i++){ ... }
```

which defines a loop that is traversed `imax` times. Ruby is cleaner, in the sense that it allows you to forget about such implementation details. The construct

```
imax.times{ ... }
```

neatly takes care of everything, while hiding the actual counting procedure. However, if you like to make the counter visible, you can do so by writing:

```
imax.times{|i| ... }
```

where `i`, or whatever name you like to choose for the variable, will become the explicit counter.

9.2 Sparse Output

Dan: So now we have to give the print statements a test which is passed only one out of ten times.

Carol: Exactly. How about this?

```
if i%10 == 0
  print(x, "  ", y, "  ", z, "  ")
  print(vx, "  ", vy, "  ", vz, "\n")
end
```

Here the symbol % gives you the reminder after a division, just as in C.

Dan: So when you write 8%3, you get 2.

Carol: Yes. And the way I wrote it above, i%10, will be equal to zero only one out of ten times, only when the number i is a multiple of ten, or in decimal notation ends in a zero.

Dan: Okay, that's hard to argue with. Let's try it. Better make sure that you land on the same last point as before. How about running the old code and the new sparse code, and comparing the last few lines?

Carol: Good idea. After our debugging sessions you've gotten a taste for testing, hey? You'll turn into a computer scientist before you know it! I'll give you what you ordered, but of course there is hardly anything that can go wrong:

```
|gravity> ruby euler_elliptic_10000_steps.rb | tail -3
2.01466014781365  0.162047884550843  0.0  -0.152387493429476  0.258735730522888  0.0
2.01450776032022  0.162306620281366  0.0  -0.152631496539003  0.258716104290758  0.0
2.01435512882368  0.162565336385657  0.0  -0.152875528688122  0.258696442895482  0.0
```

```
|gravity> ruby euler_elliptic_10000_steps_sparse.rb | tail -3
2.018682481674  0.155054729139203  0.0  -0.145810200248145  0.259252408016603  0.0
2.01721343061819  0.157646408310245  0.0  -0.14824383417573  0.259064012737983  0.0
2.01572003060918  0.160236187852851  0.0  -0.150680280478263  0.258872130993741  0.0
```

Dan: Well, *hardly* anything perhaps, but still *something* went wrong . . .

Carol: . . . yes, I spoke too soon. The points do some to be further separated from each other, but the last point from the new code doesn't quite reach the last of the many points that the old code printed.

Ah, of course! I should have thought about that. Off by one!

Erica: Off by one?

Carol: Yes, that's what we call it when you forget that Ruby, or C for that matter, is counting things starting from zero rather than from one. The first time we traverse the loop, the value of `i` is zero, the second time it is one. We want to print out the results one out of ten times. This means that each time we have traversed the loop ten times, we print. After the tenth traversal, `i = 9`, since we started with `i = 0`. Here, I'll make the change, and call the file `euler_elliptic_10000_steps_sparse_ok.rb`:

```
  if i%10 == 9
    print(x, "  ", y, "  ", z, "  ")
    print(vx, "  ", vy, "  ", vz, "\n")
  end
```

Let me try again:

```
|gravity> ruby euler_elliptic_10000_steps.rb | tail -3
2.01466014781365  0.162047884550843  0.0  -0.152387493429476  0.258735730522888  0
2.01450776032022  0.162306620281366  0.0  -0.152631496539003  0.258716104290758  0
2.01435512882368  0.162565336385657  0.0  -0.152875528688122  0.258696442895482  0
```

```
|gravity> ruby euler_elliptic_10000_steps_sparse_ok.rb | tail -3
2.01736143096325  0.157387325301357  0.0  -0.148000345061228  0.259083008888045  0
2.01587046711702  0.159977296376325  0.0  -0.150436507846502  0.258891476525706  0
2.01435512882368  0.162565336385657  0.0  -0.152875528688122  0.258696442895482  0
```

Dan: Congratulations! I guess this is called off by zero? The last points are indeed identical.

Erica: I'd call it on target. And presumably the output file is ten times smaller?

Carol: Easy to check:

```
|gravity> ruby euler_elliptic_10000_steps.rb | wc
  10001   60006  854128
```

```
|gravity> ruby euler_elliptic_10000_steps_sparse_ok.rb | wc
   1001    6006   85445
```

So it is; from more than 10,000 lines back to 1001 lines, as before.

9.3 Better and Better

Dan: Resulting in a sparser figure, I hope?

Carol: That's the idea!

```
|gravity> ruby euler_elliptic_10000_steps_sparse_ok.rb > euler_elliptic_10000_steps_sparse_ol
```

Here is the plot, in fig. 9.1.

Erica: And yes, you can again see the individual steps on the left-hand side.

Carol: It will be easy now to take shorter and shorter steps. Starting from `euler_elliptic_10000_steps_sparse_ok.rb`, which we used before, I'll make a file `euler_elliptic_100000_steps_sparse_ok.rb`, with only two lines different: the `dt` value and the `if` statement:

```
dt = 0.0001
```

```
  if i%100 == 99
    print(x, "  ", y, "  ", z, "  ")
    print(vx, "  ", vy, "  ", vz, "\n")
  end
```

Similarly, in `ruby euler_elliptic_1000000_steps_sparse_ok.rb` we have

```
dt = 0.00001
```

```
  if i%1000 == 999
    print(x, "  ", y, "  ", z, "  ")
    print(vx, "  ", vy, "  ", vz, "\n")
  end
```

I'll run the codes and show the plots, in fig. 9.2 and fig. 9.3, respectively.

```
|gravity> ruby euler_elliptic_100000_steps_sparse_ok.rb > euler_elliptic_100000_steps_sparse_
```

```
|gravity> ruby euler_elliptic_1000000_steps_sparse_ok.rb > euler_elliptic_1000000_steps_spars
```

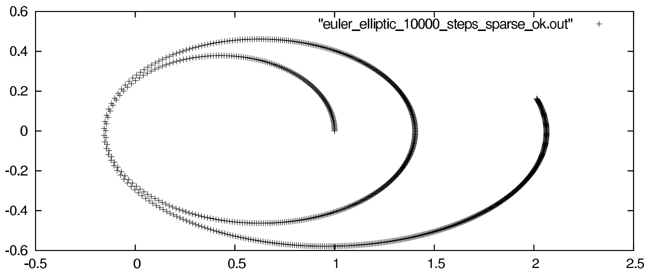

Figure 9.1: Seventh attempt at integrating the two-body problem: sparse output

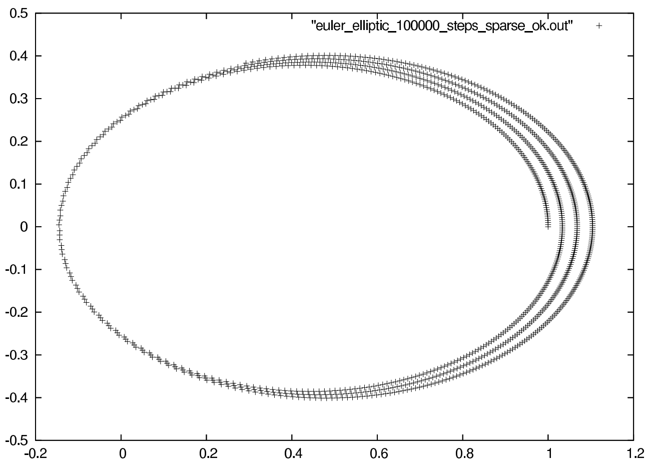

Figure 9.2: Eighth attempt at integrating the two-body problem: starting to converge.

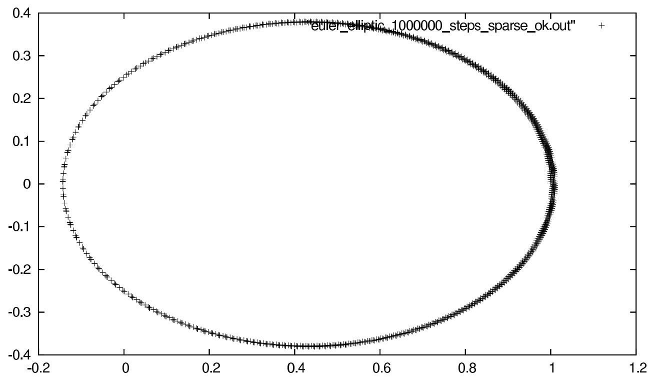

Figure 9.3: Ninth attempt at integrating the two-body problem: finally converging.

9.4 A Print Method

Dan: Beautiful. A real ellipse! Newton would have been delighted to see this. The poor guy; he had to do everything by hand.

Carol: But at least he was not spending time debugging . . .

Erica: . . . or answering email. Those were the days!

Dan: I'm not completely clear about the asymmetry in the final figure. At the left, the points are much further apart. Because all points are equally spaced in time, this means that the motion is much faster, right?

Erica: Right! Remember, what is plotted is the one-body system that stands in for the solution of the two-body problem.

Carol: You know, I would find it really helpful if we could plot the orbits of both particles separately. So far, it has made our life easier to use the $\{\mathbf{R}, \mathbf{r}\}$ coordinates, since we could choose the c.o.m. coordinate system in which $\{\mathbf{R} = 0\}$ by definition, so we only had to plot $\{\mathbf{r}\}$. But how about going back to our original coordinate system, plotting the full $\{\mathbf{r1}, \mathbf{r2}\}$ coordinates, one for each particle separately?

Erica: That can't be hard. We just have to look at the summary we wrote of our derivations, where was that, ah yes, Eq. (4.18) is what we need.

Dan: And we derived those in Eqs. (4.4) and (4.6), because Carol insisted we do so.

Carol: I'm glad I did! You see, it often pays off, if you're curious. Pure science quickly leads to applied science.

Dan: You always have such a grandiose way to put yourself on the map! But in this case you're right, we do have an application. Now how do we do this . . . oh, it's easy really: we can just plot the positions of both particles, in any order we like. As long as we plot all the points, the orbits will show up in the end.

Erica: And it is most natural to plot the position of each particle in turn, while traversing the loop once. All we have to do is to make the print statements a bit more complicated.

Carol: But I don't like to do that twice, once before we enter the loop, and once inside the loop, toward the end. It's high time to define a method.

Dan: What's a method?

Carol: It's what is called a function in C or a subroutine in Fortran, a piece of code that can be called from elsewhere, perhaps using some arguments. Here, I'll show you. When you improve a code, rule number one is: try not to break what already works. This means: be careful, take it one step at a time.

In our case, this means: before trying to go to a new coordinate system, let us first implement the method idea in the old code, then check that the old code still gives the right result, and only then try to change coordinate systems.

So far, we have solved the one-body system, using the computer program in `euler_elliptic_1000000_steps_sparse_ok.rb`. I'll copy it to a new file `euler_one_body.rb`. Now I'm going to wrap the print statements for the one-body system into a method called `print1`:

```
def print1(x,y,z,vx,vy,vz)
  print(x, "  ", y, "  ", z, "  ")
  print(vx, "  ", vy, "  ", vz, "\n")
end
```

and I will invoke this method once at the beginning, just before entering the loop:

```
print1(x,y,z,vx,vy,vz)
1000000.times{|i|
```

and once at the end of each loop traversal:

```
  print1(x,y,z,vx,vy,vz) if i%1000 == 999
```

Dan: Wait a minute, shouldn't the `if` statement come in front?

Carol: in most languages, yes, but in Ruby you instead of writing:

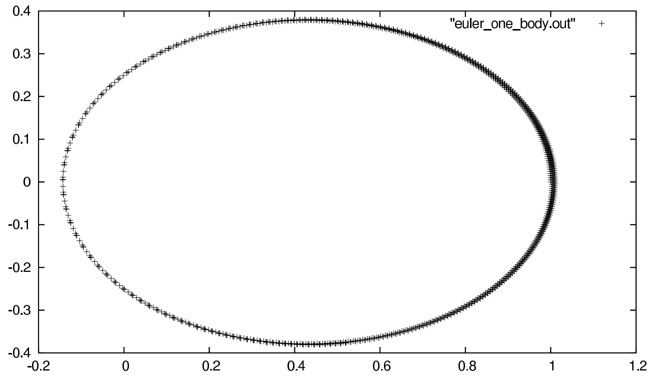

Figure 9.4: Tenth attempt at integrating the two-body problem: check of 1-body output.

```
  if a
    b
  end
```

you can also write

```
 b if a
```

if everything fits on one line, and then the `end` can be omitted.

Well, now this new code should give the same results as we had before:

```
|gravity> ruby euler_one_body.rb > euler_one_body.out
```

Erica: Sure looks the same.

Dan: If you really want to show that it's the same, why not print the last lines in each case?

Carol: Right, let's check that too:

```
|gravity> ruby euler_elliptic_1000000_steps_sparse_ok.rb | tail -3
0.496147378042769  -0.37881858244996  0.0  1.21129218162732  0.0849254022743251  0.0
0.508159065963217  -0.377892709335516  0.0  1.19108982655975  0.100148843547868  0.0
0.519970642634004  -0.376817992041834  0.0  1.17126787143698  0.114700879739653  0.0
|gravity> ruby euler_one_body.rb | tail -3
```

```
0.496147378042769  -0.37881858244996  0.0  1.21129218162732  0.0849254022743251  0
0.508159065963217  -0.377892709335516  0.0  1.19108982655975  0.100148843547868  0
0.519970642634004  -0.376817992041834  0.0  1.17126787143698  0.114700879739653  0
```

Dan: I'm happy. So now you're going to copy the code of `euler_one_body.rb` to a new file called `euler_two_body.rb` . . .

9.5 From One Body to Two Bodies

Carol: You're reading my mind. All I have to do now is implement Eq. (4.18), and its time derivative, where positions are replaced by velocities:

```
def print2(m1,m2,x,y,z,vx,vy,vz)
  mfrac1 = m1/(m1+m2)
  mfrac2 = m2/(m1+m2)
  print(-mfrac2*x, "  ", -mfrac2*y, "  ", -mfrac2*z, "  ")
  print(-mfrac2*vx, "  ", -mfrac2*vy, "  ", -mfrac2*vz, "\n")
  print(mfrac1*x, "  ", mfrac1*y, "  ", mfrac1*z, "  ")
  print(mfrac1*vx, "  ", mfrac1*vy, "  ", mfrac1*vz, "\n")
end
```

Dan: And change `print1` to `print2`.

Carol: Yes, that *plus* the fact that I now have to give two extra arguments, `m1` and `m2`. In front of the loop this becomes:

```
print2(m1,m2,x,y,z,vx,vy,vz)
1000000.times{|i|
```

and at the end inside the loop:

```
  print2(m1,m2,x,y,z,vx,vy,vz) if i%1000 == 999
```

Erica: But . . . don't we have to specify somewhere what the two masses are?

Carol: Oops! Good point. So far we've been working in a system of units in which $M_1 + M_2 = 1$, and in the c.o.m. coordinates we never had to specify what each mass value was. But now we'd better write the mass values in the initial conditions.

Erica: And for consistency, we should insist that the sum of the masses remains unity, so we only have one value that we can freely choose. For example, once we choose a value for m1, the value for m2 is fixed to be m2 = 1 - m1.

Carol: That's easy to add. How about making the masses somewhat unequal, but not hugely so? That way we can still hope to see both orbits clearly. I'll make m1 = 0.6:

```
m1 = 0.6
m2 = 1 - m1
```

Dan: Given that we use the convention $M_1 + M_2 = 1$, there is really no need to divide by this quantity, in the method print2. In fact, there was no reason to introduce the variables mfrac1 and mfrac2 for the mass fractions that were assigned to each star. With the total mass being unity, the mass fraction in each star has exactly the same value as the mass of each star itself.

Erica: Yes, that is true. However, I prefer to keep print2 the way it is, just to make the physics clear. When you write mfrac1*vx, it is clear that you are dealing with a velocity, vx, that is multiplied by a mass fraction. If you were to write simply m1*vx, you would get the same numerical value, but the casual reader would get the impression that you are now working with a momentum, rather than a velocity.

Carol: I agree. I can see Dan's argument for writing a shorter and minimal version of print2, but I, too, prefer the longer version, for clarity.

Dan: Okay, I can see the point, though I myself would prefer brevity over clarity in this case. But since I'm outvoted here, let's leave it as it is. Can you show the whole program? I'm beginning to loose track now.

Carol: Here it is:

```
include Math

x = 1
y = 0
z = 0
vx = 0
vy = 0.5
vz = 0
dt = 0.00001

m1 = 0.6
m2 = 1 - m1

def print2(m1,m2,x,y,z,vx,vy,vz)
```

```
  mfrac1 = m1/(m1+m2)
  mfrac2 = m2/(m1+m2)
  print(-mfrac2*x, "  ", -mfrac2*y, "  ", -mfrac2*z, "  ")
  print(-mfrac2*vx, "  ", -mfrac2*vy, "  ", -mfrac2*vz, "\n")
  print(mfrac1*x, "  ", mfrac1*y, "  ", mfrac1*z, "  ")
  print(mfrac1*vx, "  ", mfrac1*vy, "  ", mfrac1*vz, "\n")
end

print2(m1,m2,x,y,z,vx,vy,vz)
1000000.times{|i|
  r2 = x*x + y*y + z*z
  r3 = r2 * sqrt(r2)
  ax = - x / r3
  ay = - y / r3
  az = - z / r3
  x += vx*dt
  y += vy*dt
  z += vz*dt
  vx += ax*dt
  vy += ay*dt
  vz += az*dt
  print2(m1,m2,x,y,z,vx,vy,vz) if i%1000 == 999
}
```

And here is the output:

```
|gravity> ruby euler_two_body.rb > euler_two_body.out
```

And the results are plotted in fig. 9.5

Erica: Beautiful!

Dan: Indeed. That makes everything a lot more concrete for me. So the bigger ellipse belongs to the particle with the smaller mass, `m2`, and the smaller ellipse is for the bigger one, `m1`.

Erica: And they always face each other from different sides with respect to the origin, $\{x, y\} = \{0, 0\}$.

Carol: For now, I take your word for it, but it sure would be nice to see all that actually happening. I mean, it would be great to see the particles orbiting each other in a movie.

Erica: Definitely. But before we go into that, I suggest we move up one step, from the first-order forward Euler algorithm to a second-order algorithm. Look, we're now using a whopping one million steps just to go around a simple ellipse a few times. Clearly, forward Euler is very inefficient.

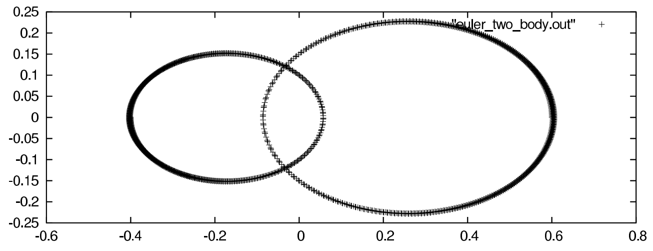

Figure 9.5: Eleventh attempt at integrating the two-body problem: check of 2-body output.

Dan: I've been wondering about that. I agree. Let's get a better scheme first, but then it will be time to see a movie.

Chapter 10

The Modified Euler Algorithm

10.1 A Wild Idea

Dan: Well, Erica, how are we going to move up to a more accurate algorithm?

Carol: You mentioned something about a second-order scheme.

Erica: Yes, and there are several different choices. With our first-order approach, we had little choice. Forward Euler was the obvious one: just follow your nose, the way it is pointed at the beginning of the step, as in fig. 5.1.

Dan: You mentioned a backward Euler as well, and even drew a picture, in in fig. 5.2.

Erica: That was only because you asked me about it! And the backward Euler scheme is not an explicit method. It is an implicit method, where you have to know the answer before give calculate it. As we discussed, you can solve that through iteration; but in that case you have to redo every step at least one more time, so you spend a lot more computer time and you still only have a first-order method, so there is really no good reason to use that method.

Carol: But wait a minute, the two types of errors in figs. 5.1 and 5.2 are clearly going in the opposite directions. I mean, forward flies out of the curve one way, and backward spirals in the other way. I wonder, can't you somehow combine the two methods and see whether we can let the two errors cancel each other?

If we combine the previous two pictures, the most natural thing would be to try *both* of the Euler types, forward and backward. Here is a sketch, in fig. 17.2. The top arrow is what we've done so far, forward Euler, simply following the tangent line of the curve. The bottom line is backward Euler, taking a step that lands on a curve with the right tangent at the end. My idea is to compute both,

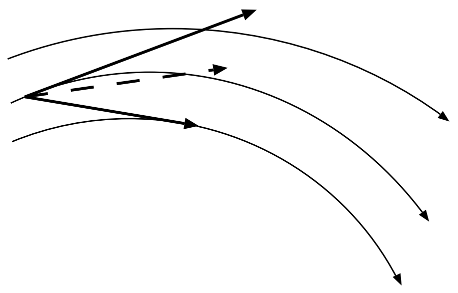

Figure 10.1: An attempt to improve the Euler methods. The top arrow shows forward Euler, and the bottom arrow backward Euler. The dashed arrow shows the average between the two, which clearly gives a better approximation to the curved lines that show the true solutions to the differential equation.

and then take the average between the two attempts. I'm sure that would give a better approximation!

Dan: But at a large cost! The backward Euler method is an implicit method, as Erica mentioned, that requires at least one extra iteration. So the bottom arrow alone is much more expensive to compute than the top arrow, and we have to compute both.

Carol: It was just a wild idea, and it may not be useful.

10.2 Backward and Forward

Erica: Actually, I like Carol's idea. In reminds me of one of the second order schemes that I learned in class. Let me just check my notes.

Aha, I found it. There is an algorithm called "Modified Euler", which starts with the forward Euler idea, and then modifies it to improve the accuracy, from first order to second order. And it seems rather similar to what Carol just sketched.

Carol: In that case, how about trying to reconstruct it for ourselves. That is more fun than copying the algorithm from a book.

Now let's see. We want to compute the dashed line in figure 17.2. How about shifting the arrow of the backward step to the end of the arrow of the forward step, as in fig. 10.2? Or to be precise, how about just taking two forward Euler steps, one after the other? The second forward step will not produce exactly the same arrow as the first backward step, but it will be almost the same arrow, and perhaps such an approximation would be good enough.

Dan: But how are you going to use that to construct the dashed line in fig. 17.2?

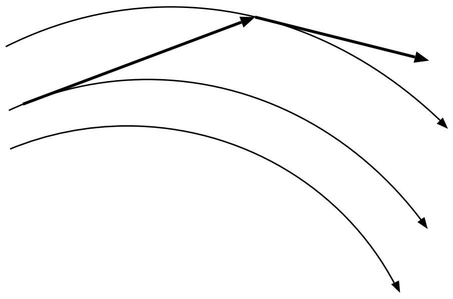

Figure 10.2: Two successive forward Euler steps.

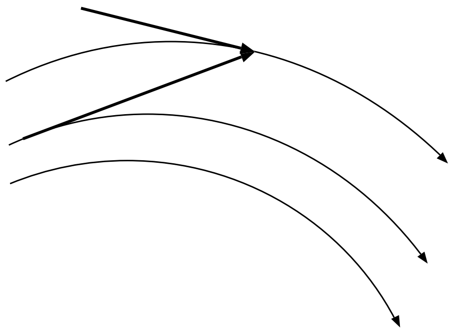

Figure 10.3: A forward Euler steps and a backward Euler step, landing at the same point.

Carol: How about shifting the second arrow back, in fig. 10.2, so that the end of the arrow falls on the same point as the end of the first arrow? In that way, we have constructed a backward Euler step that lands on the same point where our forward Euler step landed, as you can see in fig. 10.3.

As I already admitted, the top arrow in fig. 10.3 is not exactly the same arrow as the bottom arrow in fig. 17.2, but the two arrows are approximately the same, especially if our step sizes are not too large. So, in a first approximation, we can average the arrows in fig. 10.3. This will make Carol happy: no more implicit steps. We have only taken forward steps, even though we recycle the second one by interpreting it as a backward step.

The simplest way to construct the average between the two vectors is by adding them and then dividing the length by two. Here it is, in fig. 17.1.

10.3 On Shaky Ground

Dan: I don't believe it. Or what I really mean is: I cannot yet believe that this

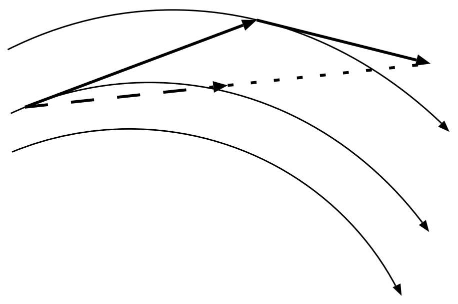

Figure 10.4: The new integration scheme produces the dashed arrow, as exactly one-half of the some of the two fully drawn arrows; the dotted arrow has the same length as the dashed arrow. This result is approximately the same as the dashed arrow in fig. 17.2.

is really correct, because I don't see any proof for it. You are waving your arms and you just hope for the best. Let's be a bit more critical here.

In figure 17.2, it was still quite plausible that the dashed arrow succeeded in canceling the opposite errors in the two solid arrows. Given that those two solid arrows, corresponding to forward Euler and backward Euler, were only first-order accurate, I can see that the error in the dashed arrow just *may* be second-order accurate. Whether the two first-order errors of the solid arrows *actually* cancel in leading order, I'm not sure, but we might be lucky.

But then you start introducing other assumptions: that you can swap the new second forward Euler arrow for the old backward Euler error, and stuff like that. I must say, here I have totally lost my intuition.

Frankly, I feel on really shaky ground, talking about an order of a differential equation. I have some vague sense of what it could mean, but I wouldn't be able to define it.

Erica: Here is the basic idea. If an algorithm is nth order, this means that it makes an error per step that is one order higher in terms of powers of the time step. What I mean is this: for a simple differential equation

$$\frac{dx}{dt} = f(x) \tag{10.1}$$

the error that we make in going from time i to time $i+1$, with $dt = t_{i+1} - t_i$, can be written as:

$$\delta x_{i+1} = B(dt)^{n+1} \tag{10.2}$$

Here the coefficient B is a function of x, but it is almost independent of the size of the time step dt, and in the limit that $dt \to 0$, it will converge to a constant value $B(x, dt) \to B(x)$, which in the general case will be proportional to the *(n+1)* th time derivative of $f(x(t))$, along the orbit $x(t)$.

In practice, we want to integrate an orbit over a given finite interval of time, for example from $t = 0$ to $t = T$. For a choice of step size dt, we then need $k = T/dt$ integration steps. If we assume that the errors simply add up, in other words, if we don't rely on the errors to cancel each other in some way, then the total integration error after k steps will be bounded by

$$\delta x(T) < kC(dt)^{n+1} = \frac{T}{dt}C(dt)^{n+1} = TC(dt)^n \tag{10.3}$$

where C is proportional to an upper bound of the absolute value of the *(n+1)* th time derivative of $f(x(t))$, along the orbit $x(t)$.

In other words, for an nth order algorithm, the error we make after integrating for a *single time step* scales like the *(n+1)* th power of the time step, and the error we make after integrating for a ¡t¿fixed finite amount of time¡/t¿ scales like the n th power of the time step.

Dan: I'm now totally confused. I don't see at all how these higher derivatives of f come in. In any case, for the time being, I would prefer to do, rather than think too much. Let's just code up and run the algorithm, and check whether it is really second order.

Erica: That's fine, and I agree, we shouldn't try to get into a complete numerical analysis course. However, I think I can see what Carol is getting at. If we apply her reasoning to the forward Euler algorithm, which is a first order algorithm, we find that the accumulated error over a fixed time interval scales like the first power of time. Yes, that makes sense: when we have made the time step ten times smaller, for example in sections 7.3 and 8.2, we have found that the error became roughly ten times smaller.

Carol: So if the modified Euler algorithm is really a second-order algorithm, we should be able to reduce the error by a factor one hundred, when we make the time step ten times smaller.

Erica: Yes, that's the idea, and that would be great! Let's write a code for it, so that we can try it out.

Dan: I'm all for writing code! Later we can always go back to see what the theory says. For me, at least, theory makes much more sense after I see at least one working application.

10.4 Specifying the Steps

Carol: It should be easy to implement this new modified Euler scheme. The picture we have drawn shows the change in position of a particle, and we should apply the same idea to the change in velocity.

For starters, let us just look at the position. First we have to introduce some notation.

Erica: In the literature, people often talk about predictor-corrector methods. The idea is that you first make a rough prediction about a future position, and then you make a correction, after you have evaluated the forces at that predicted position.

In our case, in fig. 17.1, the first solid arrow starts at the original point $\mathbf{r}_i$. Let us call the end point of that arrow $\mathbf{r}_{i+1,p}$, where the p stands for *predicted*, as the predicted result of taking a forward Euler step:

$$\mathbf{r}_{i+1,p} = \mathbf{r}_i + \mathbf{v}_i dt \tag{10.4}$$

The second arrow shows another prediction, namely for yet another forward Euler step, which lands us at $\mathbf{r}_{i+2,p}$:

$$\mathbf{r}_{i+2,p} = \mathbf{r}_{i+1,p} + \mathbf{v}_{i+1,p} dt \tag{10.5}$$

Dan: But here you are using the velocity at time $i+1$, something that you haven't calculated yet.

Erica: I know, we'll come to that in a moment, when we write down the velocity equivalent for Eq. (10.4). I just wanted to write the position part first. We can find the *corrected* new position by taking the average of the first two forward Euler steps, as indicated in fig. 17.1:

$$\begin{aligned} \mathbf{r}_{i+1,c} &= \mathbf{r}_i + \tfrac{1}{2}\left\{(\mathbf{r}_{i+1,p} - \mathbf{r}_i) + (\mathbf{r}_{i+2,p} - \mathbf{r}_{i+1,p})\right\} \\ &= \mathbf{r}_i + \tfrac{1}{2}\left(\mathbf{r}_{i+2,p} - \mathbf{r}_i\right) \\ &= \tfrac{1}{2}\left(\mathbf{r}_i + \mathbf{r}_{i+2,p}\right) \end{aligned} \tag{10.6}$$

Carol: As Dan pointed out, we have to do a similar thing for the velocities. I guess that everything carries over, but with $\mathbf{v}$ instead of $\mathbf{r}$ and $\mathbf{a}$ instead of $\mathbf{v}$.

Erica: Yes, in fact it is just a matter of differentiating the previous lines with respect to time. Putting it all together, we can calculate all that we need in the following order, from predicted to corrected quantities:

$$\begin{aligned} \mathbf{r}_{i+1,p} &= \mathbf{r}_i + \mathbf{v}_i dt \\ \mathbf{v}_{i+1,p} &= \mathbf{v}_i + \mathbf{a}_i dt \\ \mathbf{r}_{i+2,p} &= \mathbf{r}_{i+1,p} + \mathbf{v}_{i+1,p} dt \\ \mathbf{v}_{i+2,p} &= \mathbf{v}_{i+1,p} + \mathbf{a}_{i+1,p} dt \\ \mathbf{r}_{i+1,c} &= \tfrac{1}{2}\left(\mathbf{r}_i + \mathbf{r}_{i+2,p}\right) \\ \mathbf{v}_{i+1,c} &= \tfrac{1}{2}\left(\mathbf{v}_i + \mathbf{v}_{i+2,p}\right) \end{aligned} \tag{10.7}$$

Dan: Just on time delivery, as they say: $\mathbf{v}_{i+1,p}$ is calculated just before it is needed in calculating $\mathbf{r}_{i+2,p}$, just as Erica correctly predicted (no pun intended).

10.5 Implementation

Carol: Here is the new code. I'll call it `euler_modified_10000_steps_sparse.rb`. Let's hope we have properly modified the original Euler:

```
include Math

x = 1
y = 0
z = 0
vx = 0
vy = 0.5
vz = 0
dt = 0.001

print(x, "  ", y, "  ", z, "  ")
print(vx, "  ", vy, "  ", vz, "\n")

10000.times{|i|
  r2 = x*x + y*y + z*z
  r3 = r2 * sqrt(r2)
  ax = - x / r3
  ay = - y / r3
  az = - z / r3
  x1 = x + vx*dt
  y1 = y + vy*dt
  z1 = z + vz*dt
  vx1 = vx + ax*dt
  vy1 = vy + ay*dt
  vz1 = vz + az*dt
  r12 = x1*x1 + y1*y1 + z1*z1
  r13 = r12 * sqrt(r12)
  ax1 = - x1 / r13
  ay1 = - y1 / r13
  az1 = - z1 / r13
  x2 = x1 + vx1*dt
  y2 = y1 + vy1*dt
  z2 = z1 + vz1*dt
  vx2 = vx1 + ax1*dt
  vy2 = vy1 + ay1*dt
  vz2 = vz1 + az1*dt
```

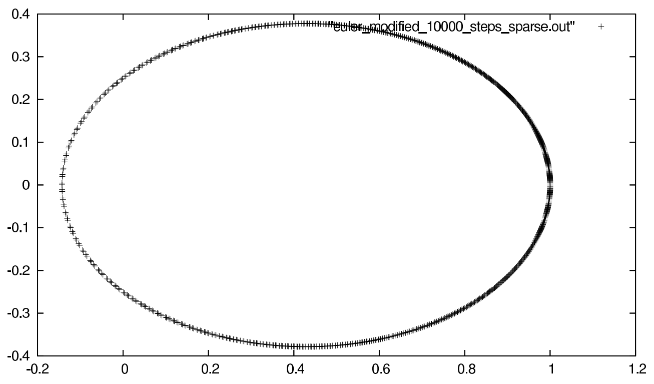

Figure 10.5: First attempt at modified Euler integration, with step size $dt = 0.001$.

