Moving Stars Around

A Preliminary Version
of what will expand into
Volumes 1,2,3
of the series

The Art of Computational Science

Piet Hut & Jun Makino

Institute for Advanced Study
1 Einstein Drive
Princeton, NJ 08540
U.S.A.
piet@ias.edu

Univ. of Tokyo, Dept. of Astronomy
7-3-1 Hongo, Bunkyo-ku
Tokyo 113-0033
JAPAN
makino@astron.s.u-tokyo.ac.jp

Preface

This book manuscript is a preliminary version of what will expand into the first three volumes of a ten-volume series *The Art of Computational Science.* We decided to make this version available on the web, so that readers can start to play with the software, and we can get early feedback concerning the direction of our series.

The Art of Computational Science

The Art of Computational Science series will provide a student with a hands-on guide to building a computational laboratory, and doing state-of-the-art research with it. The series will be self-contained: a high-school student should be able to start at page 1, and work her way through the series.

It may seem like a tall order, to use a series of books to lead a student through a process that normally takes five years, from entering college to being able to do research as a graduate student. What we offer here is a shortcut, not a replacement of a traditional curriculum. A motivated high school student or undergraduate could set aside a summer for self-study, or even better, pair up with another student, to start working through this book series. That way, the would quickly get a taste for conducting real independent research. Having acquired the taste and experience will be invaluable for them, no matter what speciality they would choose in the end.

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. Even when detailed exercises are given, together with their solutions, this still leaves out the actual experience of anybody trying to solve a problem, and getting hopelessly astray. Research is (much) more than 90% about making mistakes, and (much) less than 10% about finding something new and original. Educational learning is no different. We all learn from solving excercises, not by the answer we get, but by learning what not to do wrong in the process.

Of course, there is a good reason that text books are presented in the style they are presented. Giving the reader a real-life impression of all (or at least much) that can go wrong in solving problems would expand the size of a book by a factor of ten, if not more.

Therefore, it is certainly a good thing that most text books are as they are, providing reference material for a course, and providing the kernel of knowledge for gifted or already somewhat experienced students who are brave enough to try to learn the material on their own.

However, 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.

Still, we are faced with the problem of cramming five years of student experience in only ten volumes. Our solution has been to take as simple a problem we could think of within science. In fact it is the oldest unsolved problem ever since the day Newton formulated mathematical physics, as the basis of modern science. It is the gravitional many-body problem, in which a collection of celestial bodies move under the influence of their mutual gravitational interactions.

An example of the two-body problem is the motion of the Moon around the Earth, or that of the Earth around the Sun. An example of the three-body problem is that of the Earth, Moon, and Sun now taken together, with the perturbations of the Sun taking into account in following the Moon on its way around the earth. The Sun and the major planets, from Mercury to Neptune, offers an example of a nine-body problem. A rich star cluster, such as M15 or 47 Tuc, provides an example of a million-body problem. And our whole galaxy, as well as our neighbor the Andromeda galaxy, can be modeled as a multi-billion-body gravitational problem.

Having selected a problem that can be explained relatively easily, we plan to introduce the notion of a physical force, the notion of a gravitational force, the idea of a differential equation, as well as the concept of numerical approximations to its solution. This will all appear in Volume 1. Hands-on experience in simulating the dance of two or three bodies will be the topic of Volume 2, while the study of a 32-body system, as a model for a star cluster, will occupy Volume 3. The current manuscript provides a first draft for these three volumes combined, but as yet without the most introductory steps; we hope to provide those soon(ish), in the next few years.

Volume 4 will introduce the notion of individual time steps, a crucial ingredient for simulating larger numbers of particles. We will also focus on a number of technical details in computer science, such as the introduction of self-describing flexible data formats, the use of classes in C++, XML of I/O, more complex command-line arguments, etc. This approach will show how to build robust modules that are flexible enough to function as tools in a growing toolbox that can used to build your own software environment tailered to the type of scientific simulations you would like to perform. We will use these tools to model 256-body systems.

Volume 5 will introduce the last major ingredient needed to model large systems, namely special treatments of tight binaries. With this in place, we can go to 2048-body systems (each increase of a factor eight in 3D doubles the number of particles in each spatial dimension). We will explore applications such as core collapse, and while doing so, we will find the need to add additional ingredients, such as a special treatments of triple star systems.

Volume 6 will introduce a diversion: it will focus on three-body scattering experiments. Just as high-energy particle accelerators probe the behavior of subatomic particles, we can probe the complex interactions between single stars and double stars in the virtual laboratory of the computer, by sending 'beams' of single stars to (gravitationally) crash into 'target plates' of double stars. These experiments are useful to predict and analyse the 'microscopic' processes that happen within a star cluster scales small with respect to that of the cluster as a whole.

Volume 7 will continue the scattering treatment, but will handle the scattering of two double stars off each other, as well as the scattering of doubles and triples, and even more complex varieties, all of which occur within a large-scale simulation of a dense stellar system. As we will see, the automatization of such experiments, together with the automated reporting of the many different outcomes, will provide quite a challenge.

Volume 8 will pick up the main line of our series, the development of N-body codes with increasing sophistication (for historical reasons, the gravitional many-body problem is often called the N-body problem). This time we will start from scratch, using all the experience we have collected in volumes 1 through 7, by making a top-down design in terms of modules, while giving a complete specification of the interfaces between all these modules. This approach will allow us to mix and match different pieces written by different people, using different computer languages, and even adding different physical effects.

Volume 9 will continue this quest, by showing how such a code can be made to deal with the challenge of following close encounters, as well as tight double stars. Here we will learn to deal with various types of special treatments of neighboring particles, which is the main reasons that state-of-the-art N-body codes have grown so bulky. We will apply this code to some 16k-body systems.

Volume 10 will wrap us the quest, by introducing a way to handle even collisions between stars, through simple forms of hydrodynamics and stellar evolution. Depending on the speed of computers by the time we finish our series, we may be able to illustrate our code by taking two more steps of two in each dimension, leading us to the 128k-body system as well as the million-body system.

Extreme Research

Our aim is to break down the barrier between research and education. From an educational point of view, our approach is unusual in that we lead a student along the shortest path from an elementary introduction to the cutting edge of research. And from a research point of view, we illustrate how extensive documentation is beneficial for a research project. To put it in a more radical form: we believe that education is the key to success in research.

To express these ideas, we introduce the term *extreme research*, in analogy of the term *extreme programming*, a methodology aimed at frequent testing and fast turn-around, in which there is much interaction between users and producers of the code being developed. Typically there is a tight coupling between the producers as well, through pair programming, in which two people sit behind a computer screen, developing a piece of code together. In our case, extreme research indicates a similar type of interaction between students developing a simulation code, and between them and their supervisors and colleagues.

A key idea in our notion of extreme research is what we call *variational programming*, which we will discuss below. Briefly, variational programming invites a 'slower but faster' approach, in which we first take our time, feeling our way around a solution path, before settling on a specific approach. The familiarity with the local landscape of possible approaches typically speeds up the process of reaching a simple yet robust final product. Experience has shown us that the time lost in looking around is more than made up by the fact that we thus avoid having to backtrack excessively in the final stages.

A related example of a 'slower but faster' approach is to make it a habit of writing extensive documentation. This applies to adding detailed and frequent comments with the computer code itself, as well as writing manual pages and introductory and explenatory notes. At first, this may seem to slow down the process of rapid prototyping. In fact, by having to explain what you are doing to an anonymous reader, you often reach a much deeper understanding of the problem at hand, by being forced to make your reasoning explicit. In addition, you often find errors in your thinking as well as ways to improve aspects, even if they were correct.

Actually, wide and deep documentation goes hand in hand with variational programming. A description of anything is like drawing an outline around that thing, and the very process of doing so gives one many glimpses of the adjacent terrain. Successful documentation of a code is a form of variational description. And in our current series we have taken this approach to an extreme: one of our main motivations for writing an education ten-volume series was our desire to develop for ourselves a robust software environment for large-scale experiments in stellar dynamics, relying on a process of extreme documentation.

The central engine of this environment would be an N-body code that would be completely documented, down to every decision made. This implies a description of not only what decision was made, made also how it was made, and why. When we thought this true, we realized that we had no choice but to start from scratch. Tempting as it had been to take a code such as the Kira code, the central engine of the starlab software environment (see `http://www.manybody.org/manybody/starlab.html`), it would be impossible to reconstruct all the decisions that had gone into its ten year history. The same considerations hold even more for NBODY4 and related codes in the NBODYx family, developed and maintained by Sverre Aarseth, author of *Gravitational N-Body Simulations : Tools and*

Algorithms (Cambridge Univ. Pr., 2003), since these codes have grown over a forty-year period.

We finally decided to accept the challenge we had posed for ourselves, and to test our notion of extreme programming in practice, by indeed starting from scratch. Here is a very brief outline of our plans. The top view of our code would reveal four parts: a scheduler, a module for the global dynamics, one for the local dynamics, and one for stellar physics (stellar evolution and hydrodynamics). Each of the latter three would be built from several modules. At that level, we would thus have a dozen or more modules, each of which would ideally be so independent that they could be written by different people using different computer languages. Most importantly, these different people would have no need even to communicate with each other, since the top-down specification would indeed be completely specific about how each module would talk to each other module, and what would be expected to be delivered by each one, in what way.

Variational Programming

We have coined the term *Variational Programming* to describe our prefered approach to programming: to follow a path of least action. The name is borrow from physics, where Hamilton's variational principle states that a physical trajectory of a particle is given by a path along which the action is minimal.[1] Our goal is to be lazy in the optimal sense: to minimize the amount of time and energy to along the path of writing a piece of software, from start to finish, *i. e.* from the first idea to the completion of a robust and well-tested product, that can be easily and flexibly used in connection with other pieces of software.

As is the case in physics, the least action principle is a global one, not a local one. The challenge is to minimize the total time required to put together a whole software package. If we want to write the code for an individual model, it is by far the easiest to throw something together quickly and make it work. However, after writing a few dozen modules that way, it quickly becomes clear that making them together smoothly requires more and more work, forcing a repeated rewrite of many of the pieces that originally were easy to dash of.

Clearly, there is an optimum approach. If you spend too much time polishing each module to make it really elegant and beautiful, you will spend (almost) forever before you finish your software package. If you rush each module writing too much, and don't think carefully about how they should fit together, you spend (almost) forever chasing bugs all over the place. The question is: where, in between these two extremes, is the optimum approach, which minimizes the total amount of work?

[1]In physics, the concept of action has a precise mathematical definition, as a time integral of the Lagrangian of a system; and although in physics, too, Hamilton's principle is often called the principle of least action, strictly speaking the action should only be an extremum, so it could be a saddle point, for example, rather than a minimum. However, these technical considerations are not relevant here, since we use the term only as a metaphor.

Finding the optimum is an art, not a science. Hence the title of our series *The Art of Computational Science*. But an art requires particular skills and can be learned, ideally by exposure to many hands-on examples, the more real-life-like the better. And while being exposed to such examples, it does become clear that there are some general rules.

In our experience, which between the two of us includes half a century of frequent scientific simulation code writing, the main rule is: try to make frequent variations. Hence our term *Variational Programming*, which we will now summarize.

The first improvement over rushing into software writing is to test every step carefully, it to make sure it is correct. While this is much better than a rush job, it still is no guarantee at all that the module one is working on will function optimally in a larger setting, connected to other modules.

The next improvement is to look/feel/grope around a solution, looking at neighboring and slightly different approaches. The goal of variational programming is: pragmatic simplicity, avoiding the rigidity of an approach that is too narrow, as well as the complexity of an approach that is too baroque and general-purpose.

In summary: before writing anything, try to get a sense of the landscape of the problem. Then come up with a tentative solution, a toy model. This most likely will be wrong, the first time. But if you don't try something, you'll never get there. However, if you try something while building it in grandious ways, you will waste a lot of time if it turns out to be ill-directed. So be pragmatic: try something complex enough that it can actually do something interesting, given the landscape of the problem, but not much more than that. And above all, don't be attached to your first (few) attempt(s): be prepared to clean the decks and throw stuff away.

After one or more tries, you will get a hands-on feel for the landscape, and you will get an idea how much work will be required to go from A to B along different paths. Only then can you choose the path that (most likely) will require the least action. And while exploring that path, it remains important to keep making local variations, to continue trying to find a more optimal solution.

This holds on all levels, down to the smallest module. When you write something, first of all write in small chunks. For each chunk, while writing and while testing, think about whether it makes sense to generalize it a bit more or streamline it a bit more. Play around with it, both while writing and testing, rather than just following your first idea like an arrow let loose. Arrows are bad at following corners in the road.

So the principle of least action in programming means: Be comfortable and lazy, but only so in the long run. If you write quickly, you'll have to do a lot of nasty debugging later. For most people, this is less comfortable that writing new clean code. But even for those who like to both create and solve problems: you'll spend more time and therefore you'll have to work harder to get at your goal. If you follow the path of least action, you'll introduce bugs, for sure, but they will be interesting complex bugs, since you've avoid the annoying ones (ever spent half an hour searching for a bug, only to find that it was a matter of misspelling something somewhere in an overly long function spilling out over a few pages?).

Another way to characterize variational programming concerns the way to learn from errors. The key is to let the problem tell you how it wants to be solved. Of course, the problem won't talk to you, so you have to start with an initial attempt. But rather than trying to solve a task in the first attempt like blasting a tunnel through all the problems, it often makes more sense to try to see whether the type of problem that comes up by itself can suggest how you can change your approach a bit. That way, you often find opportunities to wind your way between the hills, rather than blasting tunnels in a straight line, as the crow flies or the mole bores, as the case may be.

Finally, we call our approach variational because making variations is in fact the only way to find the path of least action. If there where only a few paths, you could try all or most of them. The problem with paths is, there are not only infinitely many of them, but the family of paths itself even has infinitely many degrees of freedom, or more strictly speaking, about as many degrees of freedom as there are key strokes in your code. So the number of possibilities is, roughly speaking, infinity to the power infinity. The only way to find a reasonably optimal choice in such a vast sea of possibilities is to: 1) try a few global choices of different paths to see which one looks promising; 2) after settling on one, make very many local variations to explore in much greater detail the space of paths around the one you're beating. Each local variation gives you a choice, and with sufficient experience your final result will be the product of all the choice you have made along the way, selecting your approach from among a very large number of ways to write your code: a factor of a few to the power of the number of choices you have made!

An Open Source Project

Our book series explores a new approach to both research and education in computational science, bringing out simultaneously the science, arts and crafts aspects of research in the form of an the educational offering. The whole series, together with the corresponding software, will be presented on the web as an open source project. Others who share our enthusiasm are welcome to develop their own contributions. We sincerely hope that some of those will extend our approach both locally and globally: contributing modules to our problem at hand, in stellar dynamics, in stellar hydrodynamics and in stellar evolution; and starting new projects in other areas of astrophysics, indeed in other areas of science as a whole.

It would be our greatest reward to see others complement our approach. Given our emphasis on modularity and clear specification of interface protocols, each contributor will be pretty free in her or his choice of approach, from the type of computer language used to the style of programming (while hopefully adhering to the principle of least action, a principle that certainly allows many different ways to implement it). With a modest amount of care, it should not be hard to let a community of programmers and programs sprout up, servicing many different areas of science in a way that follows a similar spirit.

In addition, an open source approach has proven to provide the best quality control.

As an added benefit, this may make it easier to explore new scientific questions, by combining existing modules from different disciplines, something that is currently all but impossible, given the different ideosyncracies in the various legacy codes in each discipline. We are in the process of setting up an open source framework for our book series and for the codes that go with it. Further developments will be announced regularly on our web site `http://www.ArtCompSci.org`.

In the current concise version of the first three volumes, we have skipped a detailed explanation of many of the basic steps, such as introductions and precise definitions of the concepts of physical force and differential equations. By forging ahead quickly in that way, we ourselves can get more of an idea of where we may be heading, and we can get feedback from our readers and especially from students who will actually attempt to learn computational science from our series, and who are willing to start with a not-yet-complete product. We therefore very much appreciate hearing from our readers what they like in this volume and where they encounter difficulties of which kind. We can be reached through email at the address `comments@ArtCompSci.org`. We may not be able to answer each reaction personally; that will depend on how much time we have and on how many reactions we get. But rest assured that we will read each email comment, and implement what we learn from that email in the actual volumes that will appear in this series.

This book aims at three groups of readers. For scientists, it gives a concrete example for setting up a full 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 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. Our gamble here is that the music of the spheres may attract an audience audacious enough to follow our cosmic exploration through the various volumes in this series.

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. Perfectly functioning rockets have been blown up because of glitches in complex software systems. Newly built airports have experienced very costly delays, simply because software for baggage transport was delivered a year late. 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. Even better, helping a hand as an apprentice in the kitchen would be even better. This is exactly what this book offers.

We hope that our choice of topic, the do-it-yourself modeling of the full ten-billion-year history of a dense star cluster, will be rewarding. We offer you the controls of a state-of-the-art flight simulator that will allow you to travel through the four-dimensional space-time history of a star cluster. After zooming forwards or backwards in time by a few billions of years, you will be able to visit an interacting triple or quadruple system somewhere in the history of the star cluster. Slowing down simulated time by a factor of a trillion, you can watch the intricate and chaotic gravitational dance of the three or four stars, moving around each other in a matter of days. If you are lucky you may find a couple of neutron stars or black holes among the interacting star group, but you will have to slow down simulated time by yet another five to eight orders of magnitude, since two neutron stars can pass by each other in matters of milliseconds.

All of this and much more will be at your finger tips already through half-way the series of books we are currently writing. You will be able to use cutting-edge astrophysics research tools, together with full access to every line of code. And if you have worked your way through the books in this series, you will not only understand how the whole system works, but you will also understand and appreciate the motivation for every design and implementation decision. From that point on, you will be in a position to extend the current system and to engage in original scientific software design yourself. More importantly, you will have a complete software environment to inspire you if you want to set up your own virtual laboratory in your own scientific discipline, or your preferred business environment.

Acknowledgments: We thank Atakan Gurkan, Douglas Heggie, Vicki Johnson, and Robert Vanderbei for their comments on the manuscript.

Contents

List of Figures

List of Codes

List of Output

List of Exercises

Part I

Background

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 larger a piece of matter is, the more gravitational force it exerts on its surroundings.

This dominance of gravity at long distances makes the job of modeling a chunk of the Universe easier. 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.

This book is the first in a series of books titled *Pure Gravity*, to indicate that we are making this approximation of treating objects as gravitating masses and nothing more. The objects we will be studying are stars, and the environment we will focus on are dense stellar systems, where the stars are so close together that they will occasional collide and in general have frequent interesting and complex interactions. In a later series, *Applied Gravity*, we will look at the internal physics of those stars: how they evolve under the influence of nuclear reactions in their centers, how they may die in cosmic explosions, and what happens to their remnant cores. We will especially study how interactions between stars of all types can change their evolutionary behavior through two-body, three-body, and more complex interactions, leading to an intricate 'star cluster ecology'.

This first book, *Writing an N-Body Code*, lays 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 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

The sun is a star like any other among the hundred billion or so stars in our galaxy. It 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^{-18} per year. In other words, we would have to wait at least 10^{18} years to have an appreciable chance to witness such a collision. Given that the sun is less than five billion years old, 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 hundred 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.

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 hundredth 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 less than 10^{12} years. With globular clusters having an age of more than 10^{10} years, a typical star near the center already has a chance of more than a percent to have undergone a collision in the past.

In fact, the chances are much 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 temporary three- or four-body dance. Once engaged in such a dance, the local stellar crowding is enormously enhanced, and the chance for collisions 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 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 semimajor axis (a measure for the typical separation of the two stars) as well as the orbital eccentricity.

The goal of our book series is to develop the software tools to make it possible to simulate an entire star cluster like M15, and to analyze the resulting behavior both locally and globally.

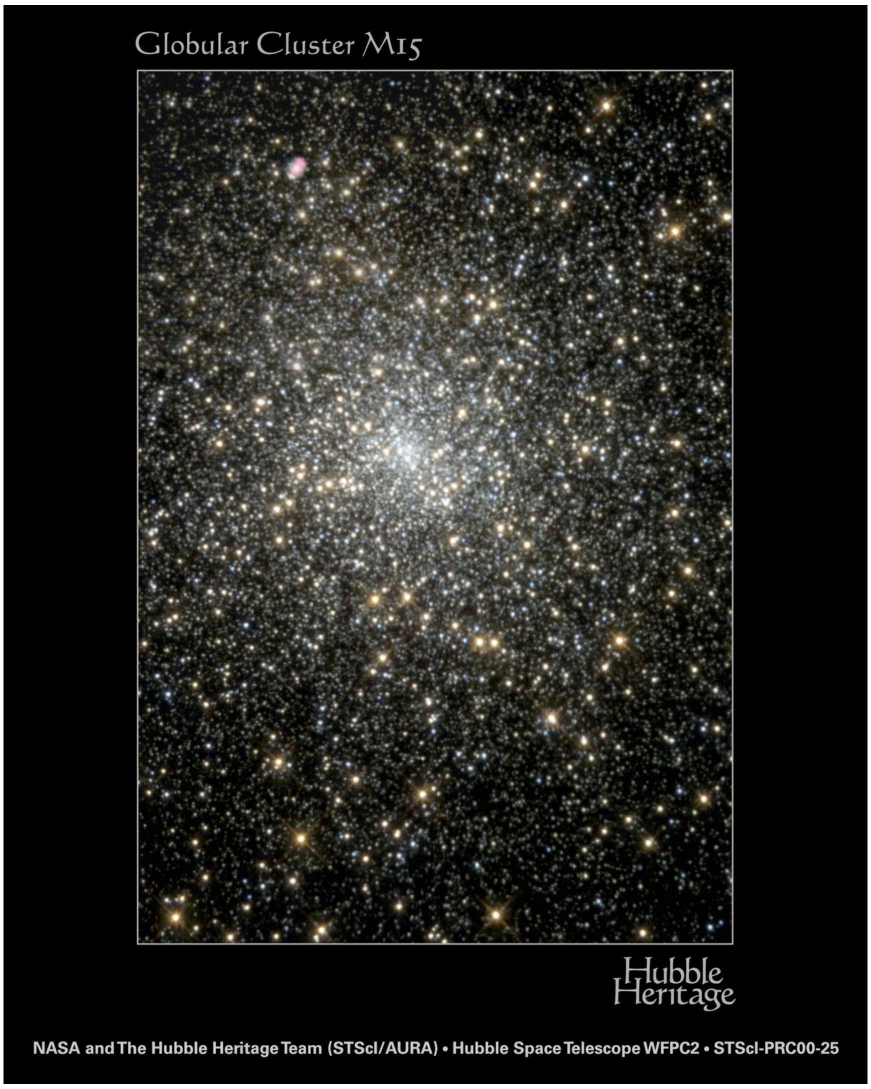
Globular Cluster M15
Hubble Heritage
NASA and The Hubble Heritage Team (STScI/AURA) • Hubble Space Telescope WFPC2 • STScI-PRC00-25

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

1.4 Galactic Nuclei

In Fig. 1.2 we see an image of the very center of our galaxy. This picture is taken with the Northern branch of the two Gemini telescopes, which is located in Hawaii on top of the mountain Mauna Kea.

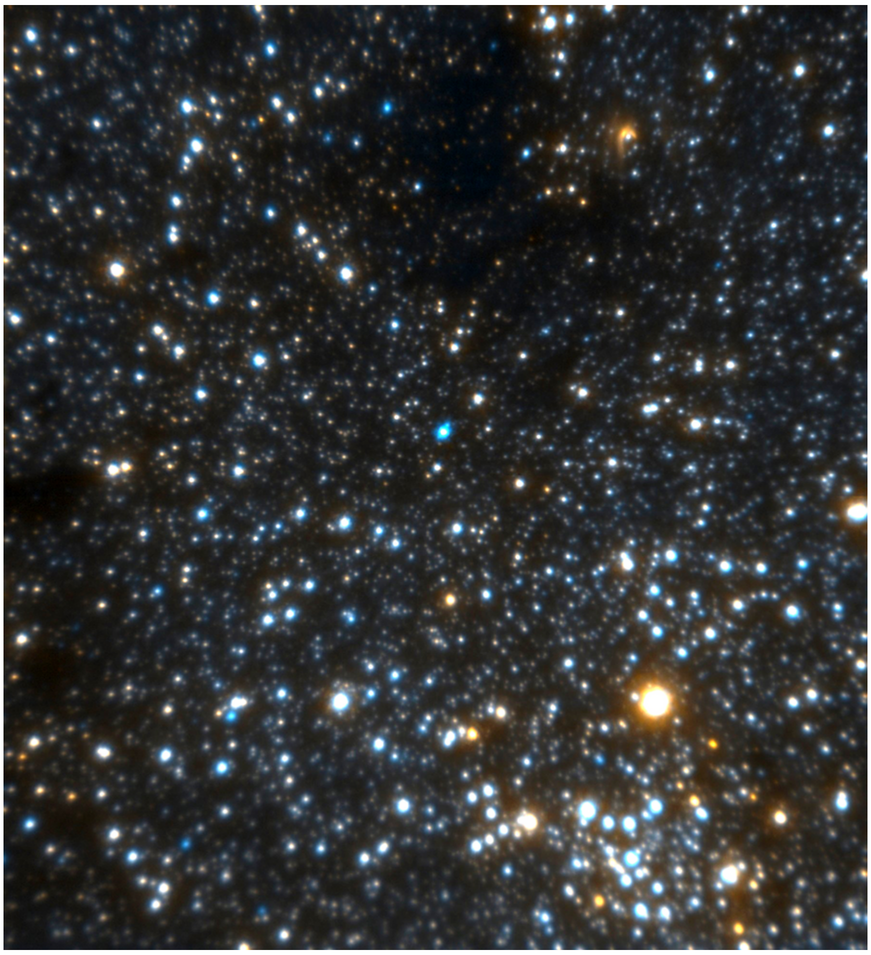

Figure 1.2: An image of the central region of our galaxy, taken with the Gemini North telescope. The center is located on the right just above the bottom edge of the image.

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. Fig. 1.2 is a false-color image, reconstructed from observations in different infrared wavelength bands.

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 also 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. As a consequence, the frequency of stellar collisions is strongly enhanced.

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 starforming 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. The four brightest stars in the center, collectively known as the Trapezium, form the most massive stars of a larger conglomeration of stars, all recently formed from the gas and dust that still surrounds them.

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. While a protostar is still in the process of contracting from the gas

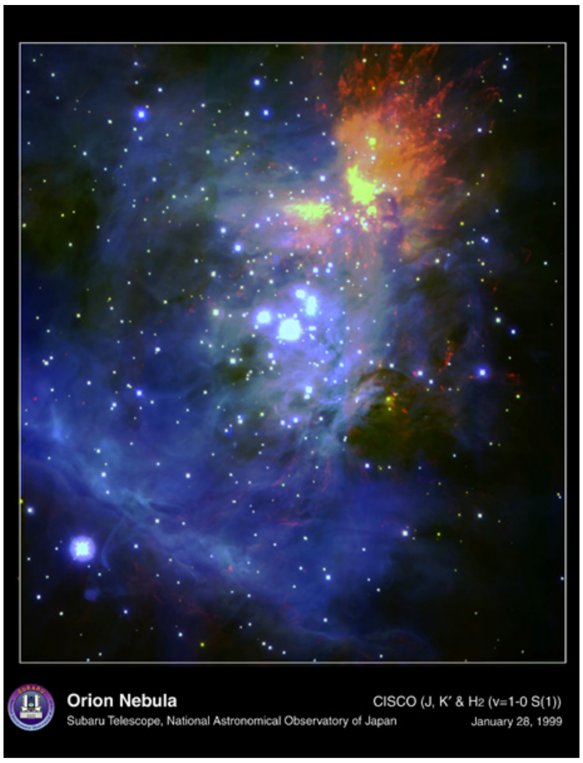
Orion Nebula
Subaru Telescope, National Astronomical Observatory of Japan
CISCO (J, K' & H2 (v=1-0 S(1))
January 28, 1999

Figure 1.3: The Orion Nebula, as seen by the Subaru telescope.

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 initial have over each other. Some of the more massive groups of newly formed stars, however, are sufficiently tightly bound to survive their environmental harassment. They form the so-called open clusters, where there name indicates that they have central densities that are typically less than what we see in globular clusters.

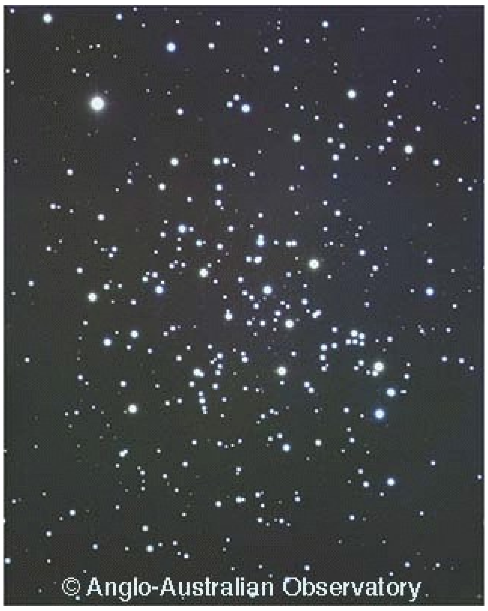

Figure 1.4: The open star cluster M67, in a picture taken by the Anglo-Australian Observatory.

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. 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.

1.7 Writing your own star cluster simulator

Astronomers have almost 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: just integrate Newton's equations of motion for each star, under the influence of the gravitational pairwise interactions of all other stars. Indeed, it is straightforward to write a simple code to do so, 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.

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 is released. Since the two stars in a binary are bound, the potential energy is larger than the kinetic energy, and the total energy is negative. 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 either crash or resort to a very slow and inefficient crawl. The changes in the relative configuration of the two stars occur so frequent that they can be easily missed. If they are modeled correctly, by choosing a tiny time step, the rest of the system will be forced to slow down, while spending a large and unnecessary amount of computer time on all the other stars that don't need such fine resolution. By the end of the sixties, these problems were 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. But this 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 crash the system if no precautions have been taken for such a situation to happen.

We can continue the list of tricks that have been invented to allow every larger and denser systems to be modeled correctly. 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.

In this first 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. In subsequent volumes in the series, we will introduce the other refinements mentioned above. 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.

Part II

Exploring N-Body Algorithms

Chapter 2

Getting Started on the N-Body problem

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, for a few billion years, say.

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 patience from the reader, since it will take a while to have a full package in hand for modeling, say, the long-term behavior of a star cluster. This drawback, we feel, is more than offset by the advantages of our approach:

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

2.1 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, Alice talks with great animation to her friends Bob and Carol. Alice is an astrophysics major, Bob is preparing for business school, and Carol majors in computer science.

Alice: Guess what! Today I was asked to choose a junior 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.

Bob: What are the interactions that are so interesting?

Alice: 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?

Alice: 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 more 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.

Bob: 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.

Alice: 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.

Alice: Yes, that would be better. In fact, I already had a brief chat with my supervisor, 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.

Bob: 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.

Bob: 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?

Alice: I asked that question, too. My supervisor told me that a through-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 fermenting. 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 write the original code won't have the time to do a complete rewrite; they 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 fermenting as you just called it?

Alice: Exactly.

Bob: You know what? Perhaps this is a silly thing to suggest, but I suddenly got an idea. It seems that Alice 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 Alice honest?

Bob: Exactly! Of course, she may well get way ahead of us into all kind 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, Alice, are we rudely intruding upon your new project?

Alice: No, on the contrary! As long as I keep my actual project separated, as Bob 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 wouldn't inject new ideas I hadn't thought about, or notice old ideas of mine that can be improved.

Bob: Okay, 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.

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

2.2 The Gravitational N-Body Problem

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

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

Bob: I understand that we are dealing with something gravitational attractions between celestial bodies, but what is the problem with that?

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

Alice: Traditionally, in mathematics and mathematical physics, when we pose a question, we call it a problem, as in a home work problem. 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.

Bob: What if the stars collide?

Alice: 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 (see volume 2). However, when we start with random initial conditions, the chance of such a collision is negligible.

Carol: But real stars are not points?

Alice: True. At 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. In practice, to discriminate between the physical modeling of stars and the replacement of them with point masses, we often call those points 'bodies'.

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.

Carol: Fine. Let's pick a language and start coding! I bet you physics types insist on using fortran?

Alice: Believe it or not, most of the code to be overhauled has been written in C++, and I suggest that we adopt the same language. It may not be exactly my favorite, but it is at least widely available, well supported, and likely to stay with us for decades.

Bob: What is C++, and why the obscure name? Makes the notion of an N-body 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 ...

Bob: ... are you making a pun on our names?

Carol: No, I'm not kidding. Then C was extended to a new language for object-oriented programming, something we'll talk about later. In a nerdy pun, the successor operation "++" from the C language was used to indicate that C++ was the successor language to C. Don't look at me, we'll have to live with it.

2.3 The Gravitational 2-Body Problem

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.C` in her favorite editor. Expectantly, she looks at Alice, sitting to her left, for instructions, but Bob first raises a hand.

Bob: 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.

Alice: 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. Instead of dealing with position $\mathbf{r}_1$ for the first particle and $\mathbf{r}_2$ for the second particle, we can write down the gravitational attraction between the two in terms of the relative position, defined as:

$$\mathbf{r} = \mathbf{r}_2 - \mathbf{r}_1 \tag{2.1}$$

Newton's gravitational equation of motion then becomes:

$$\frac{d^2}{dt^2}\mathbf{r} = -G\frac{M_1 + M_2}{r^3}\mathbf{r} \tag{2.2}$$

This is a second-order differential equation. At the left-hand side you see the second derivative of position with respect to time t. The first time derivative of position $\mathbf{r}$ is the velocity $\mathbf{v} = d\mathbf{r}/dt$ while the second derivative presented here is the acceleration $\mathbf{a} = d\mathbf{v}/dt = d^2\mathbf{r}/dt^2$. At the right hand side, the masses of the two particles are indicated by M_1 and M_2, respectively. G is the value of Newton's gravitational constant.

I'm glad you both have at least some familiarity with differential equations, in the context of classical mechanics. 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 deep 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.

Bob: The differential equation does indeed look familiar, but why is there a power 3 in the denominator? I thought that Newtonian gravity is an inverse square power, so I would have expected a power 2 down there.

Alice: Good question! We are working here in three dimensions, because that is how many dimensions space in the universe has. The bold-face notation **r** indicates that we are dealing with three-dimensional vectors. If we name the components as follows,

$$\mathbf{r} = \{x, y, z\} \tag{2.3}$$

then the scalar distance between the two particles is defined by

$$r = |\mathbf{r}| = \sqrt{x^2 + y^2 + z^2} \tag{2.4}$$

And while it is true that Newtonian gravity is a $1/r^2$ force, we have to tell the particles not only the magnitude of their mutual attraction, but also the direction in which they pull each other. This is accomplished by adding the last factor **r**. To compensate for the fact that **r** grows linearly with the distance, we have to add an extra power in the denominator.

Carol: Is there no cleaner way to write this equation, making the $1/r^2$ nature of the interaction more transparent?

Alice: Sure there is. We can introduce a so-called unit vector, which by definition has a length of one unit in our coordinate system. This is a good tool for dealing with directions without introducing changes in magnitude. The unit vector corresponding to **r** is given as $\hat{\mathbf{r}} = \mathbf{r}/r$, and the equation of motion for our 2-body problem now reads:

$$\frac{d^2}{dt^2}\mathbf{r} = -G\frac{M_1 + M_2}{r^2}\hat{\mathbf{r}} \tag{2.5}$$

Bob: That looks more like the real thing.

Carol: Yes, but it may be easier to program the previous expression, so let's keep both on the table for now, and see what's most convenient.

Alice: 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:

$$\begin{aligned} G &= 1 && (2.6) \\ M_1 + M_2 &= 1 && (2.7) \end{aligned}$$

Our original equation of motion now becomes simply:

$$\frac{d^2}{dt^2}\mathbf{r} = -\frac{\mathbf{r}}{r^3} \tag{2.8}$$

Chapter 3

Exploring $N = 2$ with a Forward-Euler Algorithm

3.1 Choosing an Algorithm

Carol: Now that we have the equations of motion for the relative position of one particle with respect to the other, I guess we need an algorithm to integrate these equations.

Alice: Indeed, and 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.

Bob: What is the order of an algorithm?

Alice: 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.

Alice: 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 turn out to be optimal for the job.

Bob: How about starting with the lowest-order integrator we can think off? 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.

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

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

Alice: 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.

Bob: 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?

Alice: You can only 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, *i. e.* make half the mistakes of each of the two, while trying to strike the right balance between forward and backward Euler?

Alice: 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:

$$\mathbf{r}_{i+1} = \mathbf{r}_i + \mathbf{v}_i dt \tag{3.1}$$

$$\mathbf{v}_{i+1} = \mathbf{v}_i + \mathbf{a}_i dt \tag{3.2}$$

for the position $\mathbf{r}$ and velocity $\mathbf{v}$ 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 $\mathbf{a}$. 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.C`, let's call it `forward_euler1.C`.

Bob: Alice, go for it!

3.2 Writing a Code

After some further discussion among the three, here is the code that Alice typed in. Actually, the first version contained a few errors, not surprisingly, but to speed up a bit, we will just show here the final product, by printing the file `forward_euler1.C`:

Code 3.1 (forward_euler1.C)

```
//-----------------------------------------------------------------------------
// forward_euler1.C
//-----------------------------------------------------------------------------
#include  <iostream>
#include  <cmath>
using namespace std;

int main()
{
    double r[3], v[3], a[3];
    double dt = 0.01;

    r[0] = 1;
    r[1] = 0;
    r[2] = 0;
    v[0] = 0;
    v[1] = 0.5;
    v[2] = 0;

    for (int ns = 0; ns < 1000; ns++){
        double r2 = r[0]*r[0] + r[1]*r[1] + r[2]*r[2];
        for (int k = 0; k < 3; k++)
            a[k] = - r[k] / (r2 * sqrt(r2));
        for (int k = 0; k < 3; k++){
            r[k] += v[k] * dt;
            v[k] += a[k] * dt;
        }
        cout << r[0] << " " << r[1] << " " << r[2] << " ";
        cout << v[0] << " " << v[1] << " " << v[2] << endl;
    }
}
//-----------------------------------------------------------------------------
```

Let us look at each line in turn. The first `#include` statement is needed in C++ in order to gain access to the I/O library, to make sense of output statements like ``` ``cout <<'' ``` at the end of the code. The second `#include` statement gives access to the math library that contains functions such as the square root `sqrt()`. The third line is a default C++ way of indicating that we use the standard name space. Later, when our software environment will have grown significantly, it will make sense to introduce our own name spaces, but for now the default choice suffices.

There is only one function in this file, the `main()` function which has to be present in every C++ program. It is defined as type `int`, which means that it returns an integer value (by default defined to be 0 upon successful completion, and nonzero otherwise). The three arrays `r[3], v[3], a[3]` store the values for the three Cartesian components of the relative position, velocity and acceleration of one particle with respect to the other. These are declared as `double`, the C++ way to indicate 8-byte floating point variables.

In this simple program there is no need for any input: all initial values are chosen already in the program. The time step `dt` is fixed to be 0.01. The relative position is chosen along the x axis at a distance of unity from the origin where the other particle resides in the relative coordinate system that we use. The velocity is chosen to be 0.5 in the direction of the positive y axis. This is less than the value $v_y = 1$ needed to sustain a circular orbit, as you can check in any book on celestial mechanics or stellar dynamics. The choice of a velocity less than the circular velocity means that we have captured the relative motion at apocenter, which is the point in the orbit at which the two particles are furthest away from each other. The word is derived from the Greek $\alpha\pi o$ *(apo)* meaning 'far (away) from'. The subsequent motion will bring the two particles closer together, until the pericenter is reached on the negative x axis, which is the point in the orbit at which the two particles are closest. This word is derived from the Greek $\pi\epsilon\rho\iota$ *(peri)* meaning 'around' or 'near'. After passing through pericenter the particles will move away from each other again.

The `for` loop takes 1000 time steps that are counted by the variable `ns`, covering a time interval of 10 units. The body of the loop shows how we first compute the square of the scalar distance between the two particles by summing the squares of the Cartesian components. We then take the square root, and multiply that with the square of the distance to get the third power of the distance, which we have seen in the denominator of Eq. 2.8 on p. 22, which we repeat here for convenience:

$$\frac{d^2}{dt^2}\mathbf{r} = -\frac{\mathbf{r}}{r^3} \tag{3.3}$$

In our code we find this equation back in component form as:

```
        a[k] = - r[k] / (r2 * sqrt(r2))
```

Finally, the integration step is completed by executing Eqs. 3.1, 3.2 on p. 24 above, to update the relative position and velocity. At the end of each time step, we print the three position and three velocity components, all on a single line.

To compile the code, we must invoke a C++ compiler. In a Linux environment, the natural compiler is the GNU C++ compiler that is called by the `g++` command:

```
g++ -o forward_euler1 forward_euler1.C
```

which will produce an executable file `forward_euler1`, if no errors are detected during compilation. Let us follow the discussion of our three friends, from the moment that their code first compiled.

3.3 Running a Code

Carol: That's it, the last bug caught, our `forward_euler1` compiled at last! Let's see what it produces. Since we asked for the output of positions and velocities for 10 time units at intervals `dt = 0.01`, our screen will be flooded by 1000 lines of output; better to redirect the output to a file. Let's call it `forward.out`, and let's have a look at the beginning and the end of the file, say five lines each:

Output 3.1

```
|gravity> forward_euler1 > forward.out
|gravity> head -5 forward.out
1 0.005 0 -0.01 0.5 0
0.9999 0.01 0 -0.0199996 0.49995 0
0.9997 0.0149995 0 -0.0300001 0.49985 0
0.9994 0.019998 0 -0.0400027 0.4997 0
0.999 0.024995 0 -0.0500088 0.4995 0
|gravity> tail -5 forward.out
7.66125 -6.25475 0 0.812523 -0.574458 0
7.66937 -6.26049 0 0.812443 -0.574393 0
7.6775 -6.26623 0 0.812364 -0.574329 0
7.68562 -6.27198 0 0.812286 -0.574264 0
7.69375 -6.27772 0 0.812207 -0.5742 0
```

Bob: Hey, Alice told us that our two particles were at apocenter initially, at distance 1, which meant that they could come closer but that they would never get further away from each other than unity in our length units. And here we have distance vectors larger than 7, actually close to 10 if I remember my Pythagoras formula well enough, in the last line.

Carol: You are right, slightly more than 9.9:

```
|gravity> bc
7.69375*7.69375 + 6.27772*6.27772
98.6035574609
sqrt(98.6035574609)
9.92993239961380603154
|gravity>
```

Bob: Unix takes a while to get used to! What does the two-letter 'word' `bc` mean?

Carol: Oh, `bc` is a more powerful version of `dc`, which stands for 'desk calculator' and is a simple utility to quickly find arithmetic answers without writing a program. Here is what the manual page for `dc` tells us:

```
|gravity> man dc
NAME
     dc - desk calculator

SYNOPSIS
     dc  [ filename ]

DESCRIPTION
     dc  is an arbitrary  precision  arithmetic  package.  Ordi-
     narily  it operates on decimal integers, but one may specify
     an input base, output  base,  and  a  number  of  fractional
     digits  to  be maintained. The overall structure of dc  is a
     stacking (reverse Polish) calculator.  If  an  argument  is
     given,  input  is  taken  from that file until its end, then
     from the standard input.

     bc  is a preprocessor for dc  that provides  infix  notation
     and  a  C-like  syntax  that implements functions.  bc  also
     provides reasonable control  structures  for  programs.  See
     bc(1).
|gravity>
```

Carol: With `dc` we could not have used the `sqrt()` function, so we used `bc`; perhaps it stands for a 'better calculator', who knows.

Bob: Well, our program compiles and our program runs, so we have passed two barriers, but now it gives answers that are not sensible. The third step that could have gone wrong is that we have somehow made a mistake in our algorithm, mistyping it, or even being misguided as to its structure. But with the simplicity of Eqs. 3.1, 3.2 it seems clear that neither is the case here. The equations tell the particle to 'follow its nose' in a straight line during each time step. What could be simpler!

Alice: You are right. After syntactic compile time errors, dynamic run time errors, and semantic errors in coding up the algorithm correctly, the fourth potential source of errors lies in using the algorithm. While it seems that we took an almost ridiculously small time step, the choice `dt = 0.01` may still be too large. After all, `dt` is the only free parameter in our program. The code is like a laboratory instrument, a black box with only one dial, so we may as well play with different settings of the dial.

Bob: Indeed, we had no particular reason to choose the value we did, rather than any other. But before we start playing, can we plot a picture of the orbit we just computed? I'd rather have a real sense of what is going on, beyond staring at a few output numbers.

Carol: Will do, but may I add another option to Alice's detective analysis? Apart from the four types of error that she mentioned, there is another possibility. We are not really sure that the program we ran was the one we wrote. The default search path on this computer may be set such that we wind up invoking another program with the same name.

Bob: Also called `forward_euler1`? That seems extremely unlikely!

Carol: But as they say in the Hitchhiker's guide to the galaxy, not impossible. Let me try, and force execution from the current directory "." by adding it to the name of our executable:

Output 3.2

```
|gravity> ./forward_euler1 | tail -2
7.68562 -6.27198 0 0.812286 -0.574264 0
7.69375 -6.27772 0 0.812207 -0.5742 0
```

Carol: Okay, no problem here. I just thought I'd bring up something I happened to learn in class today. Now let's look at a picture of the orbit. I suggest using `gnuplot`, present on any Linux running system, and something that can be easily installed on many other Unix systems as well. 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. However, 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, let me write the line `set size ratio -1` in a file called `.gnuplot` in my home directory. That way we don't have to type it each time we use gnuplot. Okay, done. Now let's have our picture:

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

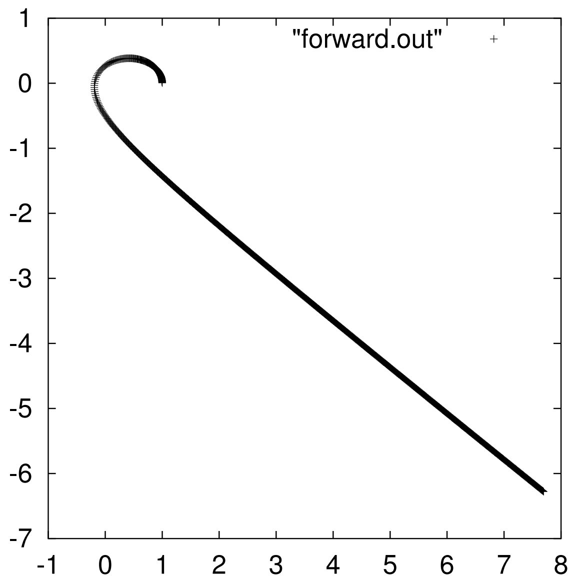

Figure 3.1: Relative orbit for the first attempt to integrate a two-body system with a forward-Euler integrator, with time step $dt = 0.01$

Bob: Wow, the two stars were flung away as by a slingshot, after only slightly more than half an orbit. I guess that indeed the time step was too large.

Alice: I think you are right. Notice that the artificial slingshot, caused by numerical errors, occurred just after pericenter, where the curvature of the orbit is highest. Stepping along tangentially to the orbit tends to let you spiral out from the original orbit, and this effect is higher when the orbital curvature is higher.

Carol: Okay, I'll change the time step size.

3.4 Extending a Code

Carol: In addition, rather than changing the program each time we change step size, I'll make it an option to be provided by the user. How about this version:

Code 3.2 (forward_euler2a.C)

```
//-----------------------------------------------------------------------
// forward_euler2a.C
//-----------------------------------------------------------------------
#include  <iostream>
#include  <cmath>
using namespace std;

int main()
{
    double r[3], v[3], a[3];
    double dt;

    cout << "Please provide a value for the time step" << endl;

    cin >> dt;

    r[0] = 1;
    r[1] = 0;
    r[2] = 0;
    v[0] = 0;
    v[1] = 0.5;
    v[2] = 0;

    for (int ns = 0; ns < 1000; ns++){
        double r2 = r[0]*r[0] + r[1]*r[1] + r[2]*r[2];
        for (int k = 0; k < 3; k++)
            a[k] = - r[k] / (r2 * sqrt(r2));
        for (int k = 0; k < 3; k++){
            r[k] += v[k] * dt;
            v[k] += a[k] * dt;
        }
        cout << r[0] << " " << r[1] << " " << r[2] << " ";
        cout << v[0] << " " << v[1] << " " << v[2] << endl;
    }
}
//-----------------------------------------------------------------------
```

Bob: And even with a polite request to remind the user what is needed. Very nice. Let's compile, run, and plot!

Carol: Okay. Let's direct to output to a file `forward2a.out`, and let's rename the old output file `forward1.out`, to get a consistent numbering system, where the number in the source file corresponds to the same number in the output file; the 'a' after '2' is added here since different input options will produce different output files.

Bob: How about making it even clearer and add the value of the time step that we input to the name?

Carol: Yes, even better. Let's start with $dt = 0.01$, the same value we used by default in `forward_euler1.C`. It never hurts to check whether we indeed get the same result. Here goes:

```
|gravity> g++ -o forward_euler2 forward_euler2.C
|gravity> mv forward.out forward1.out
|gravity> forward_euler2 > forward2_0.01.out
```

Bob: So much for the polite reminder. What is happening, is the computer so slow? Or is it just hanging?

Alice: Ah, I see. By redirecting the output to `forward2_0.01.out`, we are also redirecting our reminder to that file, spoiling the format of our nbody snapshot output, and losing our reminder.