```
  x = 0.5 * (x + x2)
  y = 0.5 * (y + y2)
  z = 0.5 * (z + z2)
  vx = 0.5 * (vx + vx2)
  vy = 0.5 * (vy + vy2)
  vz = 0.5 * (vz + vz2)
  if i%10 == 9
    print(x, "  ", y, "  ", z, "  ")
    print(vx, "  ", vy, "  ", vz, "\n")
  end
}
```

10.6 Experimentation

Carol: As you can see I am giving it time steps of size 0.001, just to be on the safe side. Remember, in the case of plain old forward Euler, when we chose that step size, we got figure 9.1. Presumably, we will get a more accurate orbit integration this time. Let's try it!

```
|gravity> ruby euler_modified_10000_steps_sparse.rb > euler_modified_10000_steps_s
```

Here are the results, in figure 10.5.

Dan: Wow!!! Too good to be true. I can't even see deviations from the true elliptic orbit! This is just as good as what we got for forward Euler with a hundred times more work, in figure 9.3.

Erica: fifty times more work, you mean. In figure 9.3, we had used time steps of 10^{-5}, a hundred times smaller than the time steps of 10^{-3} that we used in figure 10.5; but in our modified Euler case, each step requires twice as much work.

Dan: Ah, yes, you're right. Well, I certainly don't mind doing twice as much work per step, if I have to do far fewer than half the number of steps!

10.7 Simplification

Carol: Let's try to do even less work, to see how quickly things get bad. Here, I'll make the time step that is ten times larger, in the file `euler_modified_1000_steps.rb`. This also makes life a little simpler, because now we no longer have to sample: we can produce one output for each step, in order to get our required one thousand outputs:

```
include Math

x = 1
y = 0
z = 0
vx = 0
vy = 0.5
vz = 0
dt = 0.01

print(x, "  ", y, "  ", z, "  ")
print(vx, "  ", vy, "  ", vz, "\n")

1000.times{
  r2 = x*x + y*y + z*z
  r3 = r2 * sqrt(r2)
  ax = - x / r3
  ay = - y / r3
  az = - z / r3
  x1 = x + vx*dt
  y1 = y + vy*dt
  z1 = z + vz*dt
  vx1 = vx + ax*dt
  vy1 = vy + ay*dt
  vz1 = vz + az*dt
```

```
   r12 = x1*x1 + y1*y1 + z1*z1
   r13 = r12 * sqrt(r12)
   ax1 = - x1 / r13
   ay1 = - y1 / r13
   az1 = - z1 / r13
   x2 = x1 + vx1*dt
   y2 = y1 + vy1*dt
   z2 = z1 + vz1*dt
   vx2 = vx1 + ax1*dt
   vy2 = vy1 + ay1*dt
   vz2 = vz1 + az1*dt
   x = 0.5 * (x + x2)
   y = 0.5 * (y + y2)
   z = 0.5 * (z + z2)
   vx = 0.5 * (vx + vx2)
   vy = 0.5 * (vy + vy2)
   vz = 0.5 * (vz + vz2)
   print(x, "  ", y, "  ", z, "  ")
   print(vx, "  ", vy, "  ", vz, "\n")
 }
```

Carol: This approach should need just twice as much work as our very first attempt at integrating the elliptic orbit, which resulted in failure, even after we had corrected our initial typo, as we could see in figure 8.2.

```
|gravity> ruby euler_modified_1000_steps.rb > euler_modified_1000_steps.out
```

Erica: Again, this is far better than what we saw in figure 8.2. There we couldn't even complete a single orbit!

10.8 Second Order Scaling

Dan: Yes, it seems clear that our modified Euler behaves a lot better than forward Euler. But we have not yet convinced ourselves that it is really second order. We'd better test it, to make sure.

Carol: Good idea. Here is a third choice of time step, ten times smaller than our original choice, in file euler_modified_100000_steps_sparse.rb:

```
include Math

x = 1
```

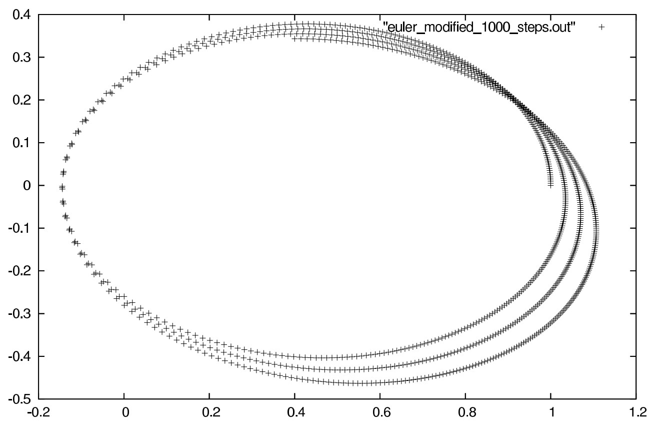

Figure 10.6: Second attempt at modified Euler integration, with step size $dt = 0.01$.

```
y = 0
z = 0
vx = 0
vy = 0.5
vz = 0
dt = 0.0001

print(x, "  ", y, "  ", z, "  ")
print(vx, "  ", vy, "  ", vz, "\n")

100000.times{|i|
  r2 = x*x + y*y + z*z
  r3 = r2 * sqrt(r2)
  ax = - x / r3
  ay = - y / r3
  az = - z / r3
  x1 = x + vx*dt
  y1 = y + vy*dt
  z1 = z + vz*dt
  vx1 = vx + ax*dt
  vy1 = vy + ay*dt
  vz1 = vz + az*dt
  r12 = x1*x1 + y1*y1 + z1*z1
  r13 = r12 * sqrt(r12)
  ax1 = - x1 / r13
```

```
   ay1 = - y1 / r13
   az1 = - z1 / r13
   x2 = x1 + vx1*dt
   y2 = y1 + vy1*dt
   z2 = z1 + vz1*dt
   vx2 = vx1 + ax1*dt
   vy2 = vy1 + ay1*dt
   vz2 = vz1 + az1*dt
   x = 0.5 * (x + x2)
   y = 0.5 * (y + y2)
   z = 0.5 * (z + z2)
   vx = 0.5 * (vx + vx2)
   vy = 0.5 * (vy + vy2)
   vz = 0.5 * (vz + vz2)
   if i%100 == 99
     print(x, "  ", y, "  ", z, "  ")
     print(vx, "  ", vy, "  ", vz, "\n")
   end
 }
```

With the three choices of time step, we can now compare the last output lines in all three cases:

```
|gravity> ruby euler_modified_1000_steps.rb | tail -1
0.400020239524913  0.343214474344616  0.0  -1.48390077762002  -0.0155803976141248
```

```
|gravity> ruby euler_modified_10000_steps_sparse.rb | tail -1
0.598149603243697  -0.361946726406968  0.0  1.03265486807376  0.21104830479922  0.
```

```
|gravity> ruby euler_modified_100000_steps_sparse.rb | tail -1
0.59961042861231  -0.360645741133914  0.0  1.03081178933713  0.213875737743879  0.
```

Well, that's pretty clear, isn't it? The difference between the last two results is about one hundred times smaller, that the difference between the first two results.

In other words, if we take the last outcome as being close to the true result, then the middle result has an error that it about one hundred times smaller than the first result. The first result has a time step that is ten times larger than the second result. Therefore, making the time step ten times smaller gives a result that is about one hundred times more accurate. We can congratulate ourselves: we have clearly crafted a second-order integration algorithm!

Chapter 11

Arrays

11.1 The DRY Principle

Dan: What a difference a second-order scheme makes! Clearly, we can the same accuracy with far fewer calculations than with a first-order scheme. I wonder whether we can go to even higher order schemes, like third order or fourth order or who knows what order.

Erica: Yes, in stellar dynamics, depending on the application, various orders are used. In star cluster calculations, for example, traditionally a fourth-order scheme has been the most popular. In contrast, in planetary dynamics, people routine use even higher-order schemes, like tenth or twelfth order schemes.

But before we go any further in climbing the ladder of integration orders, I really want to write a leapfrog code. The modified Euler version that we have discovered was interesting as such, but in astrophysics at least, the leapfrog is used much more often. Presumably it has some advantages, and in any case, I'd like to see how it behaves.

Dan: What is a *leapfrog*?

Erica: It is probably the most popular integration algorithm, not only in astrophysics, but also in molecular dynamics, as well as in other fields. It combines simplicity with long-term stability. Shall I sketch it out?

Carol: Before we do your 'before', I have an urgent wish: I want to clean up the last code we've written. If we just go on adding extra lines to produce higher-order codes, pretty soon the code becomes a bunch of spaghetti.

Look, everything we do is spelled out separately for the x coordinate, and the again for the y coordinate and then once again for the z coordinate. That's just plain silly. It violates the most basic principle in software writing, the DRY principle: Don't Repeat Yourself.

Dan: What's so wrong with repeating yourself?

Carol: Lot's of things are wrong with that! First of all, repetitions make a code unnecessarily long, and therefore harder to read. Secondly, if you want to modify a feature of a code, it is very difficult to do so correctly if that same feature is repeated elsewhere in the code, even if it is repeated in a place nearby. It is very easy to overlook the repetition, and only modify the first instance that you encounter.

Related to that is a third point: even the first time around that you write a code, if you start repeating yourself, it is quite likely that you make a mistake . . .

Erica . . . as we did in our very first code, with our typo!

Carol: Yes, indeed, I'd forgotten that already. Yes, that was a classic example of the type of penalty you can get for violating the DRY principle!

11.2 Vector Notation

Dan: When we were drawing pictures, we could look at the vectors themselves, but when we started coding, we had to go back to the components of the vectors. Are you suggesting to introduce some graphical way of coding, in which we can specify what happens directly to the vectors, as arrows, rather than to their separate x, y, and z components?

Carol: Well, in a way. Until the middle of the previous century, mathematicians often wrote vector equations in component form. But then they started more and more to use vector notation, by writing down symbols that stood for vectors as such, without any mention of components. On the level of the mathematical equations we have written down, we have used vector notation right from the beginning: we happily wrote things like $\mathbf{r}_2 = \mathbf{r}_1 + \mathbf{v}_1 dt$ on paper, but then we tediously wrote the same thing on our computer screen as:

```
x2 = x1 + vx1*dt
y2 = y1 + vy1*dt
z2 = z1 + vz1*dt
```

Erica: So you would like to write a computer code with lines like

```
r2 = r1 + v1*dt
```

where it would be understood that `r2`, *etc.*, would be an object with the three components { `x2, y2, z2` }.

Carol: Yes, exactly! But for that to work, a lot more should be understood. For example, it should also be understood that the simple `+` symbol is now a much more complicated addition operator. It should be clear to the computer that each of the components of `r1` should be added to the equivalent component of the second expression, `v1*dt`. And in that last expression the `*` symbol should in turn be understood to be a more complicated multiplication operator. Multiplying a vector `v1` with the scalar quantity `dt` should be understood as multiplying each of the components of the vector with the same scalar.

Dan: I like the idea of simplifying the code, and making it look more like the pen-and-paper expressions, but boy, the computer will have to understand a lot! Let me write down what you just said, to see whether *I* at least understand it.

Writing in the code `a = b + c` for three vector quantities `a, b, c` should be translated automatically into the following code fragment

```
ax = bx + cx
ay = by + cy
az = bz + cz
```

where `ax` is the first component of the vector `a`, `ay` is its second component, and so on. And writing in the code `a = 3*b` will be translated into

```
ax = 3 * bx
ay = 3 * by
az = 3 * bz
```

Carol: Yes, exactly. That would be nice, wouldn't it?

Erica: Well, let's try to make that work. The first thing that comes to mind is to use arrays. If we represent a vector by an array, then each element of an array can contain a component of the vector.

Dan: That makes sense. I hope Ruby has arrays, just like Fortran?

Carol: Ruby sure does. But, as you can guess, they are far more powerful. A single array can contain objects of different types in different elements, and the length of an array can grow and shrink.

Dan: Seems like overkill to me. But who cares, let's get started.

11.3 Arrays

Erica: Before rewriting our modified Euler code, let us start with the simplest case, and rewrite our original forward Euler code, `euler.rb`.

Carol: Here, let me translate that code, line for line, into array notation. That way we can make sure that we perform the same calculations. Here is file `euler_array_try.rb`:

```
include Math

r = [1, 0, 0]
v = [0, 0.5, 0]
dt = 0.01

print(r[0], "  ", r[1], "  ", r[2], "  ")
print(v[0], "  ", v[1], "  ", v[2], "\n")

1000.times{
  r2 = r[0]*r[0] + r[1]*r[1] + r[2]*r[2]
  r3 = r2 * sqrt(r2)
  a[0] = - r[0] / r3
  a[1] = - r[1] / r3
  a[2] = - r[2] / r3
  r[0] += v[0]*dt
  r[1] += v[1]*dt
  r[2] += v[2]*dt
  v[0] += a[0]*dt
  v[1] += a[1]*dt
  v[2] += a[2]*dt
  print(r[0], "  ", r[1], "  ", r[2], "  ")
  print(v[0], "  ", v[1], "  ", v[2], "\n")
}
```

As you can see, I have simply replaced `x` by `r[0]`, `y` by `r[1]`, `z` by `r[2]`, and similarly for the velocities and accelerations, I have replaced `vx` by `v[0]` and `ax` by `a[0]`, and so on for the other elements.

Dan: From your example, I guess that arrays start with element zero?

Carol: Ah, yes, that's true, like in C, where the first element of an array `foo` is `foo[0]`, unlike Fortran, where the first element is `foo[1]`.

Erica: I noticed that right at the start of the program, you have done a bit more already than simply replacing `x = 1` by `r[0] = 1`, `y = 0` by `r[1] = 0`, and `z = 0` by `r[2] = 0`.

Instead, you have directly assigned all three values in one line by writing `r = [1, 0, 0]`.

Carol: That's true too. I guess I'm getting already familiar enough with Ruby that I had not noticed that I had made a shortcut. Yes, this is a nice example of the compact way in which Ruby can assign values. In fact, this line, which is the first line of the program after the `include` statement, *defines* `r` as having the type 'array', and in fact an array of three elements. Similarly, the second line defines `v`, too, as an array containing three elements.

In the case of `r`, its three elements are integers, and in the case of `v`, the first and last elements are integers, and the middle element is a floating point number. As I had mentioned before, Ruby allows each array element to have its own dynamic type. In practice, though, as soon as we start calculating with these numbers, most of them will quickly become floating point numbers. Adding an integer and a floating point number, or multiplying an integer with a floating point number, is defined as giving a floating point number as a result, as you would expect.

11.4 Declaration

Dan: That all sounds reasonable. Shall we check whether we get the same result as we did before?

Carol: Good idea. I'll just print the last line, so that we can compare that one with our old result:

```
|gravity> ruby euler_array_try.rb | tail -1
euler_array_try.rb:13: undefined local variable or method ‘a’ for main:Object (NameError)
from euler_array_try.rb:10:in ‘times’
from euler_array_try.rb:10
1  0  0  0  0.5  0
```

Dan: Well, something went wrong right at the start. It seems like there is a problem with the acceleration array `a`. The only output line we got was from the print statement before entering the loop.

Carol: Ah, I see. The two variables `r` and `v` are recognized as arrays, because they are defined as such, in the first two assignment lines of the program. The line:

```
r = [1, 0, 0]
```

says clearly: `r` is an array with three elements, and the elements are 1, 0, and 0.

In contrast, the first time that `a` is used occurs in the line:

```
  a[0] = - r[0] / r3
```

and here we are not specifying what `a` is; we are not assigning anything to `a`. Instead, we are assigning a value to a *element* of `a`, as if `a` had already been defined as an object that has elements.

Erica: In C, you could just declare `a` to be an array. But you have told us before that in Ruby, because of dynamic typing, there was no need to declare the type of a variable. What gives?

Carol: Well, in this case we do need to give *some* extra information, otherwise the Ruby interpreter cannot possibly know what we mean. And yes, here we effectively need to declare `a` as an array.

Carol: How should we do that? We could give fake values, say `a = [0, 0, 0]` right at the beginning of the program.

Carol: That would be confusing, since a reader would wonder what the meaning would be of those three zeroes. In fact, in Ruby there is no need to specify how many elements an array has. All we need to say is that the variable `a` has the type of an array. Or more precisely, in Ruby's terminology: the class of `a` is `Array`, which is one of the built-in classes.

11.5 Classes

Dan: What is a class?

Carol: In Ruby, the word class is used to talk about the collection of possible things that have a particular type.

Dan: What exactly is a type?

Carol: The number 3 is an example of the type integer, the string "hello" is an example of the type string, and an array [3, "hello"] is an example of the type array.

In Ruby, roughly speaking, each concrete example of a type is called an object, and the collection of all possible objects of a given type is called a class. Each object of that type is called an instance of the class that corresponds to that type. In Ruby, the character string "hello" is an object, an instance of class String.

Dan: What about numbers?

Carol: In Ruby there is the class of `Fixnum`, which is the class of integers, numbers such as 7 or -5, and the class `Float`, which is the class of floating point numbers, such as 1.41421, and so on. Some classes, such as arrays, allow objects to contain objects of other classes. For example, the object `[-5, 1.41421]` is an array, which means it is an instance of the class `Array`, while the two elements are instances of the classes `Fixnum` and `Float`, respectively.

Erica: That's very different from how it is done in C++, where all the elements of an array always have the same type. In C++, you can have an array `[-5, 8, 3, ...]` and an array `[1.41421, 3.14, ...]` but you certainly cannot mix the types, by giving each element of the array a different type.

Carol: Well, in Ruby you can. This is one of the many ways in which Ruby is far more flexible than C++.

Coming back to Dan's question about classes, let's see how we can interrogate Ruby about class membership:

```
|gravity> irb
a = [-5, 1.41421]
[-5, 1.41421]
a.class
Array
a[0].class
Fixnum
a[1].class
Float
a[1].class.class
Class
a.class.class
Class
quit
```

Erica: Ah, so each class, like `Fixnum` or `Array`, or whatever is a member of the class `Class`?

Carol: Yes, more precisely, *every* constant or variable in Ruby is an object, and therefore must be an instance of a class. The number 3 is an object of class `Fixnum`, and the class `Fixnum` is an object of class `Class`. Note the convention: in Ruby the name of a class always starts with a capital letter.

Erica: And what about `Class`? What is that an instance of?

Carol: Try it, ask `irb`.

Erica: Okay:

```
|gravity> irb
3
3
3.class
Fixnum
3.class.class
Class
3.class.class.class
Class
quit
```

So `Class` is a member of class `Class`. That sounds circular!

Carol: But it isn't. This is an example of the strength of Ruby. Like Lisp and other seemingly circular languages, Ruby has the power to invoke itself, without arbitrary boundaries between metalevels. This is one of my favorite topics. Shall I explain how it works?

Dan: Not today. I got enough of an idea of what a class could be. Let's keep writing code.

Chapter 12

Array Methods

12.1 An Array Declaration

Carol: So where were we? We wanted to declare a as an array. more precisely as an instance of the Array class. Here is one way to do that:

```
a = []
```

Let me write the new version in a new file, euler_array.rb, in the hope things will be correct now:

```
include Math

r = [1, 0, 0]
v = [0, 0.5, 0]
a = []
dt = 0.01

print(r[0], "  ", r[1], "  ", r[2], "  ")
print(v[0], "  ", v[1], "  ", v[2], "\n")

1000.times{
  r2 = r[0]*r[0] + r[1]*r[1] + r[2]*r[2]
  r3 = r2 * sqrt(r2)
  a[0] = - r[0] / r3
  a[1] = - r[1] / r3
  a[2] = - r[2] / r3
  r[0] += v[0]*dt
  r[1] += v[1]*dt
  r[2] += v[2]*dt
  v[0] += a[0]*dt
  v[1] += a[1]*dt
  v[2] += a[2]*dt
  print(r[0], "  ", r[1], "  ", r[2], "  ")
  print(v[0], "  ", v[1], "  ", v[2], "\n")
}
```

12.2 Three Array Methods

Dan: Seeing is believing. Does it now work?

Carol: Let's try:

```
|gravity> ruby euler_array.rb | tail -1
7.6937453936572  -6.27772005661599  0.0  0.812206830641815  -0.574200201239989  0.
```

Dan: Great! And it would be even better if this is what we got before.

Carol: Well, let's check:

```
|gravity> ruby euler.rb | tail -1
7.6937453936572  -6.27772005661599  0.0  0.812206830641815  -0.574200201239989  0.
```

So far, so good. Okay, we got our first array-based version of forward Euler working, but it still looks just like the old version. Time to start using some of the power of Ruby's arrays.

In general, a Ruby class can have methods that are associated with that class. A while ago, we have come across a very simple in section 5.4, where we encountered the method `times` that was associated with the class `Fixnum`. By writing `10.times` we could cause a loop to be transversed ten times.

Ruby has a somewhat confusing notation *(class name)#(method name)* to describe methods that are associated with classes. The example `10.times` is a way to invoke the method `Fixnum#times`. I find it a bit confusing, because in practice you always use the dot notation *(object name).(method name)* in your code. You'll never see the `#` notation in a piece of code; you only encounter it in a manual or other text description of a code.

Back to our application. There are three methods for the class `Array` that we can use right away, namely `Array#each`, `Array#each_index` and `Array#map`.

I'll explain what they all do in a moment, but it may be easiest to show how they work in our forward Euler example, in file `euler_array_each.rb`:

```
include Math

r = [1, 0, 0]
v = [0, 0.5, 0]
dt = 0.01

r.each{|x| print(x, "  ")}
v.each{|x| print(x, "  ")}
print "\n"

1000.times{
  r2 = 0
  r.each{|x| r2 += x*x}
  r3 = r2 * sqrt(r2)
  a = r.map{|x| -x/r3}
  r.each_index{|k| r[k] += v[k]*dt}
  v.each_index{|k| v[k] += a[k]*dt}
  r.each{|x| print(x, "  ")}
  v.each{|x| print(x, "  ")}
  print "\n"
}
```

Erica: That looks nice and compact.

Dan: Does it work?

Carol: Let's see:

```
|gravity> ruby euler_array_each.rb | tail -1
7.6937453936572  -6.27772005661599  0.0  0.812206830641815  -0.574200201239989  0.0
```

12.3 The Methods `each` and `each_index`

Erica: Good! Now let's look at these magic terms. I can guess what `each` does. It seems to iterate over all the elements of an array, applying whatever appears in parentheses to each element.

Carol: Yes, indeed. And while working with a specific element, it needs to give that element a name. The name is defined between the two vertical bars that follow the opening parentheses. It works just like the lambda notion in Lisp.

Dan: I've no idea what lambda notation means, but I can see what is happening here. In the line

```
r.each{|x| print(x, "  ")}
```

writing {`|x| ...`} lets `x` stand for the element of the array `r`. First `x = r[0]`, and the `...` command then becomes `print(r[0], " ")`. Then in the next round, `x = r[1]`, and so on.

Hey, now that I'm looking at the code a bit longer, I just noticed that the construction `.each`{`|x| ...`} is actually quite similar to the construction `.times`{`...`} that we use in the loop.

Carol: Yes, in both cases we are dealing with a method, `each` and `times` that causes the statements in parentheses to be iterated. And the analogy goes further. Remember that we learned how to get sparse output? At that time we added a counter to the `times` loop, so that it became `.times`{`|i| ...`}. Just like `x` stands in for each successive array element in `r.each`{`|x| ...`}, so `i` stands in for each successive value between 0 and 999 in `1000.times`{`|i| ...`}.

Erica: As for your second magic term, the method `each_index` seems to do something similar to `each`. What's the difference?

Carol: Take the line:

```
  r.each_index{|k| r[k] += v[k]*dt}
```

There we want to add to each element of array `r` the corresponding element of array `v`, multiplied by `dt`. However, we cannot just use the `each` method, since in that case we would iterate over the *values* of `r`, and the dummy parameter, defined between vertical bars, will take on the values `r[0]`, `r[1]`, and so on. That would give us no handle on the value of the *index*, which is `0` in the first case, `1` in the second, and so on.

In the print case above, we had no need to know the value of the index of each element of the array that we were printing. But here, the value of the index is needed, otherwise we cannot line up the corresponding elements of `r` and `v`.

Erica: I see. Or at least I think I do. Let me try it out, using irb.

```
|gravity> irb
a = [4, 7, 9]
[4, 7, 9]
a.each{|x| print x, "\n"}
4
7
9
[4, 7, 9]
a.each_index{|x| print x, "\n"}
```

```
0
1
2
[4, 7, 9]
a.each_index{|x| print a[x], "\n"}
4
7
9
[4, 7, 9]
quit
```

Yes, that makes sense.

Dan: Why do we get an echo of the whole array, at the end of each result?

Carol: That's because irb always prints the value of an expression. First the expression is evaluated, and as a side effect the print statements in the expression are executed. But then a value is returned, which turns out to be the array `a` itself. That's not particularly useful here, but in general, it is convenient that irb always gives you the value of anything it deals with, without you having to add print statements everywhere.

12.4 The `map` Method

Dan: What about this mysterious `map` that you are using in line:

```
  a = r.map{|x| -x/r3}
```

Carol: Ah, that is another Lisp like feature, but don't worry about that, since you're not familiar with Lisp. The method `map`, when applied by a given array, returns a new array in which every element is the result of a mapping that is applied to the corresponding element of the old array. That sounds more complicated than it really is. Better to look at an example:

```
|gravity> irb
a = [4, 7, 9]
[4, 7, 9]
a.map{|x| x + 1}
[5, 8, 10]
a.map{|x| 2 * x}
[8, 14, 18]
quit
```

Dan: Ah, now I get it. In the first case, the mapping is simply adding the number one, and indeed, each element of the array gets increased by one. And in the second case, the mapping tells us that any element `x` is doubled to become `2 * x`, and that's exactly what happens.

Carol: Yes, and notice how convenient it is that irb echoes the value of each statement you type. You don't have to write `print a.map{|x| x + 1}`, for example.

So in our case the line

```
  a = r.map{|x| -x/r3}
```

transforms the old array `r` into a new array `a` for which each element gets a minus sign and is divided by `r3`, which is just what we needed.

Erica: Ah, look, you forgot to include the line `a = []`, and it still worked, this time. That must be because now we *are* actually producing a new array `a`, and we are no longer trying to assign values to elements of `a` as we did before.

Carol: That's right! I had not even realized that. Good. One less line to worry about.

Oh, by the way, when you look at books about Ruby, or when you happen to see someone else's code, you may come across the method `Array#collect`. That is just another name for `Array#map`. Both `collect` and `map` are interchangeable terms. This often happens in Ruby: many method names are just an alias for another method name. I guess the author of Ruby tried to please many of his friends, even though they had different preferences.

Erica: I prefer the name `map`, since it gives you the impression that some type of transformation is being performed. As for the word `collect`, it does not suggest much work being done.

Carol: I agree, and that's why I chose to use `map` here.

12.5 Defining a Method

Erica: Carol, you convinced us that we should obey the DRY principle, and indeed, we are no longer repeating ourselves on the level of vector components. But when I look at the last code that you produced, there is still a glaring violation of the DRY principle. Look at the three lines that we use to print the positions and velocities right at the beginning. The very same three lines are used inside the loop, at the end.

Carol: Right you are! Let's do something about that. Time to define a method of our own. Here, this is easy. Let's introduce a method called `print_pos_vel(r,v)`, which prints the position and velocity arrays. It has two arguments, `r` and `v`, the values of the two arrays it should print.

We can write the definition of `print_pos_vel` at the top of the file, and then we can invoke that method wherever we need it; I'll call the file `euler_array_each_def.rb`:

```
include Math

def print_pos_vel(r,v)
  r.each{|x| print(x, "  ")}
  v.each{|x| print(x, "  ")}
  print "\n"
end

r = [1, 0, 0]
v = [0, 0.5, 0]
dt = 0.01

print_pos_vel(r,v)
1000.times{
  r2 = 0
  r.each{|x| r2 += x*x}
  r3 = r2 * sqrt(r2)
  a = r.map{|x| -x/r3}
  r.each_index{|k| r[k] += v[k]*dt}
  v.each_index{|k| v[k] += a[k]*dt}
  print_pos_vel(r,v)
}
```

Erica: Good! I think we can now certify this program as DRY compliant.

Dan: Does it work?

Carol: Ah yes, to be really compliant, it'd better work. Here we go:

```
|gravity> ruby euler_array_each_def.rb | tail -1
7.6937453936572  -6.27772005661599  0.0  0.812206830641815  -0.574200201239989  0.0
```

12.6 The `Array#inject` Method

Dan: I wonder, would it be possible to make the code even shorter?

Erica: Making a code shorter doesn't necessarily make it more readable!

Dan: Sure, but I'm just curious.

Carol: Well, if you want to get fancy, there is an array method called `inject`. It's a strange name for what is something like an accumulation method. Let me show you:

```
|gravity> irb
a = [3, 4, 5]
[3, 4, 5]
a.inject(0){|sum, x| sum + x}
12
a.inject(1){|product, x| product * x}
60
quit
```

Erica: I get the idea. What `inject(p){|y, x| y @ x}` does is to give `y` the initial value `p`, and then for each array component `x`, it applies the `@` operator, whatever it is, to the arguments `y` and `x`.

Carol: Indeed. So this will allow me to make the loop part of the code a bit shorter, in `euler_array_inject1.rb`:

```
include Math

def print_pos_vel(r,v)
  [r,v].flatten.each{|x| print(x, "  ")}
  print "\n"
end

r,v = [[1, 0, 0], [0, 0.5, 0]]
dt = 0.01

print_pos_vel(r,v)
1000.times{
  r2 = r.inject(0){|sum, x| sum + x*x}
  r3 = r2 * sqrt(r2)
  a = r.map{|x| -x/r3}
  r.each_index{|k| r[k] += v[k]*dt ; v[k] += a[k]*dt}
  print_pos_vel(r,v)
}
```

Dan: I see. That got rid of the first line of the previous loop code. Does it work?

Carol: Good point, let's first test it:

```
|gravity> ruby euler_array_inject1.rb | tail -1
7.6937453936572  -6.27772005661599  0.0  0.812206830641815  -0.574200201239989  0.0
```

Same answer as before. So yes, it works.

12.7 Shorter and Shorter

Dan: And above that, you combined the assignment of the position and velocity arrays. I'm surprised that *that* works!

Carol: In general, in Ruby you can assign values to more than one variable in one statement, where Ruby assumes that the values are listed in an array:

```
|gravity> irb
a, b, c = [10, "cat", 3.14]
[10, "cat", 3.14]
a
10
b
"cat"
c
3.14
quit
```

Dan: Oh, and before that, in the `print_pos_vel` method, you've gotten rid of a line as well. What does `flatten` do?

Carol: I takes a nested array, and replaces it by a flat array, where all the components of the old tree structure are now arranged in one linear array. Here's an example:

```
|gravity> irb
[1, [[2, 3], [4,5], 6], 7].flatten
[1, 2, 3, 4, 5, 6, 7]
quit
```

Dan: And then just before the last print statement, you combine two statements into one, using a semicolon. Four little tricks, saving us four lines. I'm impressed!

Carol: Ah, but I can do better! How about this one, `euler_array_inject2.rb`?

```
include Math

def print_pos_vel(r,v)
  [r,v,"\n"].flatten.each{|x| print(x, "  ")}
end

r,v,dt = [[1, 0, 0], [0, 0.5, 0], 0.01]

print_pos_vel(r,v)
1000.times{
  r2 = r.inject(0){|sum, x| sum + x*x}
  r3 = r2 * sqrt(r2)
  a = r.map{|x| -x/r3}
  r.each_index{|k| r[k] += v[k]*dt ; v[k] += a[k]*dt}
  print_pos_vel(r,v)
}
```

12.8 Enough

Dan: Two lines less. You're getting devious! And does it work?

```
|gravity> ruby euler_array_inject2.rb | tail -1
```

I guess not. But how can it produce nothing?

Carol: Beats me. Strange. Let's show a bit more output:

```
|gravity> ruby euler_array_inject2.rb | tail -3
  7.68562253804505  -6.27197741210993  0.0  0.81228556121432  -0.5742644506056  0.
  7.6937453936572  -6.27772005661599  0.0  0.812206830641815  -0.574200201239989
```

Ah, of course, I've been a bit too clever. By adding the return character `\n` character to the same line in the printing method, I have caused an extra two blank spaces to appear in the end. Well, I can get rid of that simply by reversing the order, by printing the blank spaces first. Here is `euler_array_inject3.rb`:

```
include Math

def print_pos_vel(r,v)
  [r,v,"\n"].flatten.each{|x| print("  ", x)}
end

r,v,dt = [[1, 0, 0], [0, 0.5, 0], 0.01]

print_pos_vel(r,v)
1000.times{
  r2 = r.inject(0){|sum, x| sum + x*x}
  r3 = r2 * sqrt(r2)
  a = r.map{|x| -x/r3}
  r.each_index{|k| r[k] += v[k]*dt ; v[k] += a[k]*dt}
  print_pos_vel(r,v)
}
```

and here are the results:

```
|gravity> ruby euler_array_inject3.rb | tail -3
  7.677498893594  -6.26623412376799  0.0  0.812364445105053  -0.574328834193899  0.0
  7.68562253804505  -6.27197741210993  0.0  0.81228556121432  -0.5742644506056  0.0
  7.6937453936572  -6.27772005661599  0.0  0.812206830641815  -0.574200201239989  0.0
```

Same!

Dan: Almost the same: now every line has a few blank spaces at the start.

Carol: Actually, that looks more elegant, doesn't it?

Erica: Frankly, I'm getting a bit tired of shaving lines from codes. Stop playing, you two, and let's move one!

Chapter 13

Overloading the + Operator

13.1 A DRY Version of Modified Euler

Dan: Now how did we get into all this array stuff?

Erica: I wanted to move on to the leapfrog algorithm, but Carol brought up the DRY principle, Don't Repeat Yourself, insisting on first cleaning up the modified Euler code . . .

Carol: . . . which we haven't done yet, but now we're all set to do so! It is just a matter of translating the old file `euler_modified_1000_steps.rb`, introducing our array notation, just as we did for the code in `euler_array_each_def.rb`.

Here it is, in `euler_modified_array.rb`

```
include Math

def print_pos_vel(r,v)
  r.each{|x| print(x, "  ")}
  v.each{|x| print(x, "  ")}
  print "\n"
end

r = [1, 0, 0]
v = [0, 0.5, 0]
dt = 0.01
print_pos_vel(r,v)

1000.times{
  r2 = 0
  r.each{|x| r2 += x*x}
  r3 = r2 * sqrt(r2)
  a = r.map{|x| -x/r3}
  r1 = []
  r.each_index{|k| r1[k] = r[k] + v[k]*dt}
  v1 = []
  v.each_index{|k| v1[k] = v[k] + a[k]*dt}
  r12 = 0
  r1.each{|x| r12 += x*x}
  r13 = r12 * sqrt(r12)
  a1 = r1.map{|x| -x/r13}
  r2 = []
  r1.each_index{|k| r2[k] = r1[k] + v1[k]*dt}
  v2 = []
  v1.each_index{|k| v2[k] = v1[k] + a1[k]*dt}
  r.each_index{|k| r[k] = 0.5 * ( r[k] + r2[k] )}
  v.each_index{|k| v[k] = 0.5 * ( v[k] + v2[k] )}
  print_pos_vel(r,v)
}
```

And before Dan can ask me to do so, let me run it:

```
|gravity> ruby euler_modified_array.rb | tail -1
0.400020239524913  0.343214474344616  0.0  -1.48390077762002  -0.0155803976141248
```

I will also compare it to the previous result:

```
|gravity> ruby euler_modified_1000_steps.rb | tail -1
0.400020239524913  0.343214474344616  0.0  -1.48390077762002  -0.0155803976141248
```

13.2 Not quite DRY yet

Dan: Bravo! Same answers. And yes, the code has become shorter. I like that.

Erica: Hmmm, just making a code shorter does not necessarily make it better. You can use semicolons to put two or three lines on one line, but that doesn't make the code any prettier. Most likely, it will make the code more opaque.

In fact, I'm sorry to say, I don't find the new code easier to read. While the old code was longer, it was very easy to see what happened. But in the new code, there are all those `[k]` clumps floating around . . . I thought the whole point of using arrays was that we could hide the elements of the array!

Carol: To some extent, we have hidden things. The methods map, each and each_index can be attached directly to the arrays themselves, without showing elements. And our use of each in the print statements shows an example where there is no ugly [k] showing up at all. But I agree with you, we should be able to do better.

Erica: Can we really do much better? An array *does* seem to be the natural way to represent a physical vector quantity in a computer program. I've never seen any other way . . . ah, no, I take that back. I once saw a C++ program, where the writer had introduced a special vector class.

Carol: Can you remember what the reason was for doing so?

Erica: I'm not exactly sure now, but it may have had to do with making it easier to add vectors, by overloading the + operator, and that sort of thing.

Carol: That sounds exactly like what we need. If we take a look at the first line in our code that contains one of these ugly [k] notations that you so disliked:

```
r1 = []
r.each_index{|k| r1[k] = r[k] + v[k]*dt}
```

you really would like to write this as

```
r1 = r + v*dt
```

right?

Erica: Yes, that would be great! I would *love* to get rid of all those [k] blocks. In fact, I think that we *should* get rid of them if we want to follow the DRY principle. Look, we have been repeating this [k] thingy three times in one line, in our latest code!

13.3 Array Addition

Carol: Fair enough. Well, it's always a good idea to start simply. The simplest case I can think of is to add two vectors. If we continue to represent them as arrays, we can add arrays a1 and a2 to obtain their sum a as follows:

```
a = []
a1.each_index{|k| a[k] = a1[k] + a2[k]}
```

which includes the declaration, which is necessary if a has not yet been introduced as an array, earlier in the program.

Now what you would like to write is

```
a = a1 + a2
```

without any further complications that include references to elements `[k]` or to methods such as `each_index` or to a declaration of `a`, since after all the addition of two vectors should obviously produce a new vector. Right?

Erica: It sounds too good to be true, to be able to leave out all that crap, and to tell the computer only the minimal information needed, as if you were writing a note for yourself on a scratch pad. Do you really think you can implement all that, and even do away with the need for declarations?

Carol: I think so. First, let's see what happens if we don't make any modification. I must admit, I'm not sure what Ruby will do if we ask it to add two arrays. Well, let's find out:

```
|gravity> irb
a1 = [1, 2, 3]
[1, 2, 3]
a2 = [5, 6, 7]
[5, 6, 7]
a = a1 + a2
[1, 2, 3, 5, 6, 7]
quit
```

Dan: I guess that is *one* way to add two arrays, to just put them end to end, and string all the elements together. And for many applications that might just be the right thing to do, for example, if you have an array of the names of countries in a nation, and you want to add a few more names. But in our case, this is not what we want. We'd better get:

```
a = [1+5, 2+6, 3+7] = [6, 8, 10]
```

Carol: Of course, we can *change* the definition of "+" for array.

Dan: How can you change the definition of a built-in function?

Carol: In Ruby you can change anything, or at least almost anything! But let's take only one step at a time. The simplest and safest way to get the correct addition behavior, is to introduce a new array method. We can call it `Array#plus`, and use it to add two arrays in the way Dan just specified, according to the vector rules that we have in mind for the physical addition of two vectors.

Erica: But how can you add a new method to the `Array` class? Somebody else already has defined the `Array` class for us. I guess we'll have to dig into wherever Ruby is defined, and change the `Array` class definition?

Carol: No digging needed! The neat thing about Ruby, one of the neat things, is that it allows you to augment a class. Even if someone else had defined a class, we can always add a few lines to the class definition, for example, when we want to add a method. The different bits and pieces of the class definition can live in different places. Nothing to worry about!

It should be simple. From what I've seen so far, I guess that Ruby wants us to write something like this, in file `array_try_addition1.rb`:

```
class Array
  def plus(a)
    sum = []
    self.each_index{|k| sum[k] = self[k]+a[k]}
    return sum
  end
end
```

This should add the method `plus`, that is doing the addition, to the other existing methods in the `Array` class. In this way, we don't disturb anything in the `Array` class, so everything should work as it did before. The only difference is that we can now add arrays in the way we intend for vectors.

13.4 Who is Adding What

Erica: That's a lot shorter and simpler than I had expected. It seems to deliver all the three things you promised: it hides the all `[k]` occurrences, it hides the `each_index` and it creates a new vector, so that you don't have to declare anything. And all that in just a few lines!

Dan: Not so fast. I'm not there yet. In fact, I don't understand this at all. To begin with, shouldn't addition have two arguments? You're going to add two vectors, no?

Carol: Yes, but here we're describing addition from the point of view of an array. Given one array, all we have to do is to specify one second array, which I have called `a`, which can then be added to the given array. The given array itself simply goes by the name of `self`, a reserved word that points to the current instance of the class.

Dan: You mean that the `Array` class definition describes arrays in general, but if I deal with a particular array, `a1`, then from the point of view of `a1` itself, `a1` is called `self`?

Carol: Right.

Dan: But we want to get the result `a = a1 + a2`. So from the point of view of `a` there really are two other arrays involved.

Carol: Yes, but at first we only have a1 and a2. That's all we've got, and that's what we have to use to construct a. The way I'm trying to define addition is by starting with one of the two arrays a1, and to define a method that allows a1 to accept a second array a2. So the whole operation can then be written as

```
a1.plus(a2)
```

where a2 is the argument for the method plus that is associated with the class Array of which a1 is an instance. Now this expression will result a new instance of the Array class, and we can assign that new instance to the new variable a by writing:

```
a = a1.plus(a2)
```

Dan: You can't write

```
a = a1 plus a2
```

?

Carol: Sorry, no, you can't; that wouldn't make any sense. What you *can* do in Ruby is leave out the parentheses around the argument of a method. So instead of writing

```
a = a1.plus(a2)
```

you *can* indeed write

```
a = a1.plus a2
```

13.5 The plus Method

Dan: I'll take your word for it. So if we write this, a1 uses its own plus method, and according to the definition you wrote in the Array class, a1 first creates a new array called sum, which is an empty array, specified by writing []. Next it assigns the the value sum[k] = a1[k]+a2[k] to each component [k] of the array sum. That makes sense!

And finally, in the next line you return the value sum, before you reach the end of the method. In that way a receives the value that a1.plus a2 returns, which is the sum of a1 and a2. Okay, I got it now!

Carol: Let's hope it works, after everything I've told you!

```
|gravity> irb
require "array_try_addition1.rb"
true
a1 = [1, 2, 3]
[1, 2, 3]
a2 = [5, 6, 7]
[5, 6, 7]
a = a1.plus a2
[6, 8, 10]
quit
```

Dan: Wonderful! That's just what we ordered.

Erica: A great improvement over the old array addition!

13.6 The + Method

Dan: Still, I can't say I like your new notation. I'm still not happy about the asymmetry. Writing

```
a = a1.plus a2
```

gives the impression that `a1` is charging forward, gobbling up `a2` and then spitting out the result. You told me that we cannot write

```
a = a1 plus a2
```

and I understand that such a statement would have no clear meaning in Ruby. But is there really no way to make the expression more symmetric, rather than making `a1` the predator and `a2` the prey?

Carol: Actually, there is a way to make it at least *look* more symmetric. It is just a form of syntactic sugar, as they call it: a way to let the syntax look more tasty, without really changing the underlying code.

The idea is what is called 'overloading operators'. We can use the `+` symbol, instead of the word `plus`, and we can redefine the meaning of `+` for the case of arrays. This is what I meant when I said earlier that in Ruby you can change almost anything. I have read about that; let me see whether it works. I believe the idea is to write something like this, in file `array_try_addition2.rb`:

```
class Array
  def +(a)
    sum = []
    self.each_index{|k| sum[k] = self[k]+a[k]}
    sum
  end
end
```

13.7 A Small Matter: the Role of the Period

Dan: All you've done is to change `plus` into `+` in the second line. Can that really work?

Erica: There is one more change: you've also left out the word `return` in the third line of the definition.

Carol: Ah yes, most people writing Ruby seem to leave out `return`; it is really not necessary to add that. You just have to remember to let the last line of a definition echo the result you want to return. The result of invoking a method is to return whatever the last line of the definition evaluates to.

And yes, other than that, I've just replaced `plus` by `+`. In fact, in *all* cases where Ruby uses `+`, it is only syntactic sugar for invoking a method that is associated with the left-hand side of the `+` symbol. So even though it *looks* symmetric, it never really has been a symmetric operation.

Dan: But how can it work? I thought that you always needed to write a dot between an object and its method.

Carol: Generally, that is true, and in fact, if you want to, you still can write the `+` operator using a period like normal Ruby methods.

Dan: Let me try:

```
|gravity> irb
2.+ 3
5
8.* 4
32
quit
```

Erica: Wow, surprise. They work just like ordinary Ruby methods.

Dan: Are you sure? Isn't `2.` just translated into `2.0` so that we are only evaluating `2.0 + 3`? Let's check, by adding a space after the periods:

```
|gravity> irb
2. + 3
```

```
5
8. * 4
32
quit
```

Ah, you see, the zero is just added, like in Fortran.

Carol: I don't think so. Let me try a simpler case:

```
|gravity> irb
2.
quit
NoMethodError: undefined method ‘quit’ for 2:Fixnum
from (irb):1
from :0
quit
```

You see: `2.` is *not* translated into `2.0`, but is in fact illegal. Or more accurately, it is okay in Ruby to leave space after the period between an object and its method. Here `irb` is asking for the method name, and doesn't like the fact that I just wanted to quit; `irb` interpreted the `quit` as the method name it was waiting for.

Dan: I agree, we can now be sure that `2.+` really invokes the addition operator of the number 2. Okay, now we know.

Carol: But of course, it is much more intuitively obvious to write `2 + 3` than to write `2.+ 3`. I have mentioned earlier the principle of least surprise, introduced by Matsumoto, the designer of Ruby, as a guide line for the Ruby syntax. Even though in fact Ruby has much in common with Lisp, Matsumoto decided not to use a lisp like notation, in which `2 + 3` would have looked something like `(+ (2, 3))`, a beautifully clear notation once you get used to it, but unlike `2 + 3` not immediately obvious when somebody comes across it for the first time.

Dan: I'd say!

13.8 Testing the + Method

Carol: Well, enough talk: let's test my second version of array addition:

```
|gravity> irb
require "array_try_addition2.rb"
true
a1 = [1, 2, 3]
```

```
[1, 2, 3]
a2 = [5, 6, 7]
[5, 6, 7]
a = a1 + a2
[6, 8, 10]
quit
```

Dan: I like this a whole lot better! And I'm glad Matsumoto did not introduce four parentheses to add two things.

Carol: I like it too, but I'm afraid we can't leave things like this.

Dan: Why not?

Carol: Because we've now been tinkering with the `Array` class, we can no longer use arrays in the standard way. That's why not.

Erica: Ah, you mean that we can no longer concatenate arrays, the way we saw before, using the `+` method. What did we do again? It was something like this:

```
|gravity> irb
a1 = [1, 2, 3]
[1, 2, 3]
a2 = [4, 5, 6]
[4, 5, 6]
a = a1 + a2
[1, 2, 3, 4, 5, 6]
quit
```

Dan: Of course, that no longer will work, when we add our modification:

```
|gravity> irb
require "array_try_addition2.rb"
true
a1 = [1, 2, 3]
[1, 2, 3]
a2 = [4, 5, 6]
[4, 5, 6]
a = a1 + a2
[5, 7, 9]
quit
```

But who cares? I don't expect us to have much use for array concatenation anyway.

Carol: Ah, not so quick. I think you should care a lot! If you use *any* Ruby program, written by someone else, you don't know what that program relies on. Most likely some Ruby programs do rely on the default way of array addition, in the form of concatenation. If you're going to change the default rules, you're likely to invite disaster.

When we introduced the new `plus` method, there was no danger, since we left the existing methods, such as `+`, alone. That's fine. But tinkering with existing methods is simply a bad idea.

13.9 A `Vector` Class

Dan: Is there no way out? I like what we've done, and it would be a pity to give it up, now that we've just figured out how to do it.

Carol: Yes, there is a way. What we want to do is to introduce a new class, similar to the `Array` class, but with a different name and with somewhat different rules. To be precise, we want to define a vector class. Ruby has the convention that class names start with a capital, so a natural name for our new class would be `Vector`.

In principle, we could define our new class from scratch, but it would be a lot easier to use the features that the `Array` class already has. All we really want to do is to tame the `Array` class to behave like proper physical vectors, and we can do this by redefining only some of the array operations, such as adding two arrays, as we have just done, and multiplying an array with a scalar, and probably a few more such operations.

In Ruby, just like in C++ and many other object-oriented languages, we can do this by an approach called *inheritance*. Instead of just defining a new class

```
  class Vector
```

we can write

```
  class Vector < Array
```

which means that the new `Vector` class inherits all the features of the existing class `Array`. The `Vector` class is then called a *subclass* of the `Array` class, and the `Array` class is called a *superclass* of the `Vector` class.

Okay, let me see whether I can redefine the array addition operator, so that vectors can be added in the right way. From what I've seen so far, I guess that Ruby wants us to write something like this, in file `vector_try_addition2.rb`, to replace `array_try_addition2.rb`:

```
class Vector < Array
  def +(a)
    sum = Vector.new
    self.each_index{|k| sum[k] = self[k]+a[k]}
    sum
  end
end
```

This new class definition so far contains only one new method, by the name of `+`, that is doing the addition. Note that I create an empty new vector by writing `Vector.new` instead of `[]`, the notation we used to create a new array. In fact, `[]` is simply syntactic sugar for `Array.new`. So therefore it is a straightforward change to replace it by `Vector.new` as I've done above.

We can use this new class in the same way as we did before, but there is one difference: we have to declare all objects we will play with to be vectors. Before, we declared objects as arrays by using the `[]` notation, which is really a shorthand for `Array[]`. Now we have to specify that they are vectors by writing `Vector[]`. At least I think that's how it works. Let's try:

```
|gravity> irb
require "vector_try_addition2.rb"
true
v1 = Vector[1, 2, 3]
[1, 2, 3]
v2 = Vector[5, 6, 7]
[5, 6, 7]
v = v1 + v2
[6, 8, 10]
quit
```

Dan: That seems to do the right thing. And I guess we have now left the `Array` class alone, without changing it in any way, right?

Carol: Yes, but let us test that as well:

```
|gravity> irb
require "vector_try_addition2.rb"
true
a1 = [1, 2, 3]
[1, 2, 3]
a2 = [5, 6, 7]
[5, 6, 7]
a = a1 + a2
[1, 2, 3, 5, 6, 7]
quit
```

Dan: Good. So we're now playing it safe.

Chapter 14

A `Vector` Class with + and -

14.1 `Vector` Subtraction

Erica: Well, Carol, you've pulled a really neat trick. What a difference, being able to add vectors by writing

```
v = v1 + v2
```

rather than

```
v = []
v1.each_index{|k| v[k] = v1[k] + v2[k]}
```

Dan: Thanks, Carol! You've just made our life a whole lot easier.

Carol: Not quite yet; I'll have to do the same thing for subtraction, multiplication and addition as well. And I'm sure there are a few more things to consider, before we can claim to have a complete `Vector` class. But I, too, am encouraged with the good start we've made!

I'll open a new file `vector_try_add_sub.rb`. First thing to add, after addition, is subtraction. That part is easy:

```
class Vector < Array
  def +(a)
    sum = Vector.new
    self.each_index{|k| sum[k] = self[k]+a[k]}
    sum
  end
  def -(a)
```

```
    diff = Vector.new
    self.each_index{|k| diff[k] = self[k]-a[k]}
    diff
  end
end
```

Erica: So now we can write `v = v1 + v2` and `v = v1 - v2` But what about `v = -v1`? Or even just `v = v1`? Would that work too?

Carol: Good question! Probably not. But before getting into the why not, let's play with `irb` and see what happens:

```
|gravity> irb
require "vector_try_add_sub.rb"
true
v1 = Vector[1, 2, 3]
[1, 2, 3]
v2 = Vector[5, 6, 7]
[5, 6, 7]
v = v1 + v2
[6, 8, 10]
v = v1 - v2
[-4, -4, -4]
v = v1
[1, 2, 3]
v = -v1
NoMethodError: undefined method ‘-@’ for [1, 2, 3]:Vector
from (irb):7
from :0
quit
```

Dan: Huh? A method by the name of a minus sign followed by an `@` symbol? That's the strangest method name I've ever seen. And what can it mean that it is undefined? Should it be defined?

Erica: At least writing `v = v1` worked. So we've come halfway!

Dan: Ah, but would it also work if we would write `v = +v1`? Let me try:

```
|gravity> irb
require "vector_try_add_sub.rb"
true
v1 = Vector[1, 2, 3]
[1, 2, 3]
v2 = Vector[5, 6, 7]
```

```
[5, 6, 7]
v = +v1
NoMethodError: undefined method ‘+@’ for [1, 2, 3]:Vector
from (irb):4
from :0
quit
```

Aha! You see, we're not even half-way yet. Neither of the two work. But it's intriguing that we get a similar error message, this time with a plus sign in front of the mysterious `@` symbol.

14.2 Unary +

Carol: Let me consult the Ruby manual. Just a moment . . . aha, I see! Well, this is perhaps an exception to Ruby's principle of least surprise. The manual tells me that `-@` is the Ruby notation for a unary minus, and similarly, `+@` is the Ruby notation for a unary plus.

Dan: A unary minus?

Carol: Yes, and the word 'un' in unary here means 'one,' like in 'unit.' A unary minus is an operation that has only one operand.

Erica: As opposed to what?

Carol: As opposed to a binary minus. Most normal operations, such as addition and subtraction, as well as multiplication and division, are binary operation. Binary here means two operands. When you use a single plus sign, you add two numbers. Similarly, a minus allows you to subtract two numbers. So when you write `5 - 3 = 2` you are using a binary minus. However, when you first write `x = 5` and then `y = -x`, to give `y` the value `-5`, you are using not a binary minus, but a unary minus. The construction `-x` returns a value that has the value of the variable `x`, but with an additional minus sign.

Dan: So you are effectively multiplying `x` with the number `-1`.

Erica: Or you could say that you are subtracting `x` from `0`.

Carol: Yes, both statements are correct. But rather than introducing multiplication, it is simpler to think only about subtraction. So writing `-x` uses a unary minus, while writing `0-x` uses a binary minus, while both are denoting the same result.

Dan: But why does Ruby use such a complicated symbol, `-@`, for unary minus?

Carol: That symbol is only used when you redefine the symbol.

Here, let me try it out, in a new file `vector_try_unary.rb`. And I may as well start with the unary plus, since that seems the simplest of the two unary operations.

We have just redefined the binary plus as follows:

```
  def +(a)
    sum = Vector.new
    self.each_index{|k| sum[k] = self[k]+a[k]}
    sum
  end
```

We can now use the `+@` symbol to redefine also the unary plus for the same `Vector` class:

```
  def +@
    self
  end
```

When we use it, we don't have to add the `@` symbol, which is only used in the definition, to make the necessary distinction between the unary and binary plus.

Dan: That's it? You just return the vector itself? I guess it makes sense, but it seems almost too simple. Let's try it out:

```
|gravity> irb
require "vector_try_unary.rb"
true
v1 = Vector[1, 2, 3]
[1, 2, 3]
v = +v1
[1, 2, 3]
quit
```

Good! Now what about unary minus?

14.3 Unary -

Carol: What about it?

Dan: My first guess would be to let the method return `-self` but that's too simple, I'm sure . . .

Carol: Yes, that would beg the question! By writing `-self` you are trying to invoke the very method you are trying to define. That certainly won't work. But, hey, remember our friend `map`, which maps an operation on all elements of an array? Well, because the `Vector` class inherits the `Array` class, any method working for an array will work for a vector as well, so here we go:

```
  def -@
    self.map{|x| -x}
  end
```

And here is the reality check:

```
|gravity> irb
require "vector_try_unary.rb"
true
v1 = Vector[1, 2, 3]
[1, 2, 3]
v = -v1
[-1, -2, -3]
quit
```

Dan: It's real. Congratulations!

Erica: Can you compose these operations arbitrarily?

Carol: Of course you can. The syntax is the same, we have only overloaded the + and - operators; the way you can combine them is the same as in normal arithmetic.

Erica: Let me try:

```
|gravity> irb
require "vector_try_unary.rb"
true
v1 = Vector[1, 2, 3]
[1, 2, 3]
v2 = Vector[5, 6, 7]
[5, 6, 7]
v = -((-v1) + (+v2))
NoMethodError: undefined method ‘-@’ for [-1, -2, -3, 5, 6, 7]:Array
from (irb):4
from :0
quit
```

14.4 An Unexpected Result

Dan: So much for normal arithmetic.

Carol: That is *very* unexpected, I must say. What does the error message say? It talks about a very long array, with six components. Wait a minute, it should only talk about vectors. It seems that not all of our vectors are really members of the `Vector` class. Could it be that some of them are still `Array` members?

Erica: Easy to test:

```
|gravity> irb
require "vector_try_unary.rb"
true
v1 = Vector[1, 2, 3]
[1, 2, 3]
v2 = Vector[5, 6, 7]
[5, 6, 7]
v1.class
Vector
v2.class
Vector
(+v2).class
Vector
(-v1).class
Array
quit
```

Carol: Aha! Unexpected, yes, but it all makes sense. For the unary plus method, we just returned `self`, the object itself, which already is a member of the `Vector` class. But the way I wrote the unary minus, things are more tricky:

```
  def -@
    self.map{|x| -x}
  end
```

You see, `self` is a instance of the `Vector` class, which inherits the `Array` class, and thereby inherits all the methods of the `Array` class. But now the question is: what does a method such as `map` do? It is a member of the `Array` class, something that Ruby folks write as `Array#map` in a notation that I find somewhat confusing, but we'll have to get used to it. So, what `Array#map` does, and the only thing it *can* do, is to return an `Array` object.

Erica: And not a `Vector` object. Got it! So all we have to do is to tell the result of the `Array#map` method to become a `Vector`.

But wait a minute, we can't do that. We want to pull off a little alchemy here, turning an `Array` into a `Vector`. But doesn't this mean that the `Array` class has to learn about vectors?

14.5 Converting

Carol: Well, I *may* have a lead here. In Ruby, you will often see something like to_s as a method to write something as a string. Or more precisely, to convert it into a string.

Dan: What does it mean to *convert* something? Can you give a really simple example?

Carol: The simplest example I can think of is the way that integers are being converted into floating point numbers. I'm sure you're familiar with it. If you specify a multiplication like 3.14 * 2 to get an approximate result for 2π, the floating point number 3.14 will try to make the fixed point number 2, in integer, into a floating point number first.

In other words, 3.14, an object that is an instance of the class Float, will try to convert the number 2, an object that is an instance of the class Fixnum, into an instance of the class Float. To say it in simple terms: 3.14 will convert 2 into 2.0 and since it knows how to multiply two floating point numbers, it can then happily go ahead and apply its multiplication method.

Dan: I see. I guess I never worried about what happened when I write something like 3.14 * 2; I just expect 5.28 to come out.

Carol: 6.28.

Dan: Oops, yes, of course. But you see what I mean.

Carol: I agree, normally there is no reason to think about those things, as long as we are using predefined features that hopefully have been tested extensively. But now we are going to define our own objects, vectors, and we'd better make sure that we're doing the right thing.

Erica: You mentioned the to_s method.

Carol: Yes, for each class XXX that has a method to_s defined, as XXX#to_s, we can use to_s to convert an object of class XXX into an object of class String.

Here, let me show you:

```
|gravity> irb
3
3
3.class
Fixnum
3.to_s
"3"
3.to_s.class
String
3.14
3.14
```

```
3.14.class
Float
3.14.to_s
"3.14"
3.14.to_s.class
String
quit
```

Erica: Ah, so the "..." notation already shows that we are dealing with a character string, or string for short, and indeed, the class of "..." is String. That makes sense.

So you want to do something similar with vectors, starting with an array and then converting it into a vector, with a method like to_v.

Carol: Exactly! And I like the name you just suggested. So we have to define a method Array.to_v. Then, once we have such a method, we can use it to create a vector by writing

```
v = [1, 2, 3].to_v
```

14.6 Augmenting the Array Class

Erica: But how can we define to_v? Somebody else already has defined the Array class for us. I guess we'll have to dig into wherever Ruby is defined, and change the Array class definition?

Carol: No digging needed! The neat thing about Ruby, one of the neat things, is that it allows you to augment a class. Even if someone else had defined a class, we can always add a few lines to the class definition, for example, when we want to add a method. The different bits and pieces of the class definition can live in different places. Nothing to worry about!

It should be simple. Let me copy our previous vector_try_unary.rb into a new file vector_try.rb. Hopefully we're getting closer to the real thing!

Here is my first attempt to augment the Array class:

```
class Array
  def to_v
    Vector[*self]
  end
end
```

And now, keeping my fingers crossed:

```
|gravity> irb
require "vector_try.rb"
true
[1, 2, 3].class
Array
[1, 2, 3].to_v.class
Vector
v1 = Vector[1, 1, 1]
[1, 1, 1]
v2 = [1, 2, 3].to_v
[1, 2, 3]
v = v1 + v2
[2, 3, 4]
v.class
Vector
quit
```

Erica: Your hope was justified: `to_v` does indeed seem to produce genuine vectors. How nice, that we have the power to add to the prescribed behavior of the `Array` class!

Dan: It may be nice, but I'm afraid I don't understand yet how `to_v` works. You are returning a new vector, and that new vector should have the same numerical components as the original array, `self`, right? Now what is that little star doing there, in front of `self`?

Carol: Ah, that's a way to liberate the components. We have seen that we can create an array by writing

```
[1, 2, 3]
```

which you can view as a shorthand for

```
Array[1, 2, 3]
```

where the `Array[]` method receives a list of components and returns an array that contains those components.

Now for our vector class we can similarly write:

```
Vector[1, 2, 3]
```

in order to create vector `[1, 2, 3]`.

Now, let me come back to your question. If I start with an `Array` object `[1, 2, 3]`, which internally is addressed by `self`, and if I then were to write:

```
Vector[self]
```

that would be translated into

```
Vector[[1, 2, 3]]
```

Dan: I see: that would be a vector with one component, where the one component would be the array `[1, 2, 3]`. Got it. So we have to dissolve one layer of square brackets, effectively.

Carol: Indeed. And here is where the `*` notation comes in. Let me show you:

```
|gravity> irb
a = [1, 2, 3]
[1, 2, 3]
b = [a]
[[1, 2, 3]]
c = [*a]
[1, 2, 3]
quit
```

14.7 Fixing the Bug

Erica: So now we can go back and fix the bug in our unary minus.

Carol: Ah, yes, that's how we got started. Okay, we had in file `vector_try_unary.rb`:

```
  def -@
    self.map{|x| -x}
  end
```

In our new file, `vector_try.rb`, I can now make this:

```
  def -@
    self.map{|x| -x}.to_v
  end
```

Dan: Shall we repeat our old trial run? Here we go:

```
|gravity> irb
require "vector_try.rb"
true
v1 = Vector[1, 2, 3]
[1, 2, 3]
v2 = Vector[5, 6, 7]
[5, 6, 7]
v = -((-v1) + (+v2))
[-4, -4, -4]
quit
```

Great! Okay, now we have really covered a complete usage of `+` and `-` for vectors, the unary and binary forms for each of them.

Chapter 15

A Complete `Vector` Class

15.1 `Vector` Multiplication

Carol: Which means it is time to move on to multiplication. But here we have another problem: there is multiplication and then there is multiplication.

Dan: You mean?

Carol: We can multiply a vector with a scalar number; for a vector

```
v = [1, 2, 3]
```

we would like to see multiplication by two giving us:

```
2 * v = [2, 4, 6]
```

But in addition (no pun intended) we want to form an inner product of two vectors. In particular, we would like to get the square of the length of a vector by forming the inner product with itself:

```
v * v = 2*2 + 4*4 + 6*6 = 56
```

Or more generally, for

```
w = [10, 10, 10]
```

we want to be able to take the inner product of `v` and `w` to give us:

```
v * w = 2*10 + 4*10 + 6*10 = 120
```

We could of course define different method names for these two operations, like `multiply_scalar` and `inner_product`, but something tells me that we will be happier using `*` for both.

Dan: Certainly I will be happier that way!

Carol: Well, how about this, in `vector_try.rb`:

```
  def *(a)
    if a.class == Vector              # inner product
      product = 0
      self.each_index{|k| product += self[k]*a[k]}
    else
      product = Vector.new            # scalar product
      self.each_index{|k| product[k] = self[k]*a}
    end
    product
  end
```

Time for a workout:

```
|gravity> irb
require "vector_try.rb"
true
v1 = Vector[1, 2, 3]
[1, 2, 3]
v2 = Vector[5, 6, 7]
[5, 6, 7]
v1 * 3
[3, 6, 9]
v1 * v1
14
v1 * v2
38
v1 * v2 * 3
114
v1 * 3 * v2
114
quit
```

15.2 An Unnatural Asymmetry

Dan: So far so good, but why do you put the number 3 only at the end and in the middle, why not in front? That would seem more natural! Let me try:

```
|gravity> irb
require "vector_try.rb"
true
v1 = Vector[1, 2, 3]
[1, 2, 3]
v1 * 3
[3, 6, 9]
3 * v1
TypeError: Vector can’t be coerced into Fixnum
from (irb):4:in ‘*’
from (irb):4
from :0
quit
```

Hmmm, I guess not.

Carol: Yes, it would be more natural, but it doesn’t work. Do you see why? Remember, when we write `v2 * 3` we invoke the `*` method of the vector object `v2`. I understand that you would like to write `3 * v2`, but in that case you have a problem: you would be trying to invoke the `*` method of the number object 3 . . . .

Erica: That’s all and well, as a formal explanation, but if you can only write `[1, 2, 3] * 3` and never `3 * [1, 2, 3]`, I’m not very happy with it. The whole point was to make our life easier, and to make the software notation more natural, more close to what you would write with pen and paper . . .

Dan: I agree. Half a solution is often worse than no solution at all. Perhaps we should just return to our earlier component notation.

Carol: I can’t argue with that. But I’m not yet willing to give up. I wonder whether there is not some sort of way out. Let me see whether I can find something.

15.3 `Vector` Division

Erica: For now at least, let’s finish the job we started . . .

Carol: . . . which means that we have to add a division method as well. In the case of division, we only have to deal with scalar division.

Erica: Ah, yes, of course, you have an inner product, but not an inner quotient.

Carol: Well, not completely ‘of course’, there is something called ‘geometric algebra’. If you’re curious, you can search for it on the internet.

Dan: I’m not curious, let’s move on.

Carol: Okay, so I will punish attempts to divide two vectors by letting an error message appear. In Ruby you can use the command `raise` to halt execution and display an error message:

```
  def /(a)
    if a.class == Vector
      raise
    else
      quotient = Vector.new           # scalar quotient
      self.each_index{|k| quotient[k] = self[k]/a}
    end
    quotient
  end
```

A quick check:

```
|gravity> irb
require "vector_try.rb"
true
v1 = Vector[1, 2, 3]
[1, 2, 3]
v2 = Vector[5, 6, 7]
[5, 6, 7]
v1 / 3.0
[0.333333333333333, 0.666666666666667, 1.0]
v2 / 0.5
[10.0, 12.0, 14.0]
v1 / v2
RuntimeError:
from ./vector_try.rb:41:in ‘/’
from (irb):6
from :0
quit
```

You see, we got a run time error, because the `raise` statement had the effect of raising an error, as it is called in Ruby.

15.4 The Score: Six to One

Erica: Let me see whether I can sum up what we've learned. We've successfully implemented seven legal operations, unary/binary addition/multiplication, scalar/vector multiplication and scalar division.

Of these seven, six are okay. It is only with scalar multiplication that we encounter a problem, namely a lack of symmetry.

Dan: Can you define "okay" for the other six?

Erica: I will be explicit. I will use a notation where `v1` and `v2` are vectors, and `s` is a scalar number, which could be either an integer or a floating point number. I will call something "well defined" if it gives a specific answer, and not an error message.

Here is the whole list of the six operations that I am now considering "okay":

1) unary plus: `+v1` is okay, because it is well defined.

2) unary minus: `-v1` is okay, because it is well defined.

3) binary plus: `v1 + v2` is okay, because both `v1 + v2` and `v2 + v1` are well defined, and both give the exact same answer, as they should.

4) binary minus: `v1 - v2` is okay, because both `v1 - v2` and `v2 - v1` are well defined, and both give the exact opposite answer, as they should.

5) vector times: `v1 * v2` is okay, because both `v1 * v2` and `v2 * v1` are well defined, and both give the exact same answer, as they should.

6) scalar slash: `v1 / s` is okay, because it is well defined. The alternative, `s / v1` is not defined, and will give an error message. However, that's perfectly okay too, because we have decided that we don't allow division by vectors.

So far, the score is six to zero, and we seem to be winning. The problem is that we are losing in the case of number 7:

7) scalar times: `v1 * s` is okay, because it is well defined. The alternative, `s * v1` *should* give the same answer too, but unfortunately, in our implementation it is not defined, and in fact, as we have seen, it gives an error message.

Dan: Thanks, Erica, that's a very clear summary. So all we have to do, to save the day, is to repair `s * v1`, so that it gives a well defined result, and in fact the same result as `v1 * s`.

15.5 A Solution

Carol: Ah, of course! How stupid, we should have thought about it right away!

Dan: About what?

Carol: Class augmentation! We can just augment the number classes, telling them how to multiply with vectors! Just as we have already augmented the array class, telling it how to convert an array into a vector.

Dan: I don't see that. What exactly are you trying to repair?

Carol: Let's go back to square one. We wanted to make our scalar-vector multiplication symmetric. The problem was, when you have a vector `v` and you want to write

```
3 * v
```

you are effectively asking the number 3 to provide a method that knows how to handle multiplication of 3 with a vector. But the poor number 3 has never heard of vectors! No wonder we got an error message. Let me show again explicitly what the error message says:

```
|gravity> irb
require "vector_try.rb"
true
v = Vector[1, 2, 3]
[1, 2, 3]
v * 3
[3, 6, 9]
3 * v
TypeError: Vector can't be coerced into Fixnum
from (irb):4:in '*'
from (irb):4
from :0
quit
```

You see: it tells us that the number 3 expects to make a multiplication with something like another number. It tries to convert a vector into such an other number, but doesn't succeed.

Erica: So, what we really would like to do is to augment the rules for the class `Fixnum`, which is the class of integers, to explain how to multiply with an object of class `Vector`. We can then do the same for the class `Float`, the class of floating point numbers.

15.6 Augmenting the `Fixnum` Class

Carol: Indeed. Well, there is no stopping us now to add to the behavior of the `Fixnum` class! Here we go, in `vector.rb`:

```
class Fixnum
  alias :original_mult :*
  def *(a)
    if a.class == Vector
      a*self
    else
```

```
      original_mult(a)
    end
  end
end
```

Dan: Whoa! That's quite a bit more complicated than I had expected. You must have been reading Ruby manuals lately!

Carol: I admit, I did. Here is what happens. If we start with the number 3, and take a vector `v`, and write

```
3 * v
```

then the `*` method of the fixed point number 3 first checks whether `v` is a `Vector`. In our case, it is, so then it returns the result

```
v * self
```

which in the case of 3 is simply

```
v * 3
```

and that is something we have already defined, in the `Vector` class.

If, on the other hand we write anything else, such as

```
3 * 8.5
```

then the `*` method of the fixed point number 3 finds that `8.5` is *not* a `Vector`, so it applies the original definition of multiplication, to the number `8.5`, as it should.

Dan: So the `alias` notion means that whatever the original `*` did is now assigned to `original_mult` instead?

Carol: Exactly. Writing `alias :x :y` means that `x` becomes an alias for `y`. Or so the theory goes. Let's see what the practice tells us:

```
|gravity> irb
require "vector.rb"
true
v = [1, 2, 3].to_v
[1, 2, 3]
v * 3
[3, 6, 9]
```

```
(v * 3).class
Vector
3 * v
[3, 6, 9]
(3 * v).class
Vector
quit
```

15.7 Augmenting the Float Class

Dan: Not bad! Great! `3 * v` now produces the exact same thing as `v * 3`. Congratulations! Another hope fulfilled. And I guess you should do the same thing for floating point numbers, right?

Carol: Right. All very similar:

```
class Float
  alias :original_mult :*
  def *(a)
    if a.class == Vector
      a*self
    else
      original_mult(a)
    end
  end
end
```

This time my hope for success will be quite justified:

```
|gravity> irb
require "vector.rb"
true
v = [1, 2, 3].to_v
[1, 2, 3]
v * 3.14
[3.14, 6.28, 9.42]
(v * 3.14).class
Vector
3.14 * v
[3.14, 6.28, 9.42]
(3.14 * v).class
Vector
quit
```

And indeed, it all comes out the way it should. So I declare victory: I'm very pleased with our vector class.

15.8 Vector Class Listing

Erica: Let's get the whole listing on the screen. Here it is. In fact, it is shorter than I thought, for all the things it does:

```
class Vector < Array
  def +(a)
    sum = Vector.new
    self.each_index{|k| sum[k] = self[k]+a[k]}
    sum
  end
  def -(a)
    diff = Vector.new
    self.each_index{|k| diff[k] = self[k]-a[k]}
    diff
  end
  def +@
    self
  end
  def -@
    self.map{|x| -x}.to_v
  end
  def *(a)
    if a.class == Vector              # inner product
      product = 0
      self.each_index{|k| product += self[k]*a[k]}
    else
      product = Vector.new            # scalar product
      self.each_index{|k| product[k] = self[k]*a}
    end
    product
  end
  def /(a)
    if a.class == Vector
      raise
    else
      quotient = Vector.new           # scalar quotient
      self.each_index{|k| quotient[k] = self[k]/a}
    end
    quotient
  end
end

class Array
  def to_v
    Vector[*self]
  end
end

class Fixnum
  alias :original_mult :*
  def *(a)
    if a.class == Vector
      a*self
    else
      original_mult(a)
    end
  end
end

class Float
  alias :original_mult :*
  def *(a)
    if a.class == Vector
      a*self
    else
      original_mult(a)
    end
  end
end
```

15.9 Forward Euler in Vector Form

Dan: The whole point of this long exercise was to make our codes more readable. Here is one of the array versions we made of our first code, the forward Euler version, in `euler_array_each_def.rb`:

```
include Math

def print_pos_vel(r,v)
  r.each{|x| print(x, "  ")}
  v.each{|x| print(x, "  ")}
  print "\n"
end

r = [1, 0, 0]
v = [0, 0.5, 0]
dt = 0.01

print_pos_vel(r,v)
1000.times{
  r2 = 0
  r.each{|x| r2 += x*x}
  r3 = r2 * sqrt(r2)
  a = r.map{|x| -x/r3}
  r.each_index{|k| r[k] += v[k]*dt}
  v.each_index{|k| v[k] += a[k]*dt}
  print_pos_vel(r,v)
}
```

What would that look like in vector form?

Carol: To start with, we'll have to `require` the file `vector.rb`. We also have to convert the arrays into vectors, using `.to_v`. Then, in the loop, we can use our vector versions of addition, unary minus, multiplication and division. Let me write it all in file `euler_vector.rb`:

```
require "vector.rb"
include Math

def print_pos_vel(r,v)
  r.each{|x| print(x, "  ")}
  v.each{|x| print(x, "  ")}
  print "\n"
end

r = [1, 0, 0].to_v
v = [0, 0.5, 0].to_v
dt = 0.01

print_pos_vel(r,v)
1000.times{
  r2 = r*r
  r3 = r2 * sqrt(r2)
  a = -r/r3
  r += v*dt
  v += a*dt
  print_pos_vel(r,v)
}
```

Dan: Wait a minute, we haven't define `+=` yet!

Carol: Ah, you see, that's a nice feature of operator overloading in Ruby: once you redefine `+` the same redefinition applies to derivative expressions such as `+=`.

Dan: That's nice, if it really works. Let's compare the run with the old results:

```
|gravity> ruby euler_array_each_def.rb | tail -1
7.6937453936572  -6.27772005661599  0.0  0.812206830641815  -0.574200201239989  0.
```

```
|gravity> ruby euler_vector.rb | tail -1
7.6937453936572  -6.27772005661599  0.0  0.812206830641815  -0.574200201239989  0.
```

Wonderful. And I must say, the code does look a lot cleaner now.

Erica: Definitely. This *is* a lot more pretty. What a difference! None of all those ugly `[k]` occurrences anymore. I think it was worth all the work we put in, defining a `Vector` class.

Chapter 16

A Matter of Speed

16.1 Slowdown by a Factor Two

Dan: Clean and pretty, sure, but is it fast enough? Let's compare the speed of the old code and the new vector code. I wonder whether all this fancy vector stuff is affecting execution speed.

Carol: We can use the `time` command, to find out how much time a code spends before finishing. We can redirect the standard output to `/dev/null`, literally a null device, effectively a waste basket. This is a Unix way of throwing the ordinary output away, the output that appears on the standard output channel. In that way, we are left only with output that appears on the standard error channel, such as timing information provided by the `time` command.

Let me run all three forward Euler codes, the original version we wrote, the array version, and the vector version:

```
|gravity> time ruby euler.rb > /dev/null
0.052u 0.003s 0:00.06 83.3%0+0k 0+0io 0pf+0w
|gravity> time ruby euler_array_each_def.rb > /dev/null
0.118u 0.001s 0:00.13 84.6%0+0k 0+0io 0pf+0w
|gravity> time ruby euler_vector.rb > /dev/null
0.205u 0.004s 0:00.22 90.9%0+0k 0+0io 0pf+0w
```

Dan: It seems that using our nifty vector notation, we get a penalty in speed of a factor two. But before jumping to conclusions, can we check once more that we are really doing the same calculations in all three cases?

Carol: Sure, here are the last lines of all three calculations:

```
|gravity> ruby euler.rb | tail -1
7.6937453936572  -6.27772005661599  0.0  0.812206830641815  -0.574200201239989  0.
|gravity> ruby euler_array_each_def.rb | tail -1
7.6937453936572  -6.27772005661599  0.0  0.812206830641815  -0.574200201239989  0.
|gravity> ruby euler_vector.rb | tail -1
7.6937453936572  -6.27772005661599  0.0  0.812206830641815  -0.574200201239989  0.
```

Dan: Good! And now I have another question. Presumably even the first version is slower than an equivalent code written in Fortran or C or C++. I would like to know how much slower.

16.2 A C Version of Forward Euler

Carol: That's a good idea. Given that our first code, `euler.rb`, is rather simple, it should be easy to translate it into C. And to make the time measurement a bit more precise, I'll make the time step a hundred times smaller, so that we let the code make a hundred thousand steps.

Finally, we are interested now in execution speed, and for now at least we don't worry about the cost of frequent outputs. After all, when we switch to the real N-body problem, for $N > 2$, the total costs will be dominated by inter-particle force calculations, not by print statements.

Here is the C version, for 100,000 steps, with only one output at the end, in file `euler_100000_steps.c`:

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

int main()
{
    double r[3], v[3], a[3];
    double dt = 0.0001;
    int ns, k;

    r[0] = 1;
    r[1] = 0;
    r[2] = 0;
    v[0] = 0;
    v[1] = 0.5;
    v[2] = 0;

    for (ns = 0; ns < 100000; ns++){
        double r2 = r[0]*r[0] + r[1]*r[1] + r[2]*r[2];
        for (k = 0; k < 3; k++)
            a[k] = - r[k] / (r2 * sqrt(r2));
        for (k = 0; k < 3; k++){
            r[k] += v[k] * dt;
            v[k] += a[k] * dt;
        }
    }
    printf("%.15g %.15g %.15g %.15g %.15g %.15g\n",
          r[0], r[1], r[2], v[0], v[1], v[2]);
}
```

Let me compile and run it:

```
|gravity> gcc -o euler_100000_steps euler_100000_steps.c -lm
|gravity> time euler_100000_steps
0.292716737827072 0.382907748579753 0 -1.56551896976935 -0.313957063866527 0
0.065u 0.001s 0:00.08 75.0%0+0k 0+0io 0pf+0w
```

Dan: Pretty fast indeed! But you really should compile it with the optimizer flag `-O`, to bring out the real speed that C is capable off.

Carol: Good point! Here goes:

```
|gravity> gcc -O -o euler_100000_steps euler_100000_steps.c -lm
|gravity> time euler_100000_steps
0.292716737827197 0.382907748579793 0 -1.56551896976913 -0.313957063866306 0
0.036u 0.000s 0:00.03 100.0%0+0k 0+0io 0pf+0w
```

Dan: Even faster. One and a half times faster, it seems, but we can't really be sure, given the limited accuracy of the timing output. In any case, the C code really flies!

Erica: Note that the output values differ in the last few significant digits. That must be because optimization causes a different order of arithmetic operations, which means that the roundoff errors are different too. Since we are taking a hundred thousand steps, it is perhaps not so strange that we are losing several digits in accuracy.

Dan: Time to compare these results with the Ruby code. I have this sinking feeling that it will be muuuuuch slower.

16.3 A Simple Ruby Version

Carol: You are probably right. Here is the simplest forward Euler version, also with only one output statement at the end, in `euler_100000_steps.rb`:

```
include Math

x = 1
y = 0
z = 0
vx = 0
vy = 0.5
vz = 0
dt = 0.0001

100000.times{
  r2 = x*x + y*y + z*z
  r3 = r2 * sqrt(r2)
  ax = - x / r3
  ay = - y / r3
  az = - z / r3
  x += vx*dt
  y += vy*dt
  z += vz*dt
  vx += ax*dt
  vy += ay*dt
  vz += az*dt
}
print(x, "  ", y, "  ", z, "  ")
print(vx, "  ", vy, "  ", vz, "\n")
```

and here is the timing result:

```
|gravity> time ruby euler_100000_steps.rb
0.292716737826681  0.38290774857968  0.0  -1.56551896976999  -0.313957063867402  0
1.859u 0.007s 0:02.14 86.4%0+0k 0+0io 0pf+0w
```

Dan: A dramatic difference. As far as we can see here, Ruby is at least thirty times slower than C!

Erica: That's a bit of a shock.

Carol: Yes, I knew that Ruby would be slow, but I didn't expect it to be quite that slow. Well, at least C and Ruby give the same output results, apart from the last few digits, which change anyway in C when we switch on optimization, as we have seen. So as far as what it is the same between the optimized and unoptimized C results, Ruby produces that part exactly.

Dan: Let's complete the exercise and make similar versions for arrays and vectors.

16.4 A Ruby Array Version

Carol: Here is the array version, in `euler_array_100000_steps.rb`:

```
include Math

def print_pos_vel(r,v)
  r.each{|x| print(x, "  ")}
  v.each{|x| print(x, "  ")}
  print "\n"
end

r = [1, 0, 0]
v = [0, 0.5, 0]
dt = 0.0001

100000.times{
  r2 = 0
  r.each{|x| r2 += x*x}
  r3 = r2 * sqrt(r2)
  a = r.map{|x| -x/r3}
  r.each_index{|k| r[k] += v[k]*dt}
  v.each_index{|k| v[k] += a[k]*dt}
}
print_pos_vel(r,v)
```

and here is what timing gives:

```
|gravity> time ruby euler_array_100000_steps.rb
0.292716737826681  0.38290774857968  0.0  -1.56551896976999  -0.313957063867402  0.0
3.471u 0.047s 0:05.21 67.3%0+0k 0+0io 0pf+0w
```

Dan: So adding arrays let Ruby slow down by a factor two.

16.5 A Ruby Vector Version

Carol: And here is the vector version, in `euler_vector_100000_steps.rb`:

```
require "vector.rb"
include Math

def print_pos_vel(r,v)
  r.each{|x| print(x, "  ")}
  v.each{|x| print(x, "  ")}
  print "\n"
end

r = [1, 0, 0].to_v
v = [0, 0.5, 0].to_v
dt = 0.0001

100000.times{
  r2 = r*r
  r3 = r2 * sqrt(r2)
  a = -r/r3
  r += v*dt
  v += a*dt
}
print_pos_vel(r,v)
```

and here is how slow it runs:

```
|gravity> time ruby euler_vector_100000_steps.rb
0.292716737826681  0.38290774857968  0.0  -1.56551896976999  -0.313957063867402  0
9.061u 0.080s 0:11.47 79.6%0+0k 0+0io 0pf+0w
```

Dan: And now we're losing yet another factor of three. That's pretty terrible!

16.6 More Timing Precision

Carol: Let's try to get a bit more timing precision. Instead of taking a hundred thousands steps, we can take a million steps, to make the timing comparison more accurate. You can guess the names of the files I will create: `euler_1000000_steps.c`, `euler_1000000_steps.rb`, `euler_array_1000000_steps.rb` and `euler_vector_1000000_steps.rb`.