Carol: Yes, of course, you are right. Well, how about putting the reminder on the error stream `cerr`, rather than the output stream `cout`? While reminding you of something is not an error, it does provide a handy way to disentangle the snapshot output, meant to be read further by a computer, and the reminder output, aimed at human consumption.

Bob: Good idea. But while we do that, let's make another modification as well. I just realized that our halting criterion is completely wrong. If we specify `dt = 0.0001`, the system would bail out after only 1000 steps, as before, thus advancing the time from zero to 0.1, barely enough to see the particles move, and far too little to complete even one orbit.

Alice: Right you are. Let's put in a time counter, so that we can keep integrating till time 10, for good measure.

Carol: Fine! And I'll rename the previous version `forward_euler2a.C`, while keeping the name `forward_euler2.C` for the corrected version. When you are developing code, it is always a good idea to keep older versions lying around, even if they are wrong. After all, subsequent versions may turn out to be even more wrong, in which case you might well want to go back to the less wrong one! Here we are:

Code 3.3 (forward_euler2.C)

```
//-----------------------------------------------------------------------------
// forward_euler2.C
//-----------------------------------------------------------------------------
#include  <iostream>
#include  <cmath>
using namespace std;

int main()
{
    double r[3], v[3], a[3];
    double dt;

    cerr << "Please provide a value for the time step" << endl;

    cin >> dt;

    r[0] = 1;
    r[1] = 0;
    r[2] = 0;
    v[0] = 0;
    v[1] = 0.5;
    v[2] = 0;

    for (double t = 0; t < 10; t += dt){
        double r2 = r[0]*r[0] + r[1]*r[1] + r[2]*r[2];
        for (int k = 0; k < 3; k++)
            a[k] = - r[k] / (r2 * sqrt(r2));
        for (int k = 0; k < 3; k++){
            r[k] += v[k] * dt;
            v[k] += a[k] * dt;
        }
        cout << r[0] << " " << r[1] << " " << r[2] << " ";
        cout << v[0] << " " << v[1] << " " << v[2] << endl;
    }
}
//-----------------------------------------------------------------------------
```

Carol: And now let’s try again:

Output 3.3

```
|gravity> g++ -o forward_euler2 forward_euler2.C
|gravity> forward_euler2 > forward2_0.01.out
forward2_0.01.out: File exists.
|gravity> rm forward2_0.01.out
rm: remove ‘forward2_0.01.out\’\? y
|gravity> forward_euler2 > forward2_0.01.out
Please provide a value for the time step
0.01
|gravity> tail -2 !$
tail -2 forward2_0.01.out
7.69375 -6.27772 0 0.812207 -0.5742 0
7.70187 -6.28346 0 0.812128 -0.574136 0
```

Bob: That's curious. Almost the same results as before, but not quite. Remember, earlier we got:

Output 3.4

```
7.68562 -6.27198 0 0.812286 -0.574264 0
7.69375 -6.27772 0 0.812207 -0.5742 0
```

Bob: Ah, I see: same results, after all, but one more time step. Our next to last line is identical to our previous last line.

Carol: That would imply that the new file should be one line longer. Let's check by doing a word count (actually a line-word-character count):

Output 3.5

```
gravity> wc forward1.out  forward2_0.01.out
   1000    6000   39633 forward1.out
   1001    6006   39673 forward2_0.01.out
   2001   12006   79306 total
```

Alice: Indeed: one line more and six words more, for the extra position and velocity components. I guess our switch from integer book keeping with the number of time steps, to floating-point book keeping by using the time introduced some slight round-off which caused the program to do one more step.

Bob: Sounds plausible, but let's check it out. Seeing is believing! Shall we put in an extra output statement on `cerr`? This time we're after errors after all, namely round-off errors. Let's add the following line at the end, after the two lines starting with `cout`:

```
cerr << t << endl;
```

Carol: Done. I'll call the resulting code `forward_euler2b.C`, following our tradition to keep all version around for a while. Let's run the code again, but I won't bother about the snapshot output, since we already know what that is. Redirecting the `cout` output to `/dev/null` is a time-honored Unix way to get rid of it. The 'null' device acts like a black hole, getting rid of whatever falls into it, without a trace. We are then left only with the `cerr` output on the screen.

Output 3.6

```
|gravity> g++ -o forward_euler2b forward_euler2b.C
|gravity> forward_euler2b > /dev/null
Please provide a value for the time step
0.01
0
0.01
0.02
0.03
. . . . . .
9.96
9.97
9.98
9.99
10
```

Bob: Hah! So much for your clever theory, Alice. It stopped at `t = 10`, right on the mark. So round-off cannot have been the reason.

Alice: Not so quick in claiming victory, Bob. Look at the code. In the `for` loop the halting criterion is `t < 10`. If there would have been no round-off, the code should have stopped at `t = 9.99`, not at `t = 10`.

Bob: That makes sense. And yet there is no round-off. It seems like we are both right, but that is impossible. What gives?

Carol: In case of doubt, find it out. I have a sneaky suspicion that there is another round-off that is fooling us: the limited accuracy of the output. The quickest way to find out is to modify the time output line. Instead of simply printing the time, let's print the difference of the real time `t` and the non-round-off value `t = 10`. So our last error output line becomes:

```
cerr << t - 10.0 << endl;
```

Output 3.7

```
|gravity> g++ -o forward_euler2c forward_euler2c.C
|gravity> forward_euler2c > /dev/null
Please provide a value for the time step
0.01
-10
-9.99
-9.98
-9.97
. . . . . .
-0.04
-0.03
-0.02
-0.01
-1.68754e-13
```

Bob: Okay, Alice was right after all. Close but no cigar. After 1,000 steps, the time is updated to a hair's breadth smaller than 10, and the code takes one extra step. Clever detective work, Carol!

Carol: I'm glad my course in numerical methods was good for something!

Bob: By the way, returning to the commands you gave earlier, just before we spotted the extra output line, what was that all about, that you had to remove files by hand?

Carol: The operating system refused, fortunately, to override the existing output file that we created with our earlier version of `forward_euler2.C`. This is a good thing, since it is all to easy to override (much) earlier results that are perhaps hard to recreate. Similarly, when I gave the explicit command `rm` to remove the file, the system checked to make sure I knew what I was doing, and I answered `y` for yes.

Bob: That seems like overkill to me. You asked for a specific file to be deleted, and it checked to see whether you had learned to read and write??

Carol: For a single file this may be overkill, but often we give a command such as `rm tmp*` if we want to remove all temporary files in a directory. If you (or your cat) were to hit the space bar by mistake in the middle, you could wind up with `rm tmp *` . . .

Bob: . . . I see, and thus deleting all files in that directory. No, that would not be good, I agree.

Carol: By the way, you can choose this safety option by typing `rm -i` with option `i` for inquiry, prompting the system to question you for each file to be deleted. Or simpler, you can add this to your shell definitions, for example to your `.cshrc` file if you use a C shell.

Bob: And what about `!$`?

Carol: Oh, that is shorthand for retyping the last word in the previous command. I could have used that two lines earlier too, by typing `rm !$`. It speeds things up, just like the command `!!` which repeats the whole previous command. Also the second time I ran the program I could have simply typed `!f` which would have expanded to `forward_euler2 > forward2_0.01.out`. Or I could have type `!-2` to repeat the command that was issued two steps before. Unix has many ways to say a lot with two or three key strokes! Here is what I could have typed:

Output 3.8

```
|gravity> g++ -o forward_euler2 forward_euler2.C
|gravity> forward_euler2 > forward2_0.01.out
forward2_0.01.out: File exists.
|gravity> rm !$
rm: remove ‘forward2_0.01.out\’\? y
|gravity> !f
Please provide a value for the time step
0.01
|gravity> tail -2 !$
tail -2 forward2_0.01.out
7.69375 -6.27772 0 0.812207 -0.5742 0
7.70187 -6.28346 0 0.812128 -0.574136 0
```

3.5 Plotting and Printing

Alice: I have a different question. It was nice to see the orbit plotted on the screen, but I wonder how we can make a hard copy output. I would like to start gathering material to write a working paper about our project.

Carol: That's easy, although 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, try `"set terminal"` and `"set output"`. Let me show you.

Note: because the output of the `help` facility of `gnuplot` is rather long, we will omit most of it here by printing "......" instead. In the example below, the words typed by Carol are `help`, `set`, `terminal`, `postscript`, then she twice hit the return key without typing anything, after which she typed `output`, followed again by hitting the return key twice to get back to the command level of gnuplot.

```
gnuplot>
gnuplot> help
 `gnuplot` is a command-driven interactive function and data plotting program.

    . . . . . .

 The new `gnuplot` user should begin by reading about `plotting` (if on-line,
 type `help plotting`).

Help topics available:
    batch/interactive bugs              commands          comments
    coordinates       copyright         environment       expressions
    glossary          graphical         introduction      line-editing
    new-features      old_bugs          plotting          seeking-assistance
    set               show              startup           substitution
    syntax            time/date

Help topic: set
 The `set` command can be used to sets _lots_ of options.  No screen is
 drawn, however, until a `plot`, `splot`, or `replot` command is given.

 The `show` command shows their settings;  `show all` shows all the
 settings.

 If a variable contains time/date data, `show` will display it according to
 the format currently defined by `set timefmt`, even if that was not in effect
 when the variable was initially defined.

Subtopics available for set:
    angles            arrow             autoscale         bar
    bmargin           border            boxwidth          clabel
  . . . . . .
    terminal          tics              ticscale          ticslevel
    time              time/date_specifiers                timefmt
  . . . . . .

Subtopic of set: terminal
 `gnuplot` supports many different graphics devices.  Use `set terminal` to
 tell `gnuplot` what kind of output to generate. Use `set output` to redirect
 that output to a file or device.
```

```
    . . . . . .

Subtopics available for set terminal:
    aed512            aed767            aifm              bitgraph
    cgm               corel             dumb              dxf
    eepic             emtex             epson-180dpi      epson-60dpi
    epson-lx800       fig               gpic              hp2623a
    hp2648            hp500c            hpdj              hpgl
    hpljii            hppj              imagen            jpeg
    kc-tek40xx        km-tek40xx        latex             mf
    mif               mp                nec-cp6           okidata
    pbm               pcl5              png               postscript
    pslatex           pstex             pstricks          qms
    regis             selanar           starc             table
    tandy-60dpi       tek40xx           tek410x           texdraw
    tgif              tkcanvas          tpic              vttek
    x11               xlib

Subtopic of set terminal: postscript
 Several options may be set in the `postscript` driver.

    . . . . . .

 `eps` mode generates EPS (Encapsulated PostScript) output, which is just
 regular PostScript with some additional lines that allow the file to be
 imported into a variety of other applications.  (The added lines are
 PostScript comment lines, so the file may still be printed by itself.)  To
 get EPS output, use the `eps` mode and make only one plot per file.  In `eps`
 mode the whole plot, including the fonts, is reduced to half of the default
 size.
Subtopics available for set terminal postscript:
    editing           enhanced

Subtopic of set terminal postscript:
Subtopic of set terminal:
Subtopic of set: output
 By default, screens are displayed to the standard output. The `set output`
 command redirects the display to the specified file or device.

 Syntax:
       set output {"<filename>"}

    . . . . . .

Subtopic of set:
Help topic:
gnuplot>
gnuplot> quit
|gravity>
```

Alice: It is nice to have so much detailed information at your fingertips, but I'm glad you knew what to ask? I would never have guessed that I would have to give an arcane command like `set terminal` before adding `postscript`, which is what I wanted; and while `set output` is somewhat more logical, I wouldn't have guessed that either.

Carol: That's what friends are for: learning to work with a new package is always easiest when looking over someone's shoulder. The first time I used `gnuplot help`, I did not know that you could work your way up to higher levels by simply hitting the return key. I tried quit, exit, and a number of other things, and finally killed the program. Only later I saw someone simply typing nothing, which was the solution! Okay, let's make a postscript file `forward2_0.01.ps` so that we can print the orbit computed by invoking `forward_euler2` for step size `0.01`.

Alice: Let's make it `terminal postscript eps`. That way I can encapsulate the resulting postscript file directly into my working paper, as the help statement just told us.

Carol: Fine. Note, by the way, that gnuplot does not require us to type words like `terminal` in full: `term` is fine, or even `ter` or `te`. You might guess that `t` would not be enough to specify, given that there are other commands starting with t, like `tics`. In our case, even `t` will work, since `terminal` happens to be the first command starting with a t, alphabetically, as you can see above. However, for the human reader a good compromise is `set term post eps` which is much easier to type than `terminal postscript eps` and still easily recognizable.

```
|gravity> gnuplot
gnuplot> plot "forward2_0.01.out"
gnuplot> set term post eps
Terminal type set to 'postscript'
Options are 'eps noenhanced monochrome dashed defaultplex "Helvetica" 14'
gnuplot> set output "forward2_0.01.ps"
gnuplot> replot
gnuplot> q
|gravity>
```

Carol: Let's print it out:

```
|gravity> lpr "forward2_0.01.ps"
|gravity>
```

Alice: Great! My first figure for my working paper. I wonder how long this paper is going to be. I guess it depends on how much patience you both have – if this is fun, I could even go on to make it a book!

Bob: Or even a book series, for that matter?

Carol: Don't be ironic, who knows what our first baby steps will lead to.

3.6 Finding (slow) Convergence

Alice: Ready for a ten times smaller time step?

Carol: We aim to please!

Output 3.9

```
|gravity> forward_euler2 > forward2_0.001.out
Please provide a value for the time step
0.001
|gravity> tail -2 !$
tail -2 forward2_0.001.out
2.01436 0.162565 0 -0.152876 0.258696 0
2.0142 0.162824 0 -0.15312 0.258677 0
```

Bob: Somewhat better, but still the separation vector has a length greater than unity, which is not physical. Let's look at a picture. I saw that you wrote this line `set size ratio -1` in `.gnuplot`, so I guess you only have to issue the plot command.

```
|gravity> gnuplot
gnuplot> plot "forward2_0.001.out"
gnuplot> quit
|gravity>
```

Alice: At least we got two revolutions this time.

Carol: Let's try a couple more steps of ten refinement of the step size. But before we do that, let us be a bit more frugal in our output. I just noticed that our output files are growing alarmingly in size:

```
|gravity> ls -l *.out
-rw-r--r--    1 carol    students    39633 Dec 24 07:07 forward1.out
-rw-r--r--    1 carol    students   404204 Dec 24 11:27 forward2_0.001.out
-rw-r--r--    1 carol    students    39673 Dec 24 10:31 forward2_0.01.out
|gravity>
```

Bob: I see! Our last file had a size of 400 kbytes, which is not much of a problem. But with two more steps of refinement we would wind up with a 40 Mbyte file. Even though our disk space is large enough, it would surely slow down both gnuplot and postscript plotting. How about restricting output to occur only once every `dt_out = 0.01`, even if our time step `dt < dt_out`?

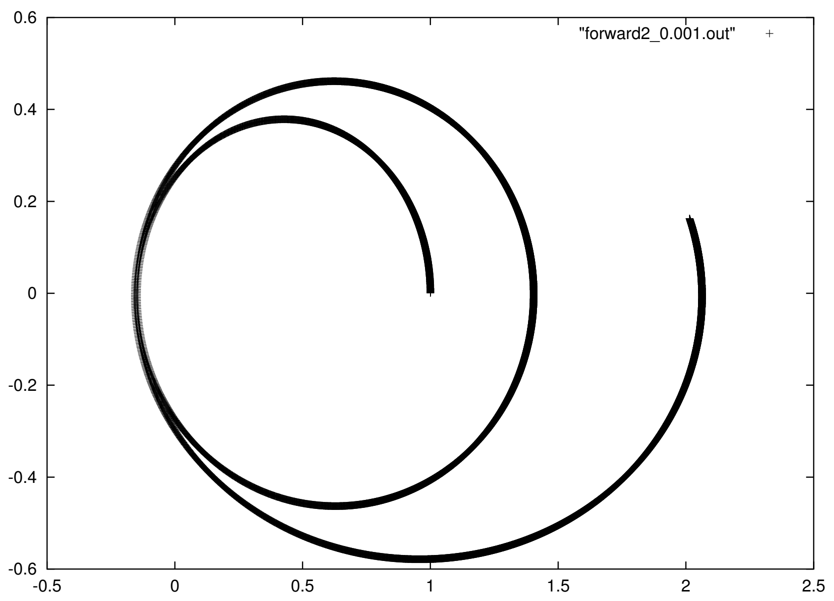

Figure 3.2: Relative orbit for the second attempt to integrate a two-body system with a forward-Euler integrator, with time step $dt = 0.001$

Carol: That's what I had in mind. Time for a new version:

Code 3.4 (forward_euler3.C)

```
//-----------------------------------------------------------------------------
// forward_euler3.C
//-----------------------------------------------------------------------------
#include  <iostream>
#include  <cmath>
using namespace std;

int main()
{
    double r[3], v[3], a[3];
    double dt;

    cerr << "Please provide a value for the time step" << endl;

    cin >> dt;
    r[0] = 1;
    r[1] = 0;
    r[2] = 0;
    v[0] = 0;
    v[1] = 0.5;
    v[2] = 0;

    double dt_out = 0.01;
    double t_out = dt_out;

    for (double t = 0; t < 10; t += dt){
        double r2 = r[0]*r[0] + r[1]*r[1] + r[2]*r[2];
        for (int k = 0; k < 3; k++)
            a[k] = - r[k] / (r2 * sqrt(r2));
        for (int k = 0; k < 3; k++){
            r[k] += v[k] * dt;
            v[k] += a[k] * dt;
        }
        if (t >= t_out){
            cout << r[0] << " " << r[1] << " " << r[2] << " ";
            cout << v[0] << " " << v[1] << " " << v[2] << endl;
            t_out += dt_out;
        }
    }
}
//-----------------------------------------------------------------------------
```

Carol: As before, let us first look at the end of the output file, to see whether the numbers at least look reasonable.

Output 3.10

```
|gravity> g++ -o forward_euler3 forward_euler3.C
|gravity> forward_euler3 > forward3_0.0001.out
Please provide a value for the time step
0.0001
|gravity> tail -2 !$
tail -2 forward3_0.0001.out
0.323206 0.388479 0 -1.51485 -0.250448 0
0.307933 0.385822 0 -1.54017 -0.28151 0
|gravity> ls -al  *.out
-rw-r--r--    1 carol    students    39633 Sep 19 13:41 forward1.out
```

```
-rw-r--r--    1 carol    students   404204 Sep 19 13:41 forward2_0.001.out
-rw-r--r--    1 carol    students    39673 Sep 19 13:41 forward2_0.01.out
-rw-r--r--    1 carol    students    40598 Sep 19 13:41 forward3_0.0001.out
```

Bob: Ah, for the first time the separation vector is consistent, at least in principle, since now it is shorter than unity.

Carol: And the output size is under control now. Time to make a plot, but I'm getting a little tired of the label written in the top right hand corner with the one lonely plot symbol. I'll add the `notitle` command.

```
|gravity> gnuplot
gnuplot> plot "forward3_0.0001.out" notitle
gnuplot> quit
|gravity>
```

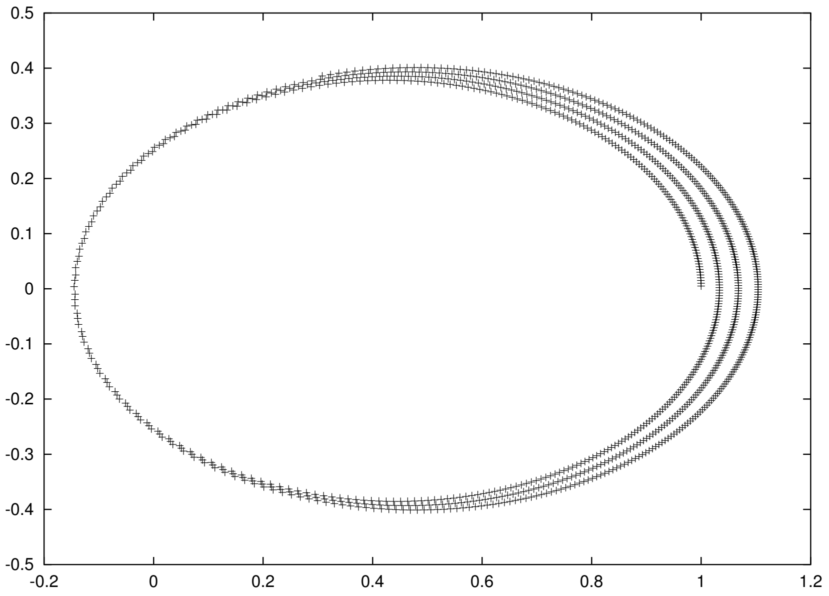

Figure 3.3: Relative orbit for the third attempt to integrate a two-body system with a forward-Euler integrator, with time step $dt = 0.0001$

Carol: Three revolutions and counting! And there does seem to be a definite convergence toward an ellipse, which is encouraging. Okay, ready for another shrinking by ten of the time step?

Output 3.11

```
|gravity> forward_euler3 > forward3_0.00001.out
Please provide a value for the time step
0.00001
|gravity> tail -2 !$
tail -2 forward3_0.00001.out
0.496172 -0.378817 0 1.21125 0.0849565 0
0.508183 -0.377891 0 1.19105 0.100179 0
```

Bob: We are finally beginning to give the computer a workout, with our request to take a million steps! But note that we are still not converging very quickly. The numbers are once more quite different from what we saw when we took 'only' a hundred thousand steps!

```
|gravity> gnuplot
gnuplot> plot "forward3_0.00001.out" notitle
gnuplot> quit
|gravity>
```

Carol: But this time the orbit looks almost natural. I bet that with another refinement of the step size by a factor ten, we would finally see convergence.

Output 3.12

```
|gravity> forward_euler3 > forward3_0.000001.out
Please provide a value for the time step
0.000001
|gravity> tail -2 !$
tail -2 forward3_0.000001.out
0.570929 -0.366472 0 1.08014 0.182609 0
0.58164 -0.364588 0 1.06201 0.19411 0
```

Bob: Disappointing, I must say. At least we got the first decimal right, more or less, but that's about it. Let's see whether the figure finally looks like an honest ellipse.

```
|gravity> gnuplot
gnuplot> plot "forward3_0.000001.out" notitle
gnuplot> quit
|gravity>
```

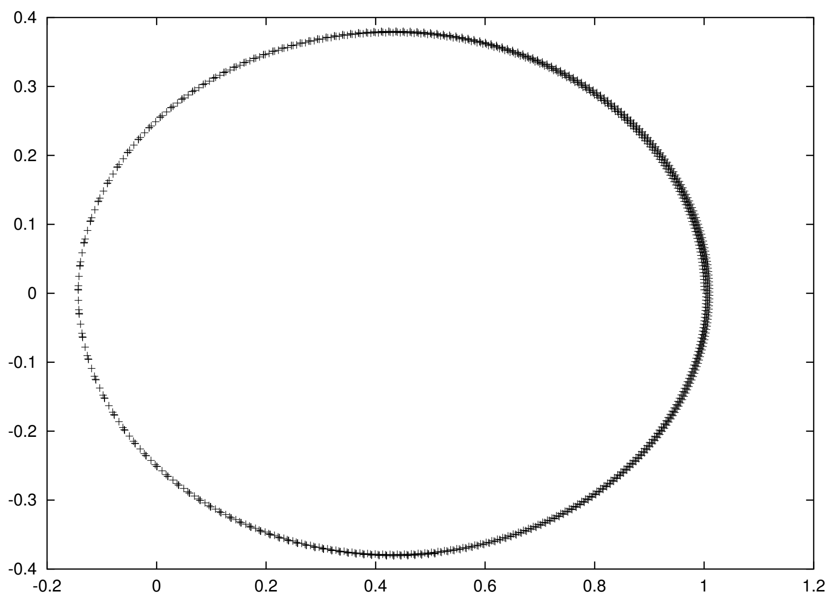

Figure 3.4: Relative orbit for the fourth attempt to integrate a two-body system with a forward-Euler integrator, with time step $dt = 0.00001$

Alice: That looks more like it. Interesting, by the way, to see the speed up of the relative motion of the two particles near pericenter, in the last three figures: you can see that from the wider spacing of the particles, something that was lost when we went to a higher density of output points.

Carol: Let's do one more step of ten refinement in step size. Modern computers should not have much trouble taking a hundred million time steps, after all. And since the plot will not change much any more, let us directly look at the last lines of the output:

Output 3.13

```
|gravity> forward_euler3 > forward3_0.0000001.out
Please provide a value for the time step
0.0000001
|gravity> tail -2 !$
tail -2 forward3_0.0000001.out
0.577881 -0.364877 0 1.06776 0.191061 0
0.588468 -0.36291 0 1.0498 0.202265 0
```

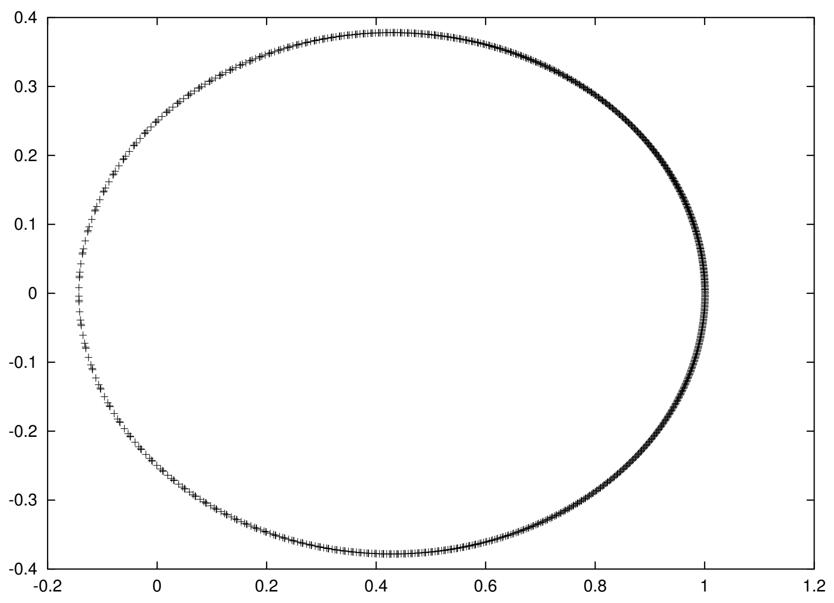

Figure 3.5: Relative orbit for the fifth attempt to integrate a two-body system with a forward-Euler integrator, with time step $dt = 0.000001$

Carol: There you go, Bob, convergence to two significant digits at least! Even so, I guess we are all convinced that first-order methods are not the way to go. Time to go to second order!

Alice: So let's write a leapfrog code. But before doing so, there is still one addition I would like to make to our poorly performing forward Euler code. You see, for the two-body problem we can get a pretty good idea of what's going on by looking at the orbit, because we know what to expect. In fact, we can solve the orbit analytically, and therefore we can compute everything to arbitrary accuracy that way, much faster and cheaper than doing it the hard way, through numerical orbit integration. The point is that for an N-body system with $N > 2$ the orbits don't look recognizable at all, and we would be hard put to judge the performance of a code from staring at orbital figures.

Bob: Ah, you mean that we should try to define, what did they call it in physics class, conserved quantities?

Alice: Exactly. And the simplest such quantity is the total energy of the N-body system. We know that it is rigorously conserved, so any deviation between beginning and end of a numerical integration session must be purely the result of numerical errors.

Carol: Okay, let's code that up too, and then call it a day. What are the equations?

3.7 Checking Energy Conservation

Alice: Here are the expressions for the kinetic energy $E_{\rm kin}$ and the potential energy $E_{\rm pot}$. The total energy is just the sum of both terms.

$$E_{\rm kin} = \frac{1}{2}\frac{M_1M_2}{M_1+M_2}v^2 \qquad (3.4)$$

$$E_{\rm pot} = -\frac{M_1M_2}{r} \qquad (3.5)$$

Alice: So far, we have used units in which $M_1 + M_2 = 1$. We have not specified the individual masses, since we did not have to; the equations of motion are invariant with respect to how we divide our unit of mass over the two bodies. But the interpretation in terms of the physical energy of the system does depend on the value of M_2, which is conventionally chosen as the less massive body. Once we choose M_2, the mass of the other body is given as $M_1 = 1 - M_2$.

Carol: I would not have guessed that the earlier invariance would be broken. Can you make that plausible by hand waving, without deriving equations?

Bob: Let me try. If we start with the Earth-Moon system, we know that the mass of the Moon is far smaller than the mass of the Earth. Their mass ratio is roughly $1 : 81$, so the Moon has not much more than one percent of the Earth's mass. In our notation, $M_2 = 1/82 = 0.012\ldots$. In the center-of-mass system, the velocities of the two bodies are inversely proportional to their masses. Kinetic energy, however, is proportional to the square of the velocities, and therefore the motion of the Moon carries about 81 times more kinetic energy than the motion of the Earth, around their common center of gravity — which lies inside the Earth, by the way, so the Earth barely moves.

Carol: Let me guess your next step. You want to compare the Earth-Moon system with another system with an even larger mass ratio? Aha, I see. Why not go all the way, and replace the Moon with a pebble, orbiting the Earth at the same distance, but without the Moon being there. The total mass is almost the same (we could even give the Earth an extra percent of mass to preserve $M_1 + M_2$), but clearly the kinetic energy of the pebble is far far smaller than the kinetic energy of the Moon. And using your argument, the pebble will still carry almost all the kinetic energy of the Earth-pebble system, as did the Moon before. The velocities of pebble and Moon are almost the same, which means that for the whole system $E_{kin} \propto M_2$ in the limit where $M_2 \ll M_1$. Indeed, that is what Eq. 3.4 tells us. Now I feel comfortable with that result.

Alice: If we continue like this, you'll both be turned into astronomers! Yes, it is always a good idea to look at a new formula, and to think of some limiting cases, to check whether the equation makes sense. Of course, making sense does not really prove that the equation is correct. We still have to check the derivation, which is given in many text books on classical dynamics. However, we are at least guarded against most typos, and more importantly, it gives us more of an idea of the physics behind the mathematics.

Bob: It is interesting that Eqs. 3.4, 3.5 have the exact same mass dependence, if we switch back again to our notation in which $M_1 + M_2 = 1$.

Alice: Yes, and we can bring that out more clearly by introducing the notion of what is called the 'reduced mass' μ of a two-body system:

$$\mu = \frac{M_1 M_2}{M_1 + M_2} \tag{3.6}$$

Carol: I see, if we define $M = M_1 + M_2$ we then get:

$$\begin{aligned} E_{\rm kin} &= \frac{1}{2}\mu v^2 && (3.7) \\ E_{\rm pot} &= -\frac{\mu M}{r} && (3.8) \end{aligned}$$

Carol: Nice and elegant!

Bob: Let me try to make it more elegant. I'm beginning to remember more and more now from my physics class. There was this notion of specific something as something per unit mass. How about defining those two energy components as specific energies, per unit reduced mass? Let us use a script $\mathcal{E}$, rather than roman E, for that notion. In our notation, with unit total mass, we will have the following specific energies:

$$\mathcal{E}_{\rm kin} = \frac{1}{2}v^2 \qquad ; \qquad \mathcal{E}_{\rm pot} = -\frac{1}{r} \tag{3.9}$$

Carol: So elegant and skinny that they almost disappear! But I suggest that we keep calling those expressions simply kinetic and potential energy, since we all know what we're talking about, rather than the mouthful 'specific kinetic and potential energy with respect to the reduced mass'.

Alice: Agreed! Time to code it up.

Carol: Does this look reasonable?

Code 3.5 (forward_euler4.C)

```
//-----------------------------------------------------------------------------
// forward_euler4.C
//-----------------------------------------------------------------------------
#include  <iostream>
#include  <cmath>
using namespace std;

int main()
{
    double r[3], v[3], a[3];
    double dt;

    cerr << "Please provide a value for the time step" << endl;

    cin >> dt;

    r[0] = 1;
    r[1] = 0;
    r[2] = 0;
    v[0] = 0;
    v[1] = 0.5;
    v[2] = 0;

    double ekin = 0.5 * (v[0]*v[0] + v[1]*v[1] + v[2]*v[2]);
    double epot = -1.0/sqrt(r[0]*r[0] + r[1]*r[1] + r[2]*r[2]);
    double e_in = ekin + epot;

    cerr << "Initial total energy E_in = " << e_in << endl;

    double dt_out = 0.01;
    double t_out = dt_out;

    for (double t = 0; t < 10; t += dt){
        double r2 = r[0]*r[0] + r[1]*r[1] + r[2]*r[2];
        for (int k = 0; k < 3; k++)
            a[k] = - r[k] / (r2 * sqrt(r2));
        for (int k = 0; k < 3; k++){
            r[k] += v[k] * dt;
            v[k] += a[k] * dt;
        }
        if (t >= t_out){
            cout << r[0] << " " << r[1] << " " << r[2] << " ";
            cout << v[0] << " " << v[1] << " " << v[2] << endl;
            t_out += dt_out;
        }
    }

    ekin = 0.5 * (v[0]*v[0] + v[1]*v[1] + v[2]*v[2]);
    epot = -1.0/sqrt(r[0]*r[0] + r[1]*r[1] + r[2]*r[2]);
    double e_out = ekin + epot;

    cerr << "Final total energy E_out = " << e_out << endl;
    cerr << "absolute energy error: E_out - E_in = " << e_out - e_in << endl;
    cerr << "relative energy error: (E_out - E_in) / E_in = "
         << (e_out - e_in) / e_in << endl;
}
//-----------------------------------------------------------------------------
```

Bob: Looks good to me. Let's start with our good old `dt = 0.01`

Output 3.14

```
|gravity> g++ -o forward_euler4 forward_euler4.C
|gravity> forward_euler4 > forward4_0.01.out
Please provide a value for the time step
0.01
Initial total energy E_in = -0.875
Final total energy E_out = 0.393987
absolute energy error: E_out - E_in = 1.26899
relative energy error: (E_out - E_in) / E_in = -1.45027
```

Bob: Well, we knew that it was bad, and the energy check confirms it. The energy even changes sign — which of course we knew, since we saw the particle escaping, and only positive-energy orbits can show escape.

Carol: Rather than producing more output files, let's just confirm we have the same output in this case, after which I suggest we call our friend `/dev/null` to the rescue.

Output 3.15

```
|gravity> tail -2 forward4_0.01.out
7.69375 -6.27772 0 0.812207 -0.5742 0
7.70187 -6.28346 0 0.812128 -0.574136 0
```

Bob: Good! The same as before, with still the one step overshoot. Let's look at the other cases.

Output 3.16

```
|gravity> forward_euler4 > /dev/null
Please provide a value for the time step
0.001
Initial total energy E_in = -0.875
Final total energy E_out = -0.449681
absolute energy error: E_out - E_in = 0.425319
relative energy error: (E_out - E_in) / E_in = -0.486079
|gravity> !!
Please provide a value for the time step
0.0001
Initial total energy E_in = -0.875
Final total energy E_out = -0.80008
absolute energy error: E_out - E_in = 0.0749198
relative energy error: (E_out - E_in) / E_in = -0.0856226
|gravity> !!
Please provide a value for the time step
0.00001
Initial total energy E_in = -0.875
Final total energy E_out = -0.864747
absolute energy error: E_out - E_in = 0.0102526
relative energy error: (E_out - E_in) / E_in = -0.0117172
|gravity> !!
Please provide a value for the time step
0.000001
Initial total energy E_in = -0.875
Final total energy E_out = -0.873971
absolute energy error: E_out - E_in = 0.00102934
relative energy error: (E_out - E_in) / E_in = -0.00117639
```

Bob: How nice to see the errors shrink as snow for the sun!

Carol: On a rather cold day, with a very low sun. Yes, it converges, but very slooooowly.

Alice: I'm glad we went all the way to ten million steps, in the last calculation, even though it took more than a minute to complete. In the last two runs we have just confirmed the first-order character of the forward Euler integration scheme. Making the time step ten times smaller makes the error ten times smaller, to within the first three significant digits! This is something that was not obvious at all during the earlier runs, where the errors were so large that nonlinear effects overwhelmed the asymptotic proportionality between error and time step, for the limit $dt \to 0$.

Carol: Great! And a great point to stop. This has been a far longer session than I had anticipated. Let's get back tomorrow, and then someone else should take over the controls.

Bob: I'd be happy to do so. This has been fun, and I look forward to writing a second-order code.

Alice: Happy dreams!

Chapter 4

Exploring $N = 2$ with a Leapfrog Algorithm

During their next session, Alice, Bob, and Carol wrote a second-order integrator, with Bob behind the key board this time. They chose the leapfrog algorithm, one of the most popular integrators for cases where only modest accuracy is needed. In some cases the problem itself does not require high accuracy. In other cases the leapfrog is used to quickly build a prototype, a toy model that can give insight in the overall structure of the problem. At a later stage, once the overall strategy has become more clear, a higher-order integrator can replace the leapfrog. Dense stellar systems are an example of the latter category, where fourth-order algorithms are almost always the engines of choice in N-body codes.

We will not follow their dialogue in detail here, but rather describe a summary of their results.

4.1 Two Ways to Write the Leapfrog

The name *leapfrog* comes from one of the ways to write this algorithm, where positions and velocities 'leap over' each other. Positions are defined at times $t_i, t_{i+1}, t_{i+2}, \ldots$, spaced at constant intervals dt, 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 $t_{i+1} - t_{i+1/2} = t_{i+1/2} - t_i = dt/2$. The leapfrog integration scheme then reads:

$$\begin{aligned}
\mathbf{r}_i &= \mathbf{r}_{i-1} + \mathbf{v}_{i-1/2}dt &(4.1)\\
\mathbf{v}_{i+1/2} &= \mathbf{v}_{i-1/2} + \mathbf{a}_i dt &(4.2)
\end{aligned}$$

Note that the accelerations $\mathbf{a}$ are defined only on integer times, just like the positions, while the velocities are defined only on half-integer times. This makes sense, given that $\mathbf{a}(\mathbf{r},\mathbf{v}) = \mathbf{a}(\mathbf{r})$: the acceleration on one particle depends only on its position with respect to all other particles, and not on its or their velocities. Only at the beginning of the integration do we have to set up the velocity at its first half-integer time step. Starting with initial conditions $\mathbf{r}_0$ and $\mathbf{v}_0$, we take the first term in the Taylor series expansion to compute the first leap value for $\mathbf{v}$:

$$\mathbf{v}_{1/2} = \mathbf{v}_0 + \mathbf{a}_0 dt/2. \tag{4.3}$$

We are then ready to apply Eq. 4.1 to compute the new position $\mathbf{r}_1$, using the first leap value for $\mathbf{v}_{1/2}$. Next we compute the acceleration $\mathbf{a}_1$, which enables us to compute the second leap value, $\mathbf{v}_{3/2}$, using Eq. 4.2, and so on.

A second way to write the leapfrog looks quite different at first sight. Defining all quantities only at integer times, we can write:

$$\begin{aligned} \mathbf{r}_{i+1} &= \mathbf{r}_i + \mathbf{v}_i dt + \mathbf{a}_i (dt)^2/2 & (4.4)\\ \mathbf{v}_{i+1} &= \mathbf{v}_i + (\mathbf{a}_i + \mathbf{a}_{i+1})dt/2 & (4.5) \end{aligned}$$

This is still the same leapfrog scheme, although represented in a different way. 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.

A most interesting way to see the equivalence of Eqs. 4.1, 4.2 and Eqs. 4.4, 4.5 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 inspect.

Starting with Eqs. 4.1, 4.2, even though it may be obvious, let us write out the time reversibility. We will take one step forward, taking a time step $+dt$, to evolve $\{\mathbf{r}_i, \mathbf{v}_{i-1/2}\}$ to $\{\mathbf{r}_{i+1}, \mathbf{v}_{i+1/2}\}$, and then we will take one step backwards, using the same scheme, taking a time step $-dt$. Clearly, the time will return to the same value since $+dt - dt = 0$, but we 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, resulting from applying Eqs. 4.1, 4.2 twice.

$$\begin{aligned} \mathbf{r}_f &= \mathbf{r}_{i+1} - \mathbf{v}_{i+1/2} dt \\ &= \left[\mathbf{r}_i + \mathbf{v}_{i+1/2} dt\right] - \mathbf{v}_{i+1/2} dt \\ &= \mathbf{r}_i & (4.6) \end{aligned}$$

$$\begin{aligned}
\mathbf{v}_f &= \mathbf{v}_{i+1/2} - \mathbf{r}_i dt \\
&= \left[\mathbf{v}_{i-1/2} + \mathbf{a}_i dt\right] - \mathbf{a}_i dt \\
&= \mathbf{v}_{i-1/2}
\end{aligned} \tag{4.7}$$

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. In a computer application, this means that we can evolve forward a thousand time steps and then evolve backward for the same length of time. Although we will make integration errors (remember, leapfrog is only second-order, and thus not very precise), those errors will exactly cancel each other, apart from possible round-off effects, due to limited machine accuracy.

Now the real fun comes in, when we inspect the equal-time Eqs. 4.4, 4.5:

$$\begin{aligned}
\mathbf{r}_f &= \mathbf{r}_{i+1} - \mathbf{v}_{i+1} dt + \mathbf{a}_{i+1}(dt)^2/2 \\
&= \left[\mathbf{r}_i + \mathbf{v}_i dt + \mathbf{a}_i (dt)^2/2\right] - \left[\mathbf{v}_i + (\mathbf{a}_i + \mathbf{a}_{i+1})dt/2\right] dt + \mathbf{a}_{i+1}(dt)^2/2 \\
&= \mathbf{r}_i
\end{aligned} \tag{4.8}$$

$$\begin{aligned}
\mathbf{v}_f &= \mathbf{v}_{i+1} - (\mathbf{a}_{i+1} + \mathbf{a}_i)dt/2 \\
&= [\mathbf{v}_i + (\mathbf{a}_i + \mathbf{a}_{i+1})dt/2] - (\mathbf{a}_{i+1} + \mathbf{a}_i)dt/2 \\
&= \mathbf{v}_i
\end{aligned} \tag{4.9}$$

In this case, too, we have exact time reversibility. Even though not immediately obvious from an inspection of Eqs. 4.4, 4.5, as soon as we write out the effects of stepping forward and backward, the cancellations become manifest.

4.2 A Simple Leapfrog for the 2-Body Problem

Our friends decided to write their leapfrog scheme using the second way of implementing the algorithm, centering all physical quantities on integer time step times.

The first leapfrog code that Bob wrote, `leapfrog1.C`, is similar to the last forward Euler code that they produced, at the end of the previous chapter, `forward_euler3.C`. There are really only two differences. One is a computation of the acceleration before we get into the integration loop, and the other is the difference in the integration steps themselves. Here is the code:

Code 4.1 (leapfrog1.C)

```
//-----------------------------------------------------------------------------
// leapfrog1.C
//-----------------------------------------------------------------------------
#include  <iostream>
#include  <cmath>
using namespace std;

int main()
{
    double r[3], v[3], a[3];
    double dt;

    cerr << "Please provide a value for the time step" << endl;

    cin >> dt;

    r[0] = 1;
    r[1] = 0;
    r[2] = 0;
    v[0] = 0;
    v[1] = 0.5;
    v[2] = 0;

    double ekin = 0.5 * (v[0]*v[0] + v[1]*v[1] + v[2]*v[2]);
    double epot = -1.0/sqrt(r[0]*r[0] + r[1]*r[1] + r[2]*r[2]);
    double e_in = ekin + epot;

    cerr << "Initial total energy E_in = " << e_in << endl;

    double r2 = r[0]*r[0] + r[1]*r[1] + r[2]*r[2];
    for (int k = 0; k < 3; k++)
        a[k] = - r[k] / (r2 * sqrt(r2));

    double dt_out = 0.01;
    double t_out = dt_out;

    for (double t = 0; t < 10; t += dt){
        for (int k = 0; k < 3; k++){
            v[k] += 0.5 * a[k] * dt;
            r[k] += v[k] * dt;
        }
        r2 = r[0]*r[0] + r[1]*r[1] + r[2]*r[2];
        for (int k = 0; k < 3; k++){
            a[k] = - r[k] / (r2 * sqrt(r2));
            v[k] += 0.5 * a[k] * dt;
        }
        if (t >= t_out){
            cout << r[0] << " " << r[1] << " " << r[2] << " ";
            cout << v[0] << " " << v[1] << " " << v[2] << endl;
            t_out += dt_out;
        }
    }

    ekin = 0.5 * (v[0]*v[0] + v[1]*v[1] + v[2]*v[2]);
    epot = -1.0/sqrt(r[0]*r[0] + r[1]*r[1] + r[2]*r[2]);
    double e_out = ekin + epot;

    cerr << "Final total energy E_out = " << e_out << endl;
    cerr << "absolute energy error: E_out - E_in = " << e_out - e_in << endl;
    cerr << "relative energy error: (E_out - E_in) / E_in = "
         << (e_out - e_in) / e_in << endl;
}
//-----------------------------------------------------------------------------
```

Let us start with the integration loop `for (double t = 0; ...`. A direction implementation of Eqs. 4.4, 4.5 would have give for the third and fourth line of the loop:

```
            r[k] += v[k] * dt + 0.5 * a[k] * dt * dt;
            v[k] += 0.5 * a[k] * dt;
```

thus updating the position completely, and leaving the second half of the velocity increment until after the new acceleration a_{i+1} is calculated, based upon the new position r_{i+1}. However, they realized that switching the order of updating position and velocity would simplify the above two lines to:

```
            v[k] += 0.5 * a[k] * dt;
            r[k] += v[k] * dt;
```

since the last term in the position update has now been taken care off already by the velocity update. In professional codes you will often find such short cuts, sometimes explained by a comment, more often not (we will address the question of code comments in the next part of this book).

The other change with respect to `forward_euler3.C` stems from the fact that we need to know the acceleration at the beginning of the loop, in order to let the velocity begin to step forward, *and* we need to calculate the acceleration at the end of the loop, in order to complete the velocity step. The simplest solution would be to calculate the acceleration twice, at the beginning and at the end of the loop. However, this would be a terrible waste of computer time. Calculating accelerations is the most expensive part of an N-body code, in fact it is the one place where all the computational muscle is needed, since it scales with the square of the number of particles. Therefore, it is important to see whether we can avoid such excess.

The solution is simple: rather than calculating the acceleration at the beginning of the loop, as we did in `forward_euler3.C`, we calculate it only at the end of the loop. In that way, the next round through the loop will conveniently start with the old value of the acceleration, while the new value will become available at the end of that round.

The only problem, and a potential source for run time bugs, is that the loop is no longer complete in and by itself. Unlike the case for the `forward_euler3.C` code, we cannot simply enter the loop at the beginning, since the accelerations `a[k]` would be undefined, and have arbitrary values. This is the reason for the three lines preceding the integration loop, where we calculate the initial acceleration between the two particles:

```
    double r2 = r[0]*r[0] + r[1]*r[1] + r[2]*r[2];
    for (int k = 0; k < 3; k++)
        a[k] = - r[k] / (r2 * sqrt(r2));
```

Just for curiosity, and also to show what type of nonsense can be produced so easily by forgetting to 'spin the wheels' once before getting into an integration loop, let us take out these three lines, and call the bug version `leapfrog1a.C`. Notice yet another common source of bugs. Deleting or commenting out the three lines above will give an error message from the compiler, with a complaint along the lines of `leapfrog1a.C:34: 'r2' undeclared`. Indeed, after deleting the three lines, we have to put the declaration of the variable `r2` back into the loop, something that was not necessary in `leapfrog1.C`, where `r2` had already been declared.

Here are the first few output steps, on one particular computer (with a bug present, all bets are off with respect to reproducibility on another computer; and even on the same computer we can get different results at different times, if our program addresses memory that has random values in it). Rather than redirecting the output to an output file, we will use the `head` command, the counterpart of the `tail` command. Typing `head -5` will give us only the first five lines of output.

```
|gravity> g++ -o leapfrog1a leapfrog1a.C
|gravity> leapfrog1a | head -5
Please provide a value for the time step
0.01
Initial total energy E_in = -0.875
1.00012 0.0102179 0.00021032 -0.00412328 0.510817 0.0105144
1.00003 0.0153255 0.000315454 -0.0141193 0.510689 0.0105118
0.999835 0.0204317 0.000420556 -0.0241158 0.510511 0.0105081
0.999544 0.0255357 0.000525616 -0.034114 0.510281 0.0105034
0.999153 0.0306373 0.000630624 -0.0441151 0.51 0.0104976
|gravity>
```

The problem is that the output does not look so unreasonable. If we would have gotten values like 10^{10} or larger somewhere, we would immediately know that we had made a blunder somewhere. But one dead giveaway is the fact that the values of position and velocity along the z axis (the third component in three dimensions) are not zero, as they should be. Since we started moving in the $\{x, y\}$ plane, with no force components acting perpendicular to the plane, it is clear that $r_z = v_z = 0$ should hold for all times.

The moral of the story is that it is always a good idea to inspect a few lines of output visually – simply by plotting the $\{x, y\}$ values of the position we might not have noticed that there was anything wrong! What must have happened is that a_z, represented by the third component `a[2]` in the acceleration array, was non-zero at the start of the program. And since we did not calculate it before using, that value first caused the velocity the deviate out of the plane, and subsequently also the position.