Here are the results:

```
|gravity> gcc -o euler_1000000_steps euler_1000000_steps.c -lm
|gravity> time euler_1000000_steps
0.519970642634788 -0.376817992041735 0 1.17126787143565 0.114700879740638 0
0.459u 0.000s 0:00.48 93.7%0+0k 0+0io 0pf+0w
```

```
|gravity> gcc -O -o euler_1000000_steps euler_1000000_steps.c -lm
|gravity> time euler_1000000_steps
0.519970642633723 -0.376817992041925 0 1.17126787143747 0.114700879739195 0
0.357u 0.000s 0:00.37 94.5%0+0k 0+0io 0pf+0w
```

```
|gravity> time ruby euler_1000000_steps.rb
0.519970642634004  -0.376817992041834  0.0  1.17126787143698  0.114700879739653  0.0
16.985u 0.064s 0:18.62 91.5%0+0k 0+0io 0pf+0w
```

```
|gravity> time ruby euler_array_1000000_steps.rb
0.519970642634004  -0.376817992041834  0.0  1.17126787143698  0.114700879739653  0.0
37.293u 0.144s 0:41.11 91.0%0+0k 0+0io 0pf+0w
```

```
|gravity> time ruby euler_vector_1000000_steps.rb
0.519970642634004  -0.376817992041834  0.0  1.17126787143698  0.114700879739653  0.0
95.045u 0.521s 1:55.34 82.8%0+0k 0+0io 0pf+0w
```

16.7 Conclusion

Dan: This confirms our earlier conclusions. At least on this particular computer, that we are now using to do some speed tests, the unoptimized C version takes 50% more time than the optimized version, the simplest Ruby version takes about 50 times more time, the Ruby array version about 100 times more, and finally the Ruby vector version takes more than 250 times more time than the optimized C version.

Carol: But even so, for short calculations, who cares if a run takes ten millisecond or a few seconds? I certainly like the power of Ruby in giving us vector classes, and a lot more goodies. We have barely scratched the surface of all the power that Ruby can give us. You should see what we can do when we really start to pass blocks to methods and . . .

Dan: . . . and then we will start drinking a lot of coffee, while waiting for results when we begin to run 100-body experiments! Is there no way to speed up Ruby calculations?

Carol: There is. By the time we use 100 particles, we are talking about $10^2.10^2 = 10^4$ force calculations for every time step. This means that the calculation of the mutually accelerations will take up almost all of the computer time. What we can do is write a short C code for computing the accelerations. It is possible to invoke such a C code from within a Ruby code. In that way, we can leave most of the Ruby code unchanged, while gaining most of the C speed.

Erica: I certainly like the flexibility of a high-level language like Ruby, at least for writing a few trial versions of a new code. In order to play around, Ruby is a lot more fun and a lot easier to use than C or C++ or Fortran. After we have constructed a general N-body code that we are really happy with, we can always translate part of it into C, as Carol just suggested.

Or, if really needed to gain speed, we could even translate the whole code into C. Translating a code will always take far less time than developing a code in the first place. And is seems pretty clear to me that development will be faster in Ruby.

Dan: I'm not so sure about all that. In any case, we got started now with Ruby, so let us see how far we get. But if and when we really get bogged down by the lack of speed of Ruby, we should not hesitate to switch to a more efficient language.

Carol: Fair enough! Let's continue our project, and apply our vector formalism to second-order integration schemes.

Chapter 17

Modified Euler in Vector Form

17.1 An Easy Translation

Dan: Now that we have a vector version of forward Euler, it's time to clean up our modified Euler code as well.

Carol: That will be an easy translation. I will start by copying the old code from `euler_modified_array.rb` into a new file, `euler_modified_vector_try1.rb`. All we have to do is to translate the code from array notation to vector notation. Same as what we did with `euler_vector.rb`. Here it is:

```
require "vector.rb"
include Math

def print_pos_vel(r,v)
  r.each{|x| print(x, "  ")}
  v.each{|x| print(x, "  ")}
  print "\n"
end

r = [1, 0, 0].to_v
v = [0, 0.5, 0].to_v
dt = 0.01
print_pos_vel(r,v)

1000.times{
  r2 = r*r
  r3 = r2 * sqrt(r2)
  a = -r/r3
  r1 = r + v*dt
  v1 = v + a*dt
  r12 = r1*r1
  r13 = r12 * sqrt(r12)
  a1 = -r1/r13
  r2 = r1 + v1*dt
  v2 = v1 + a1*dt
  r = 0.5 * ( r + r2 )
  v = 0.5 * ( v + v2 )
  print_pos_vel(r,v)
}
```

Erica: What a relief! The lines are shorter, there are fewer lines, but what is most important: the lines are easy to understand, with a direct correspondence between code and math.

Let's trace our history, in this regard. We started off writing with pen and paper:

$$\mathbf{r}_1 = \mathbf{r} + \mathbf{v}dt \qquad (17.1)$$

In our first code this became:

```
  x1 = x + vx*dt
  y1 = y + vy*dt
  z1 = z + vz*dt
```

Then in our array code it became

```
  r1 = []
  r.each_index{|k| r1[k] = r[k] + v[k]*dt}
```

and finally, in our vector code, we wrote:

```
  r1 = r + v*dt
```

which is very close indeed to what we started out with:

$$\mathbf{r}_1 = \mathbf{r} + \mathbf{v}dt \qquad (17.2)$$

Dan: It was a lot of work, but now that we got the vector class, I must admit that the code looks a lot more readable. So I guess this will make life a lot easier for us. But before we move on, does it give the correct answers?

Carol: Here's the old result, from the array code:

```
|gravity> ruby euler_modified_array.rb | tail -1
0.400020239524913  0.343214474344616  0.0  -1.48390077762002  -0.0155803976141248  0.0
```

and here's what our new vector code gives:

```
|gravity> ruby euler_modified_vector_try1.rb | tail -1
0.400020239524913  0.343214474344616  0.0  -1.48390077762002  -0.0155803976141248  0.0
```

17.2 Variable Names

Dan: Good! I'm happy.

Erica: But I'm not, at least not completely. Look, in the code we are using the variable `r2` in two very different ways. Early on, we use it to hold the value of r^2, the square of the original variable $\mathbf{r}$, defined as the inner product $r^2 = \mathbf{r} \cdot \mathbf{r}$. But later, toward the end of the loop, we use the same variable to hold value of $\mathbf{r}_{i+2,p}$, the predicted value of $\mathbf{r}_{i+2}$.

I guess Ruby doesn't mind that we assign completely different values, even with different types, first a scalar, then a vector. But I sure do mind! And someone else reading our code from scratch is likely to be confused.

Carol: You have a point there. Okay, how about calling the initial position $\mathbf{r}_0$ instead of $\mathbf{r}$? That is more consistent anyway. We can then use the variable name `r0` instead of `r` for the initial vector, and the scalar value of its square will then become `r02`. So there will be no possible confusion anymore! Here is the new listing, in file `euler_modified_vector_try2.rb`:

```
require "vector.rb"
include Math

def print_pos_vel(r,v)
  r.each{|x| print(x, "  ")}
  v.each{|x| print(x, "  ")}
  print "\n"
end
```

```
r = [1, 0, 0].to_v
v = [0, 0.5, 0].to_v
dt = 0.01
print_pos_vel(r,v)

1000.times{
  r02 = r*r
  r03 = r02 * sqrt(r02)
  a = -r/r03
  r1 = r + v*dt
  v1 = v + a*dt
  r12 = r1*r1
  r13 = r12 * sqrt(r12)
  a1 = -r1/r13
  r2 = r1 + v1*dt
  v2 = v1 + a1*dt
  r = 0.5 * ( r + r2 )
  v = 0.5 * ( v + v2 )
  print_pos_vel(r,v)
}
```

and here is the test:

```
|gravity> ruby euler_modified_vector_try2.rb | tail -1
0.400020239524913  0.343214474344616  0.0  -1.48390077762002  -0.0155803976141248
```

17.3 Consistency

Erica: Yes, that's better, and you are no longer using the same variable name for two different things. But you haven't quite done what you said you would do, namely calling the initial position $\mathbf{r}_0$ instead of $\mathbf{r}$. You have only assigned the square of $\mathbf{r}$ to r_0^2, or `r02` in the code.

Carol: That's because I wanted to continue using the original variables `r` and `v`, to keep track of the evolving code. The alternative would have been to call the running variables `r0` and `v0`, but that would be misleading, as if the particle would come back to the original position each time.

Erica: How about a compromise? We can keep the original variables `r` and `v`, but convert them to `r0` and `v0` at the beginning of the loop. Let me try this, in file `euler_modified_vector_try3.rb`:

```
require "vector.rb"
include Math

def print_pos_vel(r,v)
  r.each{|x| print(x, "  ")}
  v.each{|x| print(x, "  ")}
  print "\n"
end

r = [1, 0, 0].to_v
v = [0, 0.5, 0].to_v
dt = 0.01
print_pos_vel(r,v)

1000.times{
  r0 = r
  v0 = v

  r02 = r0*r0
  r03 = r02 * sqrt(r02)
  a0 = -r0/r03
  r1 = r0 + v0*dt
  v1 = v0 + a0*dt

  r12 = r1*r1
  r13 = r12 * sqrt(r12)
  a1 = -r1/r13
  r2 = r1 + v1*dt
  v2 = v1 + a1*dt

  r = 0.5 * ( r0 + r2 )
  v = 0.5 * ( v0 + v2 )
  print_pos_vel(r,v)
}
```

and let me test it right away:

```
|gravity> ruby euler_modified_vector_try2.rb | tail -1
0.400020239524913  0.343214474344616  0.0  -1.48390077762002  -0.0155803976141248  0.0
```

Dan: Sure, that is more consistent, but you've just made the code two lines longer! In fact, five lines longer, if you count the three blank lines. Why did you add blank lines?

Erica: I'm not really worried about the extra two lines of code. What's much more important is that in this new notation we can see clearly that we have a new target of attack for the DRY principle. Look, the two blocks of code, that I have highlighted by place blank lines around them, are nearly identical!

17.4 A Method Returning Multiple Values

Carol: Right you are! This calls for a new method. Here, let me try, in file `euler_modified_vector.rb`:

```
require "vector.rb"
include Math

def print_pos_vel(r,v)
  r.each{|x| print(x, "  ")}
  v.each{|x| print(x, "  ")}
  print "\n"
end

def step_pos_vel(r,v,dt)
  r2 = r*r
  r3 = r2 * sqrt(r2)
  a = -r/r3
  [r + v*dt, v + a*dt]
end

r = [1, 0, 0].to_v
v = [0, 0.5, 0].to_v
dt = 0.01
print_pos_vel(r,v)

1000.times{
  r1, v1 = step_pos_vel(r,v,dt)
  r2, v2 = step_pos_vel(r1,v1,dt)
  r = 0.5 * ( r + r2 )
  v = 0.5 * ( v + v2 )
  print_pos_vel(r,v)
}
```

and I'll test it right away:

```
|gravity> ruby euler_modified_vector.rb | tail -1
0.400020239524913  0.343214474344616  0.0  -1.48390077762002  -0.0155803976141248
```

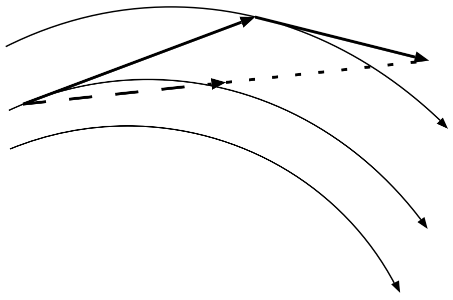
Figure 17.1: Two-step implementation of the modified Euler algorithm

---

Dan: I'm glad it works, but how does it work? It seems that your new method `step_pos_vel` returns an array.

Carol: Ah, yes. In Ruby you can return more than one value simultaneously, and the way to do that is to list them in an array. Then, when you invoke the method, you can just list the variables to which you want to assign the array values, in the same order. Very lazy, very simple, and in a way, you might say, following the principle of least surprise.

Dan: Well, yes, once you realize what is happening.

Erica: And it is quite elegant, I must say. I begin to like Ruby more and more! The inner loop now has become almost a mathematical formula, rather than a piece of computer code. You can read it aloud: step forward, step again, and then average the values from the beginning and the end, both for position and velocity, and print the results. Wonderful!

17.5 Simplification

Erica: Before we move on, there is something that bothers me, which I noticed as soon as we translated the modified Euler scheme into vector notion, in `euler_modified_vector_try1.rb`. I had not noticed any problem when we first wrote the much longer component-based version, but when it became so compact, I realized that we are being clumsy.

Carol: How so?

Erica: Look, we have effectively implemented figure 17.1, right? Let me sketch it here again:

We take two steps forward, in order to compute a single improved step.

However, we started off that whole discussion, way back when, we the simpler figure 17.2. Let me draw that one again too:

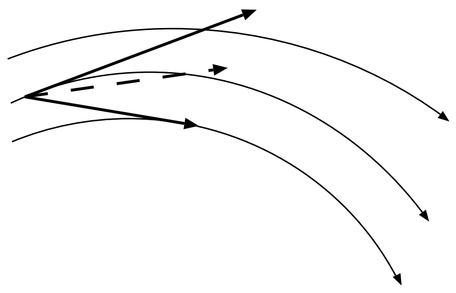

Figure 17.2: One-step implementation of the modified Euler algorithm

You see, in that original figure, we average two steps, in order to compute an improved step.

Carol: But you needed to compute the upper step, before you could compute the lower step, so really both ways of drawing the picture involve two steps.

Erica: True, but to me at least the original figure suggests a somewhat simpler procedure.

Dan: I don't care to quibble about philosophy of aesthetics. Here is the inner loop of the first vector version that we wrote for modified Euler:

```
1000.times{
  r2 = r*r
  r3 = r2 * sqrt(r2)
  a = -r/r3
  r1 = r + v*dt
  v1 = v + a*dt
  r12 = r1*r1
  r13 = r12 * sqrt(r12)
  a1 = -r1/r13
  r2 = r1 + v1*dt
  v2 = v1 + a1*dt
  r = 0.5 * ( r + r2 )
  v = 0.5 * ( v + v2 )
  print_pos_vel(r,v)
}
```

Where do you think you can simplify things?

Erica: Let me copy that file, `euler_modified_vector_try1.rb`, to `euler_modified_vector_try4.rb` and let me see whether I can change the loop, to make it look more like my understanding of the original figure, the second one above:

```
1000.times{
  r2 = r*r
  r3 = r2 * sqrt(r2)
  a = -r/r3
  r1 = r + v*dt
  r12 = r1*r1
  r13 = r12 * sqrt(r12)
  a1 = -r1/r13
  r += v*dt + 0.5*a*dt*dt
  v += 0.5*(a + a1)*dt
  print_pos_vel(r,v)
}
```

Dan: That sure looks a lot simpler. Does it give the same answer?

```
|gravity> ruby euler_modified_vector_try4.rb | tail -1
0.400020239524793  0.34321447434461  0.0  -1.48390077762024  -0.0155803976143043  0.0
```

Carol: And so it does, almost. Since some of the additions and multiplications and such are done in a different order, round-off errors may prevent us from getting the exact same results, but this is certainly close enough. And yes, I admit, this code is quite a bit simpler.

Erica: Not only that, you can now understand the mathematical structure better. The increments in position and velocity, in the last two lines, are just Taylor series expansions, up to terms that contain the accelerations. In the case of the position, the acceleration term is second order in `dt` and so the original value of the acceleration is good enough. In the case of the velocity, we need more precision, so we take the average of the forward and backward Euler values.

Carol: On the other hand, the inner loop in our code in `euler_modified_vector.rb` was even shorter. Here it is:

```
1000.times{
  r1, v1 = step_pos_vel(r,v,dt)
  r2, v2 = step_pos_vel(r1,v1,dt)
  r = 0.5 * ( r + r2 )
  v = 0.5 * ( v + v2 )
  print_pos_vel(r,v)
}
```

Dan Yes, but at the expense of introducing an extra method, making the whole code longer again.

Erica: I guess there is a trade off here. Introducing an extra method gives the elegance of seeing two steps being taken explicitly, which shortening the code as I just did brings out the Taylor series character of the result of averaging two steps.

Carol: I agree. Well, we've learned a lot, and I am actually happy to have several versions. In general, there is never just one right solution for any question concerning software writing!

Chapter 18

Leapfrog

18.1 Interleaving Positions and Velocities

Carol: Erica, we now have a second-order algorithm, modified Euler, but you mentioned an other one, quite a while ago, that you seemed to prefer.

Erica: Yes, the *leapfrog* algorithm, a nice and simple scheme. I just learned about that in class, so it is still fresh in my memory.

Dan: What a strange name. Does it let particles jump around like frogs?

Carol: Or like children jumping over each other?

Erica: Something like that, I guess. I never thought about the meaning of the name, apart from the fact that something is leaping over something, as we will see in a moment. The algorithm is used quite widely, although it has different names in different fields of science. In stellar dynamics you often hear it called leapfrog, but in molecular dynamics it is generally called the Verlet method, and I'm sure there must be other names in use in other fields.

Here is the idea. Positions are defined at times $t_i, t_{i+1}, t_{i+2}, \ldots$, spaced at constant intervals Δt, while the velocities are defined at times halfway in between, indicated by $t_{i-1/2}, t_{i+1/2}, t_{i+3/2}, \ldots$, where $t_{i+1} - t_{i+1/2} = t_{i+1/2} - t_i = \Delta t/2$.

It is these positions and velocities that 'leap over' each other. The leapfrog integration scheme reads:

$$\begin{aligned} \mathbf{r}_i &= \mathbf{r}_{i-1} + \mathbf{v}_{i-1/2}dt \\ \mathbf{v}_{i+1/2} &= \mathbf{v}_{i-1/2} + \mathbf{a}_i dt \end{aligned} \qquad (18.1)$$

Note that the accelerations **a** are defined only on integer times, just like the positions, while the velocities are defined only on half-integer times. This makes

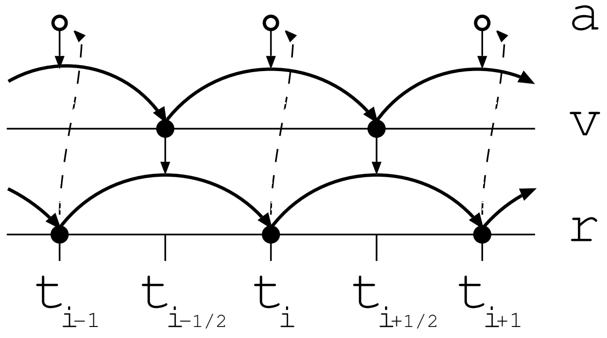

Figure 18.1: With the leapfrog algorithm, when we update a position value to a new point in time, we use the information from the velocity value in between the old and the new point in time. This is indicated by the vertical downward arrows. Similarly, when we update a velocity value to a new point in time, we use the information from the acceleration value in between the old and the new point in time. Each acceleration value is computed directly from the position value, using Newton's law of gravity. This is indicated with the dashed upward pointing arrows.

sense, given that the acceleration on one particle depends only on its position with respect to all other particles, and not on its or their velocities. To put it in mathematical terms, for many situations in physics the acceleration depends on velocity and position, as $\mathbf{a}(\mathbf{r},\mathbf{v})$. The motion of an electron in a magnetic field is one example, with the Lorentz force being velocity dependent. And in any situation in which there is friction, the friction is typically stronger for higher velocities. However, in the case of Newtonian gravity, the velocity dependence is absent: $\mathbf{a}(\mathbf{r},\mathbf{v}) = \mathbf{a}(\mathbf{r})$.

Carol: I like the nice symmetric look of Eq. (18.1). But how do you get started? If you know the positions and velocities at time t_i, how are you going to construct the velocities at time $t_{i+1/2}$?

Erica: The simplest way to extrapolate the velocities from t_i to $t_{i+1/2}$ is by using a Taylor series. and the simplest nontrivial Taylor series is the one that takes only one term beyond the initial value. It turns out that such utter simplicity is already enough!

Concretely, we can start with the initial conditions $\mathbf{r}_0$ and $\mathbf{v}_0$, and take the first term in the Taylor series expansion to compute the first leap value for $\mathbf{v}$:

$$\mathbf{v}_{1/2} = \mathbf{v}_0 + \mathbf{a}_0 \Delta t/2. \tag{18.2}$$

We are then ready to the first line of apply Eq. (18.1) to compute the new position $\mathbf{r}_1$, using the first leap value for $\mathbf{v}_{1/2}$. Next we compute the acceleration

$\mathbf{a}_1$, using Newton's law of gravitation, and this enables us to compute the second leap value, $\mathbf{v}_{3/2}$, using the second line of apply Eq. (18.1). In this way we just march on.

Carol: And when you want to stop, or pause in order to do a printout, you can again construct a Taylor series in order to synchronize the positions and velocities, I presume? If you make frequent outputs, you'll have to do a lot of Tayloring around. I wonder whether that doesn't affect accuracy. Can you estimate the errors that will be introduced that way?

Erica: Ah, the beauty here is: you do not introduce any extra errors! In fact, what we usually do is *never* use the half-integer values for the velocity in any explicit way. Here, let me rewrite the basic equations of the algorithm, in such a way that position and velocity remain synchronized, both at the beginning and at the end of each step.

$$\begin{aligned} \mathbf{r}_{i+1} &= \mathbf{r}_i + \mathbf{v}_i dt + \mathbf{a}_i (dt)^2/2 \\ \mathbf{v}_{i+1} &= \mathbf{v}_i + (\mathbf{a}_i + \mathbf{a}_{i+1}) dt/2 \end{aligned} \tag{18.3}$$

Dan: That looks totally different.

Erica: Ha, but looks deceive! Notice that the increment in $\mathbf{r}$ is given by the time step multiplied by $\mathbf{v}_i + \mathbf{a}_i dt/2$, effectively equal to $\mathbf{v}_{i+1/2}$. Similarly, the increment in $\mathbf{v}$ is given by the time step multiplied by $(\mathbf{a}_i + \mathbf{a}_{i+1})/2$, effectively equal to the intermediate value $\mathbf{a}_{i+1/2}$. In conclusion, although both positions and velocities are defined at integer times, their increments are governed by quantities approximately defined at half-integer values of time.

18.2 Time Symmetry

Dan: I'm still not quite convinced that Eq. (18.1) and Eq. (18.4) really express the same integration scheme.

Erica: An interesting way to see the equivalence of the two descriptions is to note the fact that the first two equations are explicitly time-reversible, while it is not at all obvious whether the last two equations are time-reversible. For the two systems to be equivalent, they'd better share this property. Let us check this, for both cases.

Carol: Eq. (18.1) indeed looks pretty time symmetric. Whether you jump forward or backward, in both cases you use the same middle point to jump over. So when you first jump forward and then jump backward, you come back to the same point.

Erica: Yes, but I would like to prove that, too, in a mathematical way. It is all too easy to fool yourself with purely language-based analogies.

Dan: Spoke the true scientist!

Carol: Well, I agree. In computer science too, intuition can lead you astray quite easily. How do you want to check this?

Erica: Let us first take one step forward, taking a time step $+dt$, to evolve $\{\mathbf{r}_i, \mathbf{v}_{i-1/2}\}$ to $\{\mathbf{r}_{i+1}, \mathbf{v}_{i+1/2}\}$. We can then take one step backward, using the same scheme, taking a time step of $-dt$, back in time. Clearly, after these two steps the time will return to the same value since $t_i + dt - dt = t_i$.

We now have to inspect where the final positions and velocities $\{\mathbf{r}_f(t = i), \mathbf{v}_f(t = i - 1/2)\}$ are indeed equal to their initial values $\{\mathbf{r}_i, \mathbf{v}_{i-1/2}\}$. Here is the calculation. First we apply the first line of Eq. (18.1) to compute $\mathbf{r}_f$ as the result of the step back in time, and then we again apply the first line of Eq. (18.1), to compute the forward step, and we see that indeed $\mathbf{r}_f = \mathbf{r}_i$. Next we apply the second line of Eq. (18.1) two times, to find that $\mathbf{v}_f = \mathbf{v}_i$. Here is the whole derivation:

$$\begin{aligned}
\mathbf{r}_f &= \mathbf{r}_{i+1} - \mathbf{v}_{i+1/2}dt \\
&= \left[\mathbf{r}_i + \mathbf{v}_{i+1/2}dt\right] - \mathbf{v}_{i+1/2}dt \\
&= \mathbf{r}_i
\end{aligned}$$

$$\begin{aligned}
\mathbf{v}_f &= \mathbf{v}_{i+1/2} - \mathbf{a}_i dt \\
&= \left[\mathbf{v}_{i-1/2} + \mathbf{a}_i dt\right] - \mathbf{a}_i dt \\
&= \mathbf{v}_{i-1/2}
\end{aligned}$$

Carol: Can't argue with that! This is a crystal clear derivation. In an almost trivial way, we can see clearly that time reversal causes both positions and velocities to return to their old values, not only in an approximate way, but exactly. This has amazing consequences! When we write a computer program for the leapfrog algorithm, we can evolve forward a thousand time steps and then evolve backward for the same length of time. Although we will make integration errors at every step, and although the errors will get compounded, all those errors will exactly cancel each other.

Don: Amazing indeed, but I would be really amazed if the same time symmetry would hold for that other set of equations, Eq. (18.4), that don't look time symmetric at all!

Erica: Yes, that's where the real fun comes in. The derivation is a bit longer, but equally straightforward, and the steps are all the same. Here it is:

$$\begin{aligned}
\mathbf{r}_f &= \mathbf{r}_{i+1} - \mathbf{v}_{i+1}dt + \mathbf{a}_{i+1}(dt)^2/2 \\
&= \left[\mathbf{r}_i + \mathbf{v}_i dt + \mathbf{a}_i(dt)^2/2\right] - \left[\mathbf{v}_i + (\mathbf{a}_i + \mathbf{a}_{i+1})dt/2\right]dt + \mathbf{a}_{i+1}(dt)^2/2
\end{aligned}$$

$$= \mathbf{r}_i$$

$$\begin{aligned}
\mathbf{v}_f &= \mathbf{v}_{i+1} - (\mathbf{a}_{i+1} + \mathbf{a}_i)dt/2 \\
&= [\mathbf{v}_i + (\mathbf{a}_i + \mathbf{a}_{i+1})dt/2] - (\mathbf{a}_{i+1} + \mathbf{a}_i)dt/2 \\
&= \mathbf{v}_i
\end{aligned}$$

In this case, too, we have exact time reversibility. Even though not at all obvious at first, as soon as we write out the effects of stepping forward and backward, the cancellations become manifest.

18.3 A Vector Implementation

Dan: Okay, I'm convinced now that Eq. (18.4) does the right thing. Let's code it up, and for convenience, let me write down the equations again:

$$\begin{aligned}
\mathbf{r}_{i+1} &= \mathbf{r}_i + \mathbf{v}_i dt + \mathbf{a}_i (dt)^2/2 \\
\mathbf{v}_{i+1} &= \mathbf{v}_i + (\mathbf{a}_i + \mathbf{a}_{i+1})dt/2 \qquad (18.4)
\end{aligned}$$

Obviously, this is the better set of expressions to use. It is more convenient than Eq. (18.1) since in the above equations we can forget about any leaping and frogging, and just move from time t_i to time t_{i+1}, with both positions and velocities.

Let me see whether I'm getting the hang of using vectors now. I will put it in file `leapfrog_try1.rb`:

```
require "vector.rb"
include Math

def print_pos_vel(r,v)
  r.each{|x| print(x, "  ")}
  v.each{|x| print(x, "  ")}
  print "\n"
end

r = [1, 0, 0].to_v
v = [0, 0.5, 0].to_v
dt = 0.01
print_pos_vel(r,v)

1000.times{
  r2 = r*r
```

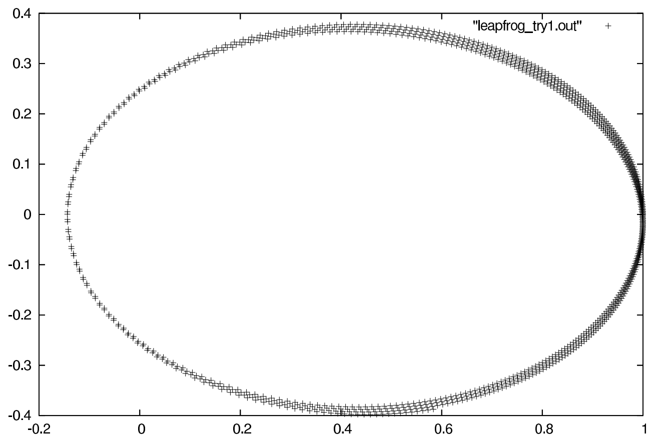

Figure 18.2: First attempt at leapfrog integration, with step size $dt = 0.01$.

```
  r3 = r2 * sqrt(r2)
  a = -r/r3
  v += 0.5*a*dt
  r += v*dt
  r2 = r*r
  r3 = r2 * sqrt(r2)
  a = -r/r3
  v += 0.5*a*dt
  print_pos_vel(r,v)
}
```

and let me make a picture right away, in figure 18.2:

```
|gravity> ruby leapfrog_try1.rb > leapfrog_try1.out
```

Carol: Now that is a clear improvement over modified Euler. Presumably both schemes are second-order, but the orbit integration is clearly more accurate in the case of the leapfrog. Modified Euler gave figure **??** for the same time step size. In fact, our leapfrog is almost as good as Modified Euler for a ten times smaller time step, as given in figure 10.5, in the sense that the orbit does not drift away.

Erica: That is in fact an essential part of the leapfrog algorithm. If it would drift in one direction, and if you would then play time backward, it would have to drift in the other direction – which means it would not be time symmetric. So because the leapfrog is time symmetric, it is impossible for the orbits to drift!

18.4 Saving Some Work

Dan: Ah, I just noticed something. In my leapfrog implementation, I compute the acceleration at the end of the loop, and then at the beginning of the next loop, I calculate the exact same acceleration once again. Since the position `r` does not change between the two calculations, the value of the acceleration `a` is bound to be the same. That's a waste of computing time!

Carol: Well, for the two-body problem, we don't have to worry too much about exactly how many milliseconds of computer time we are spending.