Here are the correct values, different also for the x and y components of position and velocity, but not dramatically so:

```
|gravity> g++ -o leapfrog1 leapfrog1.C
|gravity> leapfrog1 | head -5
Please provide a value for the time step
0.01
Initial total energy E_in = -0.875
0.9998 0.0099995 0 -0.0200019 0.4999 0
0.99955 0.014998 0 -0.0300059 0.499775 0
0.9992 0.019995 0 -0.0400138 0.4996 0
0.99875 0.02499 0 -0.0500266 0.499374 0
0.998199 0.0299825 0 -0.0600457 0.499098 0
|gravity>
```

4.3 Finding Better Convergence

In the previous chapters we saw how slow the first-order forward Euler scheme converged. The leapfrog scheme, being second order, is expected to fare better. Indeed, here are the results, in the form of energy output, output of the last couple positions and velocities, and Figs. 4.1, 4.2.

```
|gravity> leapfrog1 > leapfrog1_0.01.out
Please provide a value for the time step
0.01
Initial total energy E_in = -0.875
Final total energy E_out = -0.87497
absolute energy error: E_out - E_in = 2.98294e-05
relative energy error: (E_out - E_in) / E_in = -3.40907e-05
|gravity> tail -2 !$
tail -2 leapfrog_0.01.out
0.583527 -0.387366 0 1.03799 0.167802 0
```

```
0.593822 -0.385632 0 1.02114 0.178871 0
|gravity> leapfrog1 > leapfrog1_0.001.out
Please provide a value for the time step
0.001
Initial total energy E_in = -0.875
Final total energy E_out = -0.875
absolute energy error: E_out - E_in = 3.17517e-07
relative energy error: (E_out - E_in) / E_in = -3.62876e-07
|gravity> tail -2 !$
tail -2 leapfrog1_0.001.out
0.590112 -0.362786 0 1.04675 0.203781 0
0.600491 -0.360694 0 1.02914 0.214483 0
|gravity>
```

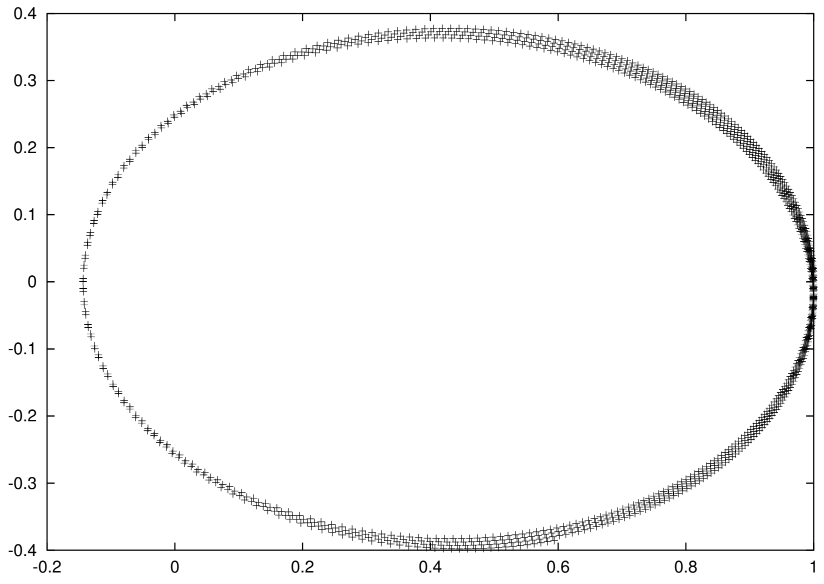

Figure 4.1: Relative orbit for the first attempt to integrate a two-body system with a leapfrog integrator, with time step $dt = 0.01$

We can see how the orbit still changes after each revolution, due to numerical errors, in Fig. 4.1. However, with a time step of `0.001`, Fig. 4.2 shows no discernible errors any more. Unlike the case with the forward Euler integrator, the last couple output values for position and velocity differ by only a few percent at most. And finally, it is clear that the leapfrog is indeed second-order: making the time step ten times smaller reduces the errors by a factor a hundred. The smaller the time step, the more accurate this factor of one hundred becomes:

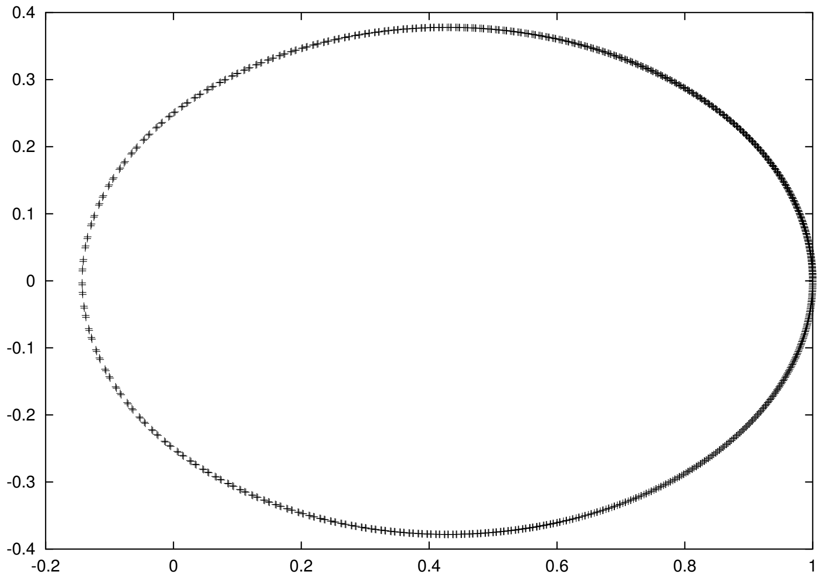

Figure 4.2: Relative orbit for the second attempt to integrate a two-body system with a leapfrog integrator, with time step $dt = 0.001$

```
|gravity> leapfrog1 > /dev/null
Please provide a value for the time step
0.0001
Initial total energy E_in = -0.875
Final total energy E_out = -0.875
absolute energy error: E_out - E_in = 3.19364e-09
relative energy error: (E_out - E_in) / E_in = -3.64987e-09
|gravity> !!
Please provide a value for the time step
0.00001
Initial total energy E_in = -0.875
Final total energy E_out = -0.875
absolute energy error: E_out - E_in = 3.19509e-11
relative energy error: (E_out - E_in) / E_in = -3.65153e-11
|gravity> !!
Please provide a value for the time step
0.000001
Initial total energy E_in = -0.875
Final total energy E_out = -0.875
absolute energy error: E_out - E_in = 8.60312e-13
relative energy error: (E_out - E_in) / E_in = -9.83214e-13
```

```
|gravity> !!
Please provide a value for the time step
0.0000001
Initial total energy E_in = -0.875
Final total energy E_out = -0.875
absolute energy error: E_out - E_in = 1.11888e-12
relative energy error: (E_out - E_in) / E_in = -1.27872e-12
```

Notice how the scaling first takes this factor 100 to much better than 1% accuracy, but then flattens off when we begin to approach machine accuracy for double precision (of order 10^{-15}). In the last case, the cumulative round-off effects are particularly severe. Not surprising, given the fact that a time step of `dt = 0.000,000,1` implied a whooping 100,000,000 integration steps, which took a quarter of an hour to complete. Whereas the scaling from the first two outputs would have predicted a final error around `3.20e-15` in absolute energy and around `3.65e-15` in relative energy, the actual result shows a turn-up of the error size, paradoxically making the calculation with one hundred million steps less accurate than the calculation with ten million steps.

The lesson is to always beware of stretching an algorithm beyond its natural area of application. In practice, we will use the leapfrog when modest accuracy is needed, with errors remaining at a level of, say, 10^{-4} to 10^{-6} or so. If much higher accuracy is needed, it is better to use a fourth-order scheme, as we will see later on in this book.

Chapter 5

Exploring $N = 3$ with a Leapfrog Algorithm

With a better algorithm in hand, our friends decided to work their way up from the two-body problem to the three-body problem. And rather than hard coding the value of N in their algorithm, they wrote a leapfrog code for the general N-body problem. The expression for the acceleration felt by particle i is given by summing together the Newtonian gravitational attraction of all other particles j, where both i and j take on values from 1 up to and including N:

$$\frac{d^2}{dt^2}\mathbf{r}_i = G \sum_{\substack{j=1 \\ j\neq i}}^{N} M_j \frac{\mathbf{r}_j - \mathbf{r}_i}{|\mathbf{r}_j - \mathbf{r}_i|^3} \tag{5.1}$$

Here M_j and $\mathbf{r}_j$ are the mass and position vector of particle j, and G is the gravitational constant. To bring out the inverse square nature of gravity, we can define $\mathbf{r}_{ji} = \mathbf{r}_j - \mathbf{r}_i$, with $r_{ji} = |\mathbf{r}_{ji}|$, and unit vector $\hat{\mathbf{r}}_{ji} = \mathbf{r}_{ji}/r_{ji}$:

$$\mathbf{a}_i = G \sum_{\substack{j=1 \\ j\neq i}}^{N} \frac{M_j}{r_{ji}^2}\,\hat{\mathbf{r}}_{ji} \tag{5.2}$$

Note that the summation excludes self-interactions: every particle feels the forces of the other $N-1$ particles, but not its own force (which would be infinitely large in case of a point mass).

5.1 A More General Leapfrog

For simplicity, Alice *et al.* decided to continue to embed the initial conditions at the beginning of their code, leaving until later the task of writing a more general input routine. It contains initial conditions corresponding to three particles moving equidistantly on a circle. Here is the code they produced. We will look at the different parts in turn in the following sections.

Code 5.1 (leapfrog2.C)

```
//-----------------------------------------------------------------------------
// leapfrog2.C
//-----------------------------------------------------------------------------
#include  <iostream>
#include  <cmath>
using namespace std;

int main()
{
    int n = 3;
    double r[n][3], v[n][3], a[n][3];
    const double m = 1;
    double dt, t_end;

    cerr << "Please provide a value for the time step" << endl;
    cin >> dt;
    cerr << "and for the duration of the run" << endl;
    cin >> t_end;

    const double pi = 2 * asin(1);
    for (int i = 0; i < n; i++){
        double phi = i * 2 * pi / 3;
        r[i][0] = cos (phi);
        r[i][1] = sin (phi);
        r[i][2] = 0;
    }

    for (int i = 0; i < n; i++)
        for (int k = 0; k < 3; k++)
            a[i][k] = 0.0;
    for (int i = 0; i < n; i++){
        for (int j = i+1; j < n; j++){
            double rji[3];
            for (int k = 0; k < 3; k++)
                rji[k] = r[j][k] - r[i][k];
            double r2 = 0;
            for (int k = 0; k < 3; k++)
                r2 += rji[k] * rji[k];
            double r3 = r2 * sqrt(r2);
            for (int k = 0; k < 3; k++){
                a[i][k] += m * rji[k] / r3;
                a[j][k] -= m * rji[k] / r3;
            }
        }
    }

    double v_abs = sqrt(-a[0][0]);
    for (int i = 0; i < n; i++){
        double phi = i * 2 * pi / 3;
        v[i][0] = - v_abs * sin (phi);
        v[i][1] = v_abs * cos (phi);
        v[i][2] = 0;
    }

    double ekin = 0, epot = 0;
    for (int i = 0; i < n; i++){
        for (int j = i+1; j < n; j++){
            double rji[3];
            for (int k = 0; k < 3; k++)
                rji[k] = r[j][k] - r[i][k];
            double r2 = 0;
            for (int k = 0; k < 3; k++)
                r2 += rji[k] * rji[k];
            double r = sqrt(r2);
            epot -= m*m/r;
        }
        for (int k = 0; k < 3; k++)
            ekin += 0.5 * m * v[i][k] * v[i][k];
    }
    double e_in = ekin + epot;
    cerr << "Initial total energy E_in = " << e_in << endl;

    double dt_out = 0.01;
    double t_out = dt_out;

    for (double t = 0; t < t_end; t += dt){
        for (int i = 0; i < n; i++){
            for (int k = 0; k < 3; k++)
                v[i][k] += a[i][k] * dt/2;
            for (int k = 0; k < 3; k++)
                r[i][k] += v[i][k] * dt;
        }
        for (int i = 0; i < n; i++)
            for (int k = 0; k < 3; k++)
                a[i][k] = 0.0;
        for (int i = 0; i < n; i++){
            for (int j = i+1; j < n; j++){
                double rji[3];
                for (int k = 0; k < 3; k++)
                    rji[k] = r[j][k] - r[i][k];
                double r2 = 0;
                for (int k = 0; k < 3; k++)
                    r2 += rji[k] * rji[k];
                double r3 = r2 * sqrt(r2);
                for (int k = 0; k < 3; k++){
                    a[i][k] += m * rji[k] / r3;
                    a[j][k] -= m * rji[k] / r3;
                }
            }
        }
        for (int i = 0; i < n; i++){
            for (int k = 0; k < 3; k++)
                v[i][k] += a[i][k] * dt/2;
        }
        if (t >= t_out){
            for (int i = 0; i < n; i++){
                for (int k = 0; k < 3; k++)
                    cout << r[i][k] << " ";
                for (int k = 0; k < 3; k++)
                    cout << v[i][k] << " ";
                cout << endl;
            }
            t_out += dt_out;
        }
    }

    epot = ekin = 0;
    for (int i = 0; i < n; i++){
        for (int j = i+1; j < n; j++){
            double rji[3];
            for (int k = 0; k < 3; k++)
                rji[k] = r[j][k] - r[i][k];
            double r2 = 0;
            for (int k = 0; k < 3; k++)
                r2 += rji[k] * rji[k];
            epot -= m*m/sqrt(r2);
        }
        for (int k = 0; k < 3; k++)
            ekin += 0.5 * m * v[i][k] * v[i][k];
    }
    double e_out = ekin + epot;

    cerr << "Final total energy E_out = " << e_out << endl;
    cerr << "absolute energy error: E_out - E_in = " << e_out - e_in << endl;
    cerr << "relative energy error: (E_out - E_in) / E_in = "
         << (e_out - e_in) / e_in << endl;
}
//-----------------------------------------------------------------------------
```

Not surprisingly, the N-body code `leapfrog2.C` is more than twice as long as the code `leapfrog1.C` which was hardwired for $N = 2$. The general lay-out is similar, so we will quickly walk through it, pointing out the salient differences.

One new feature is the expression `const` in front of the declaration of the two variables `m` and `pi`. In C++ this means that you cannot change the value, once you have assigned it. If you would try to change the value of a `const` variable, the compiler would issue an error message. But apart from making code maintenance safer by preventing someone from changing a value that should not be changed by mistake, it also makes the code more easy to read, since it tells the human reader that there is a reason to keep a particular value constant.

In our case, we have chosen to give all particles the same mass, and we do not want particles to lose or gain mass, and therefore we declare and initialize the variable storing the shared mass value as `const double m = 1`. The variable `pi` stands for the mathematical constant $\pi = 3.14159\ldots$, and is therefore declared as `const double pi = 2 * asin(1)`. The actual value is obtained from the arc sine function `asin()`, which is defined as the inverse of the sine function: when $y = \sin(x)$, then $x = \arcsin(y)$. Since $\sin(\pi/2) = 1$, we have $pi = 2\arcsin(1)$.

5.2 Coordinates in the Center of Mass System

Unlike the two-body case, there is no gain in simplicity if we would use relative coordinates for the N-body system in general. For two bodies, there is only one set of relative coordinates, while there are two sets of particle coordinates, one for each particle. For three bodies, there are three combinations of separations between individual particles, just as there are three sets of particle coordinates. In that case, it is a matter of taste which one one prefers to use. For all higher values of N, the number of relative separations is always larger than the number of particles (six versus four for $N = 4$, for example). In conclusion, from $N = 3$ onward, it makes more sense to define the positions and velocities with respect to a given coordinate system.

Although not necessary, it is often convenient to use the center of mass system for our orbit calculations. The center of mass is defined in any coordinate system as

$$\mathbf{R} = \frac{1}{M}\sum_{i=1}^{N} m_i \mathbf{r}_i \tag{5.3}$$

where N is the total number of particles, m_i and $\mathbf{r}_i$ are the mass and the position of particle i, and $M = \sum_{i=1}^{N} m_i$ is the total mass of the system. We can interpret the right hand side as a type of lever arm equation. In a one-dimensional system of weights hanging from a beam in the Earth's gravitational field, the left and right parts of the beam will be in equilibrium if we support the beam exactly at the center of mass. The same is true for a two-dimensional plank with masses.

With three dimensions, we have no room left in an extra dimension for external support, but an analogous result still holds: the motion of the center of mass is the same as if the entire mass of the system was concentrated there and acted upon by the resultant of all external forces. See any textbook on classical dynamics for a derivation of this property. In the case of an isolated N-body system, there are no external forces, and therefore the center of mass will move in a straight line.

Starting with a given coordinate system, and subtracting the center of mass position vector $\mathbf{R}$ from all particle positions allows us to construct a representation of the N-body system in its c.o.m. system (a short hand for 'center of mass'). Subtracting the c.o.m. position is not enough, however. While this causes a momentary centering, it is still quite possible that the N-body will start drifting off soon thereafter. To keep the system in place, at least on average, we also have to subtract the velocity of the c.o.m. $\mathbf{V}$ from all particle velocities. Differentiation of Eq. 5.3 gives:

$$\mathbf{V} = \frac{1}{M}\sum_{i=1}^{N} m_i \mathbf{v}_i \tag{5.4}$$

This shows, incidentally, that the total momentum of all particles is zero in the c.o.m. coordinate system. Since the c.o.m. moves in a straight line, no further corrections are necessary. Throughout this book, when we set up initial conditions for an N-body system, we will do so in the c.o.m. of that system.

At the beginning of our code, we see how positions, velocities, and accelerations are now represented by two-dimensional arrays, where the first counter indicates the number of the particle in an N-body system, and the second counter the Cartesian coordinate. Note that in C++ array counters start at zero. Therefore, the y component of the position of the 3rd particle is not stored in `a[3][2]` as one might naively think, but rather in `a[2][1]`. In a 3-body system such as the one coded here, the value `a[3][2]` is in fact undefined, since it lies outside the memory area assigned to the area; using that value would present a serious bug.

5.3 Setting up Three Stars on a Circle

Although our code is written so as to function for an arbitrary number of particles, we initialize only three particles at the beginning of `leapfrog2.C`. This can be easily changed to a larger number of particles in different configurations. A few chapters later we will see how to read in an arbitrary N-body configuration.

It was Bob who was curious to see what happened when we place three stars equally spaced along a circle. After he had coded up the two-body leapfrog, which showed him how stable a two-body orbit of two particles was, he asked Alice whether a three body configuration would be equally stable. She said it wouldn't, and they all agreed that it would be fun to see for themselves. For simplicity, they gave all particles the same mass `m = 1`.

Anticipating that the instability might take some time to develop, they decided to add one other input option, besides the time step `dt`, namely the total duration of the run `t_end`, as we can see in the code. Directly below that request, we see how they spaced the three stars around the unit circle. The first star was chosen to reside at distance unity along the positive x axis, implying a position vector $\mathbf{r} = \{1, 0, 0\}$. With the condition that the motion takes place in the $\{x, y\}$ plane, the position of the other two stars is then fixed, as shown.

Following the position assignments, the accelerations are calculated. As we saw before, we have to spin the wheels of the gravitational acceleration module once, before we can safely enter the main integration loop. As far as the code is concerned, there is no surprise here, given that we already figured out the overall structure in the two-body case. The only thing new, apart from the use of two-dimensional arrays, is the double loop over all particles in the system. The outer loop once visits each particle i, while the inner loop visits only those particles j for which $j > i$. At the end of the loop the accelerations of particles i and j on each other are calculated. The reason to avoid $j = i$ is that the particles show no self-attraction. The reason to skip $j < i$ is a matter of efficiency: once we have calculated the intermediate quantities needed to calculate `a[j][k]`, the acceleration of particle j caused by the gravitational attraction of particle i, we may as well calculate `a[j][k]` on the spot, which is the acceleration of particle i caused by the gravitational attraction of particle j. In our equal-mass case the two are in fact equal in magnitude and opposite in sign, but in the more general case where the two masses are unequal, we would introduce a mass array, and write:

```
a[i][k] += m[j] * rji[k] / r3;
a[j][k] -= m[i] * rji[k] / r3;
```

The next group of lines employs a shortcut in order to determine the initial velocities of the three particles. Since an N-body system is governed by second order differential equations, we have to supply positions and velocities for all particles as initial conditions before we can start to solve the equations of motion, which give us the accelerations. In order to guarantee that our three stars start off on a circular motion, we use the fact that the centrifugal acceleration in a uniform circular orbit is given by v^2/r. Since the gravitational centripetal acceleration has to balance the centrifugal acceleration, we can solve for v, the magnitude of the velocity, once we know a, the magnitude of the gravitational acceleration, a quantity that is equal by symmetry for all three particles, as is v and r. We find $v = \sqrt{ar}$.

In our case, the magnitude of the position of each star was chosen to be unity. Therefore, the magnitude of the velocities are $v = \sqrt{a}$. The direction of the velocity of the first particle is chosen to lie along the positive y axis. This choice determines the other two velocities `v[1][]` and `v[2][]` by successive rotations over 120 degrees, or $2\pi/3$ (by the way, while it is nice to let the computer do all the calculating, it is not difficult to derive with pen and paper that in our case $a = 1\sqrt{3}$ and thus $v = 3^{-1/4}$).

The rest of the code is again a straightforward generalization of the two-body case. Like we did before, we compute the total energy before and after the full orbit integration. Note that we could have folded the calculation of the potential energy in with the calculation of the initial accelerations. Doing so would make the program somewhat shorter, but it would make the logical structure less easy to understand. There would be no gain in overall efficiency either, since almost all the compute happens in the `for` loop which gets traversed thousands or more times. In comparison, what happens only once before and after the orbit integration can be as inefficient as we want and still not affect the total running time significantly.

5.4 Looking for Instabilities

Let us follow the conversation of Alice, Bob, and Carol, while they are using `leapfrog2.C` as a laboratory tool to investigate the stability of the behavior of three bodies initially moving uniformly and equally space along a circle. This time, Alice is behind the key board.

Alice: Having finally debugged our new 3-body leapfrog code enough to keep the compiler happy, let us now see how the three stars behave when we launch them on a circle. Here goes.

```
|gravity> g++ -o leapfrog2 leapfrog2.C
|gravity> leapfrog2 > leapfrog2_0.01_10.out
Please provide a value for the time step
0.01
and for the duration of the run
10
Initial total energy E_in = -0.866025
Final total energy E_out = -0.866025
absolute energy error: E_out - E_in = 2.72254e-10
relative energy error: (E_out - E_in) / E_in = -3.14372e-10
|gravity>
```

Bob: Wow, what a small energy error! Clearly, our leapfrog code can run circles around the forward Euler code.

Alice: What you see here is partly an artifact of going around a perfect circle. The leapfrog will do less well on an ellipse or on an arbitrary orbit. The symmetry of the circle together with the time symmetry of the leapfrog helps to cancels errors. So from the fact that the energy errors are so small, I bet that we are still moving happily around the initial circle.

Carol: I don't want to bet yet, but let us inspect. Can you show us a picture?

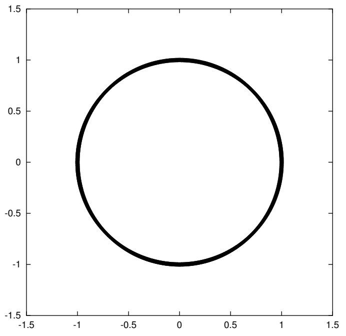

Figure 5.1: The first attempt to integrate the orbits of three stars starting off on a circle, with time step $dt = 0.01$ and a total duration of $t_{end} = 10$

Bob: You are right, Alice, as far as moving on a circle goes. Now I wonder whether you were right about the stars developing an instability. It doesn't look like it — so far!

Alice: Only one way to find out:

```
|gravity> leapfrog2 > leapfrog2_0.01_100.out
Please provide a value for the time step
0.01
and for the duration of the run
100
Initial total energy E_in = -0.866025
Final total energy E_out = -0.86602
absolute energy error: E_out - E_in = 5.63203e-06
relative energy error: (E_out - E_in) / E_in = -6.5033e-06
|gravity>
```

Bob: Okay, Alice seems to be right on both counts! First of all, the energy error is now ten thousand times larger, even though we ran only ten times longer, so that

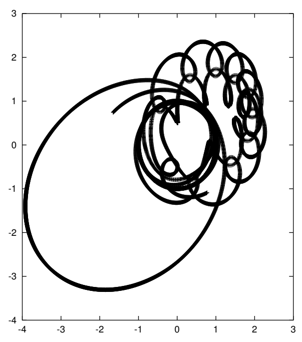

Figure 5.2: The second attempt to integrate the orbits of three stars starting off on a circle, with time step $dt = 0.01$ and a total duration of $t_{end} = 100$

confirms her circle argument. But before I congratulate her on the instability, I have one nagging question. I wonder whether this fancy three-body dance is caused by numerical instabilities caused by finite accuracy of the leapfrog, or whether it is a physical instability which real stars would undergo?

Alice: I bet the latter, since we have not yet discovered even one such system in the sky, even though we have found countless multiple stars, from triples to quadruples and quintuples and up. But why don't we check? Let's refine the time step a few times by an order of magnitude.

```
|gravity> leapfrog2 > leapfrog2_0.001_100.out
Please provide a value for the time step
0.001
and for the duration of the run
100
Initial total energy E_in = -0.866025
Final total energy E_out = -0.866025
absolute energy error: E_out - E_in = 1.58819e-07
relative energy error: (E_out - E_in) / E_in = -1.83388e-07
|gravity>
```

```
|gravity> leapfrog2 > leapfrog2_0.0001_100.out
Please provide a value for the time step
0.0001
and for the duration of the run
100
Initial total energy E_in = -0.866025
Final total energy E_out = -0.866025
absolute energy error: E_out - E_in = 1.1609e-09
```

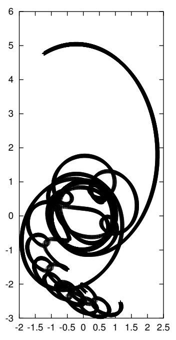

Figure 5.3: The third attempt to integrate the orbits of three stars starting off on a circle, with time step $dt = 0.001$ and a total duration of $t_{end} = 100$

```
relative energy error: (E_out - E_in) / E_in = -1.3405e-09
|gravity>
```

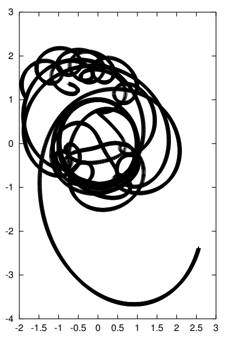

Figure 5.4: The fourth attempt to integrate the orbits of three stars starting off on a circle, with time step $dt = 0.0001$ and a total duration of $t_{end} = 100$

Bob: Well, Alice, what did I tell you? Each picture is wildly different, so I don't think that we are converging onto an actually existing unstable orbit.

Alice: Well, let's try one more refinement.

```
|gravity> leapfrog2 > leapfrog2_0.00001_100.out
Please provide a value for the time step
0.00001
and for the duration of the run
```

```
100
Initial total energy E_in = -0.866025
Final total energy E_out = -0.866025
absolute energy error: E_out - E_in = 1.17143e-10
relative energy error: (E_out - E_in) / E_in = -1.35265e-10
|gravity>
```

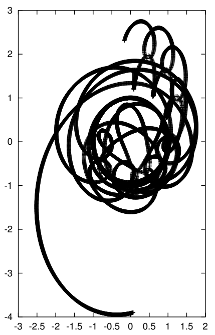

Figure 5.5: The fifth attempt to integrate the orbits of three stars starting off on a circle, with time step $dt = 0.00001$ and a total duration of $t_{end} = 100$

Alice: Well, at least the last two are somewhat similar, unlike all previous pictures. But alike they are certainly not.

Bob: Okay, one more refinement, and perhaps the whole thing will finally converge! With `dt = 0.000001`, this means a hundred million steps, but why not, computers are fast enough these days.

```
|gravity> leapfrog2 > leapfrog2_0.000001_100.out
Please provide a value for the time step
0.000001
and for the duration of the run
100
Initial total energy E_in = -0.866025
Final total energy E_out = -0.866025
absolute energy error: E_out - E_in = 8.68972e-13
relative energy error: (E_out - E_in) / E_in = -1.0034e-12
|gravity>
```

Bob: So much for similar but not alike. This one doesn't look anything like the previous one. And we are rapidly approaching machine accuracy.

Alice: Hmmmm.

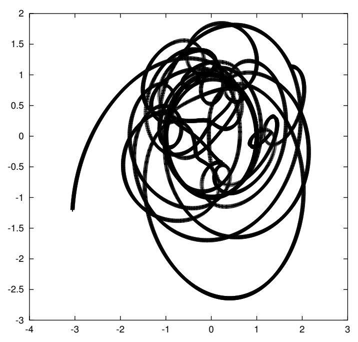

Figure 5.6: The sixth attempt to integrate the orbits of three stars starting off on a circle, with time step $dt = 0.000001$ and a total duration of $t_{end} = 100$

5.5 Priming the Pump

Carol: Hmmmm indeed. Aha! I have an idea. Even if the instability exists in the real world, it still has to be triggered in some way. In our computer it is triggered by round-off errors. In the real world, if we put a pencil on its tip straight up and let go, the pencil will fall in a random direction, triggered by the Brownian motion of the air molecules, even if we would have aligned it perfectly vertically. Brownian motion is not reproducible, but neither can you expect round-off errors to be reproducible when you change time steps.

Alice: Interesting point! Let us try. Let us give one of the stars a little perturbation in its velocity, on the order of 10^{-4}, huge compared to the round-off noise around 10^{-15}, but still larger than typical numerical errors of the leapfrog, certainly for small enough step size. Okay, here is a version, `leapfrog2a.C`. The only change I have made with respect to `leapfrog2a.C` is the addition of one line, immediately following the assignment of the velocities:

```
v[0][0] += 0.0001;
```

Bob: I see. You are priming the pump, to guide the instability. And you have put this line just after the velocity initialization, so that it appears conveniently just before the first calculation of the kinetic energy. That way, the first total energy calculation takes this displacement in the value of this one velocity component into account. By now, I do expect the system to fall apart again, and I'm really curious. If the resulting dance pattern jumps all over the place again, we definitely are dealing with errors introduced by the leapfrog. But if we converge to one particular orbit, then I concede that we have uncovered a physical instability, where a given well-defined tiny displacement leads to a well-defined resulting way in which the system falls apart. Let's have a look, Alice!

Alice: I'll start again with only ten time units, to check that our three stars complete at least a few orbits on their original circle, before going their own way.

```
|gravity> g++ -o leapfrog2a leapfrog2a.C
|gravity> leapfrog2a > leapfrog2a_0.01_10.out
Please provide a value for the time step
0.01
and for the duration of the run
10
Initial total energy E_in = -0.866025
Final total energy E_out = -0.866025
absolute energy error: E_out - E_in = 1.21321e-09
relative energy error: (E_out - E_in) / E_in = -1.40089e-09
|gravity>
```

Bob: Applying Alice's detective strategy, I would say that the energy error is still too small to show detectable deviations from a circle. Notice that the error is clearly larger than before, as we would expect, since we have given the instability a hand by starting with a gentle push.

Alice: Elementary, my dear Bob. But let us see whether you are right.

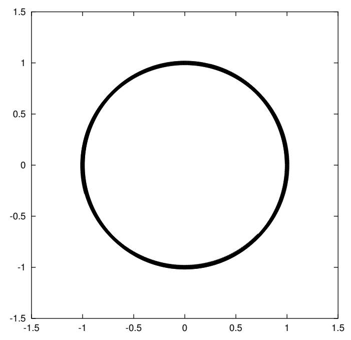

Figure 5.7: The first attempt to integrate the orbits of three stars starting off on a circle with an initial velocity perturbation of $dv_{init} = 0.0001$, time step $dt = 0.01$ and a total duration of $t_{end} = 10$

Bob: Encouraged by my success in following Alice's footsteps,, I will now predict a significant degradation in the energy conservation, together with the appearance of a wild three-body dance, when we go to 100 time units.

Alice: You should become my agent — here are the run and the plot:

```
|gravity> leapfrog2a > leapfrog2a_0.01_100.out
Please provide a value for the time step
0.01
and for the duration of the run
100
Initial total energy E_in = -0.866025
Final total energy E_out = -0.865241
absolute energy error: E_out - E_in = 0.000784568
relative energy error: (E_out - E_in) / E_in = -0.000905941
|gravity>
```

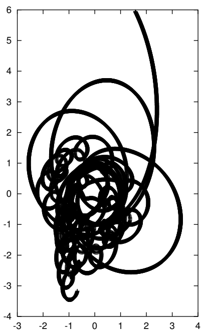

Figure 5.8: The second attempt to integrate the orbits of three stars starting off on a circle with an initial velocity perturbation of $dv_{init} = 0.0001$, time step $dt = 0.01$ and a total duration of $t_{end} = 100$

Carol: I bet Bob didn't envision *such* a big degradation in energy conservation! Let's make the time steps a factor ten smaller.

```
|gravity> leapfrog2a > leapfrog2a_0.001_100.out
Please provide a value for the time step
0.001
and for the duration of the run
100
Initial total energy E_in = -0.866025
Final total energy E_out = -0.866033
absolute energy error: E_out - E_in = -7.29447e-06
relative energy error: (E_out - E_in) / E_in = 8.42293e-06
|gravity>
```

Bob: You are right, I had expected an energy error more like the present one, already in the previous round.

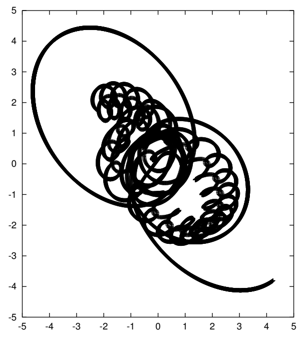

Figure 5.9: The third attempt to integrate the orbits of three stars starting off on a circle with an initial velocity perturbation of $dv_{init} = 0.0001$, time step $dt = 0.001$ and a total duration of $t_{end} = 100$

Alice: My hunch is that the previous round may have featured a particularly close encounter. Since our code does not know how to decrease the time step during close encounters, even one such encounter can spoil the energy conservation of a whole run. At some point we'll have to make our integrator smarter and more autonomous in the eye of danger.

Carol: But not now. I want to see whether we will reach convergence. So far the last two pictures don't look any more like each other than in the case before we primed the pump. Can we refine the time step by another order of magnitude?

Alice: Sure, my pleasure.

5.6 Reaching Convergence

```
|gravity> leapfrog2a > leapfrog2a_0.0001_100.out
Please provide a value for the time step
0.0001
and for the duration of the run
100
Initial total energy E_in = -0.866025
Final total energy E_out = -0.866025
absolute energy error: E_out - E_in = 2.55083e-08
relative energy error: (E_out - E_in) / E_in = -4.10015e-08
|gravity>
```

Bob: Still quite different pictures. And notice that the energy errors don't yet scale with

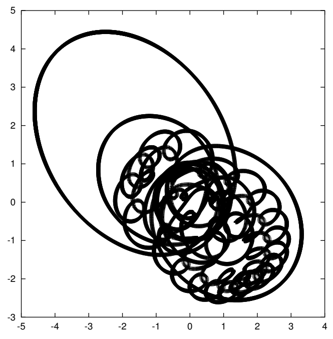

Figure 5.10: The fourth attempt to integrate the orbits of three stars starting off on a circle with an initial velocity perturbation of $dv_{init} = 0.0001$, time step $dt = 0.0001$ and a total duration of $t_{end} = 100$

a factor of one hundred, as they would once we really converge to a unique set of tracks — if there is such a thing.

Alice: Still doubting my suggestion, aren't you? I must admit, it doesn't look very good, so far. But hope springs eternal, so let's try another order of magnitude step size refinement, down to `dt = 0.000,01` which implies ten million integration steps in total for this three-body problem.

```
|gravity> leapfrog2a > leapfrog2a_0.00001_100.out
Please provide a value for the time step
0.00001
and for the duration of the run
100
Initial total energy E_in = -0.866025
Final total energy E_out = -0.866025
absolute energy error: E_out - E_in = -1.59043e-11
relative energy error: (E_out - E_in) / E_in = 1.83647e-11
|gravity>
```

Bob: Surprise! Virtually the same picture, all of a sudden! So now I agree that Alice was really right on both counts. But I have one last worry. Why has the energy error suddenly shrunk so much, by a factor of more than a thousand, where I would have expected only a hundred?

Alice: Hard to say. In the two-body case it was not so hard to track down all aspects of the integration errors, since the underlying orbits were so simple. But look here, what a mess! In order to answer your last worry, we need to build a lot more tools.

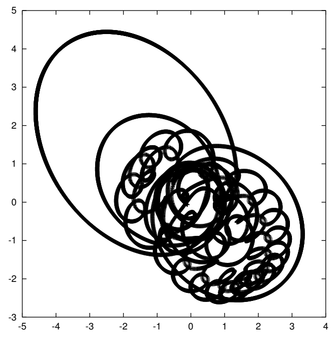

Figure 5.11: The fifth attempt to integrate the orbits of three stars starting off on a circle with an initial velocity perturbation of $dv_{init} = 0.0001$, time step $dt = 0.00001$ and a total duration of $t_{end} = 100$

Besides a switch to variable time steps in the integrator, which we'll get around to at some point, we need much better ways to dissect orbit segments, in order to focus on particularly interesting interactions. Ideally, we would like to build in some form of artificial intelligence, so that the particles themselves will know how to take notes while they are flying around and being flung around. When we can harvest their stories at the end of an integration, we will have a head start in answering questions of the type you asked. You'll have to be a bit patient though, since it will require quite a number of sessions to get to that point.

Carol: Well, I'm game. It's a lot of fun already, to see how much progress we have made. The fact that we have now reached a point where we can ask questions that call for some form of artificial intelligence to answer them is impressive enough. Besides, it means that I, too, might start to write a working paper about part of what we are doing now. One footnote, though: I wouldn't call these orbits a mess; on the contrary, I find all this wheeling around quite elegant.

Bob: Elegance is in the eye of the beholder, I guess, but I must admit that I, too, find this figure skating quite pretty. And I am desperately thinking how I could make a working paper out of all this — perhaps for an art class?

Alice: When I called the orbits a mess, I meant it in an affectionate way. It reminded me of my room, which I haven't cleaned up for a while, but that's another story. How about inviting more art, and running this thing for a hundred million time steps, by going to `t_end = 1000`?

Bob: You have my blessing.

```
|gravity> leapfrog2a > leapfrog2a_0.00001_1000.out
```

```
Please provide a value for the time step
0.00001
and for the duration of the run
1000
Initial total energy E_in = -0.866025
Final total energy E_out = -0.866025
absolute energy error: E_out - E_in = -5.78748e-10
relative energy error: (E_out - E_in) / E_in = 6.6828e-10
|gravity>
```

5.7 The End of the Story

When Alice plotted the resulting picture, they were all surprised to see an almost one-dimensional plot, barely visible on the screen. There seemed to be two particles moving away from the center, one reaching $y = 160$ on top, and one reaching $y = -320$ below. The tick marks and labels on the x axis were so compressed, they were not even readable.

Alice then pointed out that it was likely that there had been a close near-simultaneous three-body encounter, from which two stars emerged as a tight double star, a binary in astrophysical terms. Since a tight binary has a large negative binding energy, the remaining positive energy has to be carried away by the third star. Since we are plotting orbits in the c.o.m. frame, the direction of the single star's escape orbit must be exactly opposite from the direction of the escape orbit of the double star. And since the latter carries twice as much mass as the single star, its speed in the c.o.m. frame has to be half as large as that of the star escaping by itself. This explains why the track moving downwards has reached twice as large a distance from the origin as the track moving upwards. Although not visible in the terribly skinny figure, the upwards track must be that of the binary, and the downwards track that of the single star.

Given the large distance that the stars had traveled, Alice decided to be more conservative, and try to run the simulation for only 200 time units, rather than 1,000. Here is her result:

```
|gravity> leapfrog2a > leapfrog2a_0.00001_200.out
Please provide a value for the time step
0.00001
and for the duration of the run
200
Initial total energy E_in = -0.866025
Final total energy E_out = -0.866025
absolute energy error: E_out - E_in = 9.36223e-11
relative energy error: (E_out - E_in) / E_in = -1.08106e-10
|gravity>
```

The accompanying plot she made was still very skinny. They at least could see a few tick marks below along the x axis, though all the numbers there were still printed on top of each other. The upwards track reached $y = 23$, while the downwards track extended to $y = -46$. And indeed, the upwards track could now be seen to be slightly thicker than the downwards track, in agreement with Alice's prediction that that one was formed by the two stars in a tight binary.

Again shortening the duration of the run, Alice let the stars run to `t_end = 150`:

```
|gravity> leapfrog2a > leapfrog2a_0.00001_150.out
Please provide a value for the time step
0.00001
and for the duration of the run
150
Initial total energy E_in = -0.866025
Final total energy E_out = -0.866025
absolute energy error: E_out - E_in = 1.02107e-09
relative energy error: (E_out - E_in) / E_in = -1.17903e-09
|gravity>
```

And still the plot looked skinny! Most of the interaction region in the center resembled a single ink blob, and the two tracks now reached to $y = 13$ and $y = -26$, respectively. They concluded that the close encounter leading to the double escape must have taken place soon after `t = 100`. The reasoning was as follows: if the stars would move in straight line in uniform motion, an extrapolation of the last two results would lead to two short tracks already having started at `t = 100`, reaching $y = 3$ and $y = -6$, respectively. However, the particles are decelerating, due to the gravitational interaction between the single star and the double star. Therefore, the initial separation must have happened at a higher speed, moving the time of the close three-body encounter further toward the future, most likely around or slightly after `t = 100`.

To check this reasoning, Alice produced an even shorter run, this time only to `t = 120`:

```
|gravity> leapfrog2a > leapfrog2a_0.00001_120.out
Please provide a value for the time step
0.00001
and for the duration of the run
120
Initial total energy E_in = -0.866025
Final total energy E_out = -0.866025
absolute energy error: E_out - E_in = 1.48818e-11
relative energy error: (E_out - E_in) / E_in = -1.7184e-11
|gravity>
```

This finally produced a reasonable plot, with readable numbers along the x axis:

Looking back at the three overly skinny plots, not shown here, and using the numbers they read off as reported above, our friends noticed that the separation between binary and

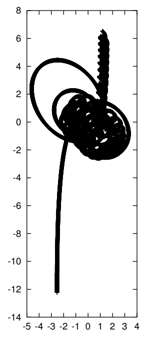

Figure 5.12: The ninth attempt to integrate the orbits of three stars starting off on a circle with an initial velocity perturbation of $dv_{init} = 0.0001$, time step $dt = 0.00001$ and a total duration of $t_{end} = 120$

single star at time `t = 1000` was about seven times larger than at time `t = 200`. Uniform motion would have predicted an increase of separations by a factor nine. The actual factor seven was recognized as evidence that the relative motion between single star and binary was still slowing down. The question was raised whether the stars would fall back again eventually, or whether they would really escape in opposite directions. To answer this question, their was no need for an extra plot. They just ran a run of a billion time steps (which took a while) and tossed out all but the last few lines of the output:

```
|gravity> leapfrog2a | tail
Please provide a value for the time step
0.00001
and for the duration of the run
10000
Initial total energy E_in = -0.866025
Final total energy E_out = -0.866025
absolute energy error: E_out - E_in = 6.29064e-09
relative energy error: (E_out - E_in) / E_in = -7.26381e-09
-68.4785 1632.06 0 -0.137523 -0.382285 0
137.396 -3263.57 0 0.0141025 -0.325753 0
-67.9165 1631.51 0 0.112159 0.719282 0
-68.4798 1632.06 0 -0.126162 -0.39353 0
137.396 -3263.57 0 0.0141025 -0.325753 0
-67.9154 1631.51 0 0.100483 0.73056 0
-68.481 1632.05 0 -0.114485 -0.404807 0
137.397 -3263.57 0 0.0141025 -0.325753 0
-67.9145 1631.52 0 0.0884761 0.741873 0
-68.4821 1632.05 0 -0.102479 -0.41612 0
|gravity>
```

This proved clear evidence for final escape of both binary and single star. Whereas the single star had separated from the binary by a distance of 320+160=480 units along the y axis at `t = 1000`, the separation had now increased to about 4900 units, in a roughly eleven times longer interval, counted from the slingshot event around `t = 100`. The conclusion was that there had been some residual slowing down, given that the separation had increased by a factor ten rather than eleven, but that escape had been established, since there was no indication that the slow-down would be enough to lead to a fall back.

After concluding once that it was high time to write better analysis tools, which would allow a more direct investigation of the questions they had now answered in a somewhat haphazard way, our friends called it quits for that night.

5.8 Three Bodies on a Figure Eight

The next time Alice, Bob, and Carol met, they decided to make one more modification to their 3-body version of the leapfrog code, before going to a more modular and flexible N-body version. Alice had mentioned a relatively recent discovery of a new solution to the centuries old three-body problem, in the form of three stars of equal mass following each other along a single orbit resembling a figure eight. Here is the first half of the code, which lists the initial conditions, followed by the computation of the initial accelerations as well as kinetic and potential energy. The second half of the code is identical to the second half of `leapfrog2.C`.

Code 5.2 (leapfrog3.C: first half)

```
//-----------------------------------------------------------------------------
// leapfrog3.C
//-----------------------------------------------------------------------------
#include  <iostream>
#include  <cmath>
using namespace std;

int main()
{
    int n = 3;
    double r[n][3], v[n][3], a[n][3];
    const double m = 1;
    double dt, t_end;

    cerr << "Please provide a value for the time step" << endl;
    cin >> dt;
    cerr << "and for the duration of the run" << endl;
    cin >> t_end;
```

```
r[0][0] = 0.9700436;
r[0][1] = -0.24308753;
r[0][2] = 0;
v[0][0] = 0.466203685;
v[0][1] = 0.43236573;
v[0][2] = 0;

r[1][0] = -r[0][0];
r[1][1] = -r[0][1];
r[1][2] = -r[0][2];
v[1][0] = v[0][0];
v[1][1] = v[0][1];
v[1][2] = v[0][2];

r[2][0] = 0;
r[2][1] = 0;
r[2][2] = 0;
v[2][0] = -2 * v[0][0];
v[2][1] = -2 * v[0][1];
v[2][2] = -2 * v[0][2];

double ekin = 0, epot = 0;
for (int i = 0; i < n; i++)
    for (int k = 0; k < 3; k++)
        a[i][k] = 0.0;
for (int i = 0; i < n; i++){
    for (int j = i+1; j < n; j++){
        double rji[3];
        for (int k = 0; k < 3; k++)
            rji[k] = r[j][k] - r[i][k];
        double r2 = 0;
        for (int k = 0; k < 3; k++)
            r2 += rji[k] * rji[k];
        double r = sqrt(r2);
        double r3 = r2 * r;
        for (int k = 0; k < 3; k++){
            a[i][k] += m * rji[k] / r3;
            a[j][k] -= m * rji[k] / r3;
        }
        epot -= m*m/r;
    }
    for (int k = 0; k < 3; k++)
        ekin += 0.5 * m * v[i][k] * v[i][k];
}
double e_in = ekin + epot;
```

```
    cerr << "Initial total energy E_in = " << e_in << endl;
    . . . .
//-----------------------------------------------------------------------------
```

They first tried their standard time step choice `dt = 0.01` for a integration until `t_end = 100`:

```
|gravity> g++ -o leapfrog3 leapfrog3.C
|gravity> leapfrog3 > leapfrog3_0.01_100.out
Please provide a value for the time step
0.01
and for the duration of the run
100
Initial total energy E_in = -1.28705
Final total energy E_out = -1.28706
absolute energy error: E_out - E_in = -1.25936e-05
relative energy error: (E_out - E_in) / E_in = 9.78486e-06
|gravity>
```

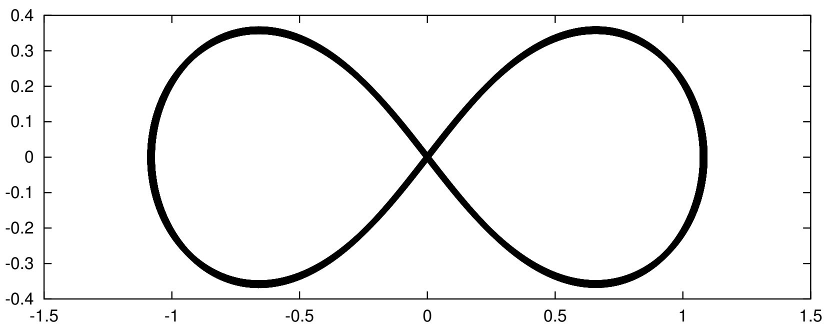

Figure 5.13: The first attempt to integrate the orbits of three stars starting off on a figure-8 orbit with time step $dt = 0.01$ and a total duration of $t_{end} = 100$

Even though the energy error was not that small, no deviation in the orbit was visible. Clearly, the three-body figure-8 orbit is far more stable than the three-body circular orbit, which fell apart well before reaching 100 time units, as we saw before. Taking a ten times smaller time step neatly decreased the errors by a factor of one hundred, also a good sign of reaching convergence in the orbits:

```
|gravity> leapfrog3 > leapfrog3_0.001_100.out
Please provide a value for the time step
0.001
and for the duration of the run
100
Initial total energy E_in = -1.28705
Final total energy E_out = -1.28705
absolute energy error: E_out - E_in = -1.19725e-07
relative energy error: (E_out - E_in) / E_in = 9.30233e-08
|gravity>
```

When plotting this last result, it looks exactly like Fig. 5.13, so we will omit it here.

Our friends then repeated the second test for instability, by 'priming the pump' again, with an offset of 10^{-4} in the x component of the first particle. As before, they added the one line

```
    v[0][0] += 0.0001;
```

immediately following the assignment of velocities at the beginning of the program. They renamed this modified code `leapfrog3a.C`. Here is what happened by `t_end = 100`:

```
|gravity> g++ -o leapfrog3a leapfrog3a.C
|gravity> leapfrog3a > leapfrog3a_0.001_100.out
Please provide a value for the time step
0.001
and for the duration of the run
100
Initial total energy E_in = -1.287
Final total energy E_out = -1.287
absolute energy error: E_out - E_in = -1.27819e-07
relative energy error: (E_out - E_in) / E_in = 9.93155e-08
|gravity>
```

Again, no visible deviations, very similar small energy errors, and no sign of any instability. This confirms what has been described in the literature, that the figure-8 orbit for three bodies is stable.

We leave it as an exercise for the reader to explore when and how the figure-8 configuration falls apart (or perhaps winds up in a higher order version of a stable orbit, with more loops and turns) upon an increase of the magnitude of the perturbation, either in one of the velocity components or in one of the position components, or in a mix of perturbations of various position and velocity components.

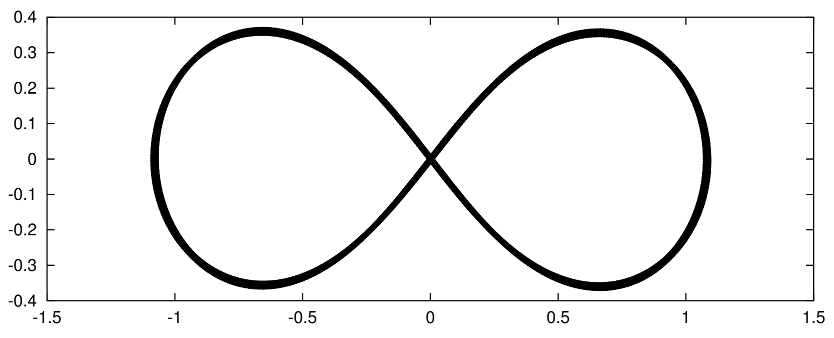

Figure 5.14: The first attempt to integrate the orbits of three stars starting off on a figure-8 orbit with an initial velocity perturbation of $dv_{init} = 0.0001$, time step $dt = 0.001$ and a total duration of $t_{end} = 100$

Chapter 6

Exploring $N = 3$ with a Hermite Algorithm

Having seen the dramatic improvement that came from switching from the forward Euler algorithm to the leapfrog, the obvious next step was to go to yet higher order algorithms. A quick look in a few books of numerical methods showed our friends that there was a bewildering choice of third- and fourth-order methods to choose from. Alice then mentioned that her thesis advisor had pointed her to an elegant and natural generalization of the leapfrog algorithm, by the name of the Hermite scheme.

6.1 A Surprisingly Simple Hermite Scheme

The most symmetric Hermite version, and the one closest resembling the leapfrog is this one:

$$\begin{aligned} \mathbf{r}_{i+1} &= \mathbf{r}_i + \tfrac{1}{2}(\mathbf{v}_i + \mathbf{v}_{i+1})dt + \tfrac{1}{12}(\mathbf{a}_i - \mathbf{a}_{i+1})(dt)^2 \qquad (6.1) \\ \mathbf{v}_{i+1} &= \mathbf{v}_i + \tfrac{1}{2}(\mathbf{a}_i + \mathbf{a}_{i+1})dt + \tfrac{1}{12}(\mathbf{j}_i - \mathbf{j}_{i+1})(dt)^2 \qquad (6.2) \end{aligned}$$

Here $\mathbf{j} = d\mathbf{a}/dt$ is the *jerk*, the time derivative of the acceleration, and therefore the third time derivative of position:

$$\mathbf{j} = \frac{d^3}{dt^3}\mathbf{r} \qquad (6.3)$$

The term 'jerk' has crept into the literature relatively recently, probably originally as a pun. If a car or train changes acceleration relatively quickly you experience not a smoothly accelerating or decelerating motion, but instead a rather 'jerky' one.

The jerk can be computed through straightforward differentiation of Newton's gravitational equations, Eq. 5.2:

$$\mathbf{j}_i = G \sum_{\substack{j=1 \\ j \neq i}}^{N} M_j \left[\frac{\mathbf{v}_{ji}}{r_{ji}^3} - 3 \frac{(\mathbf{r}_{ji} \cdot \mathbf{v}_{ji})\mathbf{r}_{ji}}{r_{ji}^5} \right] \tag{6.4}$$

where $\mathbf{v}_{ji} = \mathbf{v}_j - \mathbf{v}_i$.

As an aside, note that the jerk has one very convenient property. Although the expression above looks quite a bit more complicated than Newton's original equations, they can still be evaluated through one pass over the whole N-body system. This is no longer true for higher derivatives. For example, we can obtain the fourth derivative of the position of particle i (the *snap*, see next section) by differentiating Eq. 6.4:

$$\begin{aligned} \frac{d^4}{dt^4}\mathbf{r}_i = G \sum_{\substack{j=1 \\ j \neq i}}^{N} M_j \Bigg[& \frac{\mathbf{a}_{ji}}{r_{ji}^3} - 6 \frac{(\mathbf{r}_{ji} \cdot \mathbf{v}_{ji})}{r_{ji}^5} \mathbf{v}_{ji} \\ & + \left\{ -3 \frac{v_{ji}^2}{r_{ji}^5} - 3 \frac{(\mathbf{r}_{ji} \cdot \mathbf{a}_{ji})}{r_{ji}^5} + 15 \frac{(\mathbf{r}_{ji} \cdot \mathbf{v}_{ji})^2}{r_{ji}^7} \right\} \mathbf{r}_{ji} \Bigg] \end{aligned} \tag{6.5}$$

where $\mathbf{a}_{ji} = \mathbf{a}_j - \mathbf{a}_i$, and this is the expression that thickens the plot. Unlike the $\mathbf{r}_{ji}$ and $\mathbf{v}_{ji}$ expressions, that are given by the initial conditions, $\mathbf{a}_{ji}$ has to be calculated from the positions and velocities. However, this calculation does not only involve the pairwise attraction of particle j on particle i, but in fact all pairwise attractions of all particles on each other! This follows immediately when we write out what the shorthand implies:

$$\mathbf{a}_{ji} = \mathbf{a}_j - \mathbf{a}_i = G \sum_{\substack{k=1 \\ k \neq j}}^{N} \frac{M_k}{r_{kj}^3} \mathbf{r}_{kj} - G \sum_{\substack{k=1 \\ k \neq i}}^{N} \frac{M_k}{r_{ki}^3} \mathbf{r}_{ki} \tag{6.6}$$

When we substitute this back into Eq. 6.5, we see that we have to do a double pass over the N-body system, summing over both indices k and j in order to compute a single fourth derivative for the position of particle i.

6.2 Comparison with the Leapfrog

When we look at Eqs. 6.1, 6.2, we see some familiar features. Neglecting the higher-order term for the moment, we recognize the leapfrog: the new position is effectively determined by the mid-point velocity $v_{i+1/2}$, here approximated as the average between the two adjacent values v_i and v_{i+1}. Similarly, the new velocity is effectively determined by the mid-point acceleration.

In fact, the analogy can be made more precise. Recalling the leapfrog, as written centered on integer times, Eqs. 4.4, 4.5:

$$\begin{aligned}\mathbf{r}_{i+1} &= \mathbf{r}_i + \mathbf{v}_i dt + \mathbf{a}_i (dt)^2/2 \qquad (6.7)\\ \mathbf{v}_{i+1} &= \mathbf{v}_i + (\mathbf{a}_i + \mathbf{a}_{i+1})dt/2 \qquad (6.8)\end{aligned}$$

we can transform these back into a pseudo-leap form, without using half-integer times explicitly, by rewriting the first equation as:

$$\begin{aligned}\mathbf{r}_{i+1} &= \mathbf{r}_i + \tfrac{1}{2}(\mathbf{v}_i + \mathbf{v}_{i+1})dt + \tfrac{1}{2}(\mathbf{v}_i - \mathbf{v}_{i+1})dt + \tfrac{1}{2}\mathbf{a}_i(dt)^2\\ &= \mathbf{r}_i + \tfrac{1}{2}(\mathbf{v}_i + \mathbf{v}_{i+1})dt + \tfrac{1}{4}(-\mathbf{a}_i - \mathbf{a}_{i+1})(dt)^2 + \tfrac{1}{2}\mathbf{a}_i(dt)^2\\ &= \mathbf{r}_i + \tfrac{1}{2}(\mathbf{v}_i + \mathbf{v}_{i+1})dt + \tfrac{1}{4}(\mathbf{a}_i - \mathbf{a}_{i+1})(dt)^2\\ &= \mathbf{r}_i + \tfrac{1}{2}(\mathbf{v}_i + \mathbf{v}_{i+1})dt - \tfrac{1}{4}\mathbf{j}_i(dt)^3 \qquad (6.9)\end{aligned}$$

In the second line, we have simply rearranged terms. In the third line, we have used 6.8, and in the fourth line we have used the definition of **j**, while neglecting higher order terms in dt.

The next step is to remember that the leapfrog is a second-order scheme. The errors per step are $\propto (dt^3)$, and therefore it does not matter whether or not we include the last term $-\mathbf{j}_i(dt)^3/4$ into our leapfrog version: this term is lost in the noise, and is not going to improve the accuracy on second-level order. Therefore, we may equally well leave it out. Doing so transforms Eqs. 6.7, 6.8 into:

$$\begin{aligned}\mathbf{r}_{i+1} &= \mathbf{r}_i + \tfrac{1}{2}(\mathbf{v}_i + \mathbf{v}_{i+1})dt \qquad (6.10)\\ \mathbf{v}_{i+1} &= \mathbf{v}_i + \tfrac{1}{2}(\mathbf{a}_i + \mathbf{a}_{i+1})dt \qquad (6.11)\end{aligned}$$

Here we see explicitly that our good old leapfrog is equivalent, up to its second-order accuracy, with the leading terms $\propto (dt)$ of the Hermite algorithm, Eqs. 6.1, 6.2. It is a curiosity of the leapfrog that at first sight it resembles a first-order scheme, since the second-order terms are hidden in the 'leapy' way of using average quantities. Yet, as we have seen, the leapfrog is fully second-order.

In a very similar way, the Hermite scheme is fourth-order, even though it resembles a second-order scheme. For details we refer to the literature, but it is interesting to see in a heuristic way why this is so.

6.3 Snap, Crackle, and Pop

First a word about terminology. We will need to introduce a few extra derivatives of the position. It would be fun to give them names with a reasonable 'feel' to them, just like with jerk. What type of motion feels even more restless than jerking motion? A sudden snap comes to mind. A what changes its state more sudden than a snap — how about a crackle? And for those familiar with American rice crispies culture, a pop cannot be far away, and indeed, if something pops it really changes high derivatives of positions in a substantial way! We are not making these names up: we have seen them used a few times before, although the precise source is likely to be lost in (recent) history. So here they are:

$$\mathbf{s} = \frac{d^4}{dt^4}\mathbf{r} \quad ; \quad \mathbf{c} = \frac{d^5}{dt^5}\mathbf{r} \quad ; \quad \mathbf{p} = \frac{d^6}{dt^6}\mathbf{r} \tag{6.12}$$

snap, *crackle*, and *pop*, respectively.

We are now in a position to write the Taylor series for the four quantities that appear in Eqs. 6.1, 6.2, up to crackle:

$$\begin{aligned}
\mathbf{r}_{i+1} &= \mathbf{r}_i + \mathbf{v}_i dt + \tfrac{1}{2}\mathbf{a}_i(dt)^2 + \tfrac{1}{6}\mathbf{j}_i(dt)^3 + \tfrac{1}{24}\mathbf{s}_i(dt)^4 + \tfrac{1}{120}\mathbf{c}_i(dt)^5 &(6.13)\\
\mathbf{v}_{i+1} &= \mathbf{v}_i + \mathbf{a}_i dt + \tfrac{1}{2}\mathbf{j}_i(dt)^2 + \tfrac{1}{6}\mathbf{s}_i(dt)^3 + \tfrac{1}{24}\mathbf{c}_i(dt)^4 &(6.14)\\
\mathbf{a}_{i+1} &= \mathbf{a}_i + \mathbf{j}_i dt + \tfrac{1}{2}\mathbf{s}_i(dt)^2 + \tfrac{1}{6}\mathbf{c}_i(dt)^3 &(6.15)\\
\mathbf{j}_{i+1} &= \mathbf{j}_i + \mathbf{s}_i dt + \tfrac{1}{2}\mathbf{c}_i(dt)^2 &(6.16)
\end{aligned}$$

We can now eliminate snap and crackle at time t_i, expressing them in terms of the acceleration and jerk at times t_i and t_{i+1}, using Eqs. 6.15, 6.16. We find:

$$\begin{aligned}
\mathbf{s}_i &= 6(\mathbf{a}_{i+1} - \mathbf{a}_i)(dt)^{-2} - 2(\mathbf{j}_{i+1} + 2\mathbf{j}_i)(dt)^{-1} &(6.17)\\
\mathbf{c}_i &= -12(\mathbf{a}_{i+1} - \mathbf{a}_i)(dt)^{-3} + 6(\mathbf{j}_{i+1} + \mathbf{j}_i)(dt)^{-2} &(6.18)
\end{aligned}$$

Substituting both expressions in Eq. 6.14 we directly find:

$$\mathbf{v}_{i+1} = \mathbf{v}_i + \tfrac{1}{2}(\mathbf{a}_i + \mathbf{a}_{i+1})dt + \tfrac{1}{12}(\mathbf{j}_i - \mathbf{j}_{i+1})(dt)^2 \tag{6.19}$$

Indeed, we have recovered Eq. 6.2, and thereby explained the mysterious factor $\frac{1}{12}$ in the last term.

Let us complete our mission, by making the same derivation for the position vector, Eq. 6.1, in the Hermite scheme. Using again the snap and crackle expressions derived above, we find:

$$\mathbf{r}_{i+1} = \mathbf{r}_i + \mathbf{v}_i dt + (\tfrac{7}{20}\mathbf{a}_i + \tfrac{3}{20}\mathbf{a}_{i+1})(dt)^2 + (\tfrac{1}{20}\mathbf{j}_i - \tfrac{1}{30}\mathbf{j}_{i+1})(dt)^3 \tag{6.20}$$

Using the same trick we employed in Eq. 6.9 to factor out the velocity terms, and using Eq. 6.19, we can rewrite the above expression as:

$$\mathbf{r}_{i+1} = \mathbf{r}_i + \tfrac{1}{2}(\mathbf{v}_i + \mathbf{v}_{i+1})dt + \tfrac{1}{10}(\mathbf{a}_i - \mathbf{a}_{i+1})(dt)^2 + \tfrac{1}{120}(\mathbf{j}_i + \mathbf{j}_{i+1})(dt)^3 \tag{6.21}$$

While this result still looks quite different from Eq. 6.1, we claim that it is identical up to fourth-order in dt, which is all we need. Our final step thus parallels the discussion following Eq. 6.9 for the leapfrog, where we had to show how terms up to second-order were identical.

First we rewrite the above equation in terms of quantities defined at $t = i$:

$$\begin{aligned}
\mathbf{r}_{i+1} = \mathbf{r}_i + \tfrac{1}{2}(\mathbf{v}_i + \mathbf{v}_{i+1})dt \quad &+ \quad \tfrac{1}{10}\mathbf{a}_i(dt)^2 - \tfrac{1}{10}(\mathbf{a}_i + \mathbf{j}_i dt + \tfrac{1}{2}\mathbf{s}_i(dt)^2)(dt)^2 \\
&+ \quad \tfrac{1}{120}\mathbf{j}_i(dt)^3 + \tfrac{1}{120}(\mathbf{j}_i + \mathbf{s}_i dt)(dt)^3 \\
= \mathbf{r}_i + \tfrac{1}{2}(\mathbf{v}_i + \mathbf{v}_{i+1})dt \quad &- \quad \tfrac{1}{12}\mathbf{j}_i(dt)^3 - \tfrac{1}{24}\mathbf{s}_i(dt)^4
\end{aligned} \tag{6.22}$$

Here we have left out terms containing $\mathbf{c}_i$, since they would be proportional to at least (dt^5) and only contribute to the error noise. We can similarly write out the last term of Eq. 6.1:

$$\begin{aligned}
\tfrac{1}{12}(\mathbf{a}_i - \mathbf{a}_{i+1})(dt)^2 \quad &= \quad \tfrac{1}{12}\left(\mathbf{a}_i(dt)^2 - \tfrac{1}{12}(\mathbf{a}_i + \mathbf{j}_i dt + \tfrac{1}{2}\mathbf{s}_i(dt)^2)\right)(dt)^2 \\
&= \quad -\tfrac{1}{12}\mathbf{j}_i(dt)^3 - \tfrac{1}{24}\mathbf{s}_i(dt)^4
\end{aligned} \tag{6.23}$$

This proves the desired result:

$$\mathbf{r}_{i+1} = \mathbf{r}_i + \tfrac{1}{2}(\mathbf{v}_i + \mathbf{v}_{i+1})dt + \tfrac{1}{12}(\mathbf{a}_i - \mathbf{a}_{i+1})(dt)^2 \tag{6.24}$$

6.4 Implementing Hermite

Let us return to Alice, Bob, and Carol, who are about to implement the Hermite scheme. Since Bob was most eager to test out this new algorithm, he got his turn behind the computer.

Bob: Let's give this new Hermite scheme a stress test. I'll take `leapfrog2a.C`, the one with the off-set in initial velocity of 10^{-4}, call it `hermite1a.C` for now, and simply substitute the leapfrog equations 4.4, 4.5 by the Hermite equations 6.1, 6.2.

Carol: Why not call the code `hermite1.C`?

Bob: Even though the update seems almost trivial, I don't have the audacity to believe we'll get everything right the first time around. When it all works, we will rename the code to `hermite1.C`. So here is the new version:

Code 6.1 (hermite1a.C)

```
//-----------------------------------------------------------------------------
// hermite1a.C
//-----------------------------------------------------------------------------
#include  <iostream>
#include  <cmath>
using namespace std;

int main()
{
    int n = 3;
    double r[n][3], v[n][3], a[n][3], jk[n][3];
    const double m = 1;
    double dt, t_end;

    cerr << "Please provide a value for the time step" << endl;
    cin >> dt;
    cerr << "and for the duration of the run" << endl;
    cin >> t_end;

    const double pi = 2 * asin(1);
    for (int i = 0; i < n; i++){
        double phi = i * 2 * pi / 3;
        r[i][0] = cos (phi);
        r[i][1] = sin (phi);
        r[i][2] = 0;
    }

    for (int i = 0; i < n; i++)
        for (int k = 0; k < 3; k++)
            a[i][k] = jk[i][k] = 0.0;
    for (int i = 0; i < n; i++){
        for (int j = i+1; j < n; j++){
            double rji[3], vji[3];
            for (int k = 0; k < 3; k++){
                rji[k] = r[j][k] - r[i][k];
                vji[k] = v[j][k] - v[i][k];
            }
            double r2 = 0;
            for (int k = 0; k < 3; k++)
                r2 += rji[k] * rji[k];
            double r3 = r2 * sqrt(r2);
            double rv = 0;
            for (int k = 0; k < 3; k++)
                rv += rji[k] * vji[k];
            rv /= r2;
            for (int k = 0; k < 3; k++){
                a[i][k] += m * rji[k] / r3;
                a[j][k] -= m * rji[k] / r3;
                jk[i][k] += m * (vji[k] - 3 * rv * rji[k]) / r3;
                jk[j][k] -= m * (vji[k] - 3 * rv * rji[k]) / r3;
            }
        }
    }

    double v_abs = 1.0/sqrt(sqrt(3));
    for (int i = 0; i < n; i++){
        double phi = i * 2 * pi / 3;
        v[i][0] = - v_abs * sin (phi);
        v[i][1] = v_abs * cos (phi);
        v[i][2] = 0;
    }

    v[0][0] += 0.0001;

    double ekin = 0, epot = 0;
    for (int i = 0; i < n; i++){
        for (int j = i+1; j < n; j++){
            double rji[3];
            for (int k = 0; k < 3; k++)
                rji[k] = r[j][k] - r[i][k];
            double r2 = 0;
            for (int k = 0; k < 3; k++)
                r2 += rji[k] * rji[k];
            double r = sqrt(r2);
            epot -= m*m/r;
        }
        for (int k = 0; k < 3; k++)
            ekin += 0.5 * m * v[i][k] * v[i][k];
    }
    double e_in = ekin + epot;
    cerr << "Initial total energy E_in = " << e_in << endl;

    double dt_out = 0.01;
    double t_out = dt_out;

    double old_r[n][3], old_v[n][3], old_a[n][3], old_j[n][3];

    for (double t = 0; t < t_end; t += dt){
        for (int i = 0; i < n; i++){
            for (int k = 0; k < 3; k++){
                old_r[i][k] = r[i][k];
                old_v[i][k] = v[i][k];
                old_a[i][k] = a[i][k];
                old_j[i][k] = jk[i][k];
                r[i][k] += v[i][k]*dt + a[i][k]*dt*dt/2 + jk[i][k]*dt*dt*dt/6;
                v[i][k] += a[i][k]*dt + jk[i][k]*dt*dt/2;
            }
        }
        for (int i = 0; i < n; i++)
            for (int k = 0; k < 3; k++)
                a[i][k] = jk[i][k] = 0.0;
        for (int i = 0; i < n; i++){
            for (int j = i+1; j < n; j++){
                double rji[3], vji[3];
                for (int k = 0; k < 3; k++){
                    rji[k] = r[j][k] - r[i][k];
                    vji[k] = v[j][k] - v[i][k];
                }
                double r2 = 0;
                for (int k = 0; k < 3; k++)
                    r2 += rji[k] * rji[k];
                double r3 = r2 * sqrt(r2);
                double rv = 0;
                for (int k = 0; k < 3; k++)
                    rv += rji[k] * vji[k];
                rv /= r2;
                for (int k = 0; k < 3; k++){
                    a[i][k] += m * rji[k] / r3;
                    a[j][k] -= m * rji[k] / r3;
                    jk[i][k] += m * (vji[k] - 3 * rv * rji[k]) / r3;
                    jk[j][k] -= m * (vji[k] - 3 * rv * rji[k]) / r3;
                }
            }
        }
        for (int i = 0; i < n; i++){
            for (int k = 0; k < 3; k++){
                r[i][k] = old_r[i][k] + (old_v[i][k] + v[i][k])*dt/2
                                      + (old_a[i][k] - a[i][k])*dt*dt/12;
                v[i][k] = old_v[i][k] + (old_a[i][k] + a[i][k])*dt/2
                                      + (old_j[i][k] - jk[i][k])*dt*dt/12;
            }
        }
        if (t >= t_out){
            for (int i = 0; i < n; i++){
                for (int k = 0; k < 3; k++)
                    cout << r[i][k] << " ";
                for (int k = 0; k < 3; k++)
                    cout << v[i][k] << " ";
                cout << endl;
            }
            t_out += dt_out;
        }
    }

    epot = ekin = 0;
    for (int i = 0; i < n; i++){
        for (int j = i+1; j < n; j++){
            double rji[3];
            for (int k = 0; k < 3; k++)
                rji[k] = r[j][k] - r[i][k];
            double r2 = 0;
            for (int k = 0; k < 3; k++)
                r2 += rji[k] * rji[k];
            epot -= m*m/sqrt(r2);
        }
        for (int k = 0; k < 3; k++)
            ekin += 0.5 * m * v[i][k] * v[i][k];
    }
    double e_out = ekin + epot;

    cerr << "Final total energy E_out = " << e_out << endl;
    cerr << "absolute energy error: E_out - E_in = " << e_out - e_in << endl;
    cerr << "relative energy error: (E_out - E_in) / E_in = "
         << (e_out - e_in) / e_in << endl;
}
//-----------------------------------------------------------------------------
```

Alice: Ah, I see now that you were wise giving the code an preliminary name. Notice where you have added the jerk calculation. You replaced

```
            for (int k = 0; k < 3; k++){
                a[i][k] += m * rji[k] / r3;
                a[j][k] -= m * rji[k] / r3;
            }
```

by:

```
            for (int k = 0; k < 3; k++){
                a[i][k] += m * rji[k] / r3;
                a[j][k] -= m * rji[k] / r3;
                jk[i][k] += m * (vji[k] - 3 * rv * rji[k]) / r3;
                jk[j][k] -= m * (vji[k] - 3 * rv * rji[k]) / r3;
            }
```

in addition to some extra changes, such as introducing and calculating the variable `vji[]` for the relative velocities for particles i and j. All of this is fine, as far as I can see. The statements are correctly written and reflect the Hermite, but there is one problem. In the last two lines above you use the relative velocities $vij[]$, but at this point in the program they have not been assigned yet.

Bob: Ah, you are right. That happens just a few lines below, with this 'clever' trick of using the magnitude of the centrifugal acceleration to determine the magnitude of the velocities. Thanks! Okay, so I'll make another version, `hermite1b.C`, which has two initial loops, one for the accelerations, followed by the assignment of velocities, and then followed by the second loop which computes the jerks.

Carol: Maybe we can just run the program using our friend `/dev/null` to see how the energy behaves, before start making pictures. I'm really curious to see whether we have now reached fourth-order accuracy.

Bob: Easy to test:

```
|gravity> g++ -o hermite1b hermite1b.C
|gravity> hermite1b > /dev/null
Please provide a value for the time step
0.01
and for the duration of the run
100
Initial total energy E_in = -0.866025
Final total energy E_out = -0.975389
absolute energy error: E_out - E_in = -0.109364
relative energy error: (E_out - E_in) / E_in = 0.126282
|gravity> !!
hermite1b > /dev/null
Please provide a value for the time step
0.001
and for the duration of the run
100
Initial total energy E_in = -0.866025
Final total energy E_out = -0.866586
absolute energy error: E_out - E_in = -0.000560173
relative energy error: (E_out - E_in) / E_in = 0.000646832
|gravity> !!
hermite1b > /dev/null
Please provide a value for the time step
0.0001
and for the duration of the run
100
```

```
Initial total energy E_in = -0.866025
Final total energy E_out = -0.866026
absolute energy error: E_out - E_in = -5.52839e-07
relative energy error: (E_out - E_in) / E_in = 6.38364e-07
|gravity> !!
hermite1b > /dev/null
Please provide a value for the time step
0.00001
and for the duration of the run
100
Initial total energy E_in = -0.866025
Final total energy E_out = -0.866025
absolute energy error: E_out - E_in = -5.53711e-10
relative energy error: (E_out - E_in) / E_in = 6.39371e-10
|gravity>
```

Carol: What is this? We seem to have constructed a third-order algorithm!

Alice: Hmmm. It seems that way. Each refinement of a factor ten in step size gives a reduction of the error of a factor 1000. But this was not supposed to happen

Bob: This is strange indeed. Look, at the end, there they are, the real Hermite expressions, what can be wrong with such simple statements??

```
for (int i = 0; i < n; i++){
    for (int k = 0; k < 3; k++){
        r[i][k] = old_r[i][k] + (old_v[i][k] + v[i][k])*dt/2
                              + (old_a[i][k] - a[i][k])*dt*dt/12;
        v[i][k] = old_v[i][k] + (old_a[i][k] + a[i][k])*dt/2
                              + (old_j[i][k] - jk[i][k])*dt*dt/12;
    }
}
```

Carol: Yes, they are just what we derived before, as Eqs. 6.1, 6.2.

Alice: Could we have made a similar mistake as we did just before, when the expressions were absolutely correct, but the order of execution was wrong?

Bob: Well . . . hmmm . . . aha, of course, you are right! Look, that is exactly the problem. In the first line, where the position is updated, we indicate that we are using both the old velocity values and the new velocity values. However, the new ones have not been computed yet — that happens only on the next line! And the solution is obvious: fortunately, the second line updating the velocity does not have this problem, since it uses only accelerations and jerks, all of which have already been computed, both for the old and the new values. So we can simply swap the lines, and this should now work:

```
        for (int i = 0; i < n; i++){
            for (int k = 0; k < 3; k++){
                v[i][k] = old_v[i][k] + (old_a[i][k] + a[i][k])*dt/2
                                      + (old_j[i][k] - jk[i][k])*dt*dt/12;
                r[i][k] = old_r[i][k] + (old_v[i][k] + v[i][k])*dt/2
                                      + (old_a[i][k] - a[i][k])*dt*dt/12;
            }
        }
```

6.5 Testing the Hermite: Three Stars on a Circle

Bob: Now I'm ready to be brave and call the code `hermite1.C`. Rather than listing the whole output, let me use the `diff` program, which only lists those lines that are different.

```
|gravity> diff hermite1a.C hermite1.C
2c2
< // hermite1a.C
---
> // hermite1.C
30c30
<             a[i][k] = jk[i][k] = 0.0;
---
>             a[i][k] = 0.0;
33,34c33,34
<             double rji[3], vji[3];
<             for (int k = 0; k < 3; k++){
---
>             double rji[3];
>             for (int k = 0; k < 3; k++)
36,37d35
<                 vji[k] = v[j][k] - v[i][k];
<             }
42,45d39
<             double rv = 0;
<             for (int k = 0; k < 3; k++)
<                 rv += rji[k] * vji[k];
<             rv /= r2;
49,50d42
<                 jk[i][k] += m * (vji[k] - 3 * rv * rji[k]) / r3;
<                 jk[j][k] -= m * (vji[k] - 3 * rv * rji[k]) / r3;
64a57,81
>     for (int i = 0; i < n; i++)
>         for (int k = 0; k < 3; k++)
>             jk[i][k] = 0.0;
>     for (int i = 0; i < n; i++){
```

```
>         for (int j = i+1; j < n; j++){
>             double rji[3], vji[3];
>             for (int k = 0; k < 3; k++){
>                 rji[k] = r[j][k] - r[i][k];
>                 vji[k] = v[j][k] - v[i][k];
>             }
>             double r2 = 0;
>             for (int k = 0; k < 3; k++)
>                 r2 += rji[k] * rji[k];
>             double r3 = r2 * sqrt(r2);
>             double rv = 0;
>             for (int k = 0; k < 3; k++)
>                 rv += rji[k] * vji[k];
>             rv /= r2;
>             for (int k = 0; k < 3; k++){
>                 jk[i][k] += m * (vji[k] - 3 * rv * rji[k]) / r3;
>                 jk[j][k] -= m * (vji[k] - 3 * rv * rji[k]) / r3;
>             }
>         }
>     }
>
127,128d143
<                 r[i][k] = old_r[i][k] + (old_v[i][k] + v[i][k])*dt/2
<                                       + (old_a[i][k] - a[i][k])*dt*dt/12;
130a146,147
>                 r[i][k] = old_r[i][k] + (old_v[i][k] + v[i][k])*dt/2
>                                       + (old_a[i][k] - a[i][k])*dt*dt/12;
|gravity>
```

Carol: Yes, that is much clearer than listing the whole source code again. I can see how the main difference has been the move of the jerk calculation from the earlier part, where it was entangled with the acceleration calculation, to a separate block listed nearly at the end. At the very end, of course, there is the indication that we have swapped the order of the calculations of the positions and the velocities. The `diff` program arbitrarily took the velocity calculation as a identical standard in each file, with respect to which the shift in order of the position calculation was noted.

Alice: Soon we should start to clean up our codes, splitting them in functions at least, and probably also in different files. That way we don't have to rely on `diff` to read our own programs, since we can then take natural chunks at a time, in the form of functions. But first let's see what will happen to our figure 8 orbits.

Bob: Here are the results. Hmm, not a very good energy conservation, if you ask me.

```
|gravity> g++ -o hermite1 hermite1.C
|gravity> hermite1 > hermite1_0.01_100.out
Please provide a value for the time step
```

```
0.01
and for the duration of the run
100
Initial total energy E_in = -0.866025
Final total energy E_out = -1.08608
absolute energy error: E_out - E_in = -0.220054
relative energy error: (E_out - E_in) / E_in = 0.254096
|gravity>
```

Carol: Indeed. The leapfrog did far better at this stage. We got a relative energy error of less than 10^{-3}, and here we are faced with a relative energy error of a quarter!

Alice: That is not so strange, actually. A higher-order algorithm computes higher-order derivatives, and therefore can be extra sensitive to close encounters or other situations in which changes happen quite suddenly. Let's look at the picture, and then move on, refining our step size.

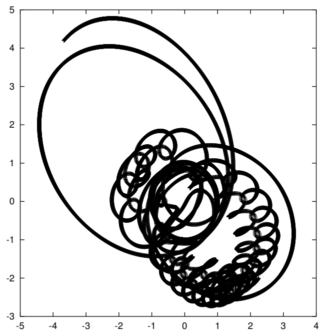

Figure 6.1: The first Hermite attempt to integrate the orbits of three stars starting off on a circle with an initial velocity perturbation of $dv_{init} = 0.0001$, time step $dt = 0.01$ and a total duration of $t_{end} = 100$

```
|gravity> hermite1 > hermite1_0.001_100.out
Please provide a value for the time step
0.001
and for the duration of the run
100
Initial total energy E_in = -0.866025
Final total energy E_out = -0.866026
absolute energy error: E_out - E_in = -9.19142e-07
relative energy error: (E_out - E_in) / E_in = 1.06133e-06
|gravity>
```

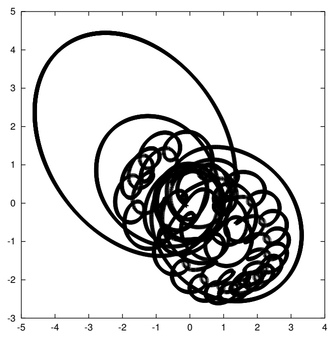

Figure 6.2: The second Hermite attempt to integrate the orbits of three stars starting off on a circle with an initial velocity perturbation of $dv_{init} = 0.0001$, time step $dt = 0.001$ and a total duration of $t_{end} = 100$

Carol: Ah, much better! Amazing.

Bob: An improvement of more than a factor 100,000 in accuracy!

Carol: And look at the picture. I cannot see any difference between Fig. 6.2 and the last picture in the series that we did with the leapfrog, Fig. 5.11!

Alice: Let's take another refinement step, to see whether we can determine the asymptotic behavior of the error growth. Clearly, our first attempt was not reliable, so we need at least a third try.

```
|gravity> hermite1 > hermite1_0.0001_100.out
Please provide a value for the time step
0.0001
and for the duration of the run
100
Initial total energy E_in = -0.866025
Final total energy E_out = -0.866025
absolute energy error: E_out - E_in = -8.49854e-12
relative energy error: (E_out - E_in) / E_in = 9.81326e-12
|gravity>
```

Carol: Okay, Bob, you can call this code `hermite1.C`, it seems to do its job.

Bob: But it does its job too well. Another five magnitudes of error improvement. Now it behaves like a fifth order code.

Alice: This seems to happen sometimes. A kth-order scheme is guaranteed only to have errors that grow not faster than kth order. However, it is possible for them to grow

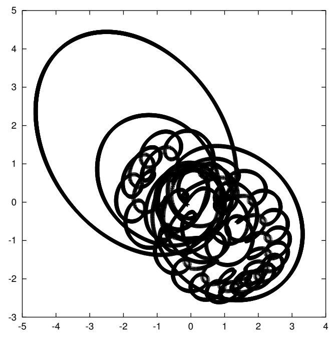

Figure 6.3: The third Hermite attempt to integrate the orbits of three stars starting off on a circle with an initial velocity perturbation of $dv_{init} = 0.0001$, time step $dt = 0.0001$ and a total duration of $t_{end} = 100$

less fast. For example, it could be that the particular orbits we are studying just happen to have some properties that lead to cancellations in some of the orders.

Carol: Is there any reason to believe that we do not have a generic system here?

Alice: Don't forget that we are studying a rather unstable system, in which it is quite likely that we will have at least one close encounter between two or three particles. So far, we have been sailing blindly, hoping that the step size we give the integrator is small enough to prevent near-collisions or other forms of strange behavior during close encounters. Soon we'll have to do better, though. It is not too hard to predict close encounters when they are about to happen, and to adapt the integration step size automatically. Only with such safety precautions does it make sense to rigorously measure the performance of the algorithm, whether it is the leapfrog or the Hermite. Without such precautions, even slight changes could show a different behavior. For example, cleaning up the code by initializing the velocities directly, instead of using the centrifugal trick, is likely to give slightly different initial conditions. I would not be surprised if such a change would give us yet another scaling of the errors, if we repeat the above measurements.

Bob: Okay, I guess we are ready to do quite a bit of cleaning up in our code. It is getting a bit too spaghetti-like for my taste already. Let's do that next time. We can improve readability and functionality at the same time, while we go along.

Carol: For now though, I feel that the Hermite should be our tool of choice. It sure seems to converge must faster. How about doing a timing test?

Bob: Good idea. Let's do it with the figure-8 orbits though. There at least we do not have any close encounters, so Alice's warnings may carry less urgency.

6.6 The Hermite Soars: Three Bodies on a Figure Eight

Bob: Here is the code. Rather than doing another `diff`, since this is a new problem I will list it in full.

Code 6.2 (hermite2.C)

```
//-----------------------------------------------------------------------------
// hermite2.C
//-----------------------------------------------------------------------------
#include  <iostream>
#include  <cmath>
using namespace std;

int main()
{
    int n = 3;
    double r[n][3], v[n][3], a[n][3], jk[n][3];
    const double m = 1;
    double dt, t_end;

    cerr << "Please provide a value for the time step" << endl;
    cin >> dt;
    cerr << "and for the duration of the run" << endl;
    cin >> t_end;

    r[0][0] = 0.9700436;
    r[0][1] = -0.24308753;
    r[0][2] = 0;
    v[0][0] = 0.466203685;
    v[0][1] = 0.43236573;
    v[0][2] = 0;

    r[1][0] = -r[0][0];
    r[1][1] = -r[0][1];
    r[1][2] = -r[0][2];
    v[1][0] = v[0][0];
    v[1][1] = v[0][1];
    v[1][2] = v[0][2];

    r[2][0] = 0;
    r[2][1] = 0;
    r[2][2] = 0;
    v[2][0] = -2 * v[0][0];
    v[2][1] = -2 * v[0][1];
    v[2][2] = -2 * v[0][2];

    for (int i = 0; i < n; i++)
        for (int k = 0; k < 3; k++)
            a[i][k] = jk[i][k] = 0.0;
    for (int i = 0; i < n; i++){
        for (int j = i+1; j < n; j++){
            double rji[3], vji[3];
            for (int k = 0; k < 3; k++){
                rji[k] = r[j][k] - r[i][k];
                vji[k] = v[j][k] - v[i][k];
            }
            double r2 = 0;
            for (int k = 0; k < 3; k++)
                r2 += rji[k] * rji[k];
            double r3 = r2 * sqrt(r2);
            double rv = 0;
            for (int k = 0; k < 3; k++)
                rv += rji[k] * vji[k];
            rv /= r2;
            for (int k = 0; k < 3; k++){
                a[i][k] += m * rji[k] / r3;
                a[j][k] -= m * rji[k] / r3;
                jk[i][k] += m * (vji[k] - 3 * rv * rji[k]) / r3;
                jk[j][k] -= m * (vji[k] - 3 * rv * rji[k]) / r3;
            }
        }
    }

    double ekin = 0, epot = 0;
    for (int i = 0; i < n; i++){
        for (int j = i+1; j < n; j++){
            double rji[3];
            for (int k = 0; k < 3; k++)
                rji[k] = r[j][k] - r[i][k];
            double r2 = 0;
            for (int k = 0; k < 3; k++)
                r2 += rji[k] * rji[k];
            double r = sqrt(r2);
            epot -= m*m/r;
        }
        for (int k = 0; k < 3; k++)
            ekin += 0.5 * m * v[i][k] * v[i][k];
    }
    double e_in = ekin + epot;
    cerr << "Initial total energy E_in = " << e_in << endl;
    double dt_out = 0.01;
    double t_out = dt_out;

    double old_r[n][3], old_v[n][3], old_a[n][3], old_j[n][3];

    for (double t = 0; t < t_end; t += dt){
        for (int i = 0; i < n; i++){
            for (int k = 0; k < 3; k++){
                old_r[i][k] = r[i][k];
                old_v[i][k] = v[i][k];
                old_a[i][k] = a[i][k];
                old_j[i][k] = jk[i][k];
                r[i][k] += v[i][k]*dt + a[i][k]*dt*dt/2 + jk[i][k]*dt*dt*dt/6;
                v[i][k] += a[i][k]*dt + jk[i][k]*dt*dt/2;
            }
        }
        for (int i = 0; i < n; i++)
            for (int k = 0; k < 3; k++)
                a[i][k] = jk[i][k] = 0.0;
        for (int i = 0; i < n; i++){
            for (int j = i+1; j < n; j++){
                double rji[3], vji[3];
                for (int k = 0; k < 3; k++){
                    rji[k] = r[j][k] - r[i][k];
                    vji[k] = v[j][k] - v[i][k];
                }
                double r2 = 0;
                for (int k = 0; k < 3; k++)
                    r2 += rji[k] * rji[k];
                double r3 = r2 * sqrt(r2);
                double rv = 0;
                for (int k = 0; k < 3; k++)
                    rv += rji[k] * vji[k];
                rv /= r2;
                for (int k = 0; k < 3; k++){
                    a[i][k] += m * rji[k] / r3;
                    a[j][k] -= m * rji[k] / r3;
                    jk[i][k] += m * (vji[k] - 3 * rv * rji[k]) / r3;
                    jk[j][k] -= m * (vji[k] - 3 * rv * rji[k]) / r3;
                }
            }
        }
        for (int i = 0; i < n; i++){
            for (int k = 0; k < 3; k++){
                v[i][k] = old_v[i][k] + (old_a[i][k] + a[i][k])*dt/2
```

```
                               + (old_j[i][k] - jk[i][k])*dt*dt/12;
              r[i][k] = old_r[i][k] + (old_v[i][k] + v[i][k])*dt/2
                               + (old_a[i][k] - a[i][k])*dt*dt/12;
          }
      }
      if (t >= t_out){
          for (int i = 0; i < n; i++){
              for (int k = 0; k < 3; k++)
                  cout << r[i][k] << " ";
              for (int k = 0; k < 3; k++)
                  cout << v[i][k] << " ";
              cout << endl;
          }
          t_out += dt_out;
      }
  }

  epot = ekin = 0;
  for (int i = 0; i < n; i++){
      for (int j = i+1; j < n; j++){
          double rji[3];
          for (int k = 0; k < 3; k++)
              rji[k] = r[j][k] - r[i][k];
          double r2 = 0;
          for (int k = 0; k < 3; k++)
              r2 += rji[k] * rji[k];
          epot -= m*m/sqrt(r2);
      }
      for (int k = 0; k < 3; k++)
          ekin += 0.5 * m * v[i][k] * v[i][k];
  }
  double e_out = ekin + epot;

  cerr << "Final total energy E_out = " << e_out << endl;
  cerr << "absolute energy error: E_out - E_in = " << e_out - e_in << endl;
  cerr << "relative energy error: (E_out - E_in) / E_in = "
       << (e_out - e_in) / e_in << endl;
}
//-----------------------------------------------------------------------------
```

Bob: And here are the results. Another fifth-order error behavior!

```
|gravity> g++ -o hermite2 hermite2.C
|gravity> hermite2 > hermite2_0.01_100.out
Please provide a value for the time step
```

```
0.01
and for the duration of the run
100
Initial total energy E_in = -1.28705
Final total energy E_out = -1.28705
absolute energy error: E_out - E_in = -3.81996e-07
relative energy error: (E_out - E_in) / E_in = 2.96801e-07
|gravity>

|gravity> hermite2 > hermite2_0.001_100.out
Please provide a value for the time step
0.001
and for the duration of the run
100
Initial total energy E_in = -1.28705
Final total energy E_out = -1.28705
absolute energy error: E_out - E_in = -4.00457e-12
relative energy error: (E_out - E_in) / E_in = 3.11144e-12
|gravity>
```

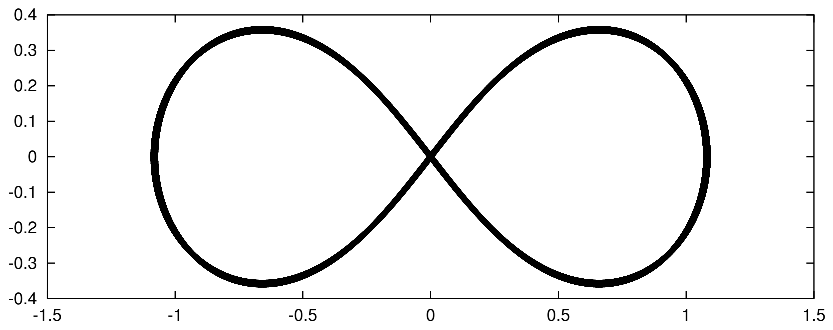

Figure 6.4: The second Hermite attempt to integrate the orbits of three stars starting off on a figure-8 orbit with an initial velocity perturbation of $dv_{init} = 0.0001$, time step $dt = 0.001$ and a total duration of $t_{end} = 100$

Alice: Well, all I can say is that the regularity of the orbit probably gives rise to cancellations. This is a well-known phenomenon for the leapfrog, for example, where sometimes the errors accumulate in the phases more than the energies of the particles. I suggest to come back to this question by the time we try our hand at larger N-body calculations starting from more random, less regular initial conditions.

Bob: Okay! And here are the timings Carol asked for:

```
|gravity> time leapfrog3a > /dev/null
Please provide a value for the time step
0.00001
and for the duration of the run
100
Initial total energy E_in = -1.287
Final total energy E_out = -1.287
absolute energy error: E_out - E_in = -1.24349e-11
relative energy error: (E_out - E_in) / E_in = 9.66196e-12
19.520u 0.100s 0:24.34 80.6% 0+0k 0+0io 168pf+0w
|gravity> time hermite2 > /dev/null
Please provide a value for the time step
0.001
and for the duration of the run
100
Initial total energy E_in = -1.28705
Final total energy E_out = -1.28705
absolute energy error: E_out - E_in = -4.00457e-12
relative energy error: (E_out - E_in) / E_in = 3.11144e-12
0.790u 0.010s 0:05.75 13.9% 0+0k 0+0io 169pf+0w
|gravity>
```

Carol: Not bad! The Hermite is more accurate, even for time steps that are a hundred times larger. Of course, each time step is more complicated than the leapfrog, so the time gain is less than a factor hundred, but still considerable, about a factor twenty-five.

Alice: For this particular case, and also without optimization switched on. Let's try compiling both programs with the $-O$ option of the g++ compiler, which should produce faster code.

```
|gravity> g++ -O -o leapfrog3a leapfrog3a.C
|gravity> g++ -O -o hermite2 hermite2.C
|gravity> time leapfrog3a > /dev/null
Please provide a value for the time step
0.00001
and for the duration of the run
100
Initial total energy E_in = -1.287
```

```
Final total energy E_out = -1.287
absolute energy error: E_out - E_in = -1.20237e-11
relative energy error: (E_out - E_in) / E_in = 9.34243e-12
10.690u 0.060s 0:13.94 77.1% 0+0k 0+0io 165pf+0w
|gravity> time hermite2 > /dev/null
Please provide a value for the time step
0.001
and for the duration of the run
100
Initial total energy E_in = -1.28705
Final total energy E_out = -1.28705
absolute energy error: E_out - E_in = -4.06408e-12
relative energy error: (E_out - E_in) / E_in = 3.15768e-12
0.610u 0.020s 0:02.98 21.1% 0+0k 0+0io 165pf+0w
|gravity>
```

Carol: Aha! Both programs ran faster, but the leapfrog more so. Now Hermite is ahead by ‘only’ a factor 18 or so, instead of 25.

Alice: Still, for this particular case only. And note that the energy errors are now slightly different from before, without the optimizer switched on. Although the optimized code should in principle give the same results as the non-optimized code if there would be no round-off errors, in practice round-off does creep in and propagate into the errors, especially when we are working at such high accuracies, where we are relatively few bits away from machine precisions. Fortunately, the effect does not seem to be too worrisome: we are talking about relative differences in energy error of only a few percent. But still this is something we clearly have to be aware of.

Bob: Okay, enough warnings and footnotes! Let’s call it a night. Next time we get together we’ll clean up the Hermite, and make it into a general working tool.

Carol: Sounds good! See you then.

Part III

Writing N-body Codes

Chapter 7

A General N-Body Hermite Code

7.1 A Wish List

In the first part of this book, we have offered a few quick-and-dirty programs that work fine for initial explorations. We were able to study stability aspects of various 2-body and 3-body systems, both with respect to numerical instabilities as well as physical instabilities. Many more situations could be easily explored, using these programs, and we hope that the readers will have tried their hand at some other configurations, starting from different initial conditions, for 2 or 3 or more bodies. Starting from `hermite2.C`, for example, it is easy to change the value of `n` in the first line, and to replace the explicit assignment of positions and velocities by other values for all st n particles.

However, it quickly becomes tedious to have to change the program, each time we want to integrate from a different starting position. Also, there are many other improvements that can be made to the code, as anyone with even modest programming experience will have noticed. In this second part of our book, we will begin to add more structure. Once we have structured our codes in a more modular and flexible way, we are in a position to carry out some real research projects with astrophysical implications. While simulating some real stars systems, we will soon realize that we will have to extend the complexity of our codes. While we will make the switch to variable time steps already in this chapter, we will later find a need to assigning individual time steps to each star. Also, we will introduce special coordinate patches for interacting group of stars. We will take these two steps in later volumes in our book series.