Erica: True, but when we go to a thousand-body problem, this will become an issue. Good point, Dan, why don't you leave one of the acceleration calculations out from the loop.

Dan: The question is, which one. If I leave out the first one, the acceleration is not yet defined, when the loop gets transversed for the very first time. But if I leave out the second one, I cannot calculate the value of the velocity at the end of the loop.

Hmmm. The second acceleration calculation is clearly essential. But . . . , aha, I see! I can take out the first acceleration calculation and place it *before* the loop. That way, the length of the computer program does not change. However, inside the loop unnecessary calculations are no longer being done.

Here is the new version, in file `leapfrog_try2.rb`:

```
require "vector.rb"
include Math

def print_pos_vel(r,v)
  r.each{|x| print(x, "  ")}
  v.each{|x| print(x, "  ")}
  print "\n"
end

r = [1, 0, 0].to_v
v = [0, 0.5, 0].to_v
dt = 0.01
print_pos_vel(r,v)

r2 = r*r
r3 = r2 * sqrt(r2)
a = -r/r3
```

```
1000.times{
  v += 0.5*a*dt
  r += v*dt
  r2 = r*r
  r3 = r2 * sqrt(r2)
  a = -r/r3
  v += 0.5*a*dt
  print_pos_vel(r,v)
}
```

Let me check first to see that we get the same result. The first code gives:

```
|gravity> ruby leapfrog_try1.rb | tail -1
0.583527377458303  -0.387366076048216  0.0  1.03799194001953  0.167802127213742  0
```

and the second version gives:

```
|gravity> ruby leapfrog_try2.rb | tail -1
0.583527377458303  -0.387366076048216  0.0  1.03799194001953  0.167802127213742  0
```

Good.

Carol: Let's check whether you really made the computation faster. We can redirect the standard output to `/dev/null`, literally a null device, effectively a waste basket, which is a Unix way of throwing the results away. That way, we are left only with output that appears on the standard error channel, such as timing information provided by the `time` command.

The first code gives:

```
|gravity> time ruby leapfrog_try1.rb > /dev/null
0.232u 0.000s 0:00.24 95.8%0+0k 0+0io 0pf+0w
```

and the second version gives:

```
|gravity> time ruby leapfrog_try2.rb > /dev/null
0.196u 0.001s 0:00.21 90.4%0+0k 0+0io 0pf+0w
```

Dan: Indeed, a bit faster. If all the computer time would have been spend on acceleration calculation, things would have sped up by a factor two, but of course, that is not the case, so the speed increase should be quite a bit less. This looks quite reasonable.

18.5 The DRY Principle Once Again

Carol: But we can make it even more clear, and we can make the loop even shorter, with the help of our old friend, the DRY principle. Look, the calculation for the acceleration, before and in the loop, contains the exact same three lines. Those lines really ask to be encapsulated in a method. Let me do that, in file `leapfrog.rb`:

```
require "vector.rb"
include Math

def print_pos_vel(r,v)
  r.each{|x| print(x, "  ")}
  v.each{|x| print(x, "  ")}
  print "\n"
end

def acc(r)
  r2 = r*r
  r3 = r2 * sqrt(r2)
  -r/r3
end

r = [1, 0, 0].to_v
v = [0, 0.5, 0].to_v
dt = 0.01
print_pos_vel(r,v)

a = acc(r)

1000.times{
  v += 0.5*a*dt
  r += v*dt
  a = acc(r)
  v += 0.5*a*dt
  print_pos_vel(r,v)
}
```

and as always, I'll test it:

```
|gravity> ruby leapfrog.rb | tail -1
0.583527377458303  -0.387366076048216  0.0  1.03799194001953  0.167802127213742  0.0
```

Chapter 19

Time Reversibility

19.1 Long Time Behavior

Dan: I must agree, that is all very nice and clean. But let's get back to the behavior of the two second-order algorithms that we have coded up so far. Time symmetry is supposed to prevent a long-term drift. I'd like to test that a bit more.

Let me take the modified Euler code, copying it from `euler_modified_vector.rb` to `euler_modified_long_time.rb`. I will let the code run ten times longer, by changing the loop defining line to:

```
10000.times{
```

Carol: I'm glad you're getting the hang of using long names. Thank you!

Dan: My pleasure. But see, I did still abbreviate a bit: I could have left the word *vector* in, but that really would have made the name too long, for my taste.

On a more important topic, I really don't like having different files lying around that are almost the same, except for just one extra 0 in one line.

Carol: We'll have to do something about that. I had already been thinking about introducing command line arguments.

Erica: What does that mean?

Carol: We really would like to specify the number of steps on the command line, as an argument. It would be much better if we could take the program `euler_modified_vector.rb` and run it for 10,000 steps, simply by invoking it as

```
|gravity> ruby euler_modified_vector.rb -n 10000
```

to indicate that now we want to take that many steps, or probably even better

```
|gravity> ruby euler_modified_vector.rb -t 100
```

to indicate that we want to run for 100 time units.

Dan: I would like that much better! Let's put that on our to-do list. But for now, let me finish my long time behavior test. I'll write `leapfrog_long_time.rb`, by modifying `leapfrog.rb` in the same way, to take 10,000 steps:

```
10000.times{
```

Our expectation would be that modified Euler will completely screw up, while the leapfrog will keep behaving relatively well. Let's see what will happen!

First I will make a picture of the long run for modified Euler, in figure 19.1:

```
|gravity> ruby euler_modified_long_time.rb > euler_modified_long_time.out
```

Next, I will make a picture of the long run for our leapfrog, in figure 19.2:

```
|gravity> ruby leapfrog_long_time.rb > leapfrog_long_time.out
```

19.2 Discussing Time Symmetry

Carol: Your expectation was right, to some extent. Modified Euler is almost literally screwing up: the orbit gets wider and wider. In contrast, the leapfrog orbit keeps the same size, which is better for sure, but why does the orbit rotate?

Erica: Well, why not? A time symmetric code cannot spiral out, since such a motion would increase the size of the orbit. If an algorithm lets an orbit grow in one direction in time, it lets the orbit grow when applied to the other direction in time as well, as so it would not be time symmetric. However, if an orbit rotates clockwise in one direction in time, you might expect the orbit to rotate counter-clockwise in the other direction in time. So time reversal will just map a leftward rotation of the whole orbit figure into a rightward rotation, and similarly rightward into leftward

Dan: I don't get that. What's so different about expanding and rotating?

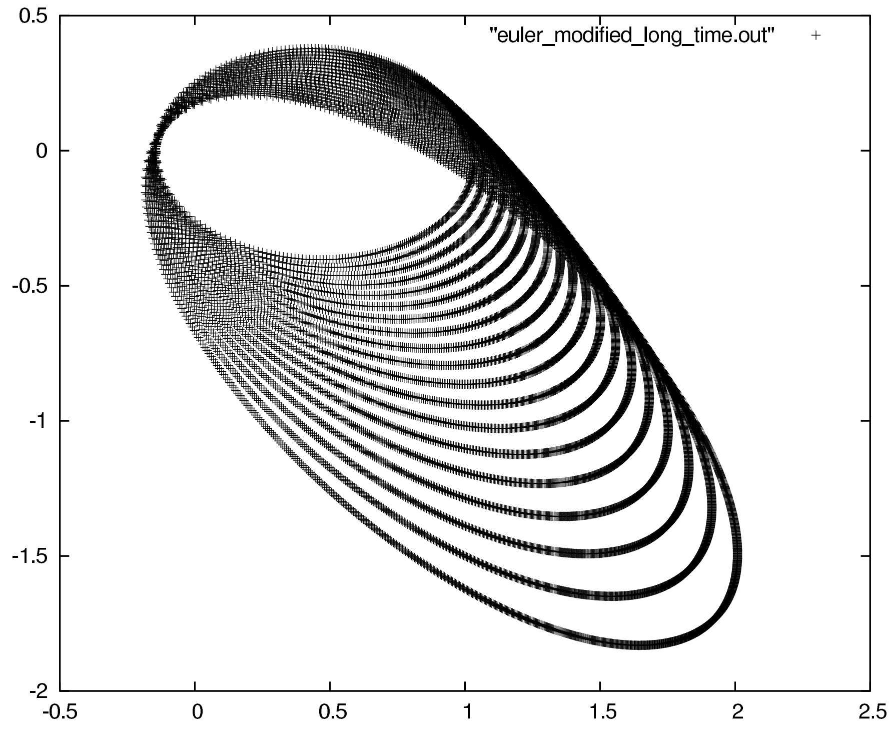

Figure 19.1: Long time integration till $t = 100$, with the modified Euler algorithm, and step size $dt = 0.01$.

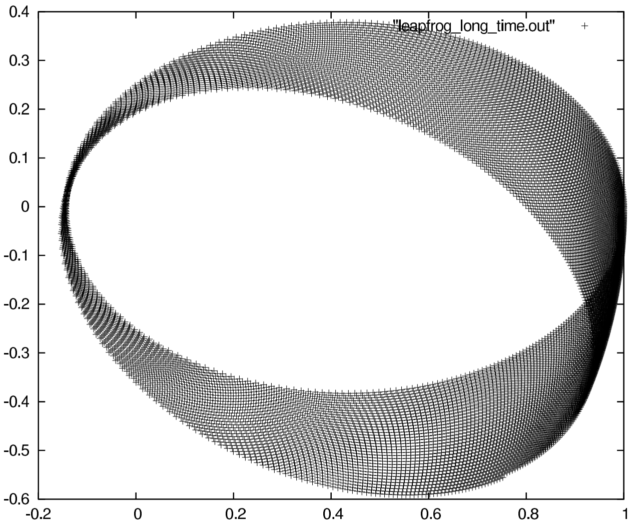

Figure 19.2: Long time integration till $t = 100$, with the leapfrog algorithm, and step size $dt = 0.01$.

Erica: The key point is that we already have a sense of direction, in our elliptic Kepler orbit. Our star moves counter-clockwise along the ellipse, and we see that the leapfrog lets the whole ellipse slowly rotate clockwise. This means that if we let our star move in the other direction, clockwise, then the leapfrog would let the whole ellipse turn slowly in counter-clockwise direction. So the leapfrog algorithm would remain time symmetric: revolve within the orbit in one direction, and the whole orbit rotates; then revolve back into the other direction and the orbit shifts back again, until it reaches the original position.

However, during the course of one revolution, the orbit neither shrinks nor expands. Since there is no prefered direction, inwards or outwards, there is nothing for the leapfrog algorithm to capitalize on. It it were to make an error in one direction in time, say expanding the orbit, it would have to make the same error when going backward in time. So after moving forward and backward in time, during both moves the orbit would have expanded, and there is no way to get back to the original starting point. In other words, that would violate time symmetry.

19.3 Testing Time Symmetry

Dan: Hmmmm. I guess. Well, let's first see how well this time symmetry idea pans out in practice. Clearly, nothing stops us from running the code backward. After taking 10,000 steps forward, we can reverse the direction by simply changing the sign of the time step value. I will do that, and I will omit the print statement in the forward loop, so that we only get to see the backward trajectory. If I would print everything on top of each other, we probably wouldn't see what was going on.

I will call the new code `leapfrog_backward.rb`, which is the same as the old code `leapfrog_long_time.rb`, except that I have replaced the original loop by the following two loops:

```
10000.times{
  v += 0.5*a*dt
  r += v*dt
  a = acc(r)
  v += 0.5*a*dt
}

dt = -dt
10000.times{
  v += 0.5*a*dt
  r += v*dt
  a = acc(r)
  v += 0.5*a*dt
  print_pos_vel(r,v)
}
```

I will plot the backward trajectory in figure 19.3:

```
|gravity> ruby leapfrog_backward.rb > leapfrog_backward.out
```

Carol: Figure 19.3 looks exactly the same as figure 19.2!

Erica: Ah, yes, but that's precisely the point. The stars are retracing their steps so accurately, we can't see the difference!

Dan: Let's check how close the stars reach their point of departure, after their long travel:

```
|gravity> ruby leapfrog_backward.rb | tail -1
0.999999999999975  -1.06571782648723e-12  0.0  2.12594872261995e-12  0.500000000000013  0.0
```

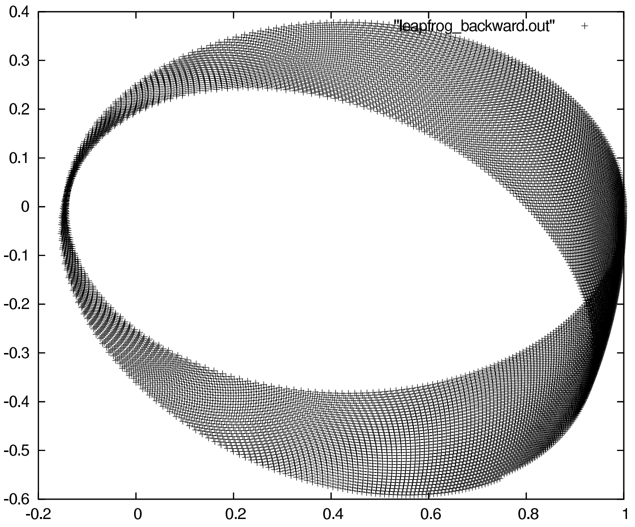

Figure 19.3: Time reversed version of the long time integration with the leapfrog algorithm, from $t = 100$ back to $t = 0$, and step size $dt = -0.01$.

Wow, that's very close to the initial position and velocity, which we specified in our code to be:

```
r = [1, 0, 0].to_v
v = [0, 0.5, 0].to_v
```

19.4 Two Ways to Go Backward

Carol: But shouldn't the velocity have the opposite sign, now that we're going backward?

Dan: No, I've gone back in time, using negative time steps, while leaving everything else the same, including the sign of the velocity. I could instead have reversed the direction of the velocity, while leaving the time step the same. That would mean that time would keep flowing forward, but the stars would move in the opposite direction, from time $t = 100$ to $t = 200$. Let me try that too, why not, in `leapfrog_onward.rb`. This code is the same as `leapfrog_backward.rb`, with the only difference being the one line in between the two loops, which now reads:

```
v = -v
```

In this case, the final position and velocity are:

```
|gravity> ruby leapfrog_onward.rb | tail -1
0.999999999999975  -1.06571782648723e-12  0.0  -2.12594872261995e-12  -0.500000000000013  -0
```

Carol: Indeed, now the velocity is reversed, while reaching the same point. Great, thanks!

Erica: It is remarkable how close we come to the starting point. And yet, it is not exactly the starting point.

Carol: The small deviations must be related to roundoff. While the algorithm itself is strictly time symmetric, in the real world of computing we typically work with double precision numbers of 64 bits. This means that floating point numbers have a finite precision, and that any calculation in floating point numbers will be rounded off to the floating point number that is closest to the true result. Since the rounding off process is not time symmetric, it will introduce a slight time asymmetry.

19.5 Testing a Lack of Time Symmetry

Dan: Before we move on, I'd like to make sure that the rival of the leapfrog, good old modified Euler, is really not time symmetric.

I will do the same as what we did for the leapfrog. I will call the new code `euler_modified_backward.rb`, which is the same as the old code `euler_modified_long_time.rb`, except that I have again replaced the original loop by these two loops:

```
10000.times{
  r1, v1 = step_pos_vel(r,v,dt)
  r2, v2 = step_pos_vel(r1,v1,dt)
  r = 0.5 * ( r + r2 )
  v = 0.5 * ( v + v2 )
}

dt = -dt
10000.times{
  r1, v1 = step_pos_vel(r,v,dt)
```

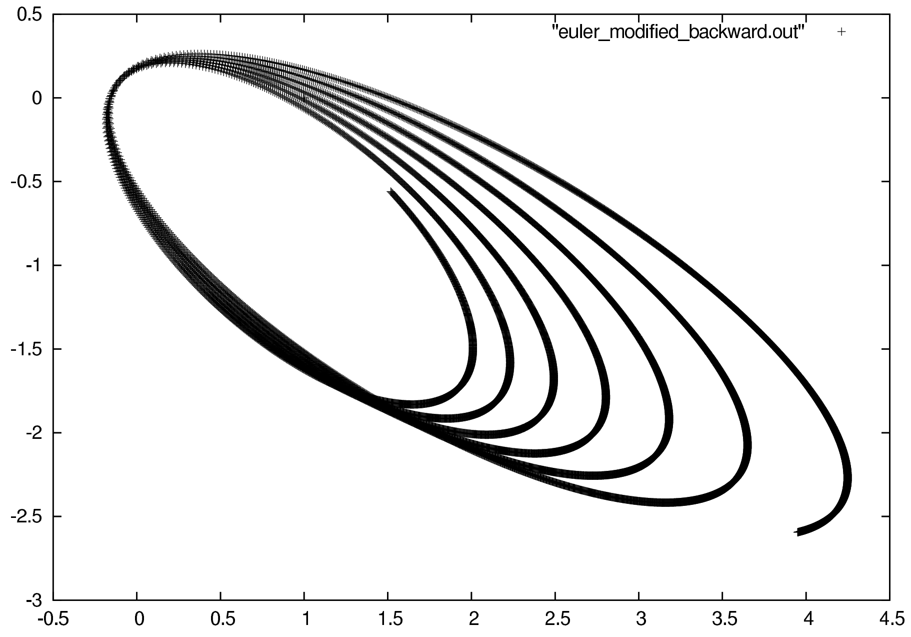

Figure 19.4: Time reversed version of the long time integration with the modified Euler algorithm, from $t = 100$ back to $t = 0$, and step size $dt = -0.01$.

```
   r2, v2 = step_pos_vel(r1,v1,dt)
   r = 0.5 * ( r + r2 )
   v = 0.5 * ( v + v2 )
   print_pos_vel(r,v)
 }
```

I will plot the backward trajectory in figure 19.4:

```
 |gravity> ruby euler_modified_backward.rb > euler_modified_backward.out
```

Carol: Figure 19.4 looks nothing like figure 19.1. Even when you reverse the direction of time, the orbit just continues to spiral out, like it did before. You have now definitely established that the modified Euler algorithm is not time symmetric!

Chapter 20

Energy Conservation

20.1 Kinetic and Potential Energy

Dan: I am glad that we now got a few second-order integration schemes. They surely are a lot more efficient than the first-order forward Euler scheme we started with!

Erica: Definitely. For the same amount of computer time, the accuracy of second-order schemes is much higher. You know, it would be nice to quantify that notion, to show exactly how accurate each scheme really is.

Carol: We have done something like that already, by checking how the endpoint of an orbit converged to a specific value, for smaller and smaller time steps.

Erica: Yes, but in that case we always needed two different choices of the time step, for two different integrations, so that we could compare the distance between the two end points. I would prefer to use a measure that tells us how good a single orbit calculation is. And this is indeed what astronomers do when they compute orbits: they pick a physical quantity that should be conversed, and they use that to get an impression of the size of the numerical errors.

Dan: What sort of quantities do you have in mind?

Erica: The typical conserved quantities for a system of interacting particles are energy and angular momentum. Of these, energy is a scalar and angular momentum is a vector. Therefore, for simplicity, people generally like to measure the change in energy, in order to get an idea of the errors introduced during orbit integration.

Dan: Okay, let's write a method to check energy conservation. For a system of two particles, how do you write down the total energy?

Erica: There are two contributions. There is the energy of motion, also called kinetic energy. This energy depends only on the speed of each particle. For a particle with mass M_i and velocity $v_i = |\mathbf{v}_i|$, the kinetic energy is

$$E_{kin,i} = \tfrac{1}{2} M_i v_i^2 \tag{20.1}$$

And then there is the energy that is given by the gravitational interaction between the particles. This is called the gravitational potential energy. For each pair of particles, say i and j, the gravitational potential energy is given by

$$E_{pot,ij} = -\frac{G M_i M_j}{r_{ij}} \tag{20.2}$$

where $r_{ij} = |\mathbf{r}_i - \mathbf{r}_j|$ is the distance between the two particles.

Dan: Why is there a minus sign?

Erica: The gravitational potential energy is normally chosen to be zero when the two particles are very far away from each other, which makes sense, since in that case there is almost no gravitational interaction. Indeed, in our expression above, for $r \to \infty$ you can see that $E_{pot,ij} \to 0$.

It is clear from the definition of $E_{kin,i}$ that the kinetic energy for each particle is always positive or zero. This implies, because the total energy is conserved, that the potential energy has to be zero or negative.

For example, if you place two particles at rest at a very large distance, the kinetic energy is zero and the potential energy is almost zero as well. Then, when the particles start falling toward each other, the kinetic energy gets larger and larger, and therefore more and more positive. The only way that the total energy can be conserved is for the potential energy to become more and more negative.

20.2 Relative Coordinates

Carol: In our computer programs we have used the one-body representation, using the relative separation $\mathbf{r}$ and relative velocity $\mathbf{v}$, rather than the individual positions $\mathbf{r}_i$ and velocities $\mathbf{v}_i$ of the particles. So we have to rewrite your expressions.

Erica: Yes, we have to transform the kinetic and potential energies from the two-body representation to the one-body representation. Let us start with the kinetic energy. Using Eq. (20.1, we get:

$$E_{kin} = E_{kin,1} + E_{kin,2} = \tfrac{1}{2} M_1 v_1^2 + \tfrac{1}{2} M_2 v_2^2 \tag{20.3}$$

We can use Eq. (4.18). When we differentiate that equation with respect to time, we get:

$$\begin{cases} \mathbf{v}_1 = -\dfrac{M_2}{M_1+M_2}\mathbf{v} \\ \\ \mathbf{v}_2 = +\dfrac{M_1}{M_1+M_2}\mathbf{v} \end{cases} \tag{20.4}$$

When we substitute these values in Eq. (20.3), we get

$$\begin{aligned} E_{kin} &= \tfrac{1}{2}M_1v_1^2 + \tfrac{1}{2}M_2v_2^2 \\ &= \tfrac{1}{2}\frac{M_1M_2^2}{(M_1+M_2)^2}v^2 + \tfrac{1}{2}\frac{M_2M_1^2}{(M_1+M_2)^2}v^2 \quad = \quad \tfrac{1}{2}\frac{M_1M_2}{M_1+M_2}v^2 \end{aligned} \tag{20.5}$$

As for the potential energy, using Eq. (20.2), we get:

$$E_{pot} = -\frac{GM_1M_2}{r} \tag{20.6}$$

and this expression already uses relative coordinates only.

In our case, we have decided to use units in which $G = M_1 + M_2 = 1$, so the last two expressions simplify to:

$$E_{kin} = \tfrac{1}{2}M_1M_2v^2 \tag{20.7}$$

$$E_{pot} = -\frac{M_1M_2}{r} \tag{20.8}$$

The total energy, which is conserved, is then

$$E_{tot} = M_1M_2\left(\tfrac{1}{2}v^2 - \frac{1}{r}\right) \tag{20.9}$$

20.3 Specific Energies

Carol: That's a bit annoying, to see that factor M_1M_2 coming in. So far, we did not have to specify the masses of the stars. Our equation of motion for Newtonian gravity, Eq. (4.22) contained only the sum of the masses. So when we choose that sum to be unity, the equation became simply Eq. (4.24), and we had gotten rid of any mention of masses.

In other words, whether we had equal masses, $M_1 = M_2 = 1/2$, whether we took one mass to be three times as large as the other, $M_1 = 3/4; M_2 = 1/4$, in both cases the orbits would be exactly the same. However, you are now telling us that the total energy will be different for those two cases. In the first case, the factor $M_1M_2 = 1/4$ whereas in the second case, it becomes $M_1M_2 = 3/16$, a smaller value.

It seems that when we want to measure energy conservation, we have to make an extra choice, for example by specifying the ratio of the two masses.

Dan: But the only thing we care about is whether the energy is conserved. All we want to know whether the term between parentheses in Eq. (20.9) remains constant, or almost constant. Who cares about the funny factor in front?

Erica: Well, yes, I basically agree, but let us try to be a bit more precise, in describing what we do. To make things clear, let me go back to our earlier description, which still contains the total mass and the gravitational constant. If you look at the text books, you will find that they introduce the so-called reduced mass μ, defined as:

$$\mu = \frac{M_1M_2}{M_1 + M_2} \tag{20.10}$$

As you can see, it has indeed the physical dimension of mass: two powers of mass divided by one power leaves mass to the power one. The total mass can be written as simply

$$M = M_1 + M_2 \tag{20.11}$$

In terms of these two quantities, we can define our two energies as:

$$E_{kin} = \tfrac{1}{2}\mu v^2 \tag{20.12}$$

and

$$E_{pot} = -\frac{G\mu M}{r} \tag{20.13}$$

So this looks like the motion of a pseudo particle with mass μ, moving in the gravitational field of another particle with mass M. The total energy is given by

$$E_{tot} = \tfrac{1}{2}\mu v^2 - \frac{G\mu M}{r} = \mu\left(\tfrac{1}{2}v^2 - \frac{GM}{r}\right) \tag{20.14}$$

We can now define the specific energy $\mathcal{E}$ as the energy per unit mass for the pseudo particle:

$$\mathcal{E}_{kin} = \frac{E_{kin}}{\mu} = \tfrac{1}{2}v^2 \tag{20.15}$$

and

$$\mathcal{E}_{pot} = \frac{E_{pot}}{\mu} = -\frac{GM}{r} \tag{20.16}$$

and with our convention $G = M_1 + M_2 = 1$, we find for the specific total energy:

$$\mathcal{E}_{tot} = \tfrac{1}{2}v^2 - \frac{1}{r} \tag{20.17}$$

This is exactly the expression in parentheses in the right-hand side of Eq. (**??**), which Dan wanted to use. And now we have a name for it: the specific energy, defined as the energy per unit reduced mass.

Dan: Well, name or no name, let's see whether it is actually conserved reasonably well in our calculations.

20.4 Diagnostics

Carol: Let us start with our simplest vector code, `euler_vector.rb`, and let us add some nice diagnostics. We definitely want to check to what extent the total energy is constant, but I would also like to see how the kinetic and potential energy are varying.

To begin with, let me define a method `energies` which returns all three energies, kinetic, potential and total, in one array:

```
def energies(r,v)
  ekin = 0.5*v*v
  epot = -1/sqrt(r*r)
  [ekin, epot, ekin+epot]
end
```

Erica: Specific energies, that is.

Carol: Yes, but I don't feel that it is necessary to add that to the method name.

Dan: I knew you would get tired of long names!

Carol: Only if there is no danger for confusion. In this case, we're not going to mix specific and absolute energies – and if we ever do, we can always make the name longer again.

Next I would like to write a method that prints diagnostics, let me just call it `print_diagnostics`, that will print out all three energies.

Erica: But what we really need is to know the *deviation* from the original energy value, to test energy conservation.

Carol: You're right. So that means that we had better measure the energy right from the start, and then remember that value. Let us call the initial energy `e0`.

Dan: That's almost too short a name, for my taste!

Carol: I was just trying to see how far I could push your taste! Now we can use the method above, picking out the last array element using the `Array` method `last`, to find the initial total energy

```
e0 = energies(r,v).last
```

before we enter the integration loop.

Now let me think a bit carefully about the layout that `print_diagnostics` should follow. We probably only need three significant digits for the relative energy change. That will be good enough to see how good or bad our energy conservation is. In Ruby you can use a C like notation to fix the output format, using expressions like `printf("%.3g", x)` to print the value of a floating point variable `x` with three significant digits.

Actually, what I will do is use `sprintf` instead of `printf`, which prints the same information onto a string, rather than directly onto the output channel. That way, I can use the Ruby `print` command, which takes multiple arguments. If `x` contains the number π, say, then writing

```
print "x = ", sprintf("%.3g\n", x)
```

gets translated into

```
print "x = ", "3.14\n"
```

and since `print` just concatenates its argument, this would give the same result as typing

```
print "x = 3.14\n"
```

so you should see

```
x = 3.14
```

on the screen, where I have added the new line character `\n` for good measure, to let the next prompt appear on a new line.

Hmmm, this ought to do it. Here is the whole code:

```
require "vector.rb"
include Math

def energies(r,v)
  ekin = 0.5*v*v
  epot = -1/sqrt(r*r)
  [ekin, epot, ekin+epot]
end

def print_pos_vel(r,v)
  r.each{|x| print(x, "  ")}
  v.each{|x| print(x, "  ")}
  print "\n"
end

def print_diagnostics(r,v,e0)
  ekin, epot, etot = energies(r,v)
  STDERR.print "  E_kin = ", sprintf("%.3g, ", ekin)
  STDERR.print "E_pot = ", sprintf("%.3g; ", epot)
  STDERR.print "E_tot = ", sprintf("%.3g\n", etot)
  STDERR.print "            E_tot - E_init = ", sprintf("%.3g, ", etot-e0)
  STDERR.print "(E_tot - E_init) / E_init = ", sprintf("%.3g\n", (etot-e0)/e0)
end

r = [1, 0, 0].to_v
v = [0, 0.5, 0].to_v
dt = 0.01
e0 = energies(r,v).last

print_pos_vel(r,v)
print_diagnostics(r,v,e0)
1000.times{
  r2 = r*r
  r3 = r2 * sqrt(r2)
  a = -r/r3
  r += v*dt
  v += a*dt
  print_pos_vel(r,v)
}
print_diagnostics(r,v,e0)
```

20.5 Checking Energy

Dan: What does STDERR mean, in front of the print statements?

Carol: That means that the information will be printed on the standard error stream. By default, information will be printed on the standard out stream. These are the two main output streams in Unix.

Dan: Why is it called *error stream*?

Carol: If you have a lot of output, you want to redirect that to a file. But if something suddenly goes wrong, you would like to be warned about that on the screen. You certainly don't want a warning message or error message to be mixed in with all the other stuff in the output file; you might never notice it there.

In our case, I would like to use this error channel to report on the behavior of the energies. In fact, we want to determine the energy errors, so it is somewhat appropriate to use the error stream, even though the name suggests that it is normally used to report real errors. But why not? We will use it here to report on small numerical errors.