Here is a quick overview of a wish list for improving the structure of our computer codes. We will address some of them fully in this book, while we leave other items partly or completely to other volumes.

comments So far, we have not included any comments in our codes. In an attempt to keep the codes short and uncluttered, we wanted to show the flow of the statements directly, given that most codes could fit on one or two pages. But when the codes became longer, it was high time to put in comments, and we will do so from now on.

functions So far, we have written each program as a single function call to `main()`, without trying to split up the program into smaller pieces. For the first few codes, this was fine, and it kept everything light weight. But for the later codes, spanning a few pages, it would have been better to start dividing the functionality over separate functions. For example, in `leapfrog2.C`, we calculate the accelerations early on in the code, and then in the same way at the end of the main integration loop. Putting those statements in a function, and calling that function once before the loop and once in the loop makes the code both easier to understand and easier to debug. In addition, it is likely that we will use such a function in other codes as well.

structured I/O In our examples so far we have used only a very rudimentary form of I/O (input/output). We wrote our results in the form of a list of positions and velocities to the standard output stream, and we wrote some energy diagnostics to the error output stream. And we used input only interactively, to prompt the user to provide a few parameters. It is much better to define a unique N-body data format, which includes other variables besides $\mathbf{r}$ and $\mathbf{v}$, such as masses, time, and perhaps additional information. Once we write the results from an integration into a file, we can then read in that file again when we want to continue that run. This leads us to:

pipes The notion of pipes in Unix allows one to redirect the output of one program as the input of another program ('piping' the results from one program to the other, as it is called). For example, it would be nice to pipe the results of a program generating initial conditions directly into an integrator, and to pipe the results of the latter into an analysis program.

command line arguments Typing in parameters by hand, after being prompted by a program, gets tedious very soon. It is also inflexible, in that it doesn't fit very well if we want to write shell scripts to run a bunch of programs in laboratory fashion. A better way to pass parameters to a program is by providing arguments directly on the command line. Unix has a default protocol for doing this, and we adapt that usage in the following programs.

using a Makefile When the number of files in our working directory grows, we may lose track of which program needs to be recompiled. To automate this process, we introduce the notion of a Makefile below. The real strength of Makefiles will become apparent only later, but already at this stage is can be helpful.

test facilities Soon our codes will reach a level of complexity where it becomes difficult to convince ourselves that the code is really doing the right thing everywhere, giving the correct answers in the end. The best approach is to develop a slew of standard tests, together with a form of scaffolding that enables these tests to be run automatically, each time we make changes to our code.

using the C++ STL So far, we have used only the bare bones part of the C++ language. In some of the programs below we will introduce a convenient extension to the C++ core language, in the form of the Standard Template Library (STL), which is included in every modern C++ compiler. It gives us a quick and well-debugged and often (but not always) efficient way to get standard tasks done quickly.

C++ classes The central feature of C++, as an object-oriented language, is the use of classes, ways to encapsulate objects. Since we need to build up some considerable experience with N-body codes in order to know what type of objects to construct, we postpone the introduction of classes until later in this book.

error checking Any robust code will do lots of error checking. Ideally, every function should make sure that the data it gets fed are of a form that is valid for the operations that it wants to do on them. Since error checking, and even better, error handling (following up an error in the proper way, once it occurs) complicates a code considerably, we postpone this until somewhat later.

more flexible data format As we discussed earlier, it would be nice to give each star considerable autonomy by building in some form of artificial intelligence to let stars decide when to do what and how to report on it. For this to work, a minimal requirement is a flexible way for reporting unforeseen events, and this requires considerable flexibility in the data formats used. We will later give an example later of 'stories' that are attached to each star's data.

more flexible command line options The Unix-based one-letter-only style of command line options that we introduce below is far from ideal. Later we will provide a more flexible way of handling arguments on the command line.

more detailed help facility For now, asking for help will result only in a list of command line options, together with a brief indication of what they do. It would be better to provide several levels of help, allowing the user to get more detailed information when needed. This leads to:

documentation At a minimum, a good software environment should have a manual page for each program. Even better, groups of programs should be described as to their purposes and the way they can work together. This leads to:

construction of a software environment At some point, when we have written various integrators and a number of programs to generate initial conditions and to analyse data, it will become too much of a clutter to keep everything in a single directory. We will need to provide more structure for the way in which we store our tools, and the way we intend them to be used. This leads to:

multiple files We mentioned under 'functions' above the desirability to recycle code by creating functions that can be used for different applications. If we compile such a function in a separate file, it will be easier to link it to other codes that use it. This leads to:

libraries An extension of the previous concept, in which a group of related functions is compiled into a library, which can then be linked to other codes that use some of the functions collected there. Having various libraries and many files requires significant bookkeeping to be done to guarantee that everything is consistent and up to date. This leads to:

version control For a software environment under development (and every healthy environment is constantly under development!), it is useful to be able to reconstruct older versions, and to keep track of the latest developments. CVS, short for Concurrent Versions System, is a useful package for doing all this. It also allows several people to write code asynchronously within the same software environment, since it will flag any collisions stemming from potential multiple edits. More recent alternatives are available as well, such as SVN, short for Subversion, which allows more flexible ways to rename files and whole directory structuress.

autoconfig A related useful facility is 'autoconfig', which allows a user to install a software environment on an (almost) arbitrary platform, without any trouble. As the name implies, this program does an automated check to see how your particular system is configured, and it then sets up your copy of the software environment in such a way that it fits your environment.

parallelization With most modern computers distributing the running of a time-intensive program over several processors, it is important to give guidance to the compiler as to how to break up a large program into chunks that can be executed safely in parallel. Later we will discuss how to modify our N-body codes to make use of both small-scale and large-scale parallelism.

special-purpose hardware Another way to greatly gain in speed is to use dedicated hardware, constructed specifically for the problem at hand. For the gravitational N-body problem, the GRAPE hardware developed at the University of Tokyo, provide such a function. We will discuss issues connected with the use of one or more GRAPE boards.

a dedicated plotting package The time will come that the use of a canned plotting package, like gnuplot, is just too inflexible for our particular needs in analyzing the results of N-body runs. At some stage we will introduce a version of a plotting package, dedicated to the analysis of stellar dynamics simulations of dense stellar systems.

a scripting language Around that time, if not earlier, the need will be felt for a scripting language that is more powerful than the simple use of shell scripts.

archiving Finally, when we have an efficient and detailed software environment for doing cutting-edge scientific research, we will want to perform large-scale simulations. When some runs will take weeks or months to run on the world's fastest computers, it is important to have ways to store the massive amounts of data in such a way that we can later query those data in flexible and efficient ways. Archiving and data retrieval as well as more fancy operations like data mining then become serious issues.

7.2 A Standard N-Body Snapshot Format

Let us take a look at the last code from Part 1, `hermite2.C`. There we had the initial conditions for the three stars on a figure-8 orbit included at the beginning of the code. Clearly, this is not a very flexible approach. It would be much better to write one N-body code which can start from an arbitrary N-body snapshot, for an arbitrary number of particles N.

Before we adapt our code, we first have to agree upon a unique format for a realization of an N-body system. If we stick with the same standard format, we can build a suite of programs that are compatible in that they all can read and write N-body snapshots in the same way. Some programs can set up initial conditions, others integrate the orbits, and yet others can analyze the resulting data.

We will start with the following simple format, which is good enough for our current purposes. Later we will introduce extensions when needed. On the first line we will print one number, an integer, N, the number of particles. On the second line we will print the time t. These two lines will be followed by N lines, one for each particle. Each of those lines will contain seven numbers: the mass of the particle, followed by the three Cartesian components of the position vector, followed similarly by the three components of the velocity vector.

For example, to store the initial conditions for the figure-8 orbits in a file, we need to write the following five lines:

```
3
0
1 0.9700436 -0.24308753 0 0.466203685 0.43236573 0
1 -0.9700436 0.24308753 0 0.466203685 0.43236573 0
1 0 0 0 -0.93240737 -0.86473146 0
```

The simplest way to read an N-body snapshot is to obtain the data from the standard input stream `cin`. For example, when we put the above five lines in a file `figure8.in`, we can then run our new Hermite version as `hermite3 < figure8.in`.

Let us see how many changes we have to make to `hermite2.C` in order to make it compatible with our new standard data format. The first problem is that we can no longer use this stream to obtain the parameters for the integrator. In `hermite2.C` we allowed the user to provide those parameters in the following lines:

```
    cerr << "Please provide a value for the time step" << endl;
    cin >> dt;
    cerr << "and for the duration of the run" << endl;
    cin >> t_end;
```

An alternative is to write the parameters as command line arguments. C++ allows us to add the following optional arguments to `main()`:

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

When we run the program, `argc`, the argument counter, will contain the number of arguments specified, while `argv[]`, the vector of arguments, will contain the command line options. For example, if we want to run `hermite3` with a time step of `dt = 0.001` for a total duration of `t_end = 10`, we could communicate this to the program by invoking it as:

```
hermite3 0.001 10 < figure8.in
```

As soon as our program starts its execution, entering `main()`, the argument counter will be set to 3, since by convention the program name is the first argument. The argument vector will contain the following values:

```
argv[0] = "hermite3"
argv[1] = "0.001"
argv[2] = "10"
```

At this point, the parameters are available for the program, but they are still written in the form of character strings. Fortunately, there are convenient library function that will convert such a string into a number. They are called st atoi for a conversion of an ASCII string to an integer, and st atof for a conversion of an ASCII string to an floating point number. We can get access to those functions by adding the line `#include <cstdlib>` to the beginning of the file.

Below is the code listing. The changes from `hermite2.C` are straightforward: as soon as we have determined the values of the command line arguments, we know the value of N and we can allocate space for the N particles. Since N is not known at compile time, we have to explicitly allocate space for the arrays, using the `new` operator. For two-dimensional arrays, this requires the length of the second array dimension to be known at compile time; fortunately that is no problem for us, since its value is known to be 3, the number of spatial dimensions in our world. Note the placement of the parentheses around `* r`, *etc.*, necessary because the array brackets `[]` have a higher precendence as operators than the `*` operator. Omitting the parentheses, as in `double *r[3]`, would be interpreted by the compiler as `double *(r[3])`, which is not what we want. By the way, strictly speaking we should deallocate the dynamically created arrays at the end of our `main()` program. However, since at that point the program exits and releases all its memory, we omit that here. In general it is good form to use a `delete` statement for every `new` statement used, in order to free the memory allocated. We will make this a habit starting in the next chapter.

The only other difference with `hermite2.C` is that we have to replace the value `m` for the mass which was shared for all particles by `m[i]` for the individual mass of particle `i`.

Code 7.1 (hermite3.C)

```
//-----------------------------------------------------------------------------
// hermite3.C
//-----------------------------------------------------------------------------
#include  <iostream>
#include  <cmath>
#include  <cstdlib>
using namespace std;

int main(int argc, char *argv[])
{
    double dt = atof(argv[1]);
    double t_end = atof(argv[2]);

    int n;
    cin >> n;

    double t;
    cin >> t;

    double * m = new double[n];
    double (* r)[3] = new double[n][3];
    double (* v)[3] = new double[n][3];
    double (* a)[3] = new double[n][3];
    double (* jk)[3] = new double[n][3];

    for (int i = 0; i < n ; i++){
        cin >> m[i];
        for (int k = 0; k < 3; k++)
            cin >> r[i][k];
        for (int k = 0; k < 3; k++)
            cin >> v[i][k];
    }

    for (int i = 0; i < n; i++)
        for (int k = 0; k < 3; k++)
            a[i][k] = jk[i][k] = 0.0;
    for (int i = 0; i < n; i++){
        for (int j = i+1; j < n; j++){
            double rji[3], vji[3];
            for (int k = 0; k < 3; k++){
                rji[k] = r[j][k] - r[i][k];
                vji[k] = v[j][k] - v[i][k];
            }
            double r2 = 0;
            for (int k = 0; k < 3; k++)
                r2 += rji[k] * rji[k];
            double r3 = r2 * sqrt(r2);
            double rv = 0;
            for (int k = 0; k < 3; k++)
                rv += rji[k] * vji[k];
            rv /= r2;
            for (int k = 0; k < 3; k++){
                a[i][k] += m[j] * rji[k] / r3;
                a[j][k] -= m[i] * rji[k] / r3;
                jk[i][k] += m[j] * (vji[k] - 3 * rv * rji[k]) / r3;
                jk[j][k] -= m[i] * (vji[k] - 3 * rv * rji[k]) / r3;
            }
        }
    }

    double ekin = 0, epot = 0;
    for (int i = 0; i < n; i++){
        for (int j = i+1; j < n; j++){
            double rji[3];
            for (int k = 0; k < 3; k++)
                rji[k] = r[j][k] - r[i][k];
            double r2 = 0;
            for (int k = 0; k < 3; k++)
                r2 += rji[k] * rji[k];
            double r = sqrt(r2);
            epot -= m[i]*m[j]/r;
        }
        for (int k = 0; k < 3; k++)
            ekin += 0.5 * m[i] * v[i][k] * v[i][k];
    }
    double e_in = ekin + epot;
    cerr << "Initial total energy E_in = " << e_in << endl;

    double dt_out = 0.01;
    double t_out = dt_out;

    double old_r[n][3], old_v[n][3], old_a[n][3], old_j[n][3];

    for (double t = 0; t < t_end; t += dt){
        for (int i = 0; i < n; i++){
            for (int k = 0; k < 3; k++){
                old_r[i][k] = r[i][k];
                old_v[i][k] = v[i][k];
                old_a[i][k] = a[i][k];
                old_j[i][k] = jk[i][k];
                r[i][k] += v[i][k]*dt + a[i][k]*dt*dt/2 + jk[i][k]*dt*dt*dt/6;
                v[i][k] += a[i][k]*dt + jk[i][k]*dt*dt/2;
            }
        }
        for (int i = 0; i < n; i++)
            for (int k = 0; k < 3; k++)
                a[i][k] = jk[i][k] = 0.0;
        for (int i = 0; i < n; i++){
            for (int j = i+1; j < n; j++){
                double rji[3], vji[3];
                for (int k = 0; k < 3; k++){
                    rji[k] = r[j][k] - r[i][k];
                    vji[k] = v[j][k] - v[i][k];
                }
                double r2 = 0;
                for (int k = 0; k < 3; k++)
                    r2 += rji[k] * rji[k];
                double r3 = r2 * sqrt(r2);
                double rv = 0;
                for (int k = 0; k < 3; k++)
                    rv += rji[k] * vji[k];
                rv /= r2;
                for (int k = 0; k < 3; k++){
                    a[i][k] += m[j] * rji[k] / r3;
                    a[j][k] -= m[i] * rji[k] / r3;
                    jk[i][k] += m[j] * (vji[k] - 3 * rv * rji[k]) / r3;
                    jk[j][k] -= m[i] * (vji[k] - 3 * rv * rji[k]) / r3;
                }
            }
        }
        for (int i = 0; i < n; i++){
            for (int k = 0; k < 3; k++){
                v[i][k] = old_v[i][k] + (old_a[i][k] + a[i][k])*dt/2
                                      + (old_j[i][k] - jk[i][k])*dt*dt/12;
                r[i][k] = old_r[i][k] + (old_v[i][k] + v[i][k])*dt/2
                                      + (old_a[i][k] - a[i][k])*dt*dt/12;
            }
        }
        if (t >= t_out){
            for (int i = 0; i < n; i++){
                for (int k = 0; k < 3; k++)
                    cout << r[i][k] << " ";
                for (int k = 0; k < 3; k++)
```

```
                    cout << v[i][k] << " ";
                cout << endl;
            }
            t_out += dt_out;
        }
    }

    epot = ekin = 0;
    for (int i = 0; i < n; i++){
        for (int j = i+1; j < n; j++){
            double rji[3];
            for (int k = 0; k < 3; k++)
                rji[k] = r[j][k] - r[i][k];
            double r2 = 0;
            for (int k = 0; k < 3; k++)
                r2 += rji[k] * rji[k];
            epot -= m[i]*m[j]/sqrt(r2);
        }
        for (int k = 0; k < 3; k++)
            ekin += 0.5 * m[i] * v[i][k] * v[i][k];
    }
    double e_out = ekin + epot;

    cerr << "Final total energy E_out = " << e_out << endl;
    cerr << "absolute energy error: E_out - E_in = " << e_out - e_in << endl;
    cerr << "relative energy error: (E_out - E_in) / E_in = "
         << (e_out - e_in) / e_in << endl;
}
//-----------------------------------------------------------------------------
```

7.3 A More Modular Approach: Functions

Now that we have a general-purpose N-body integrator, it is high time to rewrite our code in more modular fashion. The first step is to offload much of the work done in `main()` into a few functions. Obvious candidates are the loop where we calculate the accelerations and jerks of all particles, and the loop where we determine the total energy of the system. Both of these occur twice in `hermite3.C`, so by bundling each of these loops in a separate function we can shorten the length of the program. More importantly, using functions makes the program easier to understand, and therefore also easier to extend.

Here is the first part of our new version of the Hermite code, `hermite4.C`, which contains the first function that calculates the accelerations and jerks:

Code 7.2 (hermite4.C: first part)

```
//-----------------------------------------------------------------------------
// hermite4.C
//-----------------------------------------------------------------------------
#include  <iostream>
#include  <cmath>
#include  <cstdlib>
using namespace std;

void acc_and_jerk(double m[], double r[][3], double v[][3], double a[][3],
                  double jk[][3], int n)
{
    for (int i = 0; i < n; i++)
        for (int k = 0; k < 3; k++)
            a[i][k] = jk[i][k] = 0.0;
    for (int i = 0; i < n; i++){
        for (int j = i+1; j < n; j++){
            double rji[3], vji[3];
            for (int k = 0; k < 3; k++){
                rji[k] = r[j][k] - r[i][k];
                vji[k] = v[j][k] - v[i][k];
            }
            double r2 = 0;
            for (int k = 0; k < 3; k++)
                r2 += rji[k] * rji[k];
            double r3 = r2 * sqrt(r2);
            double rv = 0;
            for (int k = 0; k < 3; k++)
                rv += rji[k] * vji[k];
            rv /= r2;
            for (int k = 0; k < 3; k++){
                a[i][k] += m[j] * rji[k] / r3;
                a[j][k] -= m[i] * rji[k] / r3;
                jk[i][k] += m[j] * (vji[k] - 3 * rv * rji[k]) / r3;
                jk[j][k] -= m[i] * (vji[k] - 3 * rv * rji[k]) / r3;
            }
        }
    }
}
. . . .
//-----------------------------------------------------------------------------
```

Note that we can pass the arrays for masses, positions, *etc.*, without having to specify the total length of these arrays. This is a good thing, since at compile time we do not yet know the N value(s) with which the program will be run. In a two-dimensional array, however, we do have to specify the length of the second dimension, in order to allow the compiler to make the correct pointer calculations to map the memory. Fortunately, we will almost always work in three dimensions, so that number can be specified in, e. g. `r[][3]`.

The type of the function `acc_and_jerk` is `void`, which means that the function does not provide a return value. All the work done is stored in the arrays that contain the accelerations and jerks. Since in C++ array variables are effectively pointers, any change made locally in the function will directly affect the arrays with which the function is called further on in the program where we will see:

```
acc_and_jerk(m, r, v, a, jk, n);
```

The first five arguments will be directly visible in the body of the function, and all changes made there will affect the five variables used to call the function. The sixth and last variable `n` is passed by value. This means that changes in `n` made in `acc_and_jerk()` will not affect the original value of `n` in the function `main()` that calls `acc_and_jerk()`. In our particular case, there is no need to change `n`. If there would be, we could have passed `n` by reference, rather than by value. We will see an instance of passing by reference in the next section.

Here is the second function which calculates the total energy of the system:

Code 7.3 (hermite4.C: second part)

```
//-----------------------------------------------------------------------------
. . . .
double energy(double m[], double r[][3], double v[][3], int no)
{
    double ekin = 0, epot = 0;
    for (int i = 0; i < n; i++){
        for (int j = i+1; j < n; j++){
            double rji[3];
            for (int k = 0; k < 3; k++)
                rji[k] = r[j][k] - r[i][k];
            double r2 = 0;
            for (int k = 0; k < 3; k++)
                r2 += rji[k] * rji[k];
            epot -= m[i]*m[j]/sqrt(r2);
        }
        for (int k = 0; k < 3; k++)
            ekin += 0.5 * m[i] * v[i][k] * v[i][k];
    }
    return ekin + epot;
}
```

```
. . . .
//-----------------------------------------------------------------------------
```

The return type is `double`, which means that calling the function gives an immediate handle on the returned value, the total energy. When we call this function with

```
double e_in = energy(m, r, v);
```

the result of the calculation is directly assigned to the variable `e_in`.

Why do we add so many arguments to our functions, six and four, respectively? An alternative would have been to declare the five main data structures, from masses to jerks, as global variables at the top of our program, as follows.

Code 7.4 (global variables)

```
//-----------------------------------------------------------------------------
#include  <iostream>
#include  <cmath>
#include  <cstdlib>
using namespace std;

double * m = new double[n];
double (* r)[NDIM] = new double[n][NDIM];
double (* v)[NDIM] = new double[n][NDIM];
double (* a)[NDIM] = new double[n][NDIM];
double (* jk)[NDIM] = new double[n][NDIM];

void acc_and_jerk()
{
. . . .
}

double energy()
{
. . . .
}

. . . .
//-----------------------------------------------------------------------------
```

The use of global variables is considered bad form, for good reasons. The whole point of our exercise of splitting our program into functions is to isolate functionality. This will make it easier to understand and debug the program, and to modify or extend it later. For now the program is small enough that it is easy to keep track of the global variables. When we add more and more features, chances are that we lose track of exactly which global variables are floating around in the program. Also, it will be easy to get name conflicts, since individual functions are likely to use similar names. Finally, there is a very practical reason to stay with local variables only: some time soon we may well want to manipulate more than one N-body system, to compare and analyse them. At that time it will be impossible to use a single global set of variables to store the data. In anticipation of such complications, it is more prudent to stick to the use of local variables right from the beginning.

Here is the remainder of the program, containing the `main()` function of hermite4.C, which is now shortened to half the length it was in hermite3.C:

Code 7.5 (hermite4.C: third part)

```
//-----------------------------------------------------------------------------
. . . .
int main(int argc, char *argv[])
{
    double dt = atof(argv[1]);
    double t_end = atof(argv[2]);

    int n;
    cin >> n;

    double t;
    cin >> t;

    double * m = new double[n];
    double (* r)[3] = new double[n][3];
    double (* v)[3] = new double[n][3];
    double (* a)[3] = new double[n][3];
    double (* jk)[3] = new double[n][3];

    for (int i = 0; i < n ; i++){
        cin >> m[i];
        for (int k = 0; k < 3; k++)
            cin >> r[i][k];
        for (int k = 0; k < 3; k++)
            cin >> v[i][k];
    }

    acc_and_jerk(m, r, v, a, jk, n);
    double e_in = energy(m, r, v, n);
    cerr << "Initial total energy E_in = " << e_in << endl;

    double dt_out = 0.01;
    double t_out = dt_out;

    double old_r[n][3], old_v[n][3], old_a[n][3], old_j[n][3];

    for (double t = 0; t < t_end; t += dt){
        for (int i = 0; i < n; i++){
            for (int k = 0; k < 3; k++){
                old_r[i][k] = r[i][k];
                old_v[i][k] = v[i][k];
                old_a[i][k] = a[i][k];
                old_j[i][k] = jk[i][k];
                r[i][k] += v[i][k]*dt + a[i][k]*dt*dt/2 + jk[i][k]*dt*dt*dt/6;
                v[i][k] += a[i][k]*dt + jk[i][k]*dt*dt/2;
            }
        }
        acc_and_jerk(m, r, v, a, jk, n);
        for (int i = 0; i < n; i++){
            for (int k = 0; k < 3; k++){
                v[i][k] = old_v[i][k] + (old_a[i][k] + a[i][k])*dt/2
                                      + (old_j[i][k] - jk[i][k])*dt*dt/12;
                r[i][k] = old_r[i][k] + (old_v[i][k] + v[i][k])*dt/2
                                      + (old_a[i][k] - a[i][k])*dt*dt/12;
            }
        }
        if (t >= t_out){
            cout << n << endl;
            cout << t << endl;
            for (int i = 0; i < n; i++){
                cout << m[i] << " ";
                for (int k = 0; k < 3; k++)
                    cout << r[i][k] << " ";
                for (int k = 0; k < 3; k++)
                    cout << v[i][k] << " ";
                cout << endl;
            }
            t_out += dt_out;
        }
    }

    double e_out = energy(m, r, v, n);

    cerr << "Final total energy E_out = " << e_out << endl;
    cerr << "absolute energy error: E_out - E_in = " << e_out - e_in << endl;
    cerr << "relative energy error: (E_out - E_in) / E_in = "
         << (e_out - e_in) / e_in << endl;
}
//-----------------------------------------------------------------------------
```

Exercise 7.1 (Pythagorean problem: constant time steps)
The Pythagorean problem was introduced in the late sixties, as a severe test for the N-body codes and the computer hardware available at that time, The name stems from the way the initial conditions are set up for the three particles involved: the particles are initially positioned in a right triangle, with masses and positions chosen such that the center of mass lies in the origin of the coordinate system. The velocities are all chosen to be zero. The code `mk_pyth.C` *below generates the initial conditions.*

Experiment with the `hermite4.C` *code, to see how small the step size has to be, in order to get reasonable results. For example, starting with* `dt=0.1`*, while seemingly a small number, is clearly too large, given the large errors reported, below. How small is small enough, for* `dt`*, if we want to follow the system for 10 time units?*

```
|gravity> mk_pyth | hermite4 0.1 10
Initial total energy E_in = -12.8167
3
0.1
3 0.998076 2.99118 0 -0.0192721 -0.088297 0
4 -1.98742 -0.998076 0 0.126071 0.0192722 0
5 0.991091 -0.996247 0 -0.0892936 0.0375604 0
3
0.2
3 0.995662 2.98013 0 -0.0290448 -0.132818 0
4 -1.97162 -0.995662 0 0.190175 0.029046 0
5 0.979897 -0.991547 0 -0.134713 0.0564539 0
3
 . . . .
3
10
3 -1.74295 -7.58878 0 -0.307547 -1.1837 0
4 -353.379 265.193 0 -42.5655 31.6683 0
5 283.749 -207.601 0 34.2369 -24.6244 0
Final total energy E_out = 10077.9
absolute energy error: E_out - E_in = 10090.7
relative energy error: (E_out - E_in) / E_in = -787.31
|gravity>
```

Here is the code listing.

Code 7.6 (mk_pyth.C)

```
//-----------------------------------------------------------------------------
// mk_pyth.C:  prints initial conditions for the Pythagorean problem.
//             ref.: Szebehely, V. and Peters, C.F., 1967, Astron. J. 72, 876.
//-----------------------------------------------------------------------------
#include  <iostream>
using namespace std;

typedef double  real;                       // "real" as a general name for the
                                            // standard floating-point data type
#define  NDIM     3                         // number of spatial dimensions

int main()
{
    const int n = 3;            // number of particles
    const real t = 0;           // time
    real r[n][NDIM];            // r[i][k] : position component k for particle i
    real v[n][NDIM];            // v[i][k] : velocity component k for particle i
    real m[n];                  // m[i]  : mass for particle i

    m[0] = 3;
    m[1] = 4;
    m[2] = 5;

    r[0][0] = 1;
    r[0][1] = 3;
    r[0][2] = 0;

    r[1][0] = -2;
    r[1][1] = -1;
    r[1][2] = 0;

    r[2][0] = 1;
    r[2][1] = -1;
    r[2][2] = 0;

    for (int i = 0; i < n ; i++)
        for (int k = 0; k < NDIM; k++)
    v[i][k] = 0;

    cout.precision(16);

    cout << n << endl;                       // first output line:  N
    cout << t << endl;                       // second output line:  time
    for (int i = 0; i < n; i++){
        cout << m[i];                        // each next output line:
        for (int k = 0; k < NDIM; k++)       //   m r_x r_y r_z v_x v_y v_z
            cout << ' ' << r[i][k];
        for (int k = 0; k < NDIM; k++)
            cout << ' ' << v[i][k];
        cout << endl;
    }
}
//-----------------------------------------------------------------------------
```

7.4 Variable Time Steps: a Simple Collision Criterium

There is still one major shortcoming in `hermite4.C`, namely the use of a constant time step, which is determined at the beginning of the program, and then used unchanged during the whole orbit integration. For a specific application, where we want to follow the orbits of a small number of particles for a relatively short time, this limitation may not seem so terrible. We can simply rerun the orbit calculations for shorter and shorter time steps, until we see that the orbits no longer change significantly. In fact, this is what we have done with every application in part I. In practice, however, this way of using an integration code is very unsatisfactory.

It would be far better to use variable time steps. If none of the particles is particularly close to any of its neighbors, there is no reason to waste computer time on letting the whole system crawl forwards with tiny time steps, just to prepare for a future time where two or more particles do swing by each other in a short time interval. On the other hand, during such brief periods of fast encounter activity, even a very small initial time step may prove to be not small enough: there is no way of knowing in advance how much we have to slow down in order to guarantee good performance of our integrator. It would be far better to leave the decision concerning the size of the time step up to the computer. An autonomous choice of time step allows us to let the computer time migrate to the 'hot spots' in time where sudden fast changes demand higher time resolution.

To add a simple form of this type of cleverness is surprisingly simple. Take each pair of particles i and j, and determine the magnitudes of their separation in space, $|r_{ij}|$ as well as the magnitude of the difference between their velocities (their separation in velocity space), $|v_{ij}|$. The time scale $t_{ij} = |r_{ij}|/|v_{ij}|$ gives an estimate of the minimum time it will take for these two particle to collide. If their relative velocity vector $\mathbf{v}_{ij}$ is not closely aligned with their relative position vector $\mathbf{r}_{ij}$ and pointing in the same direction, the magnitudes $|r_{ij}|$ and $|v_{ij}|$ may not become that much smaller during the next time interval t_{ij}, but certainly the direction of both $\mathbf{r}_{ij}$ and $\mathbf{v}_{ij}$ will change significantly. In either case, it is important that the integration time step during this period is chosen to be significantly smaller than t_{ij}, so that the change per time step in relative position and velocity, both with respect to magnitude and with respect to direction, remain small. That way, our fourth-order integrator will be able to operate in a regime where further shrinking of the time step is guaranteed to give a shrinking of errors that is proportional to the fourth power of the time step size.

We can implement this idea by taking the minimal value of t_{ij}, with respect to all i and j combinations. It is not expensive to compute this minimum, since much of the work is already done anyway in the inner loop that computes accelerations and jerks. Computing this minimum in the function `acc_and_jerk()` as a side effect, we can pass its value back to the *main()* program. The code `hermite5.C` does this by adding one extra variable to the list of arguments of `acc_and_jerk()`:

Code 7.7 (hermite5.C: acc_and_jerk)

```
//-----------------------------------------------------------------------------
void acc_and_jerk(double m[], double r[][3], double v[][3], double a[][3],
                  double jk[][3], int n, double & coll_time)
{
    double coll_time_sq = 1e300;;
    double coll_est_sq;

    for (int i = 0; i < n; i++)
        for (int k = 0; k < 3; k++)
            a[i][k] = jk[i][k] = 0.0;
    for (int i = 0; i < n; i++){
        for (int j = i+1; j < n; j++){
            double rji[3], vji[3];
            for (int k = 0; k < 3; k++){
                rji[k] = r[j][k] - r[i][k];
                vji[k] = v[j][k] - v[i][k];
            }
            double r2 = 0;
            double v2 = 0;
            for (int k = 0; k < 3; k++){
                r2 += rji[k] * rji[k];
                v2 += vji[k] * vji[k];
            }
            coll_est_sq = r2/v2;
            if (coll_time_sq > coll_est_sq)
                coll_time_sq = coll_est_sq;
            double r3 = r2 * sqrt(r2);
            double rv = 0;
            for (int k = 0; k < 3; k++)
                rv += rji[k] * vji[k];
            rv /= r2;
            for (int k = 0; k < 3; k++){
                a[i][k] += m[j] * rji[k] / r3;
                a[j][k] -= m[i] * rji[k] / r3;
                jk[i][k] += m[j] * (vji[k] - 3 * rv * rji[k]) / r3;
                jk[j][k] -= m[i] * (vji[k] - 3 * rv * rji[k]) / r3;
            }
        }
    }
    coll_time = sqrt(coll_time_sq);
}
//-----------------------------------------------------------------------------
```

Note that the extra argument is declared as `double & coll_time`. Here the symbol `&` indicates that the variable `coll_time` will be passed to the fuction `acc_and_jerk()` by reference, not by value. This means that we can change `coll_time` inside the function, while the change persists after we leave the function and return to the `main()` function that called the function `acc_and_jerk()`. The variable name `coll_time_sq`, declared in the first line of `acc_and_jerk()`, is short hand for "collision time square". It is much cheaper and natural to calculate the squares of the magnitudes $|r_{ij}|$ and $|v_{ij}|$, stopping there rather than taking their square roots. The latter choice would force us to calculate roughly N^2 square roots, which are expensive to calculate, since they take much more time than additions or multiplications on most machines. All we want to know anyway is the minimum of their ratios. The same $\{i, j\}$ combination that minimizes $|r_{ij}|/|v_{ij}|$ will minimize $|r_{ij}|^2/|v_{ij}|^2$, and once we have determined that minimum, we perform the square root operation at the end of the function `acc_and_jerk()`.

Before we enter the $\{i, j\}$ loop, we have to set `coll_time_sq` to an arbitrary value that is high enough to guarantee that it exceeds the minimum value that we want to determine. Setting it to 10^{300} is surely overkill, but better safe than sorry, and according to the IEEE standard for double precision floating point numbers, this value does not yet lead to overflow. Inside the loop, we let `coll_time_sq` become shorter and shorter, through the assignment:

```
coll_time_sq = coll_est_sq;
```

each time when we find that the estimated collision time for the $\{i, j\}$ pair is shorter than the best estimate so far for `coll_time_sq`. The only other modification to `acc_and_jerk()` concerns the computation of `v2`, the square of the distance in velocity space between particles `i` and `j`.

The second function, `energy()`, remains unchanged. In the `main()` part of the program, the changes are minor, as can be seen from the result of doing a `diff` on the `main()` functions of `hermite4.C` and `hermite5.C`:

```
61c71
```

```
<     double dt = atof(argv[1]);
---
>     double dt_param = atof(argv[1]);
81c91,93
<     acc_and_jerk(m, r, v, a, jk, n);
---
>     double coll_time;
>     acc_and_jerk(m, r, v, a, jk, n, coll_time);
>     double dt = dt_param * coll_time;
102c114
<         acc_and_jerk(m, r, v, a, jk, n);
---
>         acc_and_jerk(m, r, v, a, jk, n, coll_time);
110a123,124
>  dt = dt_param * coll_time;
```

Instead of reading in the pregiven value for `dt` from the first argument on the command line, we now read in the value of `dt_param`, the factor with which we will multiply the estimate for the smallest collision time, in order to determine the next time step, as can be seen in the middle as well as in the very last line of changes reported by `diff`. The only other changes are the declaration of `double coll_time`, and the addition of that variable to the list of arguments in the two calls of the function `acc_and_jerk()`.

7.5 Variable Time Steps: Better Collision Criteria

If we sprinkle stars into space, roughly in the shape of a star cluster, with random positions and velocities, the time step criterion which we just introduced works fine. The chance that the relative velocities are all zero or extremely small is itself extremely small. However, during a long run, with millions of time steps, highly unlikely cases are bound to occur occasionally, and we should worry about them. In addition, it may happen that we start from initial conditions in which all velocities are zero. This is termed a 'cold collapse' type of initial condition, since the particle 'temperature', measured by their kinetic energies, is zero, and the particles will start of by falling roughly towards the center of the particle distribution. In such a case, `hermite5.C` would crash right at the beginning, during the attempt to calculate an infinitely long time step by dividing by zero.

We can make our code more safe, and able to handle cold starts, by adding an extra criterion. In addition to the time scale $t_{1ij} = |r_{ij}|/|v_{ij}|$, we introduce a second time scale $t_{2ij} = \sqrt{|r_{ij}|/|da_{ij}|}$, where da_{ij} is the relative acceleration between particles i and j, due to their mutual attraction. Since acceleration is the second derivative of position, we have to take a square root in order to wind up with a quantity that has the dimension of time.

Note the notation used here: in analogy with the definitions of $r_{ij} = |\mathbf{r}_i - \mathbf{r}_j|$ and $v_{ij} = |\mathbf{v}_i - \mathbf{v}_j|$, we could introduce $a_{ij} = |\mathbf{a}_i - \mathbf{a}_j|$. However, this last quantity would denote the difference between the *total* accelerations felt by particles i and j due to all other particles in the system. In contrast, we are only interested in that part of the difference in their accelerations due to their own accelerations, da_{ij}, since it is that part that is most likely to be important for neighboring particles. In other words, we neglect the gravitational tidal field contributions of all other particles while examining a particle pair. The alternative, of using a_{ij} instead of da_{ij}, would be far more expensive. We would have to compute all total accelerations first, and then do a pairwise comparison, which would entail a double pass over all particles, each with a computational cost of order N.

Here is the implementation of `hermite6.C`. With respect to `hermite5.C`, the differences are all limited to the first function, `acc_and_jerk`.

Code 7.8 (hermite6.C)

```
//-----------------------------------------------------------------------------
// hermite6.C
//-----------------------------------------------------------------------------
#include  <iostream>
#include  <cmath>
#include  <cstdlib>
using namespace std;

void acc_and_jerk(double m[], double r[][3], double v[][3], double a[][3],
                  double jk[][3], int n, double & coll_time)
{
    double coll_time_q = 1e300;;
    double coll_est_q;

    for (int i = 0; i < n; i++)
        for (int k = 0; k < 3; k++)
            a[i][k] = jk[i][k] = 0.0;
    for (int i = 0; i < n; i++){
        for (int j = i+1; j < n; j++){
            double rji[3], vji[3];
            for (int k = 0; k < 3; k++){
                rji[k] = r[j][k] - r[i][k];
                vji[k] = v[j][k] - v[i][k];
            }
            double r2 = 0;
            double v2 = 0;
            for (int k = 0; k < 3; k++){
                r2 += rji[k] * rji[k];
                v2 += vji[k] * vji[k];
            }
            coll_est_q = (r2*r2) / (v2*v2);
            if (coll_time_q > coll_est_q)
                coll_time_q = coll_est_q;
            double r3 = r2 * sqrt(r2);
            double rv = 0;
            for (int k = 0; k < 3; k++)
                rv += rji[k] * vji[k];
            rv /= r2;
            double da[3], dj[3];
            for (int k = 0; k < 3; k++){
                da[k] = rji[k] / r3;
                dj[k] = (vji[k] - 3 * rv * rji[k]) / r3;
            }
            for (int k = 0; k < 3; k++){
                a[i][k] += m[j] * da[k];
                a[j][k] -= m[i] * da[k];
                jk[i][k] += m[j] * dj[k];
                jk[j][k] -= m[i] * dj[k];
            }
            double da2 = 0;
            for (int k = 0; k < 3; k++)
                da2 += da[k] * da[k];
            double mij = m[i] + m[j];
            da2 *= mij * mij;
            coll_est_q = r2/da2;
            if (coll_time_q > coll_est_q)
                coll_time_q = coll_est_q;
        }
    }
    coll_time = sqrt(sqrt(coll_time_q));
}

double energy(double m[], double r[][3], double v[][3], int n)
{
    double ekin = 0, epot = 0;
    for (int i = 0; i < n; i++){
        for (int j = i+1; j < n; j++){
            double rji[3];
            for (int k = 0; k < 3; k++)
                rji[k] = r[j][k] - r[i][k];
            double r2 = 0;
            for (int k = 0; k < 3; k++)
                r2 += rji[k] * rji[k];
            epot -= m[i]*m[j]/sqrt(r2);
        }
        for (int k = 0; k < 3; k++)
            ekin += 0.5 * m[i] * v[i][k] * v[i][k];
    }
    return ekin + epot;
```

```
}

int main(int argc, char *argv[])
{
    double dt_param = atof(argv[1]);
    double t_end = atof(argv[2]);

    int n;
    cin >> n;

    double t;
    cin >> t;

    double * m = new double[n];
    double (* r)[3] = new double[n][3];
    double (* v)[3] = new double[n][3];
    double (* a)[3] = new double[n][3];
    double (* jk)[3] = new double[n][3];

    for (int i = 0; i < n ; i++){
        cin >> m[i];
        for (int k = 0; k < 3; k++)
            cin >> r[i][k];
        for (int k = 0; k < 3; k++)
            cin >> v[i][k];
    }

    double coll_time;
    acc_and_jerk(m, r, v, a, jk, n, coll_time);
    double dt = dt_param * coll_time;

    double e_in = energy(m, r, v, n);
    cerr << "Initial total energy E_in = " << e_in << endl;

    double dt_out = 0.01;
    double t_out = dt_out;

    double old_r[n][3], old_v[n][3], old_a[n][3], old_j[n][3];

    for (double t = 0; t < t_end; t += dt){
        for (int i = 0; i < n; i++){
            for (int k = 0; k < 3; k++){
                old_r[i][k] = r[i][k];
                old_v[i][k] = v[i][k];
                old_a[i][k] = a[i][k];
                old_j[i][k] = jk[i][k];
                r[i][k] += v[i][k]*dt + a[i][k]*dt*dt/2 + jk[i][k]*dt*dt*dt/6;
                v[i][k] += a[i][k]*dt + jk[i][k]*dt*dt/2;
            }
        }
        acc_and_jerk(m, r, v, a, jk, n, coll_time);
        for (int i = 0; i < n; i++){
            for (int k = 0; k < 3; k++){
                v[i][k] = old_v[i][k] + (old_a[i][k] + a[i][k])*dt/2
                                      + (old_j[i][k] - jk[i][k])*dt*dt/12;
                r[i][k] = old_r[i][k] + (old_v[i][k] + v[i][k])*dt/2
                                      + (old_a[i][k] - a[i][k])*dt*dt/12;
            }
        }
        dt = dt_param * coll_time;

        if (t >= t_out){
            cout << n << endl;
            cout << t << endl;
            for (int i = 0; i < n; i++){
                cout << m[i] << " ";
                for (int k = 0; k < 3; k++)
                    cout << r[i][k] << " ";
                for (int k = 0; k < 3; k++)
                    cout << v[i][k] << " ";
                cout << endl;
            }
            t_out += dt_out;
        }
    }

    double e_out = energy(m, r, v, n);

    cerr << "Final total energy E_out = " << e_out << endl;
    cerr << "absolute energy error: E_out - E_in = " << e_out - e_in << endl;
    cerr << "relative energy error: (E_out - E_in) / E_in = "
         << (e_out - e_in) / e_in << endl;
}
//------------------------------------------------------------------------------
```

Exercise 7.2 (Pythagorean problem: variable time steps)
Repeat the previous exercise, starting again with the Pythagorean initial conditions, but now use `hermite6.C`*. In order to get good energy conservation, roughly how much time do you gain when using a variable integrator?*

7.6 Further Improvements

That's all it took to change our code from a toy version of an integrator to a real N-body code with which we are ready to start doing production runs on concrete astrophysical problems.

There are two caveats. Although the `hermite6.C` code is fully able to tackle a number of small-to-moderate size N-body calculations in a reasonably efficient way, it is still possible to make significant additional speed-ups, which can make the program run orders of magnitude faster for reasonably large N values. One natural extension is to give each particle an individual time step. When two, three, or more particles have a simultaneous close encounter, it is important that they slow down enough to resolve the curvature in their orbits, but there is no reason, really, to force the rest of the system to join in the slow crawl. Efficient and accurate ways to implement individual time step schemes complicate the code significantly, and therefore we will postpone a treatment of that approach till a later volume in the current *Pure Gravity* series.

The second caveat concerns the ease of use of `hermite6.C`. As it stands, the user only has two handles on the way the code runs, through the command line arguments that initialize the variables `dt_param` and `t_end`. As soon as one starts performing laboratory experiments on N-body systems, the need will arise to have more control over, for example, the frequency of output of snapshots or the frequency of output of diagnostic information concerning energy conservation. In addition, in some cases it is convenient to echo the initial snapshot to the output as the first snapshot reported by the integrator; in other cases we may only be interested in the final snapshot, without having the output file cluttered by an extra copy of the initial conditions. Also, for debugging purposes, it would be nice to be able to specify on the command line that we would like to see information about accelerations and jerks, in addition to the standard information about positions and velocities. Finally, a 'help' option would be nice to remind us how to invoke all these options.

There are other serious shortcomings in `hermite6.C`. Although it has two functions, it is easy to make the whole code significantly more modular, and therefore easier to extend when we want to add extra functionality. In addition, it is high time to add comments, so that other users (and we ourselves in the near future) will be reminded of what is going on, both in the flow of information and control throughout the whole file, as well as inside each function.

In the next chapter, we will address this second caveat. Without changing the overall functionality of `hermite6.C`, we will wind up with a far more robust, flexible, and extendible version of the program. With that version in hand, we will then try our hand at experimentation, laboratory style, simulating the behavior of star clusters.

Chapter 8

A More Modular N-Body Hermite Code

8.1 Starting a Tool Box

In this chapter we will discuss in detail a more modular version of the Hermite code `hermite6.C`, developed in the previous chapter. The new version is called `nbody_sh1.C`. Here 'sh' stands for the shared but variable time step choice, and the number 1 indicates again that this is the first version. This code will be the first tool of a tool box that we will continue to develop in the rest of this book, as well as in following books in this series. From now on, each tool will adhere to our N-body I/O format, specified in the previous chapter (and possibly more fancy formats as well, but we will keep those more advanced versions compatible with our current bare bones format). In addition, each tool will have extensive comments, explaining both the usage and the internal structure of the code.

The new code, `nbody_sh1.C`, has roughly four times more lines than the previous version, `hermite6.C`. Almost half of these lines are either comments of blank lines, both of which help to make the code more readable and more understandable. The fact that the code itself still has more than twice the length of the previous version stems from several factors. First, the new code has nine functions, besides `main()`, while the old code had only two. Second, there are seven command line options, rather than two. Third, we now declare all functions at the top of the file. Finally, there is more diagnostics output than we had before.

Below, the full code is presented, one function at a time.

8.2 Gravitylab

Our aim is to build a powerful software environment for experiments in stellar dynamics of dense stellar systems. The idea is to build a virtual laboratory, which we will call *gravitylab*. From now on, each new tool in our tool box will have a distinctive 'gravitylab' header:

Code 8.1 (nbody_sh1.C: logo_begin)

```
              // Time-stamp: <2003-09-10 12:22:36 piet>
             //================================================================
            //                                                                |
           //           /__----__                         ........            |
          //       .           \                     ....:        :.          |
         //       :                 _\|/_         ..:                         |
        //       :                   /|\         :                     _\|/_  |
       //  ___   ___                  _____                      ___    /|\   |
      //  /     |   \    /\ \     / |   |  \  / |        /\    |   \         |
     //  |   __ |___/   /  \ \   /  |   |    \/  |       /  \   |___/         |
    //   |    | |  \   /____\ \ /   |   |    /   |      /____\  |   \     \/  |
   //     \___| |   \ /      \ V    |   |   /    |____ /      \ |___/     |   |
  //                                                                      /   |
 //              :                       _/|     :..                    |/    |
//                :..              ____/           :....          ..         |
/*   o   //          :.    _\|/_     /                   :........:           |
 *  O  ‘//\                 /|\                                               |
 *  |     /\                                                                  |
 *=============================================================================
```

We will not show these headers in future code listings, but they will be there in the source code for other tools. The three stars moving on a figure-8 orbit are inspired by the solution presented in chapter 5. They are being observed at bottom left by the small figure looking through a telescope.

Note the time stamp at the very first line. This is a handy feature of the *emacs* editor that we have used to write this book. When you add the line "`(add-hook ’write-file-hooks ’time-stamp)`" to the `.emacs` startup file, the date and time and user name will be updated automatically each time you write the file to disk.

8.3 Introductory Comments

Immediately following the gravitylab header, we see a lengthy comment block:

Code 8.2 (nbody_sh1.C: summary)

```
*============================================================================
*
*  nbody_sh1.C:  an N-body integrator with a shared but variable time step
*                (the same for all particles but its size changing in time),
*                using the Hermite integration scheme.
*
*                ref.: Hut, P., Makino, J. & McMillan, S., 1995,
*                      Astrophysical Journal Letters 443, L93-L96.
*
*  note: in this first version, all functions are included in one file,
*        without any use of a special library or header files.
*____________________________________________________________________________
*
*  usage: nbody_sh1 [-h (for help)] [-d step_size_control_parameter]
*                   [-e diagnostics_interval] [-o output_interval]
*                   [-t total_duration] [-i (start output at t = 0)]
*                   [-x (extra debugging diagnostics)]
*
*         "step_size_control_parameter" is a coefficient determining the
*            the size of the shared but variable time step for all particles
*
*         "diagnostics_interval" is the time between output of diagnostics,
*            in the form of kinetic, potential, and total energy; with the
*            -x option, a dump of the internal particle data is made as well
*
*         "output_interval" is the time between successive snapshot outputs
*
*         "total_duration" is the integration time, until the program stops
*
*         Input/output are read/written from/to the standard i/o streams.
*         Since all options have sensible defaults, the simplest way to run
*         the code is by only specifying the i/o files for the N-body
*         snapshots:
*
*            nbody_sh1 < data.in > data.out
*
*         The diagnostics information will then appear on the screen.
*         To capture the diagnostics information in a file, capture the
*         standard error stream as follows:
*
*            (nbody_sh1 < data.in > data.out) >& data.err
*
*  Note: if any of the times specified in the -e, -o, or -t options are not an
*        an integer multiple of "step", output will occur slightly later than
```

```
 *       predicted, after a full time step has been taken.  And even if they
 *       are integer multiples, round-off error may induce one extra step.
 *-----------------------------------------------------------------------------
 *
 *  External data format:
 *
 *     The program expects input of a single snapshot of an N-body system,
 *     in the following format: the number of particles in the snapshot n;
 *     the time t; mass mi, position ri and velocity vi for each particle i,
 *     with position and velocity given through their three Cartesian
 *     coordinates, divided over separate lines as follows:
 *
 *                   n
 *                   t
 *                   m1 r1_x r1_y r1_z v1_x v1_y v1_z
 *                   m2 r2_x r2_y r2_z v2_x v2_y v2_z
 *                   ...
 *                   mn rn_x rn_y rn_z vn_x vn_y vn_z
 *
 *     Output of each snapshot is written according to the same format.
 *
 *  Internal data format:
 *
 *     The data for an N-body system is stored internally as a 1-dimensional
 *     array for the masses, and 2-dimensional arrays for the positions,
 *     velocities, accelerations and jerks of all particles.
 *-----------------------------------------------------------------------------
 *
 *    version 1:  Jan 2002   Piet Hut, Jun Makino
 *-----------------------------------------------------------------------------
```

It starts with the name of the file, a brief summary with a reference to the literature, followed by a detailed description of how to use the code. For a typical user, this is all the information needed. As long as the user combines `nbody_sh1` with other tools from gravitylab, there is even no need to understand the external data format, in which the N-body snapshots are written to and read from files. For those users interested in such details, as well as in the internal format in which the data are stored during the execution of the code, the comment block contains format information near the end. The last few lines list the history and version numbers of the code.

8.4 Include Statements, Function Declarations, etc.

The first lines of real code start right after the introductory comments:

Code 8.3 (nbody_sh1.C: premain)

```
#include  <iostream>
#include  <cmath>                          // to include sqrt(), etc.
#include  <cstdlib>                        // for atoi() and atof()
#include  <unistd.h>                       // for getopt()
using namespace std;

typedef double  real;                      // "real" as a general name for the
                                           // standard floating-point data type

const int NDIM = 3;                        // number of spatial dimensions

void correct_step(real pos[][NDIM], real vel[][NDIM],
                  const real acc[][NDIM], const real jerk[][NDIM],
                  const real old_pos[][NDIM], const real old_vel[][NDIM],
                  const real old_acc[][NDIM], const real old_jerk[][NDIM],
                  int n, real dt);
void evolve(const real mass[], real pos[][NDIM], real vel[][NDIM],
            int n, real & t, real dt_param, real dt_dia, real dt_out,
            real dt_tot, bool init_out, bool x_flag);
void evolve_step(const real mass[], real pos[][NDIM], real vel[][NDIM],
                 real acc[][NDIM], real jerk[][NDIM], int n, real & t,
                 real dt, real & epot, real & coll_time);
void get_acc_jerk_pot_coll(const real mass[], const real pos[][NDIM],
                           const real vel[][NDIM], real acc[][NDIM],
                           real jerk[][NDIM], int n, real & epot,
                           real & coll_time);
void get_snapshot(real mass[], real pos[][NDIM], real vel[][NDIM], int n);
void predict_step(real pos[][NDIM], real vel[][NDIM],
                  const real acc[][NDIM], const real jerk[][NDIM],
                  int n, real dt);
void put_snapshot(const real mass[], const real pos[][NDIM],
                  const real vel[][NDIM], int n, real t);
bool read_options(int argc, char *argv[], real & dt_param, real & dt_dia,
                  real & dt_out, real & dt_tot, bool & i_flag, bool & x_flag);
void write_diagnostics(const real mass[], const real pos[][NDIM],
                       const real vel[][NDIM], const real acc[][NDIM],
                       const real jerk[][NDIM], int n, real t, real epot,
                       int nsteps, real & einit, bool init_flag,
                       bool x_flag);
```

We start with `#include` statements to various libraries. The comments on each line mention some of the functions used from those libraries. If we would leave out one of these include statements, the corresponding functions listed could not be linked, and the compiler would issue an error.

The next statement indicates that we used the standard C++ namespace. Later, when gravitylab will have grown sufficiently large, it may be useful to create our own namespaces, in order to avoid collisions with other programs that may use names that are the same as we have chosen. Right now it is too early to worry about such complications.

The `typedef` statement defines the word `real` as an alternative for the build-in function type `double`. From now on we will only use the name `real` to indicate the standard floating point type `double`. It is far more logical to talk about real numbers of type `real`, together with the integers of type `int`, without using the archaic term 'double' that stems from the expression 'double precision' (long ago, the standard precision for floating point calculations used four bytes per floating point word, leading to the expression double precision for the now-standard eight-byte word length).

Next we introduce the symbol `NDIM` for the number of dimensions. So far we have simply used the number 3 in our loops over Cartesian coordinates, but it is much better not to have any magic numbers in a code, where a magic number is defined as anything that is not 0 or 1. The term "NDIM" for the number of dimensions is far clearer than a blind "3" in the middle of a piece of code. A second advantage of introducing a symbol, rather than magic numbers, is that we can change the symbol at one place, while guaranteeing its substitution everywhere else in the code. In the vast majority of cases, we will do our simulations in three spatial dimensions, hence the assignment here of the number 3 to `NDIM` here, but we will also encounter cases where we want to do some experimentation in one or two dimensions. In that case, changing 3 to 1 or 2 in this line is all we need to do (apart from making sure that we have not used uniquely three-dimensional constructs elsewhere in the code, such as for example the use of 3D spherical harmonics).

Note that older C-style usage would have defined `NDIM` through the macro definition "`#define NDIM 3`" . Nowadays, however, it is considered good form to use the C++ expression "`const int NDIM = 3;`" . Although the use of a `#define` macro in this case is quite innocent, there are many other cases where the use of macros can lead to code that is prone to confusing errors that are hard to debug. Therefore, as a matter of style it is a good idea to avoid them as much as possible.

The following nine function declarations are necessary if we want to have the freedom to define them in an arbitrary way in the rest of the file. The problem is that the C compiler goes through the file in one single pass, from top to bottom. As long as each function is invoked only after it has been seen by the compiler, there is no problem. In the codes `hermite4.C` through `hermite6.C`, the two functions listed at the top of the files were invoked only by `main()`, which was listed last. In general, however, with many functions there may not be a unique flow of functions calls. Besides, it is easier to follow the logic of the code if we can start with `main()` at the top of the file. The latter immediately implies that we will have to declare all functions mentioned in `main()`.

This need for redundant information in the form of declarations is a weakness of C++. In general, any time that a computer language forces you to duplicate information, it brings with it the danger of errors creeping in. It is easy to change the definition of a function without changing the declaration, or *vice versa.* In some cases, the compiler may catch this, but there may be other cases where overloading of function names with different argument sets makes it impossible for the compiler to catch such mistakes. Unfortunately, we will have to live with this situation.

Another example of redundant information in our program is the description of the command line options. Almost the same words appear once in the 'usage' part of the initial commments, and twice in the function `read_options()` (for the help option and the unknown option). It is possible to capture that information in a string at the top of the program, and to echo that string in `read_options()`. We will make such a modification later.

8.5 The Function `main()`

Code 8.4 (nbody_sh1.C: main)

```
/*-----------------------------------------------------------------------------
 *  main  --  reads option values, reads a snapshot, and launches the
 *            integrator
 *-----------------------------------------------------------------------------
 */

int main(int argc, char *argv[])
{
    real  dt_param = 0.03;     // control parameter to determine time step size
    real  dt_dia = 1;          // time interval between diagnostics output
    real  dt_out = 1;          // time interval between output of snapshots
    real  dt_tot = 10;         // duration of the integration
    bool  init_out = false;    // if true: snapshot output with start at t = 0
                               //          with an echo of the input snapshot
    bool  x_flag = false;      // if true: extra debugging diagnostics output

    if (! read_options(argc, argv, dt_param, dt_dia, dt_out, dt_tot, init_out,
                       x_flag))
        return 1;                // halt criterion detected by read_options()

    int n;                       // N, number of particles in the N-body system
    cin >> n;

    real t;                      // time
    cin >> t;
    real * mass = new real[n];                // masses for all particles
    real (* pos)[NDIM] = new real[n][NDIM];   // positions for all particles
    real (* vel)[NDIM] = new real[n][NDIM];   // velocities for all particles

    get_snapshot(mass, pos, vel, n);

    evolve(mass, pos, vel, n, t, dt_param, dt_dia, dt_out, dt_tot, init_out,
           x_flag);

    delete[] mass;
    delete[] pos;
    delete[] vel;
}
```

The first six variables declared at the top of `main()` receive their values from the function `read_options()` which reads the Unix style command line arguments. Note that each variable has a default value, which is retained unless it is changed explicitly by the corresponding command. We discuss the usage of command line options in the next section.

If the function `read_options()` detects a request for help, or the invocation of a non-existent option, it will return the Boolean value `false`. In that case the statement `!read_options()` is true, and program execution is halted. In C++, returning the value 0 indicates normal successful completion of the `main()` program, while any other value indicates a failure of some kind or other. For simplicity we return here the value 1.

Once the options are interpreted, we are ready to read the N-body snapshot from the standard input (which typically is redirected to read either the contents of a file, as in `nbody_sh1 < data.in` or to receive data from another program through a pipe, as in `generate_data | nbody_sh1`). Once the number of particles `n` has been read in, we can allocate storage space to contain the masses and dynamical information for all `n` particles, as we have seen in the previous chapter. The actual initialization of the arrays is carried out by the function `get_snapshot()`.

The real work is then delegated to the function `evolve()`, which oversees the evolution in time of the N-body system. When the call to `evolve()` returns, there is nothing left to be done. For good form we then deallocate the memory that we had dynamically allocated with the `new` operator. Note the square brackets in `delete`, which tell the compiler to delete the full memory assigned to the arrays. If we would leave this out, for example in a statement `delete[] mass`, we would only free the memory for `mass[0]`. This would constitute a memory leak, since the rest of the array will still be allocated, but it will be no longer usable in our program. In our particular case, this is no problem since we are about to terminate the program anyway, but in more complex cases, such as we will encounter in the function `evolve()`, it will be important to not create memory leaks.

8.6 Command Line Options

There are six command line options, Unix style, from which we can choose. All essential options have default values, so it is perfectly possible to run our code without specifying any of them. For example, if we start with an N-body snapshot in an input file `data.in`, we can run the code to produce a stream of snapshot data in the output file `data.out`, by typing:

```
|gravity> nbody_sh1 < data.in > data.out
```

This will have the exact same effect as if we would have specified the default values for the four main options, namely the time step control parameter (0.03), the interval between diagnostics output (1 time unit), the interval between output of snapshots (1 time unit), and the duration of the integration (10 time units):

```
|gravity> nbody_sh1 -d 0.03 -e 1 -o 1 -t 10 < data.in > data.out
```

If we would like to have three times smaller time steps, twice as many diagnostics outputs and with additional information, snapshot output intervals of 5 time units but starting at `t = 0`, and a total run time of 30 time units, we have to give the following command:

```
|gravity> nbody_sh1 -d 0.01 -e 0.5 -x -o 5 -i -t 30 < data.in > data.out
```

The order of the arguments is unimportant, but each option that expects a value (the `-d, -e, -o, -t` options) should be immediately followed by its corresponding value. By the way, the value `0.03` as the default for the scale of the time step parameter is somewhat arbitrary. In practice, a value of `0.1` is often found to be too large, while `0.01` is often overkill. For example, when we start from the initial conditions for three stars on a figure 8 orbit, running `nbody_sh1` with all default values in place, we wind up at time `t = 10` with a relative energy error of order 10^{-7}.

Of course, the optimal choice of values depend strongly on the particular application, and the default values are only a hint, in a blind attempt to come up with at least somewhat reasonable starting values. It is up to the user to make sure that these values are appropriate in a given situation, and if not, to supply a better value after some experimentation.

The help option can be invoked by typing:

```
|gravity> nbody_sh1 -h
```

This will not result in program execution, only in the printing of a short message that lays out the various command line option choices. A similar message will appear when we attempt to supply an non-existent option, for example:

```
|gravity> nbody_sh1 -q
nbody_sh1: invalid option -- q
usage: nbody_sh1 [-h (for help)] [-d step_size_control_parameter]
         [-e diagnostics_interval] [-o output_interval]
         [-t total_duration] [-i (start output at t = 0)]
         [-x (extra debugging diagnostics)]
|gravity>
```

All this behavior can be inspected in the function `read_options()`:

Code 8.5 (nbody_sh1.C: read_options)

```
/*-----------------------------------------------------------------------------
 *  read_options  --  reads the command line options, and implements them.
 *
 *  note: when the help option -h is invoked, the return value is set to false,
 *        to prevent further execution of the main program; similarly, if an
 *        unknown option is used, the return value is set to false.
 *-----------------------------------------------------------------------------
 */

bool read_options(int argc, char *argv[], real & dt_param, real & dt_dia,
                  real & dt_out, real & dt_tot, bool & i_flag, bool & x_flag)
{
    int c;
    while ((c = getopt(argc, argv, "hd:e:o:t:ix")) != -1)
        switch(c){
            case 'd': dt_param = atof(optarg);
                      break;
            case 'e': dt_dia = atof(optarg);
                      break;
            case 'i': i_flag = true;
                      break;
            case 'o': dt_out = atof(optarg);
                      break;
            case 't': dt_tot = atof(optarg);
                      break;
            case 'x': x_flag = true;
                      break;
            case 'h':
            case '?': cerr << "usage: " << argv[0]
                           << " [-h (for help)]"
                           << " [-d step_size_control_parameter]\n"
                           << "         [-e diagnostics_interval]"
                           << " [-o output_interval]\n"
                           << "         [-t total_duration]"
```

```
                             << " [-i (start output at t = 0)]\n"
                             << "         [-x (extra debugging diagnostics)]"
                             << endl;
                        return false;      // execution stops after help or error
            }

    return true;                           // ready to continue program execution
}
```

Note that the six variables corresponding to the command line arguments are all passed by reference, so that the results are available to the calling program `main()`.

The function `getopt()` is a standard C library function that can be used equally well in C++ programs. Its third argument is a string which lists all command line options. Each option can only consist of a single letter. Those letters that should be followed by a value to be read in are indicated by a colon immediately following the letter. The string `"hd:e:o:t:ix"` tells us that options `h, i` and `x` do not expect additional values, while options `d, e, o` and `t` are to be followed with an argument, all of which are of type `real` in our particle case. All option arguments are by default passed as ASCII strings, so we need the function `atof()` to convert the ASCII information into the proper floating point value, as we already saw in the previous chapter.

Notice that each `case` in the body of the `switch` statement is ended by either a `return` statement or a `break` statement. The latter is necessary, since the default behavior of `switch` is to 'fall through' from one case to the next, something that is clearly not desirable here. After we jump out of the `switch` statement through a `break` command, we encounter the last statement, "`return true;`" which tells the calling program that all is well, and that execution can continue.

8.7 Snapshot Input

The code for snapshot input is straightforward:

Code 8.6 (nbody_sh1.C: get_snapshot)

```
/*-----------------------------------------------------------------------------
 *  get_snapshot  --  reads a single snapshot from the input stream cin.
 *
 *  note: in this implementation, only the particle data are read in, and it
 *        is left to the main program to first read particle number and time
 *-----------------------------------------------------------------------------
 */

void get_snapshot(real mass[], real pos[][NDIM], real vel[][NDIM], int n)
{
    for (int i = 0; i < n ; i++){
        cin >> mass[i];                        // mass of particle i
        for (int k = 0; k < NDIM; k++)
            cin >> pos[i][k];                  // position of particle i
        for (int k = 0; k < NDIM; k++)
            cin >> vel[i][k];                  // velocity of particle i
    }
}
```

Note that we do not check here whether a complete snapshot is being offered on the standard input stream in the right format. It would be better to verify, for example, that new lines
n occur in the correct places, separating each particle, and that no end-of-file condition is encountered before the whole N-body snapshot is read in. In later versions we will provide more complete error checking.

8.8 Snapshot Output

The code for snapshot output is similarly simple:

Code 8.7 (nbody_sh1.C: put_snapshot)

```
/*-----------------------------------------------------------------------------
 *  put_snapshot  --  writes a single snapshot on the output stream cout.
 *
 *  note: unlike get_snapshot(), put_snapshot handles particle number and time
 *-----------------------------------------------------------------------------
 */

void put_snapshot(const real mass[], const real pos[][NDIM],
                  const real vel[][NDIM], int n, real t)
{
    cout.precision(16);                        // full double precision

    cout << n << endl;                         // N, total particle number
    cout << t << endl;                          // current time
    for (int i = 0; i < n ; i++){
        cout << mass[i];                        // mass of particle i
        for (int k = 0; k < NDIM; k++)
            cout << ' ' << pos[i][k];           // position of particle i
        for (int k = 0; k < NDIM; k++)
            cout << ' ' << vel[i][k];           // velocity of particle i
        cout << endl;
    }
}
```

Note that the masses, positions, and velocities are all declared as `const` in the declaration of the function arguments. This means that this function is not allowed to change the values of those particular arguments. Being able to specify function arguments as `const` is a very useful C++ feature. It can help the compiler by providing extra information; it allows the compiler to flag an error if in the body of the function an attempt is made to change one of those arguments erroneously; and most importantly, it gives the human reader useful information about the intentions of the programmer.

For all these reasons, it is important to be consistent in the use of `const` specifications, and to always use `const` wherever we can. When we do this, we thereby imply that the absence of a `const` specifier for an argument means that we do want to affect the value of that particular argument. For example, in the previous function `get_snapshot()`, masses, positions, and velocities are not preceded by `const`. Indeed, all three arrays are being initialized in that function, and it is useful to be able to anticipate that already from looking at the argument list, either here or at the top of the file where all functions are declared.

The first line of the body of the function sets the precision for all subsequent output. It turns out that eight-byte double precision corresponds to about 16 digits of relative accuracy. If we would output less than 16 significant digits for each `real` variable, we would lose information. A subsequent program reading in the snapshot that we have just written out would not have access to the full information that we had before we wrote our data. On the other hand, if we would output those numbers with more than 16 digits, the extra digits would be effective garbage. While this doesn't hurt, it is a waste of space (and possibly later processing time) to go beyond 16 digits.

8.9 Reporting Diagnostics

Here is the code for the function that writes diagnostics to the standard error stream. Note the declarations of arguments: all arrays are specified to be `const`, which is appropriate since their values should only be reported, without changing them. The argument `einit` is passed by reference, since it will hold the initial value of the total energy of the system, information that should be passed back to the calling function. The other arguments are all passed by value.

Code 8.8 (nbody_sh1.C: write_diagnostics)

```
/*-----------------------------------------------------------------------------
 *  write_diagnostics  --  writes diagnostics on the error stream cerr:
 *                         current time; number of integration steps so far;
 *                         kinetic, potential, and total energy; absolute and
 *                         relative energy errors since the start of the run.
 *                         If x_flag (x for eXtra data) is true, all internal
 *                         data are dumped for each particle (mass, position,
 *                         velocity, acceleration, and jerk).
 *
 *  note: the kinetic energy is calculated here, while the potential energy is
 *        calculated in the function get_acc_jerk_pot_coll().
 *-----------------------------------------------------------------------------
 */

void write_diagnostics(const real mass[], const real pos[][NDIM],
                       const real vel[][NDIM], const real acc[][NDIM],
                       const real jerk[][NDIM], int n, real t, real epot,
                       int nsteps, real & einit, bool init_flag,
                       bool x_flag)
{
    real ekin = 0;                    // kinetic energy of the n-body system
    for (int i = 0; i < n ; i++)
        for (int k = 0; k < NDIM ; k++)
            ekin += 0.5 * mass[i] * vel[i][k] * vel[i][k];

    real etot = ekin + epot;          // total energy of the n-body system

    if (init_flag)                    // at first pass, pass the initial
        einit = etot;                 // energy back to the calling function

    cerr << "at time t = " << t << " , after " << nsteps
         << " steps :\n  E_kin = " << ekin
         << " , E_pot = " << epot
         << " , E_tot = " << etot << endl;
    cerr << "                "
         << "absolute energy error: E_tot - E_init = "
         << etot - einit << endl;
    cerr << "                "
         << "relative energy error: (E_tot - E_init) / E_init = "
         << (etot - einit) / einit << endl;

    if (x_flag){
        cerr << "  for debugging purposes, here is the internal data "
             << "representation:\n";
        for (int i = 0; i < n ; i++){
            cerr << "    internal data for particle " << i+1 << " : " << endl;
            cerr << "      ";
            cerr << mass[i];
            for (int k = 0; k < NDIM; k++)
                cerr << ' ' << pos[i][k];
            for (int k = 0; k < NDIM; k++)
                cerr << ' ' << vel[i][k];
            for (int k = 0; k < NDIM; k++)
                cerr << ' ' << acc[i][k];
            for (int k = 0; k < NDIM; k++)
                cerr << ' ' << jerk[i][k];
            cerr << endl;
        }
    }
}
```

The only calculation performed in this function is that of the kinetic energy. The potential energy is determined in the function `get_acc_jerk_pot_coll()`. The `init_flag` is set to `true` when `write_diagnostics()` is evoked for the first time, at `t = 0`. In that case, we want to pass the value of the initial total energy back to the calling function `evolve()`, which can use that information to compare it with later measured values of the total energy, in order to determine the absolute and relative amounts of energy drifts, which are a good measure of numerical accuracy.

Note that we could have defined the initial energy `einit` as a static variable inside `write_diagnostics()`. For our present purpose that would be fine, but this type of programming may easily create a future limitation. If some day we would like to compare two different N-body systems, each of which evolves, we would get into a conflict if both of them would try to access the same static variable. Therefore, for the same reason we don't use global variables in the first place, we prefer to pass `einit` as a function variable.

8.10 Orbit Integration

We now come to the function that manages the orbit evolution, driving the Hermite integrator and scheduling the various output operations:

Code 8.9 (nbody_sh1.C: evolve)

```
/*-----------------------------------------------------------------------------
 *  evolve  --  integrates an N-body system, for a total duration dt_tot.
 *              Snapshots are sent to the standard output stream once every
 *              time interval dt_out.  Diagnostics are sent to the standard
 *              error stream once every time interval dt_dia.
 *
 *  note: the integration time step, shared by all particles at any given time,
 *        is variable.  Before each integration step we use coll_time (short
 *        for collision time, an estimate of the time scale for any significant
 *        change in configuration to happen), multiplying it by dt_param (the
 *        accuracy parameter governing the size of dt in units of coll_time),
 *        to obtain the new time step size.
 *
 *  Before moving any particles, we start with an initial diagnostics output
 *  and snapshot output if desired.  In order to write the diagnostics, we
 *  first have to calculate the potential energy, with get_acc_jerk_pot_coll().
 *  That function also calculates accelerations, jerks, and an estimate for the
 *  collision time scale, all of which are needed before we can enter the main
 *  integration loop below.
 *       In the main loop, we take as many integration time steps as needed to
 *  reach the next output time, do the output required, and continue taking
 *  integration steps and invoking output this way until the final time is
 *  reached, which triggers a ‘break’ to jump out of the infinite loop set up
 *  with ‘while(true)’.
 *-----------------------------------------------------------------------------
 */

void evolve(const real mass[], real pos[][NDIM], real vel[][NDIM],
            int n, real & t, real dt_param, real dt_dia, real dt_out,
            real dt_tot, bool init_out, bool x_flag)
{
    cerr << "Starting a Hermite integration for a " << n
         << "-body system,\n  from time t = " << t
         << " with time step control parameter dt_param = " << dt_param
         << "  until time " << t + dt_tot
         << " ,\n  with diagnostics output interval dt_dia = "
         << dt_dia << ",\n  and snapshot output interval dt_out = "
         << dt_out << "." << endl;

    real (* acc)[NDIM] = new real[n][NDIM];          // accelerations and jerks
    real (* jerk)[NDIM] = new real[n][NDIM];         // for all particles
    real epot;                       // potential energy of the n-body system
    real coll_time;                  // collision (close encounter) time scale
    get_acc_jerk_pot_coll(mass, pos, vel, acc, jerk, n, epot, coll_time);

    int nsteps = 0;              // number of integration time steps completed
    real einit;                  // initial total energy of the system

    write_diagnostics(mass, pos, vel, acc, jerk, n, t, epot, nsteps, einit,
                      true, x_flag);
    if (init_out)                                  // flag for initial output
        put_snapshot(mass, pos, vel, n, t);

    real t_dia = t + dt_dia;          // next time for diagnostics output
    real t_out = t + dt_out;          // next time for snapshot output
    real t_end = t + dt_tot;          // final time, to finish the integration

    while (true){
        while (t < t_dia && t < t_out && t < t_end){
            real dt = dt_param * coll_time;
            evolve_step(mass, pos, vel, acc, jerk, n, t, dt, epot, coll_time);
            nsteps++;
        }
        if (t >= t_dia){
            write_diagnostics(mass, pos, vel, acc, jerk, n, t, epot, nsteps,
                              einit, false, x_flag);
            t_dia += dt_dia;
        }
        if (t >= t_out){
            put_snapshot(mass, pos, vel, n, t);
            t_out += dt_out;
        }
        if (t >= t_end)
            break;
    }

    delete[] acc;
    delete[] jerk;
}
```

Starting again with the argument list, we see that the mass array, as always, is defined as `const`, since we do not model a mechanism for mass loss for stars, nor do we (yet) allow collisions between stars, which could be followed by mergers that would produce a merger remnant with a mass equal to the sum of the masses of the two stars. The only place where we do not define the mass array as `const` is in the function `get_snapshot`, where the mass values are read in from the standard input stream. Note that the time `t` is passed by reference. In our current program, this is not necessary, since the value of `t` is not used in `main()`, where execution is halted immediately upon completion of the call to `evolve()`. However, in future extensions we may well add further commands in `main()`, and in that case it would be useful to have the value of the current time available.

As we have seen before, before we can enter the integration loop we have to start with an initial call to the function computing the accelerations and jerks. This function, `get_acc_jerk_pot_coll()` does what its name suggest: besides calculating accelerations and jerks, it also reports the value of the total potential energy of the system as well as the value of the time scale on which a 'collision' between particles can occur, *i. e.* a significant change of order unity in the local configuration of at least two particles. The latter information, stored in the variable `coll_time`, will be needed in the main integration loop in order to determine the size of the first time step. Accelerations and jerks are needed for the first part of the first integration time step, and the potential energy is used in the initial call to `write_diagnostics()`, following the first call to `get_acc_jerk_pot_coll()`.

In addition, if the user has specified the `init_out` flag to be true, the input values of the N-body system are echoed as they are on the output stream; the default behavior is to wait with output until some integration steps have been taken. This is a sensible default, since in many cases we are only interested in one final output snapshot, which can then served as the input for a later invocation of the integrator. If we invoke our program with the same value for the snapshot output interval as the duration of the run, we guarantee that only one final output will be made. An example usage of this type is:

```
|gravity> nbody_sh1 -d 0.01 -e 2 -o 40 -t 40 < data.in > data.out
```

Before entering the main integration loop, we schedule the next times for diagnostics and snapshot output, as well as the final halting time. The loop itself is an infinite loop, governed by the tautological `while (true)`, which is obviously always the case. The standard C/C++ trick to define an infinite loop uses an empty for loop, in the form `for(;;)`, but that expression is less transparent, whereas `while (true)` leaves no doubt as to it being an infinite loop. The only way to jump out of this infinite loop is at the end of the loop: when time progresses past the halting time `t_end`, the `break` statement causes control flow to continue past the loop.

The first time we enter the loop, the second `while` argument will be evaluated as `true`, unless one of the three values `dt_dia, dt_out` or `dt_tot` would be zero or negative, which would be nonsensical values. Ideally, we should check somewhere that all command line option arguments fall within reasonable ranges. Since in the present code we have already introduced so many new features, we will not include such a defensive programming style at this point. However, later on we will insist on checking all values which reach a program through an interface, such as presented by command line options. For now, we will live with the danger of a non-positive value for either `dt_dia` or `dt_out`, which combined with a positive value for `dt_tot` would lead to an infinite number of output attempts, without the time ever advancing.

With natural choices of parameters, the majority of loop cycles will not lead to any output. In those cases a new time step size is determined, and the function `evolve_step()` is called, which as the name implies will advance the system by one integration step, and in addition update the time by an amount `dt`. Sooner or later it will be time for output or for ending the run. In either case, the second `while` statement will evaluate as `false`, no integration time step will be taken and therefore the time will not be advanced either. Instead, the required output will be done and/or the integration will be finished altogether. If the run is not yet finished, the next cycle in the infinite loop will lead to another integration step, and so on.

Note the freeing up of memory for acceleration and jerk arrays, at the end of `evolve()`. As in the case of the memory allocation in `main()`, this is not strictly necessary, since the program is about to finish, but again it is certainly good form to include these statements here.

8.11 Taking a Single Integration Step

In the function `evolve_step()`, we encounter the first case where specific memory allocation and deallocation occurs more often than once during a run:

Code 8.10 (nbody_sh1.C: evolve_step)

```
/*-----------------------------------------------------------------------------
 *  evolve_step  --  takes one integration step for an N-body system, using the
 *                   Hermite algorithm.
 *-----------------------------------------------------------------------------
 */

void evolve_step(const real mass[], real pos[][NDIM], real vel[][NDIM],
                 real acc[][NDIM], real jerk[][NDIM], int n, real & t,
                 real dt, real & epot, real & coll_time)
{
    real (* old_pos)[NDIM] = new real[n][NDIM];
    real (* old_vel)[NDIM] = new real[n][NDIM];
    real (* old_acc)[NDIM] = new real[n][NDIM];
    real (* old_jerk)[NDIM] = new real[n][NDIM];

    for (int i = 0; i < n ; i++)
        for (int k = 0; k < NDIM ; k++){
          old_pos[i][k] = pos[i][k];
          old_vel[i][k] = vel[i][k];
          old_acc[i][k] = acc[i][k];
          old_jerk[i][k] = jerk[i][k];
        }

    predict_step(pos, vel, acc, jerk, n, dt);
    get_acc_jerk_pot_coll(mass, pos, vel, acc, jerk, n, epot, coll_time);
    correct_step(pos, vel, acc, jerk, old_pos, old_vel, old_acc, old_jerk,
                 n, dt);
    t += dt;

    delete[] old_pos;
    delete[] old_vel;
    delete[] old_acc;
    delete[] old_jerk;
}
```

As we have seen already in chapter 6, the Hermite code requires knowledge of the values of all four dynamical variables at the previous time step, indicated here by the prefix `old_`. Since we do not want to introduce global variables, and since these variables are not needed outside the context of the current function, we allocate the memory in the first four lines, and free up those memory locations in the last four lines. If we now would omit those last four lines, the resulting memory leak could let us run into serious trouble. For example, taking a million time steps with a hundred-body system would cause us to loose $4 * NDIM = 12$ words or $12 * 8 = 96$ bytes for each particle for each time step, leading to a total memory loss of $96 * 10^2 * 10^6$ bytes or roughly ten Gbytes, which may well be larger than the core memory of the computer at hand.

Again, it would be very good if we would check with each call to `new` whether there is still enough memory available. Since we do not do that here, a memory leak will suddenly cause the program to crash, without giving us any clue of where to look. Even using a debugger may not help, since the actual crash may well occur somewhere else, where a small amount of legitimate memory is requested, only to find out that all memory has just been exhausted elsewhere in the code. Once more, we will postpone but not neglect this type of defensive programming.

After the current values of the dynamical variables have been passed to the `old_` copies, we take the first half of a Hermite pass, in a call to `predict_step()`, followed by a recalculation of accelerations and jerks, as well as potential energy and collision time scale. We are then ready to complete the Hermite step through a call to `correct_step()`, and update the time `t`.

8.12 The Predictor Step

The first half of a Hermite step is particularly simple, nothing more than a rather short Taylor series development:

Code 8.11 (nbody_sh1.C: predict_step)
```
/*-----------------------------------------------------------------------------
 *  predict_step  --  takes the first approximation of one Hermite integration
 *                    step, advancing the positions and velocities through a
 *                    Taylor series development up to the order of the jerks.
 *-----------------------------------------------------------------------------
 */

void predict_step(real pos[][NDIM], real vel[][NDIM],
                  const real acc[][NDIM], const real jerk[][NDIM],
                  int n, real dt)
{
    for (int i = 0; i < n ; i++)
        for (int k = 0; k < NDIM ; k++){
            pos[i][k] += vel[i][k]*dt + acc[i][k]*dt*dt/2
                                      + jerk[i][k]*dt*dt*dt/6;
            vel[i][k] += acc[i][k]*dt + jerk[i][k]*dt*dt/2;
        }
}
```

Notice how much we can already read off from the way the arguments to `predict_step()` are declared: accelerations and jerks are passed as `const` variables, whereas positions and velocities are not. This implies that the latter two are updated, whereas the former two are used to provide information for the update, without being changed themselves. This of course is exactly what happens.

8.13 The Corrector Step

The second half of a Hermite step is again a Taylor series development, this time to a higher order than in the predictor step, even though this is not obvious from the way it is written. We refer to the discussion in the beginning of chapter 6, where the Taylor series character of the corrector step is made explicit. Here is the code:

Code 8.12 (nbody_sh1.C: correct_step)

```
/*-----------------------------------------------------------------------------
 *  correct_step  --  takes one iteration to improve the new values of position
 *                    and velocities, effectively by using a higher-order
 *                    Taylor series constructed from the terms up to jerk at
 *                    the beginning and the end of the time step.
 *-----------------------------------------------------------------------------
 */

void correct_step(real pos[][NDIM], real vel[][NDIM],
                  const real acc[][NDIM], const real jerk[][NDIM],
                  const real old_pos[][NDIM], const real old_vel[][NDIM],
                  const real old_acc[][NDIM], const real old_jerk[][NDIM],
                  int n, real dt)
{
    for (int i = 0; i < n ; i++)
        for (int k = 0; k < NDIM ; k++){
            vel[i][k] = old_vel[i][k] + (old_acc[i][k] + acc[i][k])*dt/2
                                      + (old_jerk[i][k] - jerk[i][k])*dt*dt/12;
            pos[i][k] = old_pos[i][k] + (old_vel[i][k] + vel[i][k])*dt/2
                                      + (old_acc[i][k] - acc[i][k])*dt*dt/12;
        }
}
```

8.14 Where All the Work is Done

We now arrive at the core function of `nbody_sh1.C`, where all the hard work is being done. In addition, this function is both the longest and the most complicated among the ten functions in the file. The main reason for the complexity is that we are trying to accomplish four things in one function, as the name indicates. While calculating accelerations and jerks are logically related, the calculation of the potential energy and the collision time is more a matter of convenience with little natural or logical relation to the calculation of the first two. The main reason for bundling these four operations is efficiency. Here is the code:

Code 8.13 (nbody_sh1.C: get_acc_jerk_pot_coll)

```
/*-----------------------------------------------------------------------------
 *  get_acc_jerk_pot_coll  --  calculates accelerations and jerks, and as side
 *                             effects also calculates potential energy and
 *                             the time scale coll_time for significant changes
 *                             in local configurations to occur.
 *
 *                                                  --                     --
 *                                                 |          -->  -->       |
 *               M                           M     |           r  . v        |
 *   -->          j    -->       -->          j    | -->        ji   ji -->  |
 *    a   ==  --------  r    ;    j   ==  -------- |  v   - 3 ---------  r   |
 *     ji     |-->  |3   ji        ji     |-->  |3 |   ji      |-->  |2   ji |
 *            | r   |                     | r   |  |           | r   |       |
 *            |  ji |                     |  ji |  |__         |  ji |     __|
 *
 *  note: it would be cleaner to calculate potential energy and collision time
 *        in a separate function.  However, the current function is by far the
 *        most time consuming part of the whole program, with a double loop
 *        over all particles that is executed every time step.  Splitting off
 *        some of the work to another function would significantly increase
 *        the total computer time (by an amount close to a factor two).
 *
 *  We determine the values of all four quantities of interest by walking
 *  through the system in a double {i,j} loop.  The first three, acceleration,
 *  jerk, and potential energy, are calculated by adding successive terms;
 *  the last, the estimate for the collision time, is found by determining the
 *  minimum value over all particle pairs and over the two choices of collision
 *  time, position/velocity and sqrt(position/acceleration), where position and
 *  velocity indicate their relative values between the two particles, while
 *  acceleration indicates their pairwise acceleration.  At the start, the
 *  first three quantities are set to zero, to prepare for accumulation, while
 *  the last one is set to a very large number, to prepare for minimization.
 *       The integration loops only over half of the pairs, with j > i, since
 *  the contributions to the acceleration and jerk of particle j on particle i
 *  is the same as those of particle i on particle j, apart from a minus sign
 *  and a different mass factor.
 *-----------------------------------------------------------------------------
 */

void get_acc_jerk_pot_coll(const real mass[], const real pos[][NDIM],
                           const real vel[][NDIM], real acc[][NDIM],
                           real jerk[][NDIM], int n, real & epot,
                           real & coll_time)
{
    for (int i = 0; i < n ; i++)
        for (int k = 0; k < NDIM ; k++)
            acc[i][k] = jerk[i][k] = 0;
    epot = 0;
    const real VERY_LARGE_NUMBER = 1e300;
    real coll_time_q = VERY_LARGE_NUMBER;      // collision time to 4th power
    real coll_est_q;                          // collision time scale estimate
                                              // to 4th power (quartic)
    for (int i = 0; i < n ; i++){
        for (int j = i+1; j < n ; j++){           // rji[] is the vector from
            real rji[NDIM];                       // particle i to particle j
            real vji[NDIM];                       // vji[] = d rji[] / d t
            for (int k = 0; k < NDIM ; k++){
                rji[k] = pos[j][k] - pos[i][k];
                vji[k] = vel[j][k] - vel[i][k];
            }
            real r2 = 0;                          // | rji |^2
            real v2 = 0;                          // | vji |^2
            real rv_r2 = 0;                       // ( rij . vij ) / | rji |^2
            for (int k = 0; k < NDIM ; k++){
                r2 += rji[k] * rji[k];
                v2 += vji[k] * vji[k];
                rv_r2 += rji[k] * vji[k];
            }
            rv_r2 /= r2;
            real r = sqrt(r2);                    // | rji |
            real r3 = r * r2;                     // | rji |^3

// add the {i,j} contribution to the total potential energy for the system:

            epot -= mass[i] * mass[j] / r;

// add the {j (i)} contribution to the {i (j)} values of acceleration and jerk:

            real da[NDIM];                        // main terms in pairwise
            real dj[NDIM];                        // acceleration and jerk
            for (int k = 0; k < NDIM ; k++){
                da[k] = rji[k] / r3;                          // see equations
                dj[k] = (vji[k] - 3 * rv_r2 * rji[k]) / r3;    // in the header
            }
            for (int k = 0; k < NDIM ; k++){
                acc[i][k] += mass[j] * da[k];                // using symmetry
                acc[j][k] -= mass[i] * da[k];                // find pairwise
                jerk[i][k] += mass[j] * dj[k];               // acceleration
                jerk[j][k] -= mass[i] * dj[k];               // and jerk
            }

// first collision time estimate, based on unaccelerated linear motion:

            coll_est_q = (r2*r2) / (v2*v2);
            if (coll_time_q > coll_est_q)
                coll_time_q = coll_est_q;
// second collision time estimate, based on free fall:

            real da2 = 0;                                  // da2 becomes the
            for (int k = 0; k < NDIM ; k++)                // square of the
                da2 += da[k] * da[k];                      // pair-wise accel-
            double mij = mass[i] + mass[j];                // eration between
            da2 *= mij * mij;                              // particles i and j

            coll_est_q = r2/da2;
            if (coll_time_q > coll_est_q)
                coll_time_q = coll_est_q;
        }
    }                                               // from q for quartic back
    coll_time = sqrt(sqrt(coll_time_q));            // to linear collision time
}
```

Notice the distribution of `const` declarations here, which is just the opposite from what we saw in `predict_step()` and `correct_step()`. In the latter two accelerations and jerk were `const` while positions and velocities were updated. Here the roles are reversed. In addition, there are two variables that are called by reference, `epot` and `coll_time`, which enable the information about potential energy and collision time to flow back to the calling function `evolve_step()` and from there back to `evolve()`, where they are used, as we have seen above.

After preparing the proper initial values for the four variables of interest, we enter the $\{i, j\}$ loop running over all particle pairs. As we have seen in the previous two chapters, we first compute a number of auxiliary quantities before we are ready to calculate first the contribution of a pair of particles to the potential energy and then their mutual contributions to each others acceleration and jerk.

At the end of the loop, we compute the two different collision time step estimates, in the same way we discovered at the end of the previous chapter. The first estimate follows the approximate of unperturbed linear motion, extrapolating current separation and rate of change of separation in order to guess when the particles will change their relative configuration substantially. The second estimate neglects the current rate of change of the pairwise separation, estimating instead the free-fall time of the two particles, in case they would start off at rest. In practice, the smaller of the two estimates provides a reasonably safe estimate for the time scale on which significant changes in configuration can occur.

8.15 Closing Logo

At the very end of our file, we add a simpler version of the gravitylab logo that we encountered at the top of the file:

Code 8.14 (nbody_sh1.C: logo_end)

```
/*----------------------------------------------------------------------------
 *                                                                   \\   o
 *  end of file:  nbody_sh1.C                                        /\\'  O
 *                                                                  /\     |
 *============================================================================
 */
```

It contains the name of the file, for consistency, and it guarantees that no part of the file has been truncated in a process of copying, editing or transmission over the net. While such mishaps are very rare nowadays, they still can occur occasionally, and it seems prudent to mark the intended end of the file. Meanwhile, our intrepid observer has changed directions from which to observe the world.

Part IV

Performing N-body Experiments

Chapter 9

Setting up a Star Cluster

9.1 A Model for a Star Cluster

Bob: Now we have a nifty N-body integrator. So what shall we do with it?

Carol: So far, we have only worked with two or three stars. How about throwing in a whole bunch?

Bob: Fine, but we have to decide how to set them up. We can't very well put them all on a circle. It wouldn't be stable anyway, as we have seen already with three particles.

Carol: I can't think of any other particular pattern either. Perhaps we should just pick random positions. But random within a certain region, I suppose, not all over the universe, since then they wouldn't interact.

Alice: How about letting nature guide us, and constructing a very simple model for a star cluster. Let us assume that a group of stars are born together in a small corner of a galaxy.

Bob: Sounds good. How shall we model that?

Alice: It would be simplest to group the stars in a sphere, and so assume a homogeneous distribution, i.e. constant density. In the general case, we should assign each star not only an initial position, but also an initial velocity. Let us again start with the simplest possible assumption, namely that of zero velocities for all stars. In other words, all stars are born at rest.

Carol: Wouldn't they all start falling to the middle, right away, under the influence of their mutually attractive forces of gravity?

Bob: I would think so. That means that the system will shrink at first. But what will happen next? Will it shrink forever, or start expanding, or oscillating, or what? We will have to do real experiments to find out.

Carol: First we have to construct the initial state, as described above. We have to introduce a random number generator, and then use that to choose a random position within a sphere. We may as well choose coordinates such that we will start with a sphere of unit radius.

9.2 Implementation: a Sphere in Cold Start

After a lengthy session, our friends came up with a code, tentatively called `sphere1a.C`. Having had the experience of writing the `nbody_sh1.C` code, it was not to heard to write the bulk of the code, with the `main` driver, the output routine, and so on. The only really new ingredients were the random number generator and the function `sphere` that gave all particles their initial conditions.

Alice: Well, I think we did a nice job. We're beginning to get good at this!

Bob: Yes, it's a lot less daunting that I had thought. Once you have one moderately complex code written and tested and documented, it gets a lot easier to write the next one. You can simply start with what you already have, and make a variation.

Carol: You mentioned documentation. We have written comments here and there, and at the head of each function, but perhaps we should get into the habit of providing a little more information, as an introduction, or a form of primer or manual, besides the code comments themselves?

Bob: By our guest!

Alice: Yes, I think you should do that, Carol. After all, you are supposed to have learned all that good stuff in your classes!

Carol: So much for making a good suggestion, only to get saddled with the work. Oh well, I actually like summarizing what I've done, once I'm happy with it, so let me give it a try. I'll keep it short, though, for now. Here goes.

We list here the full code for `sphere.C`, split up into its functions. First the information at the head of the file:

Code 9.1 (sphere1a.C: summary)

```
*=============================================================================
*
*  sphere.C:  generates initial conditions for a cold sphere
*
*-----------------------------------------------------------------------------
*
*  usage: sphere -n number_of_particles
*                   [-h (for help)] [-s random_number_generator_seed]
*
*        The number of particles has to be specified, since there is no
*        natural default value.  If no seed is specified for the random
*        number generator, a random value for the seed is chosen which
*        depends on the unix clock and will be different every second.
*
*        Example:  "sphere -n 3 > data.out"  will produce a file in the
*        following format, for particle number, time, and masses mi,
*        positions ri, and velocities vi for particles i:
*
*                    3
*                    0
*                    m1 r1_x r1_y r1_z v1_x v1_y v1_z
*                    m2 r2_x r2_y r2_z v2_x v2_y v2_z
*                    m3 r3_x r3_y r3_z v3_x v3_y v3_z
*-----------------------------------------------------------------------------
*
*  Our sphere has unit radius and constant density.  The total mass is unity,
*  so each particle will have mass 1/N, for N particles in total.  Each
*  particle will be sprinkled somewhere randomly within the unit sphere, with
*  initial velocity zero.
*-----------------------------------------------------------------------------
*
*   version 1:  Jan 2003   Piet Hut, Jun Makino
*-----------------------------------------------------------------------------
```

This comment block is followed by the #include directives and other general definitions, and then the declarations of all functions.

Code 9.2 (sphere1a.C: premain)

```
#include  <iostream>
#include  <cmath>                  // to include sqrt(), etc.
```

```
#include  <cstdlib>                  // formerly <cstdlib.h>; for atoi(), atof()
                                    // and rand(), srand()
#include  <unistd.h>                // for getopt()
#include  <time.h>                  // for time()
using namespace std;

typedef double  real;                      // "real" as a general name for the
                                           // standard floating-point data type

const int NDIM = 3;                        // number of spatial dimensions

bool read_options(int argc, char *argv[], int &n, int &seed);
real randunit(int seed);
real randinter(real a, real b);
bool sphere(real mass[], real pos[][NDIM], real vel[][NDIM], int n);
void put_snapshot(real mass[], real pos[][NDIM],
                  real vel[][NDIM], int n, real t);
```

In the *main* function, the reader is asked to provide a value for the desired number of particles, and optionally a seed for the random number generator. When you want to do a number of independent runs, it is better not to specify such a seed, since then each run will receive a different random seed. If then you want to rerun a particular run, you can specify the seed that happened to be used in that run. You can find the value of the seed used at the beginning of the output of each run, so even if you did not specify one, you will still be able to run any run again.

Note that in reality there is no such thing as a truly random number; instead we use a pseudo-random number, generated by an algorithm designed to generate numbers that are (hopefully) random-look-alike enough for our purpose. As a technical detail, two runs will only get a different random seed as long as they start more than one second apart, since the seed is constructed from the unix clock, read off in seconds.

Code 9.3 (sphere1a.C: main)

```
/*-----------------------------------------------------------------------------
 *  main  --  read in option values, invoke the model builder
 *-----------------------------------------------------------------------------
 */

int main(int argc, char *argv[])
{
    int n = 0;                         // N, number of particles; start with an
```

```
           // unphysical value, to allow us to force
           // the user to make a definite choice.
    int seed = 0;                     // seed for the random number generator;
           // a default of zero will be replaced by
           // the time taken from the unix clock.

    if (! read_options(argc, argv, n, seed))
        return 1;

    if (n <= 0){
cerr << "a value of N = " << n << " is not allowed." << endl;
return 1;
    }

    if (seed == 0)       /* no particular positive seed provided?           */
        seed = time(0);  /* then give a random value, different every second */

    cerr << "seed = " << seed << endl;

    randunit(seed);

    real * mass = new real[n];                  // masses for all particles
    real (* pos)[NDIM] = new real[n][NDIM];     // positions for all particles
    real (* vel)[NDIM] = new real[n][NDIM];     // velocities for all particles

    if (! sphere(mass, pos, vel, n))
return 1;
    put_snapshot(mass, pos, vel, n, 0);
    return 0;
}
```

The following function is similar to what we used in the previous chapter, for `nbody_sh1.C`.

Code 9.4 (sphere1a.C: read_options)

```
/*-----------------------------------------------------------------------------
 *  read_options  --  reads the command line options, and implements them.
 *
 *  note: when the help option -h is invoked, the return value is set to false,
 *        to prevent further execution of the main program; similarly, if an
 *        unknown option is used, the return value is set to false.
 *-----------------------------------------------------------------------------
 */

bool read_options(int argc, char *argv[], int &n, int &seed)
{
    int c;
    while ((c = getopt(argc, argv, "hn:s:")) != -1)
        switch(c){
            case 'n': n = atoi(optarg);
                      break;
            case 's': seed = atoi(optarg);
                      break;
            case 'h':
            case '?': cerr << "usage: " << argv[0]
                           << " [-h (for help)] [-n number_of_particles]\n"
                           << "         [-s random_number_generator_seed]"
                           << endl;
                      return false;      // execution stops after help or error
            }

    return true;                         // ready to continue program execution
}
```

Here is a pseudo-random number generator, which we took from the old Unix example, one which returns a real number between 0 and 1.