Dan: So you report the values of the various energy contributions only at the beginning and at the end of the run.

Carol: For now, that is good enough. At least it's a start. But let me check to see whether all this works. We don't need the positions and velocities for now, so I will redirect those to our waste basket `/dev/null`

```
|gravity> ruby euler_energy_try1.rb > /dev/null
  E_kin = 0.125, E_pot = -1; E_tot = -0.875
            E_tot - E_init = 0, (E_tot - E_init) / E_init = -0
  E_kin = 0.495, E_pot = -0.101; E_tot = 0.394
            E_tot - E_init = 1.27, (E_tot - E_init) / E_init = -1.45
```

Dan: That does look pretty, I must say. But look, the energy is totally different, at the beginning and at the end of the run.

Erica: As it should be: remember, this was our very first run, when we used a time step that was too big to integrate an elliptic orbit! We made a huge error at pericenter. In fact, we can now see that the energy changed sign. We started with a bound orbit, with a total energy that was negative. But at the end of the integration the energy has become positive! That means that the particles can escape to infinity.

Carol: Why is that?

Erica: When the particles are very far away from each other, the potential energy becomes negligible, and the energy is dominated by the kinetic energy. Since kinetic energy cannot be negative, such a wide separation is impossible if the total energy is negative. But for zero or positive total energy, there is nothing that can prevent the two particles to fly away from each other completely. And clearly, that is what happened here, as a result of numerical errors.

Dan: Before drawing too many conclusions, we'd better check whether we still are talking about the same orbit as we did before.

Carol: My pleasure. Here is what the old code gave:

```
|gravity> ruby euler_vector.rb > euler_vector.out
|gravity> head -1 euler_vector.out
1  0  0  0  0.5  0
|gravity> tail -1 euler_vector.out
7.6937453936572  -6.27772005661599  0.0  0.812206830641815  -0.574200201239989  0.0
```

And here is what the diagnostics produces, also at the very beginning and end of the output file:

```
|gravity> ruby euler_energy_try1.rb > euler_energy_try1.out
  E_kin = 0.125, E_pot = -1; E_tot = -0.875
            E_tot - E_init = 0, (E_tot - E_init) / E_init = -0
  E_kin = 0.495, E_pot = -0.101; E_tot = 0.394
            E_tot - E_init = 1.27, (E_tot - E_init) / E_init = -1.45
|gravity> head -1 euler_energy_try1.out
1  0  0  0  0.5  0
|gravity> tail -1 euler_energy_try1.out
7.6937453936572  -6.27772005661599  0.0  0.812206830641815  -0.574200201239989  0.0
```

Dan: Good! Exactly the same.

20.6 Error Growth

Erica: We know that our first orbit integration produced large errors, and we have quantified that by looking at the final energy error, at the end of our orbit integration. But it would be a lot more instructive to see how the energy error is growing in time.

Carol: Easily done: in file `euler_energy_try2.rb`, I will modify our `print_pos_vel` method to include the total energy value as well, calling it `print_pos_vel_energy` instead:

```
def print_pos_vel_energy(r,v,e0)
  r.each{|x| print(x, "  ")}
  v.each{|x| print(x, "  ")}
  etot = energies(r,v).last
  print (etot-e0)/e0
  print "\n"
end
```

As you can see, I am printing the energy last, after the positions and velocities. And of course, in the code I'm replacing the old name by the new name in the two invocations, just before entering the loop and at the end of the loop. Let's run it:

```
|gravity> ruby euler_energy_try2.rb > euler_energy_try2.out
  E_kin = 0.125, E_pot = -1; E_tot = -0.875
            E_tot - E_init = 0, (E_tot - E_init) / E_init = -0
  E_kin = 0.495, E_pot = -0.101; E_tot = 0.394
            E_tot - E_init = 1.27, (E_tot - E_init) / E_init = -1.45
|gravity> head -1 euler_energy_try2.out
1  0  0  0  0.5  0  -0.0
|gravity> tail -1 euler_energy_try2.out
7.6937453936572  -6.27772005661599  0.0  0.812206830641815  -0.574200201239989  0.
```

Erica: I don't like the way these numbers are rolling on and on. We don't really need that much precision in the positions and velocities, if we just want to make a pretty picture. Four or five digits should be more than enough.

Carol: That's easy to fix. In file `euler_energy_try3.rb` I will change the frequent output method as follows:

```
def print_pos_vel_energy(r,v,e0)
  r.each{|x| printf("%.5g  ", x)}
  v.each{|x| printf("%.5g  ", x)}
  etot = energies(r,v).last
  print (etot-e0)/e0
  print "\n"
end
```

This should look better now:

```
|gravity> ruby euler_energy_try3.rb > euler_energy_try3.out
  E_kin = 0.125, E_pot = -1; E_tot = -0.875
            E_tot - E_init = 0, (E_tot - E_init) / E_init = -0
  E_kin = 0.495, E_pot = -0.101; E_tot = 0.394
            E_tot - E_init = 1.27, (E_tot - E_init) / E_init = -1.45
|gravity> head -1 euler_energy_try3.out
1  0  0  0  0.5  0  -0.0
|gravity> tail -1 euler_energy_try3.out
7.6937  -6.2777  0  0.81221  -0.5742  0  -1.45027113997184
```

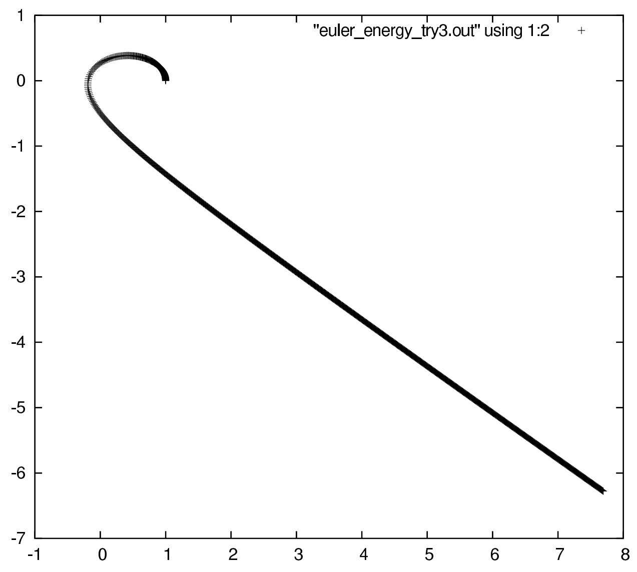

Figure 20.1: Trajectory of our very first forward Euler integration attempt.

20.7 Pericenter Troubles

Carol: Let's figure out what magic incantations `gnuplot` wants us to give it, to plot the energy as a function of time. To start with, let me remember how we have plotted the orbit. Ah yes, we have been using the fact that gnuplot use the first two columns, if you don't specify anything otherwise. Instead of relaying on that default choice, let's plot the orbit again, this time giving gnuplot explicit instructions to use the data from columns 1 and 2:

```
|gravity> gnuplot
gnuplot> set size ratio -1
gnuplot> plot "euler_energy_try3.out" using 1:2
gnuplot> quit
```

So this gives figure 20.1, and indeed, it looks just like before, when we produced figure 7.1.

Now to make Carol happy, we will plot the values of the total energy, which reside in column 7.

Dan: But wait, that is different. First we were plotting `y` as a function of `x`. Now you are going to plot the energy `E` as a function of what? Of time, I guess.

Carol: Yes, that would be the most obvious choice. And because we are using constant time steps, that boils down to plotting `E` as a function of output number, if we number the output lines successively. And indeed, gnuplot does have a way to use the output line number as the thing to plot along the horizontal axis: if you specify the value 0 as a column number, the output line number will be used.

Dan: Ah, that makes sense, and that is easy to remember. If you have an output line that reads, say, in the first three columns:

```
8 20 3
4 5 6
9 2 1
...
```

then it is as if gnuplot itself adds the line numbers to the left:

```
1 8 20 3
2 4 5 6
3 9 2 1
...
```

and now the column numbering starts at 0 instead of at 1.

Carol: Yes, come to think of it, that must be the reason they introduced that notation. Well, let me try:

```
|gravity> gnuplot
gnuplot> set size ratio -1
gnuplot> plot "euler_energy_try3.out" using 0:7
gnuplot> quit
```

Erica: Beautiful! Just as we expected, the main error is generated when the two stars pass close to each other, at pericenter. But I had not expected the error to be so sensitive to the distance between the stars. The error is generated almost instantaneously!

Carol: Why would that be?

Erica: When the two stars come closer, the orbit becomes more curved, and in addition, the speed becomes larger. So for both reasons, there is more of a change in the orbit during a constant interval in time. It would be great if we could use a smaller time step during pericenter passage, and I'm sure we'll get to that point, later on. But for now, as long as we are using constant time steps,

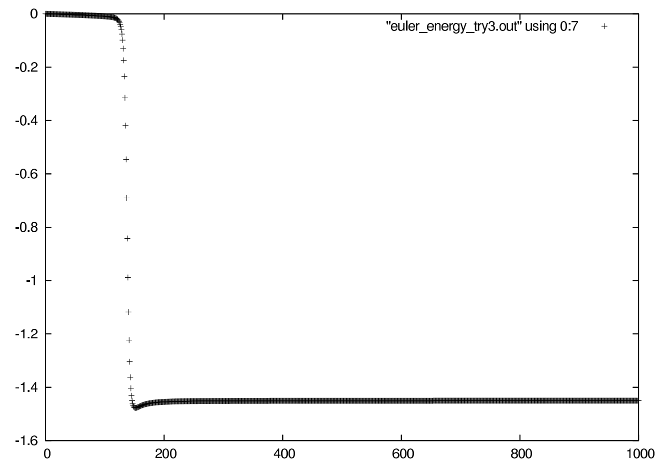

Figure 20.2: Error growth for our very first forward Euler integration attempt.

a higher speed means that each step will cover a larger distance in space. So we are in a situation that we are actually taking longer steps in space exactly there where the orbit is curved most.

Dan: Not a good thing to do.

Erica: I agree, but it's the simplest thing to do. We can later try to be more clever.

Chapter 21

Scaling of Energy Errors

21.1 A Matter of Time

Carol: So now we have seen how we are making a really big error, when we use forward Euler with a time step that is really too large for a first-order integration scheme. But that's not what we are really interested in. We want to study the behavior of errors in the case where the time steps are small enough to give reasonable orbit pictures.

Erica: Yes, and then we want to compare first-order and second-order integration schemes, to check whether the energy errors scale in different ways. First we should continue to look at forward Euler, but with smaller time steps.

Dan: You know, I really get tired of writing a whole new file, each time we change a parameter, like the size of a time step. Can't we let the program ask for the time step size, so that we can type it in while the program is running?

Carol: Good idea. That is a much cleaner approach. And while we're at it, why not ask for the total duration of the integration too. And that reminds me, I really wasn't very happy with the way we have forced the code to give sparse output, at every interval of `\Delta t = 0.01`.

Dan: Remind me, what did we do there?

Carol: Take file `euler_elliptic_100000_steps_sparse_ok.rb`

Dan: Ah, yes, of course, how can I forget such a name!

Carol: Well, as the name says, that code took 100,000 steps, during a total time of 10 time units. With output intervals of length 0.01, this means that we needed only 1,000 outputs. In other words, we needed only to print the results once every 100 steps. We did this with the following rather clumsy trick, using the remainder operator `%`:

```
  if i%100 == 99
    print(x, "  ", y, "  ", z, "  ")
    print(vx, "  ", vy, "  ", vz, "\n")
  end
```

I suggest that instead we introduce the time explicitly. Notice that so far we have used a variable dt to keep the value of the time step size, but we have never kept track of the time itself. Let us introduce a variable t to indicate the time that has passed since the start of the integration. We can then specify a desired interval between outputs, dt_out for short. And to keep track of when the next output is scheduled to happen, we can use a variable t_out. Whenever the time t reaches t_out or goes past it, we need to do another output.

Of course, our diagnostics method should now print the value of the time as well. What else do we need to change. The main loop now becomes a test to see whether the time t has passed to or beyond the final time t_end, specified by the user. And after each output statement, t_out has to be incremented to the next scheduled output time, by adding dt_out to its value. Well, that must be about it, yes?

21.2 A New Control Structure

Let me open a file euler_energy_try4.rb and type it all in:

```
require "vector.rb"
include Math

def energies(r,v)
  ekin = 0.5*v*v
  epot = -1/sqrt(r*r)
  [ekin, epot, ekin+epot]
end

def print_pos_vel_energy(r,v,e0)
  r.each{|x| printf("%.5g  ", x)}
  v.each{|x| printf("%.5g  ", x)}
  etot = energies(r,v).last
  print (etot-e0)/e0
  print "\n"
end

def print_diagnostics(t,r,v,e0)
  ekin, epot, etot = energies(r,v)
  STDERR.print "  t = ", sprintf("%.3g, ", t)
  STDERR.print "  E_kin = ", sprintf("%.3g, ", ekin)
  STDERR.print "E_pot = ", sprintf("%.3g; ", epot)
  STDERR.print "E_tot = ", sprintf("%.3g\n", etot)
  STDERR.print "           E_tot - E_init = ", sprintf("%.3g, ", etot-e0)
  STDERR.print "(E_tot - E_init) / E_init = ", sprintf("%.3g\n", (etot-e0)/e0)
end

r = [1, 0, 0].to_v
v = [0, 0.5, 0].to_v
e0 = energies(r,v).last

t = 0
t_out = 0
dt_out = 0.01
STDERR.print "time step = ?\n"
dt = gets.to_f
STDERR.print "final time = ?\n"
t_end = gets.to_f

print_pos_vel_energy(r,v,e0)
t_out += dt_out
print_diagnostics(t,r,v,e0)

while t < t_end
  r2 = r*r
  r3 = r2 * sqrt(r2)
  a = -r/r3
  r += v*dt
  v += a*dt
  t += dt
  if t >= t_out
    print_pos_vel_energy(r,v,e0)
    t_out += dt_out
  end
end
print_diagnostics(t,r,v,e0)
```

Dan: What is `gets`?

Carol: That is a Ruby input statement, short for 'get string'. It reads the next line from the command line. So if you type a single value, and then hit the enter key, `gets` gobbles up the number you have typed, but packaged as a string. For example, when the code asks you for a time step, and you type

```
0.01
```

then `gets` returns the string `"0.01"`, made up of four characters. What we really want is a number, and the method `to_f` is the built-in Ruby way to convert a string into a floating point number; it is short for '(convert) to float'.

21.3 Overshooting

Erica: Let's give it the same values as before, to see whether we get the same output.

Carol: This is what we found before:

```
|gravity> ruby euler_energy_try3.rb > euler_energy_try3.out
  E_kin = 0.125, E_pot = -1; E_tot = -0.875
            E_tot - E_init = 0, (E_tot - E_init) / E_init = -0
  E_kin = 0.495, E_pot = -0.101; E_tot = 0.394
            E_tot - E_init = 1.27, (E_tot - E_init) / E_init = -1.45
```

And let us see what our new program gives:

```
|gravity> ruby euler_energy_try4.rb > euler_energy_try4.out
time step = ?
0.01
final time = ?
10
  t = 0,   E_kin = 0.125, E_pot = -1; E_tot = -0.875
            E_tot - E_init = 0, (E_tot - E_init) / E_init = -0
  t = 10,   E_kin = 0.495, E_pot = -0.101; E_tot = 0.394
            E_tot - E_init = 1.27, (E_tot - E_init) / E_init = -1.45
```

Dan: At least the diagnostics output is the same. How about the output files?

Carol: I'll do a `diff`:

```
|gravity> diff euler_energy_try3.out euler_energy_try4.out
1001a1002
> 7.7019  -6.2835  0  0.81213  -0.57414  0  -1.45027103130336
```

He, that is strange. Our friend `diff` claims that the two files are identical except for the fact that our latest code produces one more line of output! Let me check it with a word count:

```
|gravity> wc euler_energy_try[34].out
    1001    7007   59973 euler_energy_try3.out
    1002    7014   60033 euler_energy_try4.out
    2003   14021  120006
```

And what do you know, yes, 1001 lines in `euler_energy_try3.out` as it should be, moving from times 0 till 10 with steps of 0.01, but why does `euler_energy_try4.out` have 1002 lines??

Erica: That last program must be taking one more step, beyond time 10. Can you show the last few lines for both output files?

Carol: Sure:

```
|gravity> tail -3 euler_energy_try3.out
7.6775  -6.2662  0  0.81236  -0.57433  0  -1.45027135833743
7.6856  -6.272  0  0.81229  -0.57426  0  -1.45027124898271
7.6937  -6.2777  0  0.81221  -0.5742  0  -1.45027113997184
|gravity> tail -3 euler_energy_try4.out
7.6856  -6.272  0  0.81229  -0.57426  0  -1.45027124898271
7.6937  -6.2777  0  0.81221  -0.5742  0  -1.45027113997184
7.7019  -6.2835  0  0.81213  -0.57414  0  -1.45027103130336
```

Just like `diff` told us, the last few lines are identical, except for the fact that `euler_energy_try4.out` shows one extra step. You must be right: it looks like the code didn't know how to stop in time.

21.4 Knowing When To Stop

Erica: I wonder why it overshot.

Carol: Let me put some debug statements in there, for now, just to see what the code thinks it is doing, toward the end. Right at the beginning of the loop, after the `while` line, I'll ask for the two time values to be printed out, the running time `t` and the end time `t_end`, in file `euler_energy_try5.rb`:

```
while t < t_end
  print "t = ", t, " and t_end = ", t_end, "\n"
```

Here we go:

```
|gravity> ruby euler_energy_try5.rb > euler_energy_try5.out
time step = ?
0.01
final time = ?
10
  t = 0,   E_kin = 0.125, E_pot = -1; E_tot = -0.875
            E_tot - E_init = 0, (E_tot - E_init) / E_init = -0
  t = 10,   E_kin = 0.495, E_pot = -0.101; E_tot = 0.394
            E_tot - E_init = 1.27, (E_tot - E_init) / E_init = -1.45
```

```
|gravity> tail euler_energy_try5.out
t = 9.95999999999983 and t_end = 10.0
7.6694  -6.2605  0  0.81244  -0.57439  0  -1.45027146803748
t = 9.96999999999983 and t_end = 10.0
7.6775  -6.2662  0  0.81236  -0.57433  0  -1.45027135833743
t = 9.97999999999983 and t_end = 10.0
7.6856  -6.272  0  0.81229  -0.57426  0  -1.45027124898271
t = 9.98999999999983 and t_end = 10.0
7.6937  -6.2777  0  0.81221  -0.5742  0  -1.45027113997184
t = 9.99999999999983 and t_end = 10.0
7.7019  -6.2835  0  0.81213  -0.57414  0  -1.45027103130336
```

Erica: Aha! The problem is roundoff, that explains everything! The time variable `t` is a floating point variable, and instead of reaching the exact time 10, after 1,000 steps, it comes ever so close, but not quite at the right point. Therefore, when it runs the loop test, it decides that the time has to be incremented by another time step, and it then overshoots.

Carol: That suggests a simple solution. How about testing whether the time has reached not exactly the end time, but close enough? Close enough could mean half a time step. Let's try! And I'll be bold and call the next file `euler_energy.rb`, in the hope we now get it right. I will write the loop continuation test like this:

```
while t < t_end - 0.5*dt
```

That should do it:

```
|gravity> ruby euler_energy.rb > euler_energy.out
time step = ?
0.01
final time = ?
```

```
10
  t = 0,   E_kin = 0.125, E_pot = -1; E_tot = -0.875
            E_tot - E_init = 0, (E_tot - E_init) / E_init = -0
  t = 10,   E_kin = 0.495, E_pot = -0.101; E_tot = 0.394
            E_tot - E_init = 1.27, (E_tot - E_init) / E_init = -1.45
```

```
|gravity> diff euler_energy_try3.out euler_energy.out
```

Dan: And it did it. No differences. Congratulations!

Carol: Let me be double sure:

```
|gravity> tail -1 euler_energy_try3.out
7.6937  -6.2777  0  0.81221  -0.5742  0  -1.45027113997184
|gravity> tail -1 euler_energy.out
7.6937  -6.2777  0  0.81221  -0.5742  0  -1.45027113997184
```

Good. So now we have a new tool, allowing us to change two parameters, without having to change the source code each time. Progress!

21.5 Linear Scaling

Dan: Now the point of all this was to check whether the energy errors in forward Euler scale linearly with the time step size. Let's try a few values.

Carol: Sure thing. And now that we can control both the time step and the duration of the integration, let's speed things up a bit, and integrate for only 0.1 time unit. Starting with the previous time step we get:

```
|gravity> ruby euler_energy.rb > euler_energy.out
time step = ?
0.01
final time = ?
0.1
  t = 0,   E_kin = 0.125, E_pot = -1; E_tot = -0.875
            E_tot - E_init = 0, (E_tot - E_init) / E_init = -0
  t = 0.1,   E_kin = 0.129, E_pot = -1; E_tot = -0.874
            E_tot - E_init = 0.000626, (E_tot - E_init) / E_init = -0.000715
```

Making the time step ten times shorter, we find:

```
|gravity> ruby euler_energy.rb > euler_energy.out
time step = ?
0.001
final time = ?
0.1
  t = 0,   E_kin = 0.125, E_pot = -1; E_tot = -0.875
            E_tot - E_init = 0, (E_tot - E_init) / E_init = -0
  t = 0.1,   E_kin = 0.129, E_pot = -1; E_tot = -0.875
            E_tot - E_init = 6.26e-05, (E_tot - E_init) / E_init = -7.16e-05
```

And making it yet ten times shorter gives:

```
|gravity> ruby euler_energy.rb > euler_energy.out
time step = ?
0.0001
final time = ?
0.1
  t = 0,   E_kin = 0.125, E_pot = -1; E_tot = -0.875
            E_tot - E_init = 0, (E_tot - E_init) / E_init = -0
  t = 0.1,   E_kin = 0.129, E_pot = -1; E_tot = -0.875
            E_tot - E_init = 6.26e-06, (E_tot - E_init) / E_init = -7.16e-06
```

Dan: Pretty linear, all right.

Carol: Let's jump to a hundred times smaller time step, to see whether the error still becomes a hundred times smaller:

```
|gravity> ruby euler_energy.rb > euler_energy.out
time step = ?
0.000001
final time = ?
0.1
  t = 0,   E_kin = 0.125, E_pot = -1; E_tot = -0.875
            E_tot - E_init = 0, (E_tot - E_init) / E_init = -0
  t = 0.1,   E_kin = 0.129, E_pot = -1; E_tot = -0.875
            E_tot - E_init = 6.26e-08, (E_tot - E_init) / E_init = -7.16e-08
```

Dan: And so it does.

21.6 Picture Time

Erica: I'd like to see how the energy error behaves in time, over a few full orbits, and with better accuracy.

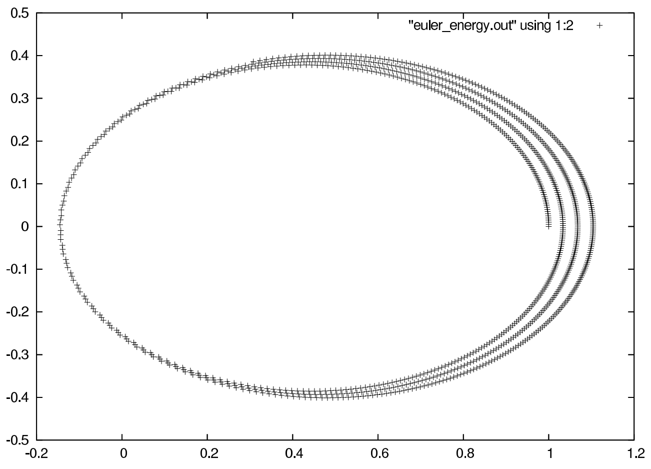

Figure 21.1: Trajectory of a more accurate forward Euler integration, with $dt = 0.0001$.

Carol: Okay, I'll take ten time units again for the total orbit integration, and a time step of 0.0001. Just to remind us of what the orbit looked like, I'll plot it again, in fig 21.1.

```
|gravity> ruby euler_energy.rb > euler_energy.out
time step = ?
0.0001
final time = ?
10
  t = 0,   E_kin = 0.125, E_pot = -1; E_tot = -0.875
            E_tot - E_init = 0, (E_tot - E_init) / E_init = -0
  t = 10,   E_kin = 1.27, E_pot = -2.07; E_tot = -0.8
            E_tot - E_init = 0.0749, (E_tot - E_init) / E_init = -0.0856
```

Erica: Ah, yes, that time step was just about short enough to begin to see the intended orbit, without too much drift.

Carol: And here is how the error grows, as a function of time, in fig 21.2.

Erica: Even though the orbit behaves a lot better now, it is clear that the energy errors are still being generated mostly around pericenter passage. In between those close encounters, the energy is very well conserved. But whenever the two stars swing around each other, the energy drifts in a systematic and cumulative way.

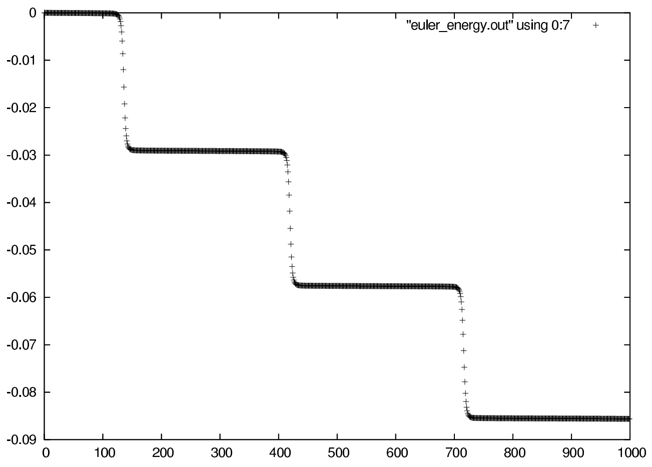

Figure 21.2: Energy error growth for a more accurate forward Euler integration, with $dt = 0.0001$.

Dan: Yes, dramatically so! I can see why people like to use individual time steps. If you use thrown in a few more time steps during close encounters, you can get very much more accuracy as a return for investing very little extra computer time.

Chapter 22

Error Scaling for 2nd-Order Schemes

22.1 Modified Euler

Dan: Let's have a look at our second-order integration schemes. If I understand things correctly, they are supposed to improve energy quadratically, right? When we make the time step ten times smaller, the energy error should become one hundred times smaller.

Carol: That's the expectation, yes. But first I have to write the codes. I will start with the modified Euler algorithm, for which we had written the vector version of the code in file `euler_modified_vector.rb`. I will open a new file `euler_modified_energy.rb`, and add the same type of energy output statements and diagnostics as we have done for the forward Euler case. Here we go:

```
require "vector.rb"
include Math

def step_pos_vel(r,v,dt)
  r2 = r*r
  r3 = r2 * sqrt(r2)
  a = -r/r3
  [r + v*dt, v + a*dt]
end

def energies(r,v)
  ekin = 0.5*v*v
  epot = -1/sqrt(r*r)
  [ekin, epot, ekin+epot]
end

def print_pos_vel_energy(r,v,e0)
  r.each{|x| printf("%.5g  ", x)}
  v.each{|x| printf("%.5g  ", x)}
  etot = energies(r,v).last
  print (etot-e0)/e0
  print "\n"
end

def print_diagnostics(t,r,v,e0)
  ekin, epot, etot = energies(r,v)
  STDERR.print "  t = ", sprintf("%.3g, ", t)
  STDERR.print "  E_kin = ", sprintf("%.3g, ", ekin)
  STDERR.print "E_pot = ", sprintf("%.3g; ", epot)
  STDERR.print "E_tot = ", sprintf("%.3g\n", etot)
  STDERR.print "            E_tot - E_init = ", sprintf("%.3g, ", etot-e0)
  STDERR.print "(E_tot - E_init) / E_init = ", sprintf("%.3g\n", (etot-e0)/e0)
end

r = [1, 0, 0].to_v
v = [0, 0.5, 0].to_v
e0 = energies(r,v).last

t = 0
t_out = 0
dt_out = 0.01
STDERR.print "time step = ?\n"
dt = gets.to_f
STDERR.print "final time = ?\n"
t_end = gets.to_f

print_pos_vel_energy(r,v,e0)
t_out += dt_out
print_diagnostics(t,r,v,e0)

while t < t_end - 0.5*dt
  r1, v1 = step_pos_vel(r,v,dt)
  r2, v2 = step_pos_vel(r1,v1,dt)
  r = 0.5 * ( r + r2 )
  v = 0.5 * ( v + v2 )
  t += dt
  if t >= t_out
    print_pos_vel_energy(r,v,e0)
    t_out += dt_out
  end
end
print_diagnostics(t,r,v,e0)
```

22.2 Energy Error Scaling

Dan: That all looks pretty straightforward.