Code 9.5 (sphere1a.C: randunit)

```
/*-----------------------------------------------------------------------------
 *  randunit  --  returns a random real number within the unit interval
 *                note: based on      @(#)rand.c   4.1 (Berkeley) 12/21/80,
 *                      but returning a positive number smaller than unity.
 *
 *  note: to initialize the random number generator, invoke it with an nonzero
 *        argument, which will then become the seed;
 *        to run the random number generator, invoke it with argument 0.
 *-----------------------------------------------------------------------------
 */
real randunit(int seed)
{
    const real MAXN = 2147483647;  // the maximum value which rand() can return

    static int randx;
    if (seed)
        {
        randx = seed;
        return(0.0);        // to make the compiler happy, we return a value,
        }                   // even though it will not be used in this case
    else
        return((real)((randx = randx * 1103515245 + 12345) & 0x7fffffff)/MAXN);
}
```

This more general version of a pseudo-random number generator allows specification of the upper and lower bounds of the interval.

Code 9.6 (sphere1a.C: randinter)

```
/*-----------------------------------------------------------------------------
 *  randinter  --  returns a random real number within an interval [a,b]
 *                 by invoking  randunit() .
 *-----------------------------------------------------------------------------
 */
real  randinter(real a, real b)
    {
    return(a + (b-a)*randunit(0));
    }
```

The following function does all the work, places all N particles homogeneously distributed in a sphere with radius 1.

Code 9.7 (sphere1a.C: sphere)

```
/*-----------------------------------------------------------------------------
 *  sphere  --  constructs a homogeneous sphere, with unit radius and unit
 *              total mass, and particle velocities zero.
 *-----------------------------------------------------------------------------
 */

bool sphere(real mass[], real pos[][NDIM], real vel[][NDIM], int n)
{
    if (NDIM != 3){
```

```
cerr << "sphere: NDIM = " << NDIM << "not supported " << endl;
return false;
    }

    for (int i = 0; i < n; i++)
        mass[i] = 1.0 / (real) n;

    const real PI = 3.14159265358979323846;

    for (int i = 0; i < n; i++){
real r = pow(randinter(0.0, 1.0), 1.0/3.0);
real theta = randinter(0.0, PI);
real phi = randinter(0.0, 2*PI);
pos[i][0] = r*sin(theta)*cos(phi);
pos[i][1] = r*sin(theta)*sin(phi);
pos[i][2] = r*cos(theta);
for (int k = 0; k < NDIM; k++)
    vel[i][k] = 0;
    }
    return true;
}
```

The resulting N-body system is then ready for output.

Code 9.8 (sphere1a.C: put_snapshot)

```
/*-----------------------------------------------------------------------------
 *  put_snapshot  --  write a single snapshot on the output stream cout, in
 *                    the same format as described above for get_snapshot.
 *
 *  note: we use "const" here for the arguments, since they are not intended
 *        to be altered by a call to put_snapshot.
 *-----------------------------------------------------------------------------
 */

void put_snapshot(real mass[], real pos[][NDIM],
                  real vel[][NDIM], int n, real t)
{
    cout.precision(16);

    cout << n << endl;
    cout << t << endl;
    for (int i = 0; i < n ; i++){
        cout << mass[i];
        for (int k = 0; k < NDIM; k++)
            cout << ' ' << pos[i][k];
        for (int k = 0; k < NDIM; k++)
            cout << ' ' << vel[i][k];
        cout << endl;
    }
}
```

9.3 Testing, testing, . . .

Bob: Short and sweet, as they say. But perhaps a bit too short, especially on explaining how exactly we managed to put particles in a sphere using a, what did you call it, spherical coordinate system?

Alice: Yes, that is a coordinate system that is a lot more convenient for dealing with spheres than the usual Cartesian coordinate system is; the latter would of course be better if we would build a star cluster with the shape of a box.

Bob: I'd prefer the shape of a diamond over that of a box. But seriously, how I am to know for sure that the equations you gave me from your physics course are actually correct?

Carol: The answer they keep repeating in *my* courses: testing, testing, . . .

Bob: Okay, let's just run the program with a few particles, okay? By just looking at the numbers we may get an idea of whether we are at least approximately right. Alice, you're at the keyboard. Have a go at it!

Alice: Fine. How about ten particles?

```
|gravity> sphere1a -n 10
seed = 1063894860
10
0
0.1 0.4540028492193794 0.3498769522044687 0.7969703663542103 0 0 0
0.1 0.03987844095394553 -0.08119205595973951 -0.6615273048865606 0 0 0
0.1 -0.07696021131813481 -0.8459387776849843 0.3519731575017572 0 0 0
0.1 -0.03159529410632059 -0.2076818100526642 0.7862706899063225 0 0 0
0.1 0.02919115871639223 -0.228846199565124 0.2176806328285098 0 0 0
0.1 0.531084745743273 -0.1003377930427043 -0.05342966914946958 0 0 0
0.1 0.7662014881610754 -0.3620973663404717 0.1670139579482853 0 0 0
```

```
0.1 0.2522400702554067 -0.6696003994776 -0.08285809759917392 0 0 0
0.1 0.2785462467572272 0.06099199162060588 0.536196594372378 0 0 0
0.1 -0.510335018159465 0.06233465247130211 0.7635335045998497 0 0 0
|gravity>
```

Bob: At least the velocities are zero, and the position coordinates are smaller than one.

Carol: And larger than minus one. So far so good.

Alice: Not only that, if one of the numbers comes close to one in absolute value, the other numbers are smaller. Indeed, it seems that the sum of the squares of the position coordinates is always smaller than unity, as it should be: the particles neatly lie in a sphere.

Bob: But do they lie homogeneously in a sphere?

Carol: I see large and small numbers, positive and negative numbers, so that all suggest that things are pretty random.

Alice: Bob is right, we really should check for homogeneity. Random is not enough, since a random distribution could still be skewed one way or other.

Bob: How do you test a sphere for homogeneity?

Carol: I've done some video game programming, where the hardest part was always to get the 3D aspects right, especially rotations. They even had us learn about quaternions!

Bob: quarter whats?

Carol: Quaternions. Four-dimensional numbers, analogies of complex numbers, that are two-dimensional when expressed in terms of real components. Really neat stuff.

Bob: For now, let's keep it simple. How about just making a picture of it? It would be a two-dimensional projection, but if the three-dimensional distribution would be wildly inhomogeneous, you would think that it would show up even in projection.

Alice: Good idea. Let us use our gnuplot once more. But now we have to rethink how to get our data into the gnuplot program. In the past, we just took the output file, since the first two numbers on each line happened to be the x and y coordinates. But with our new fancy integrator, the first two lines contain the number of particles and the time, respectively, and then each line contains first the mass, and only then positions and velocities.

Carol: This is a good use for the `tail` command in unix.

Bob: Aren't we interested to modify the head part of the file, rather than the tail? I thought `tail -20`, for example, gives you only the last 20 lines of a file.

Carol: True, but there is a + option as well. If you type `tail +3`, you get the whole file, but starting at the third line, in other words skipping the first two lines; all the rest is than the 'tail' of the file.

Alice: So we will run our `sphere1a` code and then pipe the results into tail:

```
|gravity> sphere1a -n 10000 | tail +3 > test1a_10000.out
seed = 1063986021
|gravity> head -3 !$
head -3 test1a_10000.out
0.0001 0.5710408801016649 -0.3784997794699326 0.1014650450926402 0 0 0
0.0001 0.9204463477655462 0.1904132590373278 -0.1316466915348289 0 0 0
0.0001 -0.06164472062611391 0.1087990388945161 0.7259008589013708 0 0 0
|gravity> tail -3 test1a_10000.out
0.0001 0.01308756829247314 -0.06368169834698917 0.7647793506402052 0 0 0
0.0001 0.001461596716547156 0.1127385579637729 0.6880562520087973 0 0 0
0.0001 0.05993388699838697 -0.3903340594881157 0.6016407318791096 0 0 0
|gravity> wc !$
wc test1a_10000.out
  10000   70000  718902 test1a_10000.out
|gravity>
```

Bob: Good! Indeed there are ten thousand lines, and the first few and last few lines all have the right form. So we must be doing something right!

Alice: Now let us see how to get gnuplot to show the data from the second and third column, the x and y components of the positions.

```
|gravity> gnuplot
Terminal type set to 'x11'
gnuplot> plot "test1a_10000.out" using 2:3 notitle
gnuplot>
```

Carol: (fig. 9.1) Looks good to me. The particles are closer together in the center than near the edge, but that must be because we see more volume there, when we project the sphere on a two-dimensional screen.

Alice: Yes, near the edge your line of sight only intersect a small sliver of the sphere. Although the density is homogeneous, that still gives you only a few particles.

Bob: I guess we all believe our *sphere.C* program to be correct now, but just to make sure, let us plot another view, now that we have an easy way to do that.

Carol: You are hard to satisfy! But sure, why not. Let take the x and z components of the positions, this time.

Alice: Okay, this is how to produce that picture:

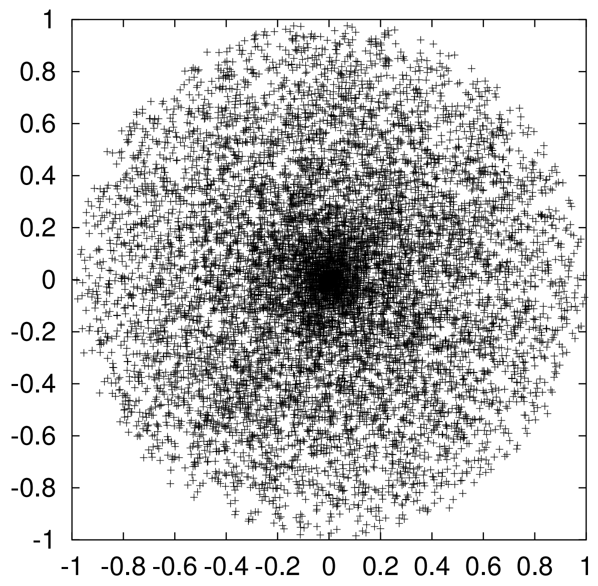

Figure 9.1: Output of `sphere1a`, with 10,000 particles, projected onto the $\{x, y\}$ plane

```
|gravity> gnuplot
Terminal type set to 'x11'
gnuplot> plot "test1a_10000.out" using 2:4 notitle
gnuplot>
```

Carol: (fig. 9.2) Wow! That's weird! What are those extra particles doing there, along the vertical line in the middle of the picture? That doesn't look at all like a projection of a homogeneous sphere.

Bob: (smiling) So much for chiding me about testing too much.

Carol: Well, I must say, I'm glad now that you were hard to satisfy. If you wouldn't have insisted we would have moved right along. Who knows how long it would have taken us to find that bug, whatever it is . . .

Alice: . . . if we would have ever found it. This is something scare about using computers: you know you have done something wrong, when you see that things don't look right, but the other way around does not hold. Things can look pretty right all right, and still be wrong.

Bob: That reminds me of a philosopher, was it Popper, who said that you can never verify a theory. You can only falsify it.

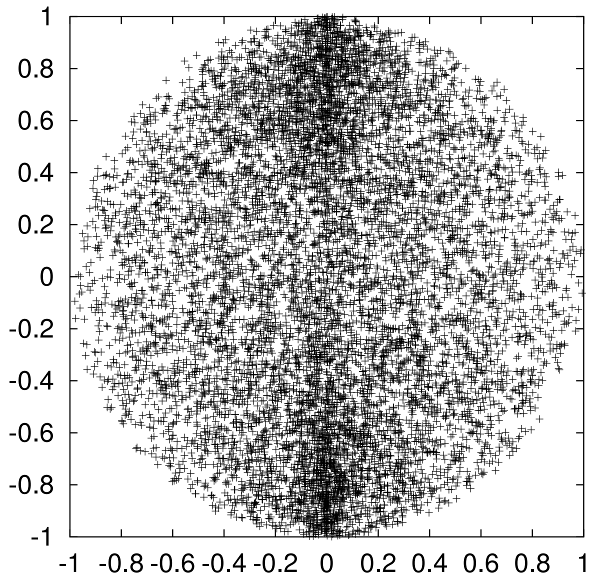

Figure 9.2: Output of `sphere1a`, with 10,000 particles, projected onto the $\{x, z\}$ plane

Carol: Yes, that was him. The best you can do is to make a theory more and more plausible. And yes, the same holds true for programming. There is a whole branch of computer science that deals with proving the correctness of code. However, they only talk about the internal correctness of a code. And that's the only thing they can possibly talk about, since no compiler can ever test whether or not the external information has been put in correctly. If the physics input is wrong, there is no logical way for any software program to find that out.

Bob: True, although here we are really working with math more than physics, assigning points to positions in space. But I see your point. Still, physics is presumably consistent, so even there one can apply consistency checks.

Carol: Yes, in a large system you must be right, but I'm not sure how far you can get with such a small code as `sphere.C`. It's a bit embarrassing, frankly, to get such a wrong result with so few lines of code.

Alice: Oh, believe me, I've made bigger mistakes in smaller codes. But let's put the philosophy aside for now, and let's fix things! But first I want to know precisely what's going on. Let's take somewhat fewer particles, and let us look at both picture again, just to have another way of approach this mystery.

Carol: Here you are, with a thousand particles, instead of ten thousand. First the x and *y* components of the positions.

```
|gravity> sphere1a -n 1000 | tail +3 > test1a_1000.out
seed = 1063987133
|gravity> gnuplot
Terminal type set to 'x11'
gnuplot> plot "test1a_1000.out" using 2:3 notitle
gnuplot>
```

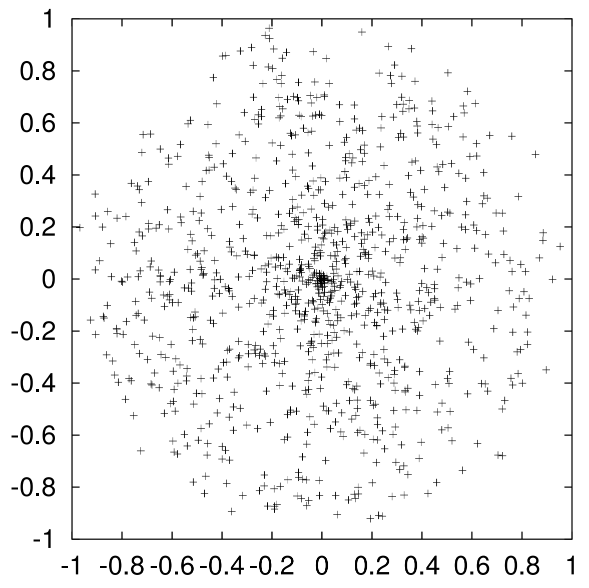

Figure 9.3: Output of `sphere1a`, with 1,000 particles, projected onto the $\{x, y\}$ plane

Bob: (fig. 9.3) Now this looks okay again, as before. However, I must say, I'm a little puzzled by the fact that there are so many particles in the center. To have the density petering off toward the edge seems reasonable, as we discussed a little earlier. But why should there be a spike in particle density near the center?

Carol: Good question. Let's first look at the $\{x, z\}$ view.

```
|gravity> gnuplot
Terminal type set to 'x11'
gnuplot> plot "test1a_1000.out" using 2:4 notitle
gnuplot>
```

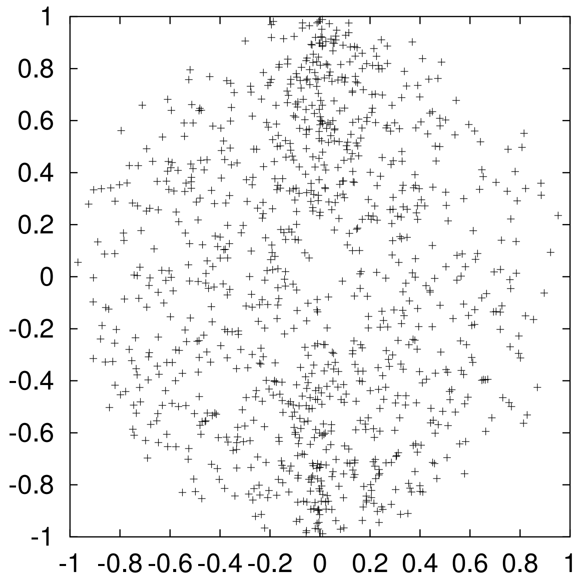

Figure 9.4: Output of `sphere1a`, with 1,000 particles, projected onto the $\{x, z\}$ plane

Carol: (fig. 9.4) Same problem: too many particles near the central vertical.

Bob: Aha! Now I can answer my own question. Of course! If there are really too many particles near the central vertical line in the $\{x, z\}$ plane, then that excess of particles will all be projected near the center of the $\{x, y\}$ projection.

Alice: Good point indeed! At least the bug we have found is consistent. I am glad we are dealing with only one problem, rather than too!

Carol: But I would prefer to have no bug at all. Let's have another look at the code.

9.4 Chasing the Bug

Bob: Well, the one part which I must admit I did not fully understand is where the angles are used to create particle positions in, what did you call it again?

Alice: spherical coordinates. A useful way to label particles by their distance from the center, and by the two angles needed in addition to locate them uniquely.

Carol: Useful, yes, if they do what they are supposed to do. Bob may be right, that is a likely place to have made a mistake. Let us look at how exactly we made that transformation. Here is how we assigned the positions and velocities to all the particles.

Code 9.9 (sphere1a_bugfragment.C: put_snapshot)

```
    for (int i = 0; i < n; i++){
real r = pow(randinter(0.0, 1.0), 1.0/3.0);
real theta = randinter(0.0, PI);
real phi = randinter(0.0, 2*PI);
pos[i][0] = r*sin(theta)*cos(phi);
pos[i][1] = r*sin(theta)*sin(phi);
pos[i][2] = r*cos(theta);
for (int k = 0; k < NDIM; k++)
    vel[i][k] = 0;
    }
```

Alice: At first sight, it looks rather reasonable. I clearly recognize the spherical coordinate angles in the position assignments: the sines and cosines of θ and ϕ are all in the right place.

Carol: And the ranges above those three lines are correct too: θ moves from the north pole to the south pole, so to speak, which makes an arc of 180^o, starting at $\theta = 0$, as it should; similarly ϕ moves all the way around along the equator, describing an angle of 360^o. What could possibly be wrong?

Bob: Why are the r values distributed according to a one-third power?

Alice: That's because the size of a volume element increases with the cube of r. For example, if you double the radius of a three-dimensional object, you make the volume eight times large. In order to keep the same density of points, you have to put eight times as many particles.

Bob: I see. You want to distribute the particles uniformly random in r^3, not in r. So this means that r, which is the third root of r^3, has to follow the third root of a uniformly distributed variable.

Alice: You got it.

Bob: And the angles don't need such a trick, because r does not change while we move around the latitude θ or the longitude ϕ.

Alice: Right you are, that is indeed . . . OOPS . . . no . . . correction . . . that is not indeed right, that is wrong. Now I see our mistake!

Carol: But look, if you go around the equator, how can it make any difference to where on the equator you are?

Alice: It doesn't, but try going to one of the poles.

Carol: Ah, now I get it. Moving from the equator to the pole, the lines of constant longitude converge together, and there is less and less room between them, the closer you get to the pole.

Alice: And mathematically you express that by the expression for a volume element dV as:

$$d\,V \;=\; r^2 \sin\theta \; dr \; d\theta \; d\phi \tag{9.1}$$

Alice: And for our application, it is easier to write this as

$$d\,V \;=\; d(\,r^3)\;\; d(\,cos\theta\,)\;\; d\phi \tag{9.2}$$

Bob: So this means that we have to invert the cosine factor for θ, just as we inverted the third power for r in the line above in our code.

Alice: Exactly. And the inverse of '$\cos\theta$' is '$\mathrm{acos}\,\theta$, sometimes also written as '$\arccos\theta$.' If we sprinkle particles along latitude in such a way that we do it as the arc cosine of θ, we will no longer overpopulate the poles.

Carol: Ah, so that is what cause the crowding at the poles, something that showed up only in the $\{x, z\}$ plot.

Bob: Because in the $\{x, y\}$ plot the poles are projected in the middle of the equatorial plane. The bug was less obvious there, although it did create the spike in the center, which we all overlooked in the first figure, which was a bit crowded with $10,000$ particles.

Alice: Well, I'm really glad we have found the bug, and it is easy to fix. Let us call the corrected code `sphere1.C`. The only difference is the line for assigning random number values to θ:

```
|gravity> diff sphere1.C sphere1a.C
197c197
<  real theta = acos(randinter(-1.0, 1.0));
---
>  real theta = randinter(0.0, PI);
|gravity>
\cba
```

```
\abc

\carol
Let's see whether things come out better.

\cba

\begin{small}
\begin{verbatim}
|gravity> sphere1 -n 1000 | tail +3 > test1_1000.out
seed = 1063988487
|gravity> gnuplot
Terminal type set to 'x11'
gnuplot> plot "test1_1000.out" using 2:3 notitle
gnuplot>
```

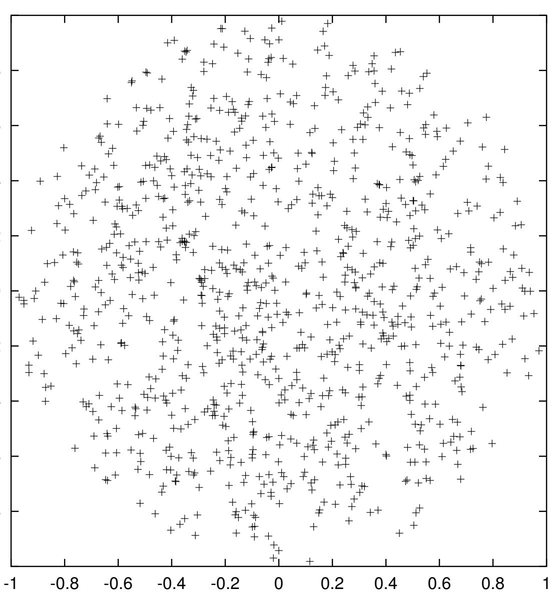

Figure 9.5: Output of `sphere1`, with 1,000 particles, projected onto the $\{x, y\}$ plane

Bob: (fig. 9.5) What a difference a line of code can make! No spike in the center any more. Everything looks so much smoother.

Carol: Indeed. There is only a mild edge effect, with somewhat fewer particles far away from the center, but the contrast between center and edge is much less than it was before.

Alice: This suggest that our central vertical line has cleared away. Let's make sure.

```
|gravity> gnuplot
Terminal type set to 'x11'
gnuplot> plot "test1_1000.out" using 2:4 notitle
gnuplot>
```

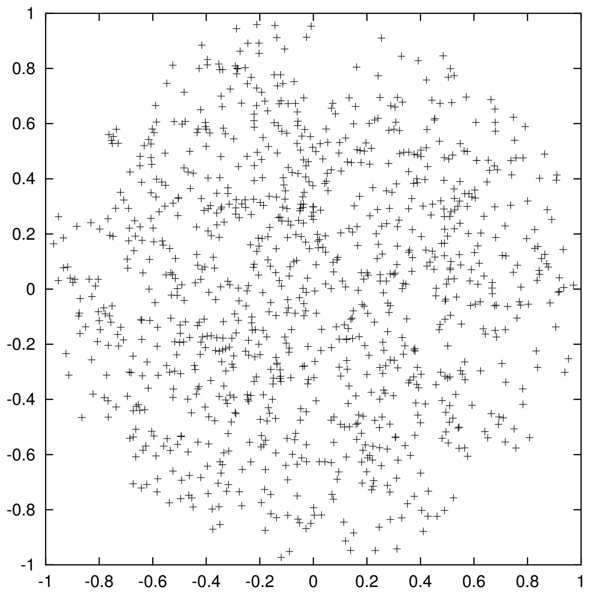

Figure 9.6: Output of `sphere1`, with 1,000 particles, projected onto the $\{x, z\}$ plane

Bob: (fig. 9.6) Indeed, a clean bill of health for `sphere1.C`

Carol: The moral of the story seems to be to try to test each module before you start putting things together, no matter how tempting it is to quickly write a few pieces and do some fun experiment with it.

Bob: Yes, and this must be equally true for real experiments as well as for virtual experiments in cyberspace. This reminds me of what I just heard from a friend of mine, a graduate student in physics. He talked about an experiment he was involved in. The wanted to test a new magnet, to be used to deflect a particle beam, in his case a beam of polarized electrons. After finishing the construction of the magnet setup, they bought a standard device that produced polarized electrons of the right type, hooked it up to the magnet, and switched everything on. If all had gone well, fine, but in their case things did not go well. The result is not what they expected, and they didn't know at that point whether there was something wrong with the magnet, with the gadget producing the electrons, with the connection between the two devices, or with the apparatus reading out the results (or even with the operator of the equipment who might have had a bad day and therefore made some kind of mistake).

Alice: It would have been worse, much worse actually, if all would have seemed fine at first, but only because some subtle bug in one of the devices produced an error that happened to be canceled more or less by another error in another part of the setup.

Bob: That was his conclusion too. He told me he realized that it is crucially important to test one by one all the steps in constructing an experimental setup. The first step should have been to make sure that the gadget generating the electrons is really doing what they expected and hoped it would do. In other words, they should first have carefully measured the properties of the electron beam *before* it entered the magnet setup. Only when they really felt comfortable that their gadget was working as advertised would it make sense to connect it to the magnet, and to start analyzing the beam coming out of the magnet.

Carol: I think the analogy is a good one, and that what is true for a traditional lab is true for our virtual lab as well. We did not really confront this problem earlier, since we dealt only with two or three particles, for which we wrote the initial conditions by hand. In contrast, we now have reached a point where for the first time we are generating initial conditions automatically. And anything that is done automatically can go wrong in all kind of unexpected ways.

Alice: Well, it is still possible to make mistakes when entering numbers by hand. I sure have done that more often than I'd like to admit.

Carol: Of course, but when you are dealing with a two-body or three-body system, you can track the orbits individually on the screen, and if you are careful, you will notice if the particles behave completely different from what you expected. But as soon as we start generating initial conditions automatically in large numbers, for a 25-body system for example, the situation is drastically different. It is no longer possible to look at the spaghetti of tangled orbits on a screen and to decide whether all 25 particles do even remotely what they are supposed to do. And indeed, the whole reason to use a computer, rather than pen and paper, is to solve a problem that is too complex to predict analytically beforehand. So you don't know in detail what to expect, and you may not notice if any part of the system behaves differently from how it should behave.

Bob: I bet your must have heard about modular testing in one of your computer science classes.

Carol: Yes, but frankly I thought they exaggerated a bit. The typical home work problems were often so simple that it was easy to see whether you got it right or not. But with our example today, I can see how essential it is to test each and every module you write, before hooking it up to other modules. If you test modules one by one, then it is most likely that a new bug in a complex of modules is generated by the latest module you have just added; or at least is caused by an unexpected interaction between that new module and what was already tested before. In both cases, you are far better off than if you suddenly have to test a whole conglomeration of modules, where errors could be hidden in any of them.

Alice: That last scenario is a nightmare, and a good way to quickly loose interest in computer programming.

Exercise 9.1 (Rejection technique)
Inspect the following code. It uses a completely different technique to generate initial conditions for a homogeneous sphere. It is simpler in that it does not use any trigonometric functions, but only the usual Cartesian coordinates. This sounds almost too good to be true. Can you figure out how the rejection technique works? Try it out and compare its results with that of `sphere1.C`.

Code 9.10 (sphere2.C)

```
              // Time-stamp: <2003-09-28 09:53:51 piet>
             //================================================================
            //                                                                |
           //           /__----__                         ........            |
          //       .           \                     ....:        :.          |
         //       :                 _\|/_         ..:                         |
        //       :                   /|\         :                     _\|/_  |
       //  ___   ___                  _____                      ___    /|\   |
      //  /     |   \    /\ \     / |   |   \  / |        /\    |   \         |
     //  |   __ |___/   /  \ \   /  |   |    \/  |       /  \   |___/         |
    //   |    | |  \   /____\ \ /   |   |    /   |      /____\  |   \     \/  |
   //     \___| |   \ /      \ V    |   |   /    |____ /      \ |___/     |   |
  //                                                                      /   |
 //              :                       _/|     :..                    |/    |
//                :..               ____/           :....          ..         |
/*   o   //          :.    _\|/_     /                   :........:           |
 *  O  `//\                 /|\                                               |
 *  |     /\                                                                  |
 *=============================================================================
 *
 *  sphere2.C:  generates initial conditions for a cold sphere
 *
 *_____________________________________________________________________________
 *
 *  usage: sphere2 -n number_of_particles
 *                   [-h (for help)] [-s random_number_generator_seed]
 *
 *        The number of particles has to be specified, since there is no
 *        natural default value.  If no seed is specified for the random
 *        number generator, a random value for the seed is chosen which
 *        depends on the unix clock and will be different every second.
 *
 *        Example:  "sphere2 -n 3 > data.out"  will produce a file in the
 *        following format, for particle number, time, and masses mi,
 *        positions ri, and velocities vi for particles i:
 *
 *                     3
 *                     0
 *                     m1 r1_x r1_y r1_z v1_x v1_y v1_z
 *                     m2 r2_x r2_y r2_z v2_x v2_y v2_z
 *                     m3 r3_x r3_y r3_z v3_x v3_y v3_z
 *-----------------------------------------------------------------------------
 *
 *  Our sphere has unit radius and constant density.  The total mass is unity,
 *  so each particle will have mass 1/N, for N particles in total.  Each
 *  particle will be sprinkled somewhere randomly within the unit sphere, with
 *  initial velocity zero.
 *-----------------------------------------------------------------------------
 *
 *    version 1:  Jan 2003   Piet Hut, Jun Makino
 *-----------------------------------------------------------------------------
 */

#include  <iostream>
#include  <cmath>                  // to include sqrt(), etc.
#include  <cstdlib>                // formerly <cstdlib.h>; for atoi(), atof()
                                   // and rand(), srand()
#include  <unistd.h>               // for getopt()
#include  <time.h>                 // for time()
using namespace std;

typedef double  real;                     // "real" as a general name for the
                                          // standard floating-point data type

const int NDIM = 3;                       // number of spatial dimensions

bool read_options(int argc, char *argv[], int &n, int &seed);
real randunit(int seed);
real randinter(real a, real b);
void sphere(real mass[], real pos[][NDIM], real vel[][NDIM], int n);
void put_snapshot(real mass[], real pos[][NDIM],
                    real vel[][NDIM], int n, real t);

/*-----------------------------------------------------------------------------
 *  main  --  read in option values, invoke the model builder
 *-----------------------------------------------------------------------------
 */

int main(int argc, char *argv[])
{
    int n = 0;                       // N, number of particles; start with an
           // unphysical value, to allow us to force
           // the user to make a definite choice.
    int seed = 0;                    // seed for the random number generator;
           // a default of zero will be replaced by
           // the time taken from the unix clock.

    if (! read_options(argc, argv, n, seed))
        return 1;

    if (n <= 0){
cerr << "a value of N = " << n << " is not allowed." << endl;
return 1;
    }

    if (seed == 0)       /* no particular positive seed provided?            */
        seed = time(0);  /* then give a random value, different every second */

    cerr << "seed = " << seed << endl;

    randunit(seed);

    real * mass = new real[n];                  // masses for all particles
    real (* pos)[NDIM] = new real[n][NDIM];     // positions for all particles
    real (* vel)[NDIM] = new real[n][NDIM];     // velocities for all particles

    sphere(mass, pos, vel, n);

    put_snapshot(mass, pos, vel, n, 0);
}

/*-----------------------------------------------------------------------------
 *  read_options  --  reads the command line options, and implements them.
 *
 *  note: when the help option -h is invoked, the return value is set to false,
 *        to prevent further execution of the main program; similarly, if an
 *        unknown option is used, the return value is set to false.
 *-----------------------------------------------------------------------------
 */

bool read_options(int argc, char *argv[], int &n, int &seed)
{
    int c;
    while ((c = getopt(argc, argv, "hn:s:")) != -1)
        switch(c){
            case 'n': n = atoi(optarg);
                      break;
            case 's': seed = atoi(optarg);
                      break;
            case 'h':
            case '?': cerr << "usage: " << argv[0]
                           << " [-h (for help)] [-n number_of_particles]\n"
                           << "         [-s random_number_generator_seed]"
                           << endl;
                      return false;      // execution stops after help or error
            }

    return true;                         // ready to continue program execution
}

/*-----------------------------------------------------------------------------
 *  randunit  --  returns a random real number within the unit interval
 *                note: based on      @(#)rand.c   4.1 (Berkeley) 12/21/80,
 *                      but returning a positive number smaller than unity.
 *
 *  note: to initialize the random number generator, invoke it with an nonzero
 *        argument, which will then become the seed;
 *        to run the random number generator, invoke it with argument 0.
 *-----------------------------------------------------------------------------
 */
real randunit(int seed)
{
    const real MAXN = 2147483647;  // the maximum value which rand() can return

    static int randx;

    if (seed)
        {
        randx = seed;
        return(0.0);        // to make the compiler happy, we return a value,
        }                   // even though it will not be used in this case
    else
        return((real)((randx = randx * 1103515245 + 12345) & 0x7fffffff)/MAXN);
}

/*-----------------------------------------------------------------------------
 *  randinter  --  returns a random real number within an interval [a,b]
 *                 by invoking  randunit() .
 *-----------------------------------------------------------------------------
 */
real  randinter(real a, real b)
    {
    return(a + (b-a)*randunit(0));
    }

/*-----------------------------------------------------------------------------
 *  sphere  --  constructs a homogeneous sphere, with unit radius and unit
 *              total mass, and particle velocities zero.
 *
 *  note: we use a rejection technique, in which we choose particle positions
 *        at random within a cube that encloses our sphere.  After each choice
 *        of a new position, we check whether that position lies within our
 *        sphere.  If it does, we accept that particle; if not, we reject it.
 *-----------------------------------------------------------------------------
 */

void sphere(real mass[], real pos[][NDIM], real vel[][NDIM], int n)
{
    for (int i = 0; i < n; i++)
        mass[i] = 1.0 / (real) n;

    for (int i = 0; i < n; i++){
real rsq;
do{
    rsq = 0;
    for (int k = 0; k < NDIM; k++){
pos[i][k] = randinter(-1.0, 1.0);
rsq += pos[i][k] * pos[i][k];
    }
}while (rsq > 1);
    }

    for (int i = 0; i < n; i++)
for (int k = 0; k < NDIM; k++)
    vel[i][k] = 0;
}

/*-----------------------------------------------------------------------------
 *  put_snapshot  --  write a single snapshot on the output stream cout, in
 *                    the same format as described above for get_snapshot.
 *
 *  note: we use "const" here for the arguments, since they are not intended
 *        to be altered by a call to put_snapshot.
 *-----------------------------------------------------------------------------
 */

void put_snapshot(real mass[], real pos[][NDIM],
                  real vel[][NDIM], int n, real t)
{
    cout.precision(16);

    cout << n << endl;
    cout << t << endl;
    for (int i = 0; i < n ; i++){
        cout << mass[i];
        for (int k = 0; k < NDIM; k++)
            cout << ' ' << pos[i][k];
        for (int k = 0; k < NDIM; k++)
            cout << ' ' << vel[i][k];
        cout << endl;
    }
}

/*-----------------------------------------------------------------------------
 *                                                                    \\   o
 *  end of file:  sphere2.C                                           /\\'  O
 *                                                                   /\     |
 *=============================================================================
 */
```

Chapter 10

A 25-body Example

10.1 A 25-body run

Bob: Time to start a real experiment! This will be our first chance to put our `nbody_sh1.C` engine to good use. I remember that we had a number of options built in, but for simplicity, let us just use the default settings.

Carol: Fine. Let's see. that means letting the system run for 10 time units, with energy diagnostics being output every time unit, and all that with a standard step size control parameter of 0.03.

```
|gravity> sphere -n 25 | ../chap8/nbody_sh1 > s25.out
seed = 1063318557
Starting a Hermite integration for a 25-body system,
  from time t = 0 with time step control parameter dt_param = 0.03  until time 10 ,
  with diagnostics output interval dt_dia = 1,
  and snapshot output interval dt_out = 1.
at time t = 0 , after 0 steps :
  E_kin = 0 , E_pot = -0.571622 , E_tot = -0.571622
                absolute energy error: E_tot - E_init = 0
                relative energy error: (E_tot - E_init) / E_init = -0
at time t = 1.00026 , after 4154 steps :
  E_kin = 1.57024 , E_pot = -2.14216 , E_tot = -0.571911
                absolute energy error: E_tot - E_init = -0.000289307
                relative energy error: (E_tot - E_init) / E_init = 0.000506117
at time t = 2.00002 , after 15063 steps :
  E_kin = 0.574657 , E_pot = -1.14657 , E_tot = -0.571912
                absolute energy error: E_tot - E_init = -0.000289794
                relative energy error: (E_tot - E_init) / E_init = 0.000506968
at time t = 3.00001 , after 24785 steps :
  E_kin = 0.735507 , E_pot = -1.30742 , E_tot = -0.571913
```

```
                absolute energy error: E_tot - E_init = -0.000290734
                relative energy error: (E_tot - E_init) / E_init = 0.000508612
at time t = 4.00002 , after 37188 steps :
  E_kin = 1.1189 , E_pot = -1.69081 , E_tot = -0.571914
                absolute energy error: E_tot - E_init = -0.000291627
                relative energy error: (E_tot - E_init) / E_init = 0.000510175
at time t = 5.00002 , after 60274 steps :
  E_kin = 1.07524 , E_pot = -1.64716 , E_tot = -0.571917
                absolute energy error: E_tot - E_init = -0.000294566
                relative energy error: (E_tot - E_init) / E_init = 0.000515317
at time t = 6 , after 109752 steps :
  E_kin = 1.68677 , E_pot = -2.25869 , E_tot = -0.571926
                absolute energy error: E_tot - E_init = -0.000304007
                relative energy error: (E_tot - E_init) / E_init = 0.000531832
at time t = 7 , after 445974 steps :
  E_kin = 1.9559 , E_pot = -2.52792 , E_tot = -0.572019
                absolute energy error: E_tot - E_init = -0.000396555
                relative energy error: (E_tot - E_init) / E_init = 0.000693737
at time t = 8 , after 805101 steps :
  E_kin = 0.579634 , E_pot = -1.15175 , E_tot = -0.572118
                absolute energy error: E_tot - E_init = -0.000495676
                relative energy error: (E_tot - E_init) / E_init = 0.000867139
at time t = 9 , after 1164240 steps :
  E_kin = 1.51451 , E_pot = -2.08673 , E_tot = -0.572217
                absolute energy error: E_tot - E_init = -0.000595069
                relative energy error: (E_tot - E_init) / E_init = 0.00104102
at time t = 10 , after 1523388 steps :
  E_kin = 2.16613 , E_pot = -2.73845 , E_tot = -0.572317
                absolute energy error: E_tot - E_init = -0.000694481
                relative energy error: (E_tot - E_init) / E_init = 0.00121493
|gravity>
```

Alice: This is fun. We are now really moving a bunch of stars around!

Carol: But the energy conservation is not great: at the end the accumulated energy error has reached more than 0.1% of the initial energy value.

Bob: Also, something funny happened around $t = 7$: until then each time unit of integration required at most a few tens of thousands of steps, but suddenly between $t = 6$ and $t = 7$ a few hundred thousand steps were needed. And each subsequent time unit turned out to be equally computationally expensive.

Carol: Let us first address the question of energy conservation, by doing a new run with a three times smaller accuracy parameter. To cut down a bit on output, let us report energy changes only once every two time units, as follows.

```
|gravity> sphere -n 25 | ../chap8/nbody_sh1 -d 0.01 -e 2 > s25a.out
seed = 1063319094
```

```
Starting a Hermite integration for a 25-body system,
  from time t = 0 with time step control parameter dt_param = 0.01  until time 10 ,
  with diagnostics output interval dt_dia = 2,
  and snapshot output interval dt_out = 1.
at time t = 0 , after 0 steps :
  E_kin = 0 , E_pot = -0.603695 , E_tot = -0.603695
                absolute energy error: E_tot - E_init = 0
                relative energy error: (E_tot - E_init) / E_init = -0
at time t = 2.00003 , after 30794 steps :
  E_kin = 0.603447 , E_pot = -1.20714 , E_tot = -0.603695
                absolute energy error: E_tot - E_init = -9.9949e-09
                relative energy error: (E_tot - E_init) / E_init = 1.65562e-08
at time t = 4.00005 , after 56055 steps :
  E_kin = 0.711514 , E_pot = -1.31521 , E_tot = -0.603695
                absolute energy error: E_tot - E_init = -1.09129e-08
                relative energy error: (E_tot - E_init) / E_init = 1.80768e-08
at time t = 6.00024 , after 91101 steps :
  E_kin = 0.309542 , E_pot = -0.913237 , E_tot = -0.603695
                absolute energy error: E_tot - E_init = -1.33915e-08
                relative energy error: (E_tot - E_init) / E_init = 2.21825e-08
at time t = 8.00012 , after 117285 steps :
  E_kin = 0.584493 , E_pot = -1.18819 , E_tot = -0.603695
                absolute energy error: E_tot - E_init = -1.38143e-08
                relative energy error: (E_tot - E_init) / E_init = 2.28829e-08
at time t = 10 , after 146608 steps :
  E_kin = 0.905758 , E_pot = -1.50945 , E_tot = -0.603695
                absolute energy error: E_tot - E_init = -1.48435e-08
                relative energy error: (E_tot - E_init) / E_init = 2.45878e-08
|gravity>
```

Bob: How curious! Indeed, the energy error is far smaller now, as we had hoped, but the total number of steps needed has been cut dramatically, by more than a factor ten, from more than one and a half million to less than hundred-fifty thousand. All this while we would naively have expected an *increase* by a factor three! What is going on?

Alice: We have started with a different random number seed. Perhaps there are large fluctuations from run to run in the difficulty of the integration? Let us do a few more runs, to compare. Since we are mostly interested in the bottom line, let us give output only at the end of the run, while throwing away the actual snapshot outputs.

```
|gravity> sphere -n 25 | ../chap8/nbody_sh1 -d 0.01 -e 10 > /dev/null
seed = 1063319209
Starting a Hermite integration for a 25-body system,
  from time t = 0 with time step control parameter dt_param = 0.01  until time 10 ,
  with diagnostics output interval dt_dia = 10,
  and snapshot output interval dt_out = 1.
```

```
at time t = 0 , after 0 steps :
  E_kin = 0 , E_pot = -0.527251 , E_tot = -0.527251
               absolute energy error: E_tot - E_init = 0
               relative energy error: (E_tot - E_init) / E_init = -0
at time t = 10 , after 802102 steps :
  E_kin = 0.665259 , E_pot = -1.19251 , E_tot = -0.527251
               absolute energy error: E_tot - E_init = -1.22308e-07
               relative energy error: (E_tot - E_init) / E_init = 2.31974e-07
|gravity> !!
sphere -n 25 | ../chap8/nbody_sh1 -d 0.01 -e 10 > /dev/null
seed = 1063319458
Starting a Hermite integration for a 25-body system,
  from time t = 0 with time step control parameter dt_param = 0.01  until time 10 ,
  with diagnostics output interval dt_dia = 10,
  and snapshot output interval dt_out = 1.
at time t = 0 , after 0 steps :
  E_kin = 0 , E_pot = -0.557163 , E_tot = -0.557163
               absolute energy error: E_tot - E_init = 0
               relative energy error: (E_tot - E_init) / E_init = -0
at time t = 10 , after 1140982 steps :
  E_kin = 0.723355 , E_pot = -1.28052 , E_tot = -0.557164
               absolute energy error: E_tot - E_init = -5.67934e-07
               relative energy error: (E_tot - E_init) / E_init = 1.01933e-06
|gravity>
```

Carol: You are right about the fluctuations, the run-to-run differences are huge.

Alice: Let us see whether we can reason our way to an answer. We are using an integrator that determines its step size automatically, depending on how close individual stars approach each other. This means that the expensive runs that take so long must contain situations where stars are very close together for very long times.

Bob: How can stars stay together so long? They don't stick together, do they? We are using point particles with zero diameter, so they can't collide.

Alice: Two stars don't stay together for long when they pass by each other occasionally. The only explanation I can think of is that two or more stars are captured in very tight bound systems, a situation that would force the shared time step to become very small, slowing down all other stars as well.

Bob: Ah, and as long as those stars stay together, the computation remains very expensive. And a very close double star must be stable, as long as no other star comes close enough to disrupt it.

Alice: And the smaller the double star is, the smaller a target it forms, and the less likely it is that it will be disrupted. At the same time, by being smaller such a double star forces the whole system on its knees, by pushing the shared time step to a very small value. These two properties conspire to produce large fluctuations!

Carol: An interesting hypothesis, but is it true? Clearly, we have to investigate this. By now I don't want to rely on plausibility any more, given how we almost made a big mistaken.

Alice: I would like to to make sure too. Now we have to find a way to check whether whether our system forms double stars – or binaries, as we astronomers tend to call them.

Bob: That reminds me of the binars, from Startrek; I guess they formed a type of double star too. But seriously, how about just making a movie, and see whether stars are pairing up. Can you do that with gnuplot?

Carol: I believe so. But we'll have to do some `awk` work to make it work.

Bob: Sounds pretty awkward to me, since I don't know anything about `awk`.

Carol: Yeah, it takes a while to get the hang of it. But it's a good investment of time, for sure: once you know your way around with `awk`, you can quickly manipulate all find of data without the need to write ponderous programs in C++ or similar languages. While `awk` is not a very fast language, in many cases we don't particularly care about speed, if it is just a matter of a few straightforward conversions.

Alice: Do I hear you volunteering for writing the scripts to let us allow to watch movies of our N-body run?

Carol: Okay, I'll do so. In fact, I have to do an `awk` project anyway, as homework for a class, so this will be it.

10.2 Making gnuplot movies

The next day Alice, Bob, and Carol get together again.

Bob: Well, Carol, did you do your homework?

Carol: Sure thing. Here is what I wrote. The first `awk` script here, `split_snapshot1.awk`, does what its name suggests: it takes a whole stream of snapshots, and puts it into different files.

Bob: This means that we can run our integrator `nbody_sh1.C` and separate all the snapshots that it produces, in order to make a movie in the form of a long row of individual photographs.

Carol: Yes, that's the idea. Here it is.

Code 10.1 (split_snapshot1.awk)

```
#!/bin/awk -f
#
# split_snapshot1.awk
#
# This awk script takes the output of an N-body integrator, in the form of
# a stream of snapshots that have been output at successive time intervals.
# It cuts the stream up into individual snapshots, and write each snapshot
# into an individual file.  If the snapshot stream resides in a file named
# "snapshots", then each file is given a name "snapshots.number" where
# "number" starts with the value 0 for the first snapshot (typically the
# initial condition at time 0), and then counts the subsequent snapshots
# as 0,1,2,...
#
# usage: awk -f split_snapshot1.awk snapshots
# However, this script is not really intended to be used standalone.
# Typically, use "makemovie.csh" which invokes this script.
#
# where "snapshots" is the name of the file containing the stream of snapshots.
#
BEGIN{
    isnap = 0;
    ip=0;
    print ARGC, ARGV[1];
    filename = ARGV[1];
}
{
    n[isnap]=$1;
    getline;
    t[isnap]=$1;
    fname = "tmp_" filename "." isnap
    for(i=0;i<n[isnap];i++){
        getline;
        print $0 > fname;
    }
    isnap++;
}
```

Bob: I'm afraid I can't follow the details, since I'm not familiar with `awk`. I will definitely read up on it, though, since it does seem like a powerful language. Everything is certainly a lot shorter than when you would have written it in C++!

Carol: Of course, it is a lot slower than C++, but in this case we don't care, because their is not much work involved.

Alice: But we probably wouldn't want to write an N-body integrator in awk, since there speed is of the essence. Though it would be an interesting exercise for Bob, to prove that he has mastered the language ; >).

Carol: Here is my second `awk` script, `makemoviescript1.awk`. What it does is to automatically generate all the gnuplot commands for each snapshot. So if we first use `split_snapshot1.awk` and then `makemoviescript1.awk` we can watch the movie.

Code 10.2 (makemoviescript1.awk)

```
#!/bin/awk -f
#
# makemoviescript1.awk
#
# This awk script takes a list of filenames, each containing one N-body
# snapshot.  The files are produced by the awk script split_snapshot1.awk
# from an N-body snapshot stream.  Commands are then generated for gnuplot
# to show a movie, at a rate of one frame per second, where successive
# frames show successive snapshots.
#
# usage: this script is not intended to be used standalone.
# instead, use "makemovie.csh" which invokes this script.
#
BEGIN{
    print "set size ratio 1";
    print "set xrange [-3:3]";
    print "set yrange [-3:3]";
}
{
    print "plot \"" $1 "\" using 2:3 notitle pointtype 1 pointsize 2";
    print "pause 1";
}
END{
   print "pause -1 \"Hit return to exit\""
}
```

Bob: At least I can get the idea, looking at the lines: this handy little script saves a lot of time, automatically generating the gnu plot commands, once a second, while giving you the time to see each frame separately.

Carol: And to make everything more userfriendly, I even wrote a `csh` script to invoke both `awk` scripts in the proper way and order. As you can see, all you have to do is to type `makemovie1.csh filename` where `filename` is the name of the output file from the integrator. Here is the script.