Carol: And now it is just a matter of making the time steps smaller and smaller:

```
|gravity> ruby euler_modified_energy.rb > /dev/null
time step = ?
0.01
final time = ?
0.1
  t = 0,   E_kin = 0.125, E_pot = -1; E_tot = -0.875
            E_tot - E_init = 0, (E_tot - E_init) / E_init = -0
  t = 0.1,   E_kin = 0.129, E_pot = -1; E_tot = -0.875
            E_tot - E_init = -8.33e-08, (E_tot - E_init) / E_init = 9.52e-08
```

```
|gravity> ruby euler_modified_energy.rb > /dev/null
time step = ?
0.001
final time = ?
0.1
  t = 0,   E_kin = 0.125, E_pot = -1; E_tot = -0.875
            E_tot - E_init = 0, (E_tot - E_init) / E_init = -0
  t = 0.1,   E_kin = 0.129, E_pot = -1; E_tot = -0.875
            E_tot - E_init = -7.19e-10, (E_tot - E_init) / E_init = 8.22e-10
```

```
|gravity> ruby euler_modified_energy.rb > /dev/null
time step = ?
0.0001
final time = ?
0.1
  t = 0,   E_kin = 0.125, E_pot = -1; E_tot = -0.875
            E_tot - E_init = 0, (E_tot - E_init) / E_init = -0
  t = 0.1,   E_kin = 0.129, E_pot = -1; E_tot = -0.875
            E_tot - E_init = -7.07e-12, (E_tot - E_init) / E_init = 8.08e-12
```

```
|gravity> ruby euler_modified_energy.rb > /dev/null
time step = ?
0.00001
final time = ?
0.1
  t = 0,   E_kin = 0.125, E_pot = -1; E_tot = -0.875
            E_tot - E_init = 0, (E_tot - E_init) / E_init = -0
  t = 0.1,   E_kin = 0.129, E_pot = -1; E_tot = -0.875
            E_tot - E_init = -4.25e-14, (E_tot - E_init) / E_init = 4.86e-14
```

```
|gravity> ruby euler_modified_energy.rb > /dev/null
time step = ?
0.000001
final time = ?
0.1
  t = 0,   E_kin = 0.125, E_pot = -1; E_tot = -0.875
            E_tot - E_init = 0, (E_tot - E_init) / E_init = -0
  t = 0.1,   E_kin = 0.129, E_pot = -1; E_tot = -0.875
            E_tot - E_init = -1.41e-13, (E_tot - E_init) / E_init = 1.61e-13
```

Erica: Up till the last run, it looked almost too good to be true. We must have hit roundoff, I guess.

Carol: Well, yes, with double precision you can't get much further than 10^{-15} in relative accuracy for a single calculation. I'm surprised we got as close as we did. Most of the roundoff errors must have cancelled, in the 10,000 steps we took in the next to last integration. But in the last run, where we took 100,000 steps, we accumulated more roundoff errors. When you are adding more steps, you'll get more roundoff, no matter how accurate each individual step may be.

Dan: But wait, just one thing: we haven't checked yet whether we are still getting the same results as before.

Carol: Ah, yes, safety first! The old code gave:

```
|gravity> ruby euler_modified_vector.rb | tail -1
0.400020239524913  0.343214474344616  0.0  -1.48390077762002  -0.0155803976141248
```

and we should get the same result for our new code, if we give it the same parameters:

```
|gravity> ruby euler_modified_energy.rb | tail -1
0.400020239524913  0.343214474344616  0.0  -1.48390077762002  -0.0155803976141248  0.0
```

Dan: All is well.

22.3 Leapfrog

Carol: Finally, time to let the leapfrog algorithm tell us whether it is really a second-order algorithm as well. I will start with `leapfrog.rb`. I will open a new file `leapfrog_energy.rb`, and again I will add the same type of energy output statements and diagnostics. Here it is:

```
require "vector.rb"
include Math

def acc(r)
  r2 = r*r
  r3 = r2 * sqrt(r2)
  -r/r3
end

def energies(r,v)
  ekin = 0.5*v*v
  epot = -1/sqrt(r*r)
  [ekin, epot, ekin+epot]
end

def print_pos_vel_energy(r,v,e0)
  r.each{|x| printf("%.5g  ", x)}
  v.each{|x| printf("%.5g  ", x)}
  etot = energies(r,v).last
  print (etot-e0)/e0
  print "\n"
end

def print_diagnostics(t,r,v,e0)
  ekin, epot, etot = energies(r,v)
  STDERR.print "  t = ", sprintf("%.3g, ", t)
  STDERR.print "  E_kin = ", sprintf("%.3g, ", ekin)
  STDERR.print "E_pot = ", sprintf("%.3g; ", epot)
  STDERR.print "E_tot = ", sprintf("%.3g\n", etot)
  STDERR.print "             E_tot - E_init = ", sprintf("%.3g, ", etot-e0)
  STDERR.print "(E_tot - E_init) / E_init = ", sprintf("%.3g\n", (etot-e0)/e0)
```

```
end

r = [1, 0, 0].to_v
v = [0, 0.5, 0].to_v
e0 = energies(r,v).last

t = 0
t_out = 0
dt_out = 0.01
STDERR.print "time step = ?\n"
dt = gets.to_f
STDERR.print "final time = ?\n"
t_end = gets.to_f

print_pos_vel_energy(r,v,e0)
t_out += dt_out
print_diagnostics(t,r,v,e0)

a = acc(r)
while t < t_end - 0.5*dt
  v += 0.5*a*dt
  r += v*dt
  a = acc(r)
  v += 0.5*a*dt
  t += dt
  if t >= t_out
    print_pos_vel_energy(r,v,e0)
    t_out += dt_out
  end
end
print_diagnostics(t,r,v,e0)
```

22.4 Another Error Scaling Exercise

Carol: I'll just use the same parameters as I did while testing the modified Euler scheme, for making the time steps smaller and smaller:

```
|gravity> ruby leapfrog_energy.rb > /dev/null
time step = ?
0.01
final time = ?
0.1
  t = 0,   E_kin = 0.125, E_pot = -1; E_tot = -0.875
```

```
            E_tot - E_init = 0, (E_tot - E_init) / E_init = -0
  t = 0.1,   E_kin = 0.129, E_pot = -1; E_tot = -0.875
            E_tot - E_init = 1.19e-07, (E_tot - E_init) / E_init = -1.36e-07
```

```
|gravity> ruby leapfrog_energy.rb > /dev/null
time step = ?
0.001
final time = ?
0.1
  t = 0,   E_kin = 0.125, E_pot = -1; E_tot = -0.875
            E_tot - E_init = 0, (E_tot - E_init) / E_init = -0
  t = 0.1,   E_kin = 0.129, E_pot = -1; E_tot = -0.875
            E_tot - E_init = 1.19e-09, (E_tot - E_init) / E_init = -1.36e-09
```

```
|gravity> ruby leapfrog_energy.rb > /dev/null
time step = ?
0.0001
final time = ?
0.1
  t = 0,   E_kin = 0.125, E_pot = -1; E_tot = -0.875
            E_tot - E_init = 0, (E_tot - E_init) / E_init = -0
  t = 0.1,   E_kin = 0.129, E_pot = -1; E_tot = -0.875
            E_tot - E_init = 1.19e-11, (E_tot - E_init) / E_init = -1.36e-11
```

```
|gravity> ruby leapfrog_energy.rb > /dev/null
time step = ?
0.00001
final time = ?
0.1
  t = 0,   E_kin = 0.125, E_pot = -1; E_tot = -0.875
            E_tot - E_init = 0, (E_tot - E_init) / E_init = -0
  t = 0.1,   E_kin = 0.129, E_pot = -1; E_tot = -0.875
            E_tot - E_init = 1.15e-13, (E_tot - E_init) / E_init = -1.31e-13
```

```
|gravity> ruby leapfrog_energy.rb > /dev/null
time step = ?
0.000001
final time = ?
0.1
  t = 0,   E_kin = 0.125, E_pot = -1; E_tot = -0.875
            E_tot - E_init = 0, (E_tot - E_init) / E_init = -0
  t = 0.1,   E_kin = 0.129, E_pot = -1; E_tot = -0.875
            E_tot - E_init = -6.22e-15, (E_tot - E_init) / E_init = 7.11e-15
```

22.5 Roundoff Kicks In

Erica: An amazing accuracy, and that after 100,000 steps! What happened? I would have thought that the cumulative effects of 100,000 roundoff errors would have spoiled the fun.

Carol: We were probably just lucky in the way the roundoff errors canceled. Notice that the energy error at first got 100 times smaller, each time we made the time step 10 times smaller, as it should for a second-order algorithm. But in the last round, the improvement was a lot less than a factor 100.

We must be really close to the roundoff barrier now. Let me just make the time step a factor two smaller. That should make the total error grow again. Wanna bet?

Dan: No, but I do wanna see whether you're right.

Carol: Here we go;

```
|gravity> ruby leapfrog_energy.rb > /dev/null
time step = ?
0.0000005
final time = ?
0.1
  t = 0,   E_kin = 0.125, E_pot = -1; E_tot = -0.875
            E_tot - E_init = 0, (E_tot - E_init) / E_init = -0
  t = 0.1,   E_kin = 0.129, E_pot = -1; E_tot = -0.875
            E_tot - E_init = 4.66e-15, (E_tot - E_init) / E_init = -5.33e-15
```

Dan: I should have accepted your challenge, and made a bet against you!

Carol: I must say, I'm surprised that the roundoff errors cancel so well. But this just can't go on. If shrink the time step factor by another factor of two.

```
|gravity> ruby leapfrog_energy.rb > /dev/null
time step = ?
0.00000025
final time = ?
0.1
  t = 0,   E_kin = 0.125, E_pot = -1; E_tot = -0.875
           E_tot - E_init = 0, (E_tot - E_init) / E_init = -0
  t = 0.1,   E_kin = 0.129, E_pot = -1; E_tot = -0.875
           E_tot - E_init = -2.74e-14, (E_tot - E_init) / E_init = 3.13e-14
```

So there! Now the errors are finally accumulating enough to show a worse performance.

Dan: But you didn't feel confident enough to ask us to bet, this time.

Carol: I should have! And yes, before you ask me, let us check whether we still get the same output as before. What we got was:

```
|gravity> ruby leapfrog.rb | tail -1
0.583527377458303  -0.387366076048216  0.0  1.03799194001953  0.167802127213742  0.0
```

And what our new code gives is:

```
|gravity> ruby leapfrog_energy.rb | tail -1
0.583527377458303  -0.387366076048216  0.0  1.03799194001953  0.167802127213742  0.0
```

Good.

Chapter 23

Error Behavior for 2nd-Order Schemes

23.1 Modified Euler: Energy Error Peaks

Erica: I want to see how the error accumulates over a few orbits. We saw in figure 21.2 how the total energy error grows monotonically, with energy conservation getting a whack each time te stars pass close to each other. I wonder whether our two second-order schemes show a similar behavior, or whether things are getting more complicated.

Carol: Easy to do. As before, I will remind us what the orbit looked like, by plotting the $\{x, y\}$ coordinates of the position, in fig 23.1.

```
|gravity> ruby euler_modified_energy.rb > euler_modified_energy.out
time step = ?
0.001
final time = ?
10
  t = 0,   E_kin = 0.125, E_pot = -1; E_tot = -0.875
            E_tot - E_init = 0, (E_tot - E_init) / E_init = -0
  t = 10,   E_kin = 0.555, E_pot = -1.43; E_tot = -0.875
            E_tot - E_init = 0.000118, (E_tot - E_init) / E_init = -0.000135
```

Erica: I remember now: even with a rather long time step, we got a very nice looking orbit. You can barely see how it drifts away from the ideal ellipse.

Carol: And here is how the error grows, as a function of time, in fig 23.2.

Dan: Wow, that's a *very* different behavior.

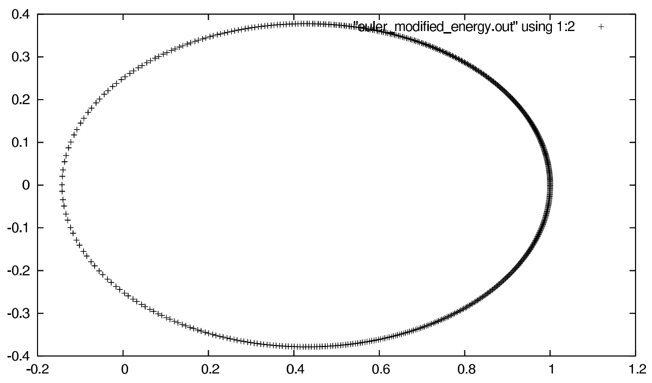

Figure 23.1: Trajectory using a modified Euler algorithm. with $dt = 0.001$.

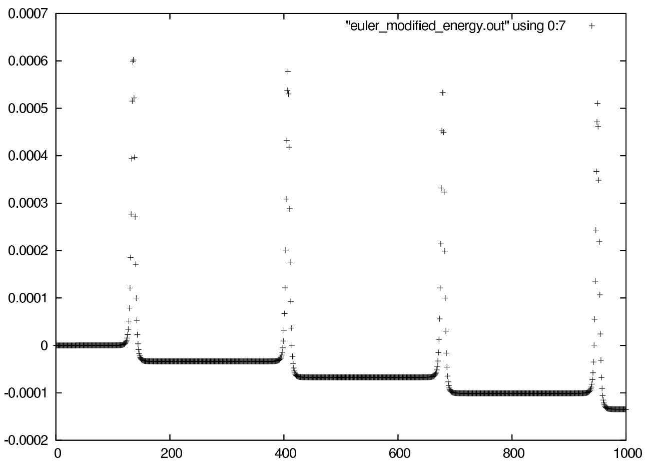

Figure 23.2: Energy error growth using a modified Euler algorithm. with $dt = 0.001$.

23.2 Almost Too Good

Erica: I think I know what's going on. The original forward Euler algorithm was making terrible errors at pericenter. I'm sure that the backward Euler algorithm would be making similarly terrible errors when the stars pass each other at pericenter. Now the modified Euler scheme works so well because it almost cancels those two errors.

In other words, during pericenter passage, the errors in forward and in backward Euler grow enormously, and the attempt at canceling is relatively less successful. But once we emerge from the close encounter, the attempts at canceling have paid off, and the net result is a much more accurate energy conservation.

Carol: Hmm, that sounds too much like a hand-waving argument to me. I would be more conservative, and just say that second-order algorithms are more complicated to begin with, so I would expect them to have more complex error behavior as well. Your particular explanation may well be right, but can you prove that?

Erica: I'm not sure how to *prove* it. It is more of a hunch.

Dan: Let's not get too technical here. We want to move stars around, and we don't need to become full-time numerical analysts.

Erica: But I would like to see what happens when we make the time step smaller.

Carol: Okay, I'll make the time step ten time smaller, and plot the results in fig 23.3.

```
|gravity> ruby euler_modified_energy.rb > euler_modified_energy2.out
time step = ?
0.0001
final time = ?
10
  t = 0,   E_kin = 0.125, E_pot = -1; E_tot = -0.875
            E_tot - E_init = 0, (E_tot - E_init) / E_init = -0
  t = 10,   E_kin = 0.554, E_pot = -1.43; E_tot = -0.875
            E_tot - E_init = 1.17e-07, (E_tot - E_init) / E_init = -1.34e-07
```

Dan: Heh, look, compared to the error peaks at pericenter passage, the total error drift looks a lot less than in the previous figure.

Erica: But the error peaks scale like a second-order algorithm: they have become 100 times less high.

Carol: So the net error after the whole run must have scaled better than second-order. And indeed, look at the output we got when I did the runs: after ten

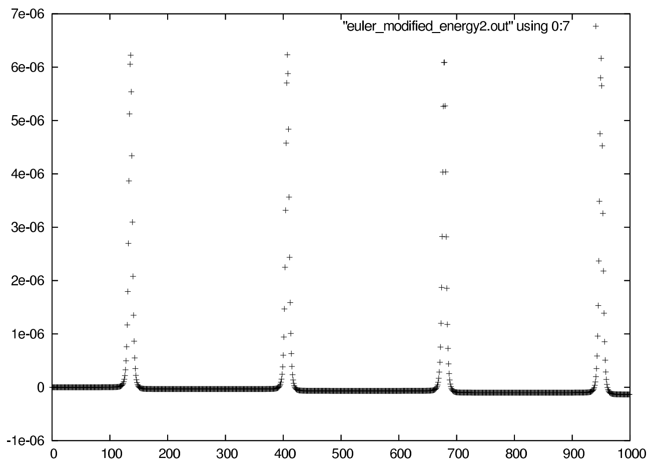

Figure 23.3: Energy error growth using a modified Euler algorithm. with $dt = 0.0001$.

time units, the energy error became a factor thousand smaller, when I decreased the time step by a factor ten!

Dan: Almost too good to be true.

Carol: Well, a second-order scheme is guaranteed to be at least second order; there is no guarantee that it doesn't do better than that. It may be the particular configuration of the orbit that gives us extra error cancellation, for free. Who knows?

Dan: Let's move on and see what the leapfrog algorithm shows us.

23.3 Leapfrog: Peaks on Top of a Flat Valley

Carol: Okay. First I'll show the orbit, in fig 23.4, using `leapfrog_energy.rb`:

```
|gravity> ruby leapfrog_energy.rb > leapfrog_energy.out
time step = ?
0.001
final time = ?
10
  t = 0,   E_kin = 0.125, E_pot = -1; E_tot = -0.875
            E_tot - E_init = 0, (E_tot - E_init) / E_init = -0
  t = 10,   E_kin = 0.554, E_pot = -1.43; E_tot = -0.875
```

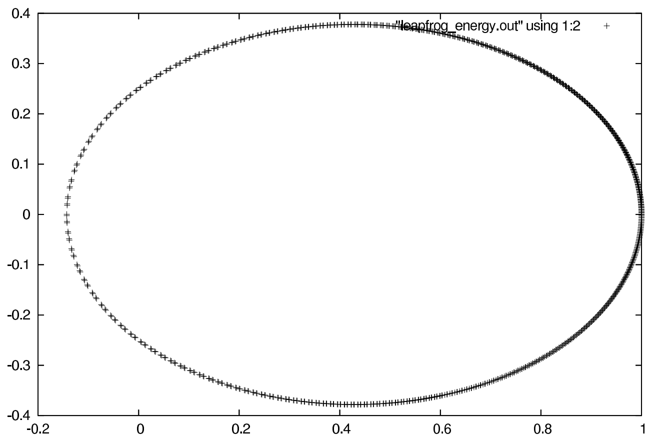

Figure 23.4: Trajectory using a leapfrog algorithm. with $dt = 0.001$.

```
E_tot - E_init = 3.2e-07, (E_tot - E_init) / E_init = -3.65e-07
```

Erica: Even better looking than the equivalent modified Euler orbit.

Carol: And here is how the error grows, as a function of time, in fig 23.5.

Dan: Now the valleys in between the peaks are all of the same height. I can't see any change from the one to the other.

Erica: That makes sense, actually. Remember, the leapfrog is time symmetric. Imagine that the energy errors increased in one direction in time. We could then reverse the direction of time after a few orbits, and we would play the tape backward, returning to our original starting point. But that would mean that in the backward direction, the energy errors would *decrease.* So that would spoil time symmetry: if the errors were to increase in one direction in time, they should increase in the other direction as well. The only solution that is really time symmetric is to let the energy errors remain constant, neither decreasing nor increasing.

Carol: Apart from roundoff.

Erica: Yes, roundoff is not guaranteed to be time symmetric. But as long as we stay away from relative errors of the order of 10^{-15}, what I said should hold accurately. This most be the explanation for the fact that the baseline errors in fig 23.5, in between the periastron peaks, remain so level.

Carol: Time to check what happens for a ten times smaller time step:

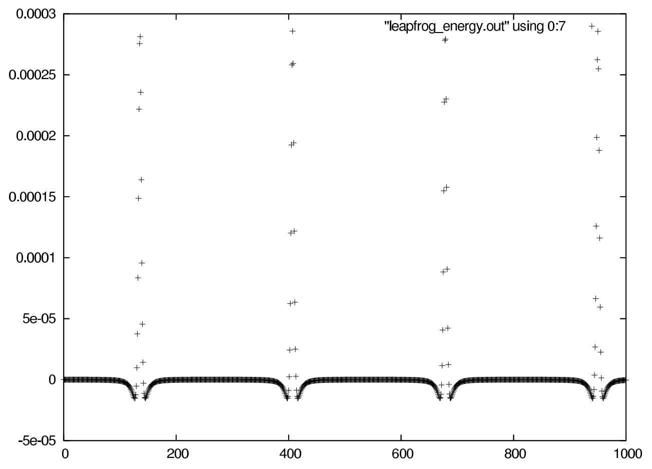

Figure 23.5: Energy error growth using a leapfrog algorithm. with $dt = 0.001$.

```
|gravity> ruby leapfrog_energy.rb > leapfrog_energy2.out
time step = ?
0.0001
final time = ?
10
  t = 0,    E_kin = 0.125, E_pot = -1; E_tot = -0.875
             E_tot - E_init = 0, (E_tot - E_init) / E_init = -0
  t = 10,    E_kin = 0.554, E_pot = -1.43; E_tot = -0.875
             E_tot - E_init = 3.2e-09, (E_tot - E_init) / E_init = -3.65e-09
```

Everything does seem to scale as we expect from a second-order scheme: the height of the peaks is a hundred times less, in fig 23.6 compared to fig 23.5.

Erica: And so is the scaling for the total error at the end of the whole run: it, too, is a hundred times smaller.

23.4 Time Symmetry

Dan: But wait a minute, you just argued so eloquently that the leapfrog algorithm should make almost no error, in the long run.

Erica: Yes, either in the long run, or after completing exactly one orbit – or any integer number of orbits, for that matter. But in our case, we haven't done either. After ten time units, we did not return to the exact same place.

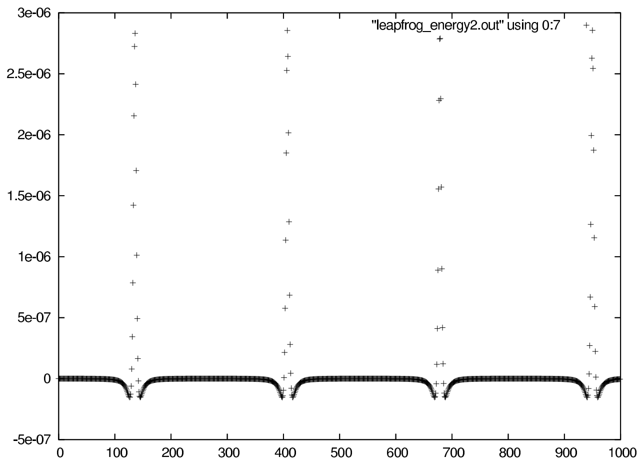

Figure 23.6: Energy error growth using a leapfrog algorithm. with $dt = 0.0001$.

You see, during one orbit, the leapfrog can make all kind of errors. It is only after one full orbit that time symmetry guarantees that the total energy neither increases or decreases. And compared to other algorithms, the leapfrog would do better in the long run, even if you would not stop after an exact integer number of orbits, compared to a non-time-symmetric scheme, such as modified Euler. The latter would just keep building up errors at every orbit, adding the same systematic contribution each time.

Dan: I'd like to see that. Let's integrate for an exact number of orbits.

Erica: Good idea. I also like to test the idea, to see how well it works in practice. But first we have to know how long it takes for our stars to revolve around each other.

Let's see. In the circular case that we have studied before, while we were debugging, we started with the following initial conditions, where I am adding the subscript c for circular:

$$\mathbf{r}_c(0) = \{1, 0, 0\} \qquad ; \qquad \mathbf{v}_c(0) = \{0, 1, 0\} \tag{23.1}$$

With Eq. (20.17), we have for the total energy, actually the total specific energy:

$$E_c = \tfrac{1}{2}v_c^2(0) - \frac{1}{r_c(0)} = \tfrac{1}{2} - 1 = -\tfrac{1}{2} \tag{23.2}$$

On a circular orbit, the speed remains constant. Since the circumference of the orbit has a length $L = 2\pi$. The orbital period, the time to go around the whole orbit, is therefore:

$$T_c = \frac{L}{v_c(0)} = \frac{2\pi}{1} = 2\pi \tag{23.3}$$

We can now do a similar analysis for the current case in which we are integrating an eccentric orbit, for which I will use the subscript e. We start with the same position, but with an initial speed that is only half as large:

$$\mathbf{r}_e(0) = \{1,0,0\} \qquad ; \qquad \mathbf{v}_e(0) = \{0,\tfrac{1}{2},0\} \tag{23.4}$$

This means that the total energy is now:

$$E_e = \tfrac{1}{2}v_e^2(0) - \frac{1}{r_e(0)} = \frac{1}{8} - 1 = -\frac{7}{8} \tag{23.5}$$

For the circular orbit, the diameter has a length of 2, and therefore the semi-major axis has a length $a_c = 1$. Since the semi-major axis a of a Kepler orbit is inversely proportional to the total energy, we can use the results from the circular orbit to find the semi-major axis for our eccentric orbit:

$$a_e = \frac{E_c}{E_e}a_c = \frac{-\frac{1}{2}}{-\frac{7}{8}}1 = -\frac{4}{7} \tag{23.6}$$

Finally, Kepler's third law tells us that the orbital period scales with the three-halves power of the energy, so $T_e/T_c = (a_e/a_c)^{3/2}$, or:

$$T_e = \left(\frac{a_e}{a_c}\right)^{3/2} T_c = 2\left(\frac{4}{7}\right)^{3/2}\pi \tag{23.7}$$

23.5 Surprisingly High Accuracy

Dan: Let's bring up a calculator on your screen.

Carol: Why not stay with Ruby and use `irb`? We can use the fact that acos(0) $= \pi/2$:

```
|gravity> irb
include Math
Object
pi = 2*acos(0)
3.14159265358979
t = (4.0/7.0)**1.5 * 2 * pi
```

```
2.7140809410828
quit
```

Dan: Ah, so four orbits would be close to our total integration time of ten time units, but just a bit longer.

Erica: Yes, and indeed, if you look at fig. 21.1, you can see that our stars have almost completed four orbits by time $t = 10$, but not quite yet.

Carol: Let's see whether we can find a good time to stop. Since we do an output every $\Delta t = 0.01$, it would be nice to find an integer number of orbits that would also be close to a multiple of Δt, so that we can end the integration at that time. I'll try a few values:

```
|gravity> irb
include Math
Object
pi = 2*acos(0)
3.14159265358979
t = (4.0/7.0)**1.5 * 2 * pi
2.7140809410828
4 * t
10.8563237643312
6 * t
16.2844856464968
7 * t
18.9985665875796
quit
```

Ah, seven orbits brings us very close to $t = 19.00$. Okay, let me integrate for 19 time units:

```
|gravity> ruby leapfrog_energy.rb > /dev/null
time step = ?
0.001
final time = ?
19
  t = 0,   E_kin = 0.125, E_pot = -1; E_tot = -0.875
            E_tot - E_init = 0, (E_tot - E_init) / E_init = -0
  t = 19,   E_kin = 0.125, E_pot = -1; E_tot = -0.875
            E_tot - E_init = 2.5e-13, (E_tot - E_init) / E_init = -2.85e-13
```

Erica: That's an amazing accuracy, for such a large time step! Can you try an even larger time step?

Carol: Sure, why not ten times larger:

```
|gravity> ruby leapfrog_energy.rb > /dev/null
time step = ?
0.01
final time = ?
19
  t = 0,   E_kin = 0.125, E_pot = -1; E_tot = -0.875
            E_tot - E_init = 0, (E_tot - E_init) / E_init = -0
  t = 19,   E_kin = 0.125, E_pot = -1; E_tot = -0.875
            E_tot - E_init = 1.02e-10, (E_tot - E_init) / E_init = -1.17e-10
```

Erica: Still a very good result. Remind me, what did we get when we did our shorter standard integration of ten time units?

Carol: Here it is

```
|gravity> ruby leapfrog_energy.rb > /dev/null
time step = ?
0.01
final time = ?
10
  t = 0,   E_kin = 0.125, E_pot = -1; E_tot = -0.875
            E_tot - E_init = 0, (E_tot - E_init) / E_init = -0
  t = 10,   E_kin = 0.553, E_pot = -1.43; E_tot = -0.875
            E_tot - E_init = 3.18e-05, (E_tot - E_init) / E_init = -3.63e-05
```

And indeed a lot worse than integrating for 19 time units. I begin to see the strength of time symmetric integration schemes! Many orders of magnitude gain in final accuracy, as long as you return to the same location in the orbit that you started from!

23.6 Squaring Off

Erica: How about running it ten times longer? I'm curious to see what will happen.

Carol: Let's find another example of an integer number of orbits, close to a multiple of $\Delta t = 0.01$. Here is one: 201 orbits will take a total time $t = 545.530269157643$, close to $t = 545.53$. That should be good enough. Here goes:

```
|gravity> ruby leapfrog_energy.rb > /dev/null
time step = ?
0.01
final time = ?
545.53
  t = 0,   E_kin = 0.125, E_pot = -1; E_tot = -0.875
           E_tot - E_init = 0, (E_tot - E_init) / E_init = -0
  t = 546,   E_kin = 0.126, E_pot = -1; E_tot = -0.875
           E_tot - E_init = 2.21e-08, (E_tot - E_init) / E_init = -2.53e-08
```

Erica: Clearly, we're losing accuracy, but I bet we're doing a lot better with leapfrog than with modified Euler here!

Dan: So now you're starting to bet as well. I think you're right, but let's check:

```
|gravity> ruby euler_modified_energy.rb > /dev/null
time step = ?
0.01
final time = ?
545.53
  t = 0,   E_kin = 0.125, E_pot = -1; E_tot = -0.875
           E_tot - E_init = 0, (E_tot - E_init) / E_init = -0
  t = 546,   E_kin = 0.0293, E_pot = -0.0886; E_tot = -0.0594
           E_tot - E_init = 0.816, (E_tot - E_init) / E_init = -0.932
```

That's pretty dramatic, I'd say! Modified Euler just falls apart, after a couple hundred orbits, for such a large time step. And the leapfrog just keeps going.

Carol: At least for periodic orbits, such as this one. But I must say, I'm impressed too.

Erica: I had wondered why so many people use the leapfrog algorithm. I'm beginning to see that it has some real advantages!

Carol: So what's next. Do we want to start playing with third-order or fourth-order integration schemes?

Dan: I'd rather go beyond the two-body problem, to the real N-body problem. It's time to do start simulating a real star cluster, rather than just one double star!

Chapter 24

Literature References

[to be provided]