Code 10.3 (makemovie1.csh)

```
#!/bin/csh -f
#
# makemovie1.csh
#
# This csh script takes the output of an N-body integrator, in the form of
# a stream of snapshots that have been output at successive time intervals.
# As a result of running this script, a gnuplot movie will run at a rate of
# one frame per second, where successive frames show successive snapshots.
#
# Usage: makemovie1.csh snapshots
# where "snapshots" is the name of the file containing the stream of snapshots.
#
set snapfile = $1
rm  tmp_${snapfile}.*
awk -f split_snapshot1.awk $snapfile
set snapno = 0
set tmpsnap = tmp_${snapfile}
rm tmp1
while ( -e  ${tmpsnap}.${snapno} )
echo  ${tmpsnap}.${snapno}  >> tmp1
@ snapno = $snapno + 1
echo snapno = $snapno
end
awk -f makemoviescript1.awk tmp1 > tmp2.gnu
gnuplot  tmp2.gnu
```

Bob: I can't wait to try it out. May I?

Carol: Be my guest!

Bob: Let me try to see first what the command line arguments were for our integrator, by giving it the `-h` option. I want to switch off all the standard energy error messages, except for the last one. How many particles shall we use?

Alice: Let's stick with 25, for now.

Bob: Okay, first we'll do the run.

```
|gravity> ../chap8/nbody_sh1 -h
usage: ../chap8/nbody_sh1 [-h (for help)] [-d step_size_control_parameter]
         [-e diagnostics_interval] [-o output_interval]
         [-t total_duration] [-i (start output at t = 0)]
         [-x (extra debugging diagnostics)]
|gravity> ../chap9/sphere -n 25 | ../chap8/nbody_sh1 -e 10 > nbody1.out
```

```
seed = 1064696533
Starting a Hermite integration for a 25-body system,
  from time t = 0 with time step control parameter dt_param = 0.03  until time 10 ,
  with diagnostics output interval dt_dia = 10,
  and snapshot output interval dt_out = 1.
at time t = 0 , after 0 steps :
  E_kin = 0 , E_pot = -0.576237 , E_tot = -0.576237
                absolute energy error: E_tot - E_init = 0
                relative energy error: (E_tot - E_init) / E_init = -0
at time t = 10 , after 81537 steps :
  E_kin = 0.617884 , E_pot = -1.19413 , E_tot = -0.576245
                absolute energy error: E_tot - E_init = -7.40051e-06
                relative energy error: (E_tot - E_init) / E_init = 1.28428e-05
|gravity>
```

Bob: Not bad, a relative energy error of about one part in a hundred thousand. Now let's see whether Carol is a good movie director!

10.3 Performing an experiment

Carol: I don't vouch for anything, but let's see what comes up.

```
|gravity> makemovie1.csh nbody1.out
rm: No match.
2 nbody1.out
snapno = 1
snapno = 2
snapno = 3
snapno = 4
snapno = 5
snapno = 6
snapno = 7
snapno = 8
snapno = 9
snapno = 10
Hit return to exit
|gravity>
```

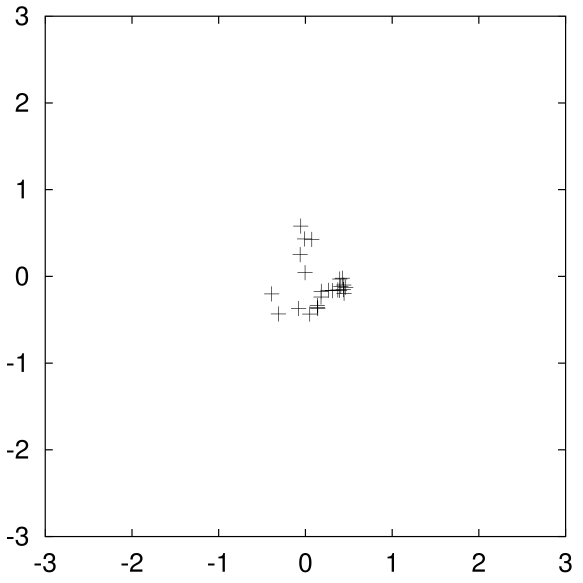

Figure 10.1: 1st frame of the first movie

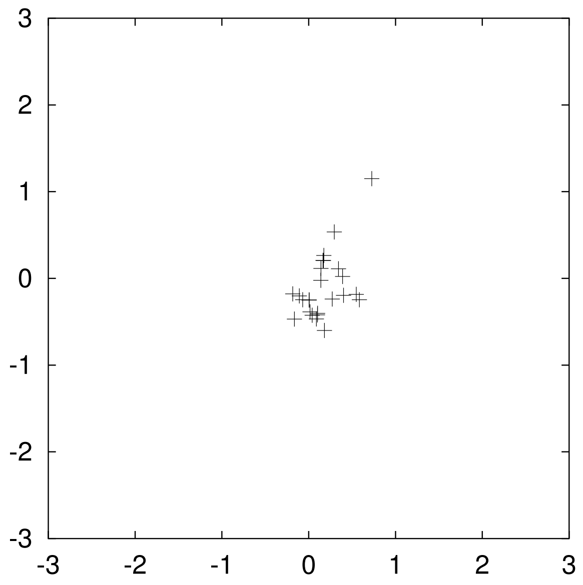

Figure 10.2: 2nd frame of the first movie

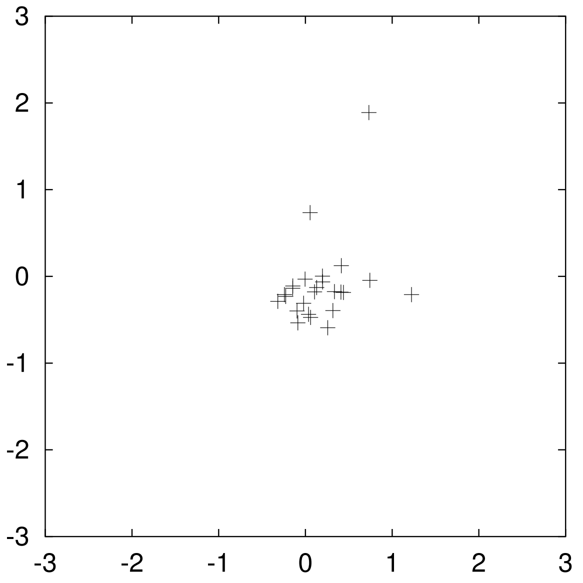

Figure 10.3: 3rd frame of the first movie

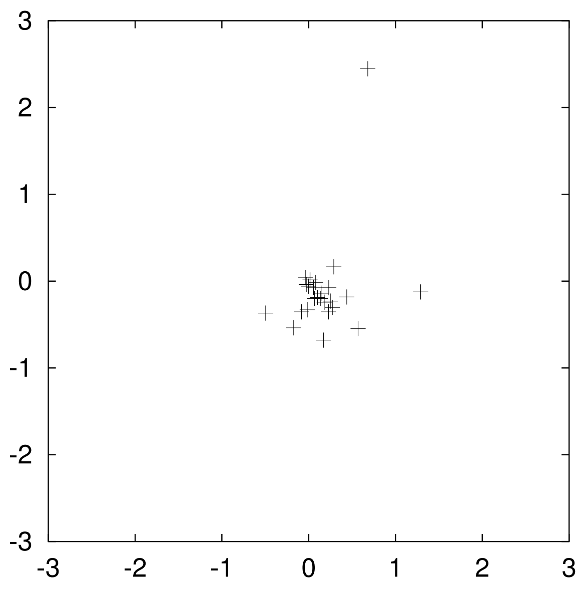

Figure 10.4: 4th frame of the first movie

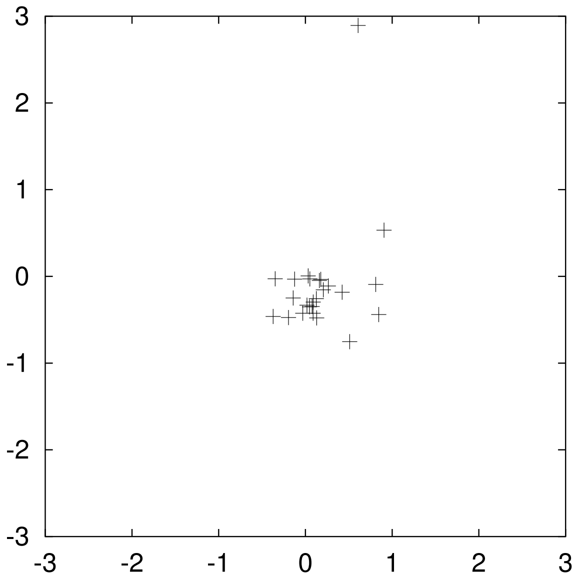

Figure 10.5: 5th frame of the first movie

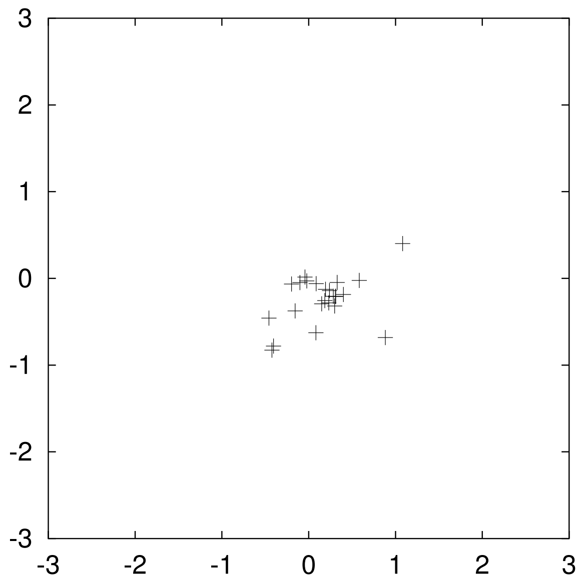

Figure 10.6: 6th frame of the first movie

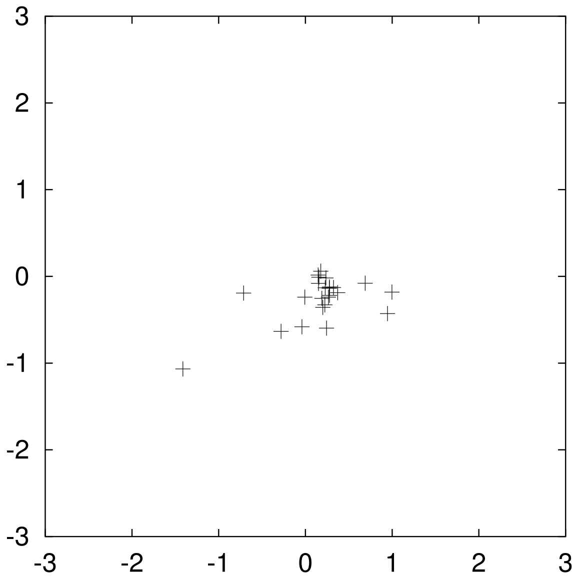

Figure 10.7: 7th frame of the first movie

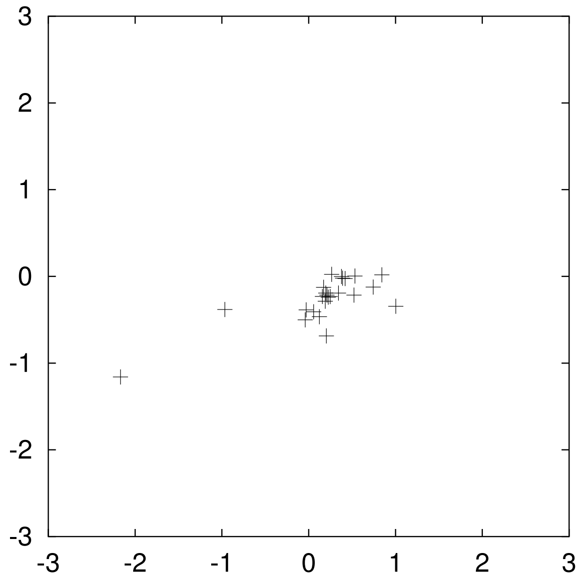

Figure 10.8: 8th frame of the first movie

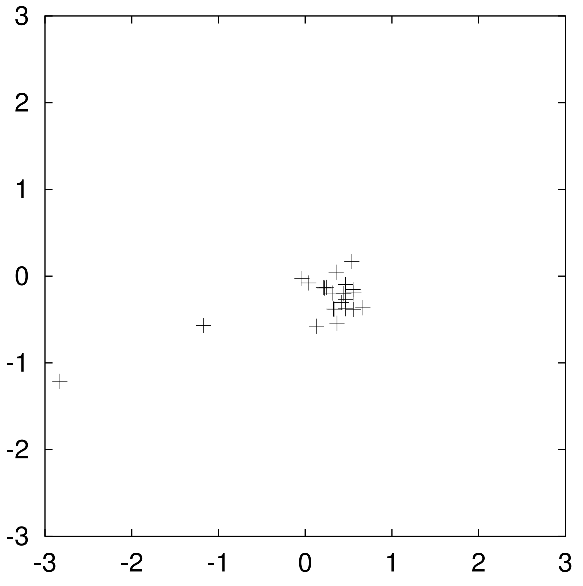

Figure 10.9: 9th frame of the first movie

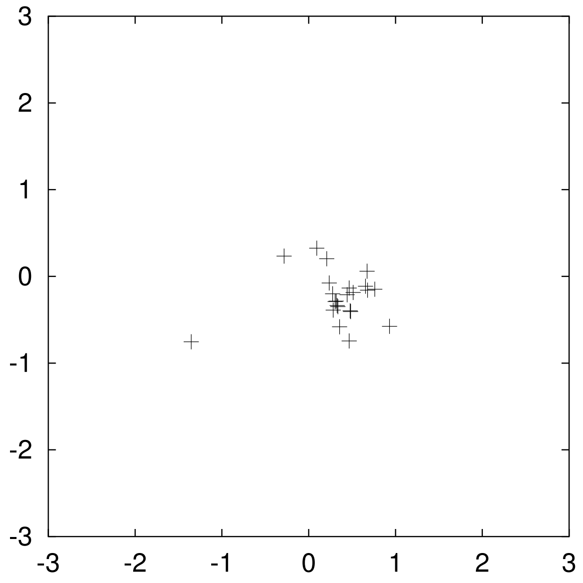

Figure 10.10: 10th frame of the first movie

Carol: I guess I passed, yes? You want my autograph?

Alice: Did you see individual stars escaping? Real astrophysics going on here, lady and gentleman!

Bob: Yes, I saw some stars boiling off into deep space, but I must admit, I couldn't quite follow what was happening in the inner core. It was worse than the old slapstick: everything was moving just too fast.

Carol: Lets get ten times as many snapshot frames, but decreasing the output time intervals from the default of one time unit to one tenth of a time unit.

Bob: Let's see, right, that was option `-o 0.1`, with `o` standing for output interval. If I work with this code a few more times I'll get familiar with all the options. Let's see what happens now.

```
|gravity> ../chap9/sphere -n 25 | ../chap8/nbody_sh1 -o 0.1 -e 10 > nbody2.out
seed = 1064697607
Starting a Hermite integration for a 25-body system,
  from time t = 0 with time step control parameter dt_param = 0.03  until time 10 ,
  with diagnostics output interval dt_dia = 10,
  and snapshot output interval dt_out = 0.1.
at time t = 0 , after 0 steps :
  E_kin = 0 , E_pot = -0.53095 , E_tot = -0.53095
                absolute energy error: E_tot - E_init = 0
                relative energy error: (E_tot - E_init) / E_init = -0
at time t = 10 , after 102258 steps :
  E_kin = 0.523367 , E_pot = -1.05432 , E_tot = -0.530958
                absolute energy error: E_tot - E_init = -8.39424e-06
                relative energy error: (E_tot - E_init) / E_init = 1.58099e-05
|gravity>
```

Carol: Again, a reasonable relative energy, comparable to what we had before. Now let us follow all those stars step by step.

```
|gravity> makemovie1.csh nbody2.out
rm: No match.
2 nbody2.out
snapno = 1
snapno = 2
snapno = 3
 . . . .
snapno = 98
snapno = 99
snapno = 100
Hit return to exit
|gravity>
```

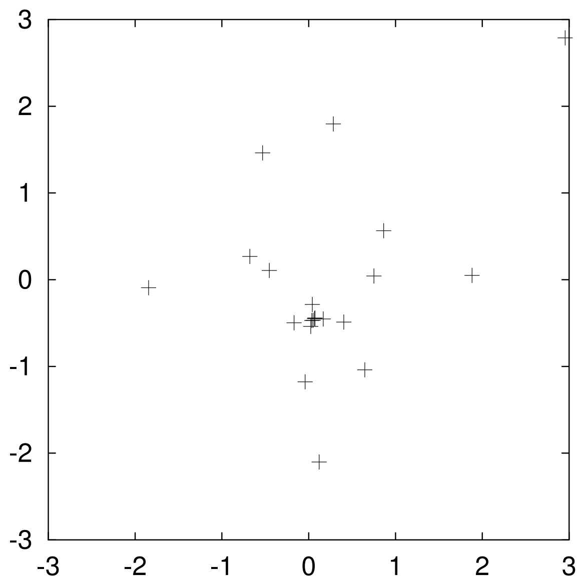

Figure 10.11: Last frame of the second movie, from the data stored in `nbody2.out`

Bob: That's more like it. Now we can really follow what's going on.

Carol: Well, yes and no. We can now follow the individual particles all right, but what about those binaries that Alice had promised us, as the explanation for the mysterious increase in computer time that we saw yesterday?

Alice: Well, I'm not sure. But my guess would be that those binaries are too small to see easily, on the scale we have plotted them. After all, they were so compute expensive exactly because the two stars of such a binary were bound in a short orbit – or so we hypothesized.

Bob: Even so, wouldn't we have clearly seen that there were places where two symbols on the screen almost overlapped?

Carol: I do believe I saw a few cases like that, but I can't be completely sure.

Alice: The other problem is that binaries are more massive than single stars, so they tend to sink to the center of a star cluster, where everything is more crowded. This is due to equipartition.

Bob: And what is equipartition?

Alice: It is an effect that is similar to what happens with molecules in the air. Carbon dioxide molecules, for example, are heavier than other molecules in air, and therefore they tend to stay lower.

Carol: Yes, I remember my high school teacher taking a beaker with carbon dioxide, which he had obtained from a pressure cylinder. And then he poured the invisible gas over a flame, dousing it immediately. It was very impressive.

Bob: But how come that carbon dioxide does not fall out of the air? I thought it was mixed rather thoroughly in the air around us.

Alice: That's because air currents tend to mix all gases, light or heavy. Even grass pollen, which are far more heavier than any gas molecule are carried along with the slightest breeze.

Carol: Tell me all about it. My allergy season is about to come up again, which is no fun.

Alice: However, when you carefully pour carbon dioxide, it will stay at the bottom of a room for a while. Similarly, a lighter gas, like helium, will flow up – which is the reason that a helium balloon indeed rises, even though the plastic surrounding the helium is heavier than air.

Bob: But what does all that have to do with binaries sinking to the center of a star cluster?

Alice: The combined gravitational force of all the stars points toward the center of the cluster. Heavier 'molecules', in our case binaries, therefore tend to sink to the center, just like carbon dioxide molecules tend to sink to the floor of a lab.

Carol: All nice and fine, but we're now speculating on and on about binaries, and we're not yet sure whether we've seen one. Aren't we running the danger of discussing how many angles can dance on the tip of a needle?

Bob: In order to dance together, perhaps angles have to form binaries too?

Carol: You're now imagining binaries everywhere. I want proof.

Alice: I have an idea. Rather than trying to peer in more and more details at those movie frames, why not build a binary detector!

Bob: A binary detector?

Alice: Yes! Just like physicists build elementary particle detectors, which they use in their accelerators to see what they've got, after smashing subatomic particles into each other.

Bob: That makes sense. After all, we have build a star accelerator, in the form of our N-body integrator `nbody_sh1.C`. Our movie was in fact a star detector. So it would be a logical next step to build a more sophisticated and specialized detector, one that would only be triggered by binaries.

Carol: That sounds like fun. But how to go about it?

Alice: The idea is simple. You just check all possible pairs of particles, and check to see which ones are bound together, forming a binary.

Carol: That's reasonable, since even with 25 particles, and 100 snapshots, you only have to check $25 * 25 * 100 = 62,500$ pairs, and with so many millions of floating point operations per second, even on laptops, that should be no problem.

Bob: After all, at every time step we had to compute the pairwise forces, which meant 625 pairwise forces far much more often than 100 times in our run. So time won't be a problem. But what type of algorithm would you use?

Alice: I'm afraid we would have to start with a reasonable amount of celestial mechanics. And rather than trying to derive all the relationships from scratch, let's get together again tomorrow. I'll bring my celestial mechanics book with me, and then we can figure it out.

Chapter 11

Fishing for Binaries

11.1 Binary Dynamics

The next day, our three friends get together again. Alice looks a bit sleepy, obviously has stayed up late the previous night.

Carol: Fortunately, we have a blackboard in this room.

Bob: And more important, we have Alice in the room!

Alice: But only a half-awake Alice. Yesterday evening it took me longer than I thought to figure everything out. What I found looks easy, it always does in the end, but I didn't realize how much I had forgotten about the classical dynamics course that I had followed in my freshman year.

Bob: Some day I'd like to know how you derived whatever you derived, but for now, I'm really eager to fish for binaries, so let's not worry too much about derivations. Perhaps you can just show us what you found.

Carol: I second that. Let's see whether it works first, and if it turns out to be useful, I'd love to learn more about the details.

Alice: Fine with me, and just as well, since I don't think I'm quite together enough to give a complete pedagogical derivation.

Alice walks to the blackboard, and writes down the equations for the energy E and angular momentum L for a two-body system. Here the masses of the two bodies are M_1 and M_2. The relative position vector, pointing from M_1 to M_2, is given by $\mathbf{r}$, and its time derivative is the relative velocity vector $\mathbf{v}$. As before, r is the absolute value of $\mathbf{r}$, a scalar, equal to the length of the vector $\mathbf{r}$, and similar v is the length of $\mathbf{v}$. The energy is given by the sum of the negative potential energy and the positive kinetic energy. The angular momentum points along the outer product, also called the cross product, of the relative position and velocity vectors. This implies that it points in a direction perpendicular to the orbit of the two bodies.

$$E = -\,G\,\frac{M_1M_2}{r} \;+\; \tfrac{1}{2}\,\frac{M_1M_2}{M_1+M_2}\,v^2 \tag{11.1}$$

$$L = \frac{M_1M_2}{M_1+M_2}\,\mathbf{r}\times\mathbf{v} \tag{11.2}$$

Note that we could have defined $\mathbf{r}$ equally well as pointing from M_2 to M_1, as long as $\mathbf{v}$ would still be the time derivative of $\mathbf{r}$. In the definition of the energy, the lengths of r and v would not change, and in the definition of the angular momentum, the two minus signs that would thus be introduced at the right would cancel each other.

These equations simplify considerably when we introducing the symbol μ for the reduced mass, defined as follows:

$$\mu = \frac{M_1M_2}{M_1+M_2} \tag{11.3}$$

we can now write the energy per unit of reduced mass as

$$\tilde{E} \equiv \frac{E}{\mu} = -\,G\,\frac{M_1+M_2}{r} \;+\; \tfrac{1}{2}\,v^2 \tag{11.4}$$

and similarly the angular momentum per unit of reduced mass as

$$\tilde{L} \equiv \frac{L}{\mu} = \mathbf{r}\times\mathbf{v} \tag{11.5}$$

Alice: The shape of a binary orbit is given by the values of the semi-major axis a and the eccentricity e. We can find those values in two steps. First we invert the expression that gives $\tilde{E}$ in terms of a, to obtain an expression for a:

$$\tilde{E} = -\,G\,\frac{M_1+M_2}{2a} \tag{11.6}$$

$$a = -\,G\,\frac{M_1+M_2}{2\tilde{E}} \tag{11.7}$$

Alice: And similarly, we invert the expression that gives $\tilde{L}$ in terms of a and e, to obtain an expression for e, or to keep it simple, for e^2 (we can always take the square root later):

$$\tilde{L}^2 = G\,(M_1 + M_2)a(1 - e^2) \tag{11.8}$$

$$e^2 = 1 - \frac{\tilde{L}^2}{G(M_1 + M_2)a} \tag{11.9}$$

Alice: The interpretation is as follows: as the name suggests, a is half the length of the longest axis of the ellipse that describes the relative orbit of the two bodies; in this way $2a$ is size of the orbit. The eccentricity e indicates the relative displacement of the focus of the orbit from the center of the orbit. The closest approach between the two bodies occurs at a distance of ae, also called the pericenter distance, while the largest separation occurs at a distance of $a(1 - e)$, the apocenter distance.

11.2 Finding Binaries

Carol: Thanks, Alice, let's code it up.

While Bob and Alice look on, and occasionally give some comments, Carol produces the following code. First, she summarized the structure and usage:

Code 11.1 (find_binaries1.C: summary)

```
*============================================================================
*
*  find_binaries1.C:  a tool to determine which pairs of particles are mutually
*                    bound, i.e. form a (temporary) binary.
*                    The program accepts a stream of N-body snapshots, and
*                    outputs a list of binaries for each snapshot.
*
*  note: in this first version, all functions are included in one file,
*        without any use of a special library or header files.
*----------------------------------------------------------------------------
*
*  usage: find_binaries1 [-h (for help)]
*
*         Input/output are read/written from/to the standard i/o streams.
*         Since there are no options, the code can simply be run by
*         specifying the input file for the N-body snapshots:
*
*            find_binaries1 < data.in
*
*         This will produce data on the screen.  In order to capture the data
*        in an output file "binarylist.out", use:
*
*           find_binaries1 < data.in > binarylist.out
*-----------------------------------------------------------------------------
*
*  External data format:
*
*     The program expects input of a single snapshot of an N-body system,
*     in the following format: the number of particles in the snapshot n;
*     the time t; mass mi, position ri and velocity vi for each particle i,
*     with position and velocity given through their three Cartesian
*     coordinates, divided over separate lines as follows:
*
*                     n
*                     t
*                     m1 r1_x r1_y r1_z v1_x v1_y v1_z
*                     m2 r2_x r2_y r2_z v2_x v2_y v2_z
*                     ...
*                     mn rn_x rn_y rn_z vn_x vn_y vn_z
*
*     Output for each snapshot consists of a list of binaries, each one
*     defined by the identity {id1,id2} of its members and its semimajor
*     axis abin and eccentricity ebin, as follows:
*
*                     time = t
*                     star1 = id1  star2 = id2  a = abin  e = ebin
*
*     Each line "time = t" is followed by zero, one, or more binary
*     listings, depending on how many binaries there are found in the
*     corresponding snapshot.
*
*  Internal data format:
*
*     The data for an N-body system is stored internally as 1-dimensional
*     arrays for the masses and densities, and 2-dimensional arrays for the
*     positions and velocities of all particles.
*-----------------------------------------------------------------------------
*
*    version 1:  Sep 2003   Piet Hut, Jun Makino
*-----------------------------------------------------------------------------
```

followed by the usual include stuff and declarations, etc.:

Code 11.2 (find_binaries1.C: premain)

```
#include  <iostream>
#include  <cmath>                           // to include sqrt(), etc.
#include  <cstdlib>                         // for atoi() and atof()
#include  <unistd.h>                        // for getopt()
using namespace std;

typedef double  real;                       // "real" as a general name for the
                                            // standard floating-point data type

const int NDIM = 3;                         // number of spatial dimensions

bool read_options(int argc, char *argv[]);
int get_snapshot(real * * mass, real (* * pos)[NDIM], real (* * vel)[NDIM],
                 int & n, real & t);
void delete_snapshot(const real mass[], const real pos[][NDIM],
                     const real vel[][NDIM]);
bool find_binaries(real mass[], real pos[][NDIM], real vel[][NDIM], int n,
                   real t);
```

The main part of the code is rather simple: one by one the snapshots are read in and a search for binaries is done, by invoking the function `find_binaries`:

Code 11.3 (find_binaries1.C: main)

```
/*-----------------------------------------------------------------------------
 *  main  --  reads option values, and starts a loop; in each round of the loop
 *            a new shapshot is read, the binaries are found and reported.
 *-----------------------------------------------------------------------------
 */

int main(int argc, char *argv[])
{
    if (! read_options(argc, argv))
        return 1;               // halt criterion detected by read_options()

    real * mass;                // masses for all particles
    real (* pos)[NDIM];         // positions for all particles
    real (* vel)[NDIM];         // velocities for all particles

    int n;                      // N, number of particles in the N-body system
    real t;                       // time

    while(get_snapshot(&mass, &pos, &vel, n, t)){
        if (! find_binaries(mass, pos, vel, n, t))
    return 1;
        delete_snapshot(mass, pos, vel);
    }
    return 0;
}
```

There are no command line arguments, apart from the standard help option:

Code 11.4 (find_binaries1.C: read_options)

```
/*-----------------------------------------------------------------------------
 *  read_options  --  reads the command line options, and implements them.
 *
 *  note: when the help option -h is invoked, the return value is set to false,
 *        to prevent further execution of the main program; similarly, if an
 *        unknown option is used, the return value is set to false.
 *-----------------------------------------------------------------------------
 */

bool read_options(int argc, char *argv[])
{
    int c;
    while ((c = getopt(argc, argv, "h")) != -1)
        switch(c){
            case 'h':
            case '?': cerr << "usage: " << argv[0]
                           << " [-h (for help)]"
                           << endl;
                      return false;      // execution stops after help or error
            }

    return true;                         // ready to continue program execution
}
```

We have seen before how to read in a snapshot:

Code 11.5 (find_binaries1.C: get_snapshot)

```
/*-----------------------------------------------------------------------------
 *  get_snapshot  --  reads a single snapshot from the input stream cin
 *
 *  note: memory allocation for masses, positions and velocities is done here
 *        after reading in the number of particles (n).
 *        If the end of file is reached, get_snapshot() returns 0;
 *        after successful completion, get_snapshot() returns 1.
 *-----------------------------------------------------------------------------
 */

int get_snapshot(real * * mass, real (* * pos)[NDIM], real (* * vel)[NDIM],
 int & n, real & t)
{
    cin >> n;
    if (cin.fail())
        return 0;
    cin >> t;

    *mass = new real[n];                  // masses for all particles
    *pos = new real[n][NDIM];             // positions for all particles
    *vel = new real[n][NDIM];             // velocities for all particles

    for (int i = 0; i < n ; i++){
        cin >> (*mass)[i];                        // mass of particle i
        for (int k = 0; k < NDIM; k++)
            cin >> (*pos)[i][k];                  // position of particle i
        for (int k = 0; k < NDIM; k++)
            cin >> (*vel)[i][k];                  // velocity of particle i
    }
    return 1;
}
```

Also, we have seen before how to delete in a snapshot once it is no longer needed:

Code 11.6 (find_binaries1.C: delete_snapshot)

```
/*-----------------------------------------------------------------------------
 *  delete_snapshot  --  frees up the memory that was allocated to the masses,
 *                       positions, and velocities for the particles in a
 *                       snapshot.
 *-----------------------------------------------------------------------------
 */

void delete_snapshot(const real mass[], const real pos[][NDIM],
     const real vel[][NDIM])
{
    delete[] mass;
    delete[] pos;
    delete[] vel;
}
```

Finally, here is the code where the real work is done. The equations written above are used directly to find and report the values for the semi-major axis a and eccentricity e. Each particle pair that has a negative relative energy is considered to be a bound system, in other words a binary:

Code 11.7 (find_binaries1.C: find_binaries)

```
/*-----------------------------------------------------------------------------
 *  find_binaries  --  for each particle pair, check whether their relative
 *                     energy is negative.  If so, determine the semi-major
 *                     axis a and eccentricity e, and print those.  The values
 *                     for a and e are determined from the energy and angular
 *                     momentum of the particle pair, expressed in units of
 *                     their reduced mass.
 *-----------------------------------------------------------------------------
 */

bool find_binaries(real mass[], real pos[][NDIM], real vel[][NDIM], int n,
                   real t)
{
    cout << "time = " << t << endl;
    for (int i = 0; i < n-1 ; i++){
        for (int j = i+1; j < n ; j++){
            real delr[NDIM];
            real delv[NDIM];
            real delr_sq = 0;
            real delv_sq = 0;
            for (int k = 0; k < NDIM ; k++){
                delr[k] = pos[j][k]-pos[i][k];
                delv[k] = vel[j][k]-vel[i][k];
                delr_sq += delr[k] * delr[k];
                delv_sq += delv[k] * delv[k];
            }
            real m_tot = mass[i] + mass[j];
            real Etilde;        // energy per unit of reduced mass
            Etilde = - m_tot / sqrt(delr_sq) + 0.5 * delv_sq;
            if (Etilde < 0.0){
                real Ltilde_sq = 0.0;    // square of the value of the angular
                                         // momentum per unit of reduced mass
                if (NDIM == 2){
                    real r_cross_v_component = delr[0]*delv[1]-delr[1]*delv[0];
                    Ltilde_sq = r_cross_v_component * r_cross_v_component;
                }
                else if (NDIM == 3){
                    real r_cross_v_component = delr[0]*delv[1]-delr[1]*delv[0];
                    Ltilde_sq = r_cross_v_component * r_cross_v_component;
                    r_cross_v_component = delr[1]*delv[2]-delr[2]*delv[1];
                    Ltilde_sq += r_cross_v_component * r_cross_v_component;
                    r_cross_v_component = delr[2]*delv[0]-delr[0]*delv[2];
                    Ltilde_sq += r_cross_v_component * r_cross_v_component;
                }
                else {
                    cerr << "find_binaries: NDIM = " << NDIM
                         << "not supported " << endl;
                    return false;
                }
                real a = - 0.5 * m_tot / Etilde;
                real e_sq = 1 - Ltilde_sq/(m_tot * a);
                real e;
                if (e_sq > 0.0)
                    e = sqrt(e_sq);
                else               // in case of roundoff errors that may
                    e = 0.0;       // lead to e_sq being slightly negative

                cout << "star1 = " << i << "  star2 = " << j
                     << "  a = " << a << "  e = " << e << endl;
            }
        }
    }
    return true;
}
```

Bob: I'm really curious now what will happen. How about starting carefully, in a number of easy steps, since we're entering new and unknown terrain here.

Carol: Now you begin to talk like a computer scientist! But I agree, let's feel our way around, to make sure we understand what we're doing.

Alice: The obvious first step would be to look at the output of `sphere1.C`, to see whether there would be any binaries already present.

Bob: Let's choose a particle seed, so that we can reproduce our results.

```
|gravity> ../chap9/sphere1 -s 42 -n 25 | find_binaries1
seed = 42
time = 0
star1 = 0  star2 = 1  a = 0.856428  e = 1
star1 = 0  star2 = 2  a = 0.397074  e = 1
star1 = 0  star2 = 3  a = 0.331345  e = 1
 . . . .
star1 = 22  star2 = 23  a = 0.683746  e = 1
star1 = 22  star2 = 24  a = 0.562686  e = 1
star1 = 23  star2 = 24  a = 0.330094  e = 1
|gravity>
```

Carol: Wow, that went by too quick, but there sure were a lot of so-called binaries. How many were there altogether?

Bob: Let see. Here is the answer:

```
|gravity> ../chap9/sphere1 -s 42 -n 25 | find_binaries1 | wc
seed = 42
    301    3603   12941
|gravity>
```

Bob: We know that there is a line `time = 0`, so if we leave that line out, we realize that we have just found a grand total of 300 binaries. And all that with only 25 stars ?!?

Alice: I see what the problem is. We begin with a cold start, remember? All velocities are set equal to zero, initially.

Carol: So this means the kinetic energy of any and every pair of stars is identically zero.

Bob: And therefore the total energy is negative, since potential energy is always negative. So we should have found $25 * 25 = 625$ binaries, no?

Alice: You can't be a binary with yourself, so there are only $25 * 24$ candidates. However, that still leaves us with 600 binaries.

Carol: Ah, but remember, I tested each pair only once, independent of whether I started with the first or the second star. In other words, for the first star, I did 24 tests with all other stars; for the second star I needed only to test binarity with respect to the other 23 stars; and so on, until the next-to-last star only had to be tested with respect to the last star, and we were done. This means that there are only $(25 * 24)/2 = 300$ star pairs that were tested.

Bob: Indeed just what we found. Phew, I'm relieved. At least the output makes sense, even this is not quite what we intended.

Alice: Notice the typical semi-major axis values in our output: all these binary-wanna-be systems claim to have a values spread more or less equally between 0 and 1. In practice, any stable binary would have an orbit far smaller than the size of the system.

Carol: How about adding an option to the code through which we exclude all binaries that are larger than a certain cut-off value?

Bob: That should help, yes, let's try that.

11.3 Finding Tight Binaries

Bob and Alice again watch Carol, who quickly copies the file `find_binaries1.C` to a new file `find_binaries2.C`, and then starts editing the new file, in order to add a cut-off option.

Carol: Here is the new main part. As you can see, I have added the cut-off value `a_max` for the semi-major axis of the binaries to be reported as an optional argument. Running `find_binaries2.C` in the same way as `find_binaries1.C` should give the same output, since the default cutoff is a `VERY_LARGE_NUMBER`, in our case equal to 10^{300}, which means that no binary found will be rejected.

Code 11.8 (find_binaries2.C: main)

```
/*-----------------------------------------------------------------------------
 *  main  --  reads option values, and starts a loop; in each round of the loop
 *            a new shapshot is read, the binaries are found and reported.
 *-----------------------------------------------------------------------------
 */

int main(int argc, char *argv[])
{
    const real VERY_LARGE_NUMBER = 1e300;
    real a_max = VERY_LARGE_NUMBER;      // maximum value for semi-major axis a
                             // of binaries, to be reported by this program
    if (! read_options(argc, argv, a_max))
        return 1;                // halt criterion detected by read_options()

    real * mass;                 // masses for all particles
    real (* pos)[NDIM];          // positions for all particles
    real (* vel)[NDIM];          // velocities for all particles
    int n;                       // N, number of particles in the N-body system
    real t;                      // time

    while(get_snapshot(&mass, &pos, &vel, n, t)){
        if (! find_binaries(mass, pos, vel, n, t, a_max))
    return 1;
delete_snapshot(mass, pos, vel);
    }
    return 0;
}
```

Here the option is added to the parsing of the command line argument:

Code 11.9 (find_binaries2.C: read_options)

```
/*-----------------------------------------------------------------------------
 *  read_options  --  reads the command line options, and implements them.
 *
 *  note: when the help option -h is invoked, the return value is set to false,
 *        to prevent further execution of the main program; similarly, if an
 *        unknown option is used, the return value is set to false.
 *-----------------------------------------------------------------------------
 */

bool read_options(int argc, char *argv[], real & a_max)
{
    int c;
    while ((c = getopt(argc, argv, "ha:")) != -1)
        switch(c){
            case 'a': a_max = atof(optarg);
                      break;
            case 'h':
            case '?': cerr << "usage: " << argv[0]
                           << " [-a maximum-value-for-semi-major-axis]"
                           << " [-h (for help)]"
                           << endl;
                      return false;      // execution stops after help or error
            }

    return true;                         // ready to continue program execution
}
```

And here the extra test is implemented in the actual search routine.

Code 11.10 (find_binaries2.C: find_binaries)

```
/*-----------------------------------------------------------------------------
 *  find_binaries  --  for each particle pair, check whether their relative
 *                     energy is negative.  If so, determine the semi-major
 *                     axis a and eccentricity e, and print those, but only
 *                     if a < a_max, the maximum value for a.  The values
 *                     for a and e are determined from the energy and angular
 *                     momentum of the particle pair, expressed in units of
 *                     their reduced mass.
 *-----------------------------------------------------------------------------
 */

bool find_binaries(real mass[], real pos[][NDIM], real vel[][NDIM], int n,
   real t, real a_max)
{
    cout << "time = " << t << endl;
    for (int i = 0; i < n-1 ; i++){
for (int j = i+1; j < n ; j++){
    real delr[NDIM];
    real delv[NDIM];
            real delr_sq = 0;
            real delv_sq = 0;
            for (int k = 0; k < NDIM ; k++){
delr[k] = pos[j][k]-pos[i][k];
delv[k] = vel[j][k]-vel[i][k];
                delr_sq += delr[k] * delr[k];
                delv_sq += delv[k] * delv[k];
    }
    real m_tot = mass[i] + mass[j];
    real Etilde;        // energy per unit of reduced mass
    Etilde = - m_tot / sqrt(delr_sq) + 0.5 * delv_sq;
    if (Etilde < 0.0){
real Ltilde_sq = 0.0;    // square of the value of the angular
                                 // momentum per unit of reduced mass
if (NDIM == 2){
    real r_cross_v_component = delr[0]*delv[1]-delr[1]*delv[0];
    Ltilde_sq = r_cross_v_component * r_cross_v_component;
}
else if (NDIM == 3){
    real r_cross_v_component = delr[0]*delv[1]-delr[1]*delv[0];
    Ltilde_sq = r_cross_v_component * r_cross_v_component;
    r_cross_v_component = delr[1]*delv[2]-delr[2]*delv[1];
    Ltilde_sq += r_cross_v_component * r_cross_v_component;
    r_cross_v_component = delr[2]*delv[0]-delr[0]*delv[2];
    Ltilde_sq += r_cross_v_component * r_cross_v_component;
}
else {
    cerr << "find_binaries: NDIM = " << NDIM
 << "not supported " << endl;
    return false;
}
real a = - 0.5 * m_tot / Etilde;
real e_sq = 1 - Ltilde_sq/(m_tot * a);
real e;
if (e_sq > 0.0)
    e = sqrt(e_sq);
else               // in case of roundoff errors that may
    e = 0.0;       // lead to e_sq being slightly negative

if (a < a_max)
    cout << "star1 = " << i << "  star2 = " << j
 << "  a = " << a << "  e = " << e << endl;
    }
}
    }
    return true;
}
```

Bob: Let's try to see first whether your claim is correct, that `find_binaries2.C` gives the same output as `find_binaries1.C`.

Carol: Okay, here is the word count:

```
|gravity> ../chap9/sphere1 -s 42 -n 25 | find_binaries2 | wc
seed = 42
    301    3603   12941
|gravity>
```

Carol: And to make really sure, here is a diff between the two outputs:

```
|gravity> ../chap9/sphere1 -s 42 -n 25 | find_binaries1 > tmp1
seed = 42
|gravity> ../chap9/sphere1 -s 42 -n 25 | find_binaries2 > tmp2
seed = 42
|gravity> diff tmp1 tmp2
|gravity>
```

Bob: I'm convinced! Now what shall we choose for `a_max`? How about $a_{max} = 0.1$?

Carol: Fine, but let's start with a word count.

```
|gravity> ../chap9/sphere1 -s 42 -n 25 | find_binaries2 -a 0.1 | wc
seed = 42
      4      39     142
|gravity>
```

Bob: A lot better already. Discounting again the line `time = 0`, we have found 3 tight binaries.

Alice: But even those binaries are likely to be not physical, but just chance positionings of two particles that happen to be within a distance of 0.1 of each other, in our units. Since the velocities are zero, any particle pair that close will be counted. We have only shown that 3 out of 300 pairs are this close, just 1% of the possible pair choices.

Bob: I would have expected something like 0.1%, one in a thousand, since the room each particle has for harboring a neighbor at such a small distance is the volume of a sphere with radius 0.1. And in three dimensions that volume is a thousand times smaller than the volume of the sphere in which particles are located, which has radius unity.

Carol: Ah, but there are 25 particles that you can start with; what you just derived was the chance to find a binary companion around a particular star.

Alice: And then we have to divide by two to avoid each pair from being counted twice. So we wind up with an estimate of 1.25%, so we could have expected $0.0125 * 300 = 4$ stars, on average. To find 3 seems close enough.

Bob: Here are a few more, now for random seeds

```
|gravity> ../chap9/sphere1 -n 25 | find_binaries2 -a 0.1 | wc
seed = 1064763074
      5      51     184
|gravity> !!
../chap9/sphere1 -n 25 | find_binaries2 -a 0.1 | wc
seed = 1064763076
      4      39     142
|gravity> !!
../chap9/sphere1 -n 25 | find_binaries2 -a 0.1 | wc
seed = 1064763078
      3      27      98
|gravity> !!
../chap9/sphere1 -n 25 | find_binaries2 -a 0.1 | wc
seed = 1064763082
      3      27      97
|gravity> !!
../chap9/sphere1 -n 25 | find_binaries2 -a 0.1 | wc
```

```
seed = 1064763084
      6      63     230
|gravity>  !!
../chap9/sphere1 -n 25 | find_binaries2 -a 0.1 | wc
seed = 1064763087
      3      27      98
|gravity>  !!
../chap9/sphere1 -n 25 | find_binaries2 -a 0.1 | wc
seed = 1064763090
      5      51     187
|gravity>  !!
../chap9/sphere1 -n 25 | find_binaries2 -a 0.1 | wc
seed = 1064763093
      4      39     140
|gravity>  !!
../chap9/sphere1 -n 25 | find_binaries2 -a 0.1 | wc
seed = 1064763097
      3      27      99
|gravity>  !!
../chap9/sphere1 -n 25 | find_binaries2 -a 0.1 | wc
seed = 1064763098
      4      39     142
|gravity>
|gravity>
```

Bob: The average is still 3, not 4. In any case, statistics is a very tricky subject, and I wouldn't be surprised if the true value would be slightly different from our rough estimate of 4. For example, there may be edge effects involved, for particles near the surface of the sphere. But I would be uncomfortable if the outcome would have been vastly different from 4 on average.

Alice: Now that we're happy with the statistics, can you show the actual output?

Bob: Ah, of course, here it is:

```
|gravity> ../chap9/sphere1 -s 42 -n 25 | find_binaries2 -a 0.1
seed = 42
time = 0
star1 = 6  star2 = 12  a = 0.0271531  e = 1
star1 = 8  star2 = 23  a = 0.0783721  e = 1
star1 = 13  star2 = 14  a = 0.0386034  e = 1
|gravity>
```

Bob: Looks pretty random to me. And the eccentricities are all unity, as I should have predicted, had I thought about it, given that the velocities are zero. Particles falling toward each other move on a straight line, the extreme case of an ellipse with eccentricity 1.

11.4 Dynamically Produced Binaries

Carol: Why don't we do a real run now.

Bob: I'll use our workhorse, `nbody_sh1.C` with options `-i` to include output for time 0 as well, and `-e 10` to allow only energy diagnostics at the beginning and at the end so as not to clutter the screen.

```
|gravity> ../chap9/sphere1 -s 42 -n 25 | ../chap8/nbody_sh1 -e 10 -i | find_binaries2 -a 0.1
seed = 42
Starting a Hermite integration for a 25-body system,
  from time t = 0 with time step control parameter dt_param = 0.03  until time 10 ,
  with diagnostics output interval dt_dia = 10,
  and snapshot output interval dt_out = 1.
at time t = 0 , after 0 steps :
  E_kin = 0 , E_pot = -0.61885 , E_tot = -0.61885
                absolute energy error: E_tot - E_init = 0
                relative energy error: (E_tot - E_init) / E_init = -0
time = 0
star1 = 6  star2 = 12  a = 0.0271531  e = 1
star1 = 8  star2 = 23  a = 0.0783721  e = 1
star1 = 13  star2 = 14  a = 0.0386034  e = 1
time = 1.00008
star1 = 16  star2 = 21  a = 0.0601398  e = 0.612398
time = 2.00009
star1 = 1  star2 = 3  a = 0.0816514  e = 0.823595
star1 = 1  star2 = 4  a = 0.0200926  e = 0.945035
star1 = 5  star2 = 8  a = 0.0336744  e = 0.798637
star1 = 8  star2 = 23  a = 0.00884901  e = 0.744231
time = 3.00005
star1 = 0  star2 = 1  a = 0.048932  e = 0.926806
star1 = 0  star2 = 21  a = 0.0669483  e = 0.681452
star1 = 1  star2 = 3  a = 0.0668783  e = 0.994049
star1 = 1  star2 = 21  a = 0.041487  e = 0.87713
star1 = 5  star2 = 8  a = 0.0541946  e = 0.841467
star1 = 5  star2 = 23  a = 0.00772702  e = 0.905568
time = 4.00008
star1 = 0  star2 = 3  a = 0.080123  e = 0.872677
star1 = 0  star2 = 21  a = 0.0268892  e = 0.940599
star1 = 2  star2 = 11  a = 0.0061008  e = 0.807254
star1 = 2  star2 = 19  a = 0.0910958  e = 0.653393
time = 5.00001
star1 = 2  star2 = 19  a = 0.0226411  e = 0.927754
star1 = 3  star2 = 21  a = 0.0149472  e = 0.595139
star1 = 5  star2 = 17  a = 0.00895071  e = 0.422472
time = 6.00001
star1 = 2  star2 = 19  a = 0.0023379  e = 0.508743
star1 = 3  star2 = 21  a = 0.0159819  e = 0.556314
```

```
time = 7.00001
star1 = 0  star2 = 21  a = 0.0180656  e = 0.303921
star1 = 2  star2 = 19  a = 0.00233757  e = 0.507285
time = 8.00002
star1 = 0  star2 = 4  a = 0.0258779  e = 0.996719
star1 = 0  star2 = 21  a = 0.0279722  e = 0.883476
star1 = 2  star2 = 19  a = 0.00233754  e = 0.507268
star1 = 4  star2 = 21  a = 0.0805251  e = 0.390524
time = 9.00001
star1 = 0  star2 = 21  a = 0.0132626  e = 0.650153
star1 = 2  star2 = 19  a = 0.0023375  e = 0.506389
at time t = 10 , after 498929 steps :
  E_kin = 0.672558 , E_pot = -1.1658 , E_tot = -0.493243
                absolute energy error: E_tot - E_init = 0.125607
                relative energy error: (E_tot - E_init) / E_init = -0.202968
time = 10
star1 = 0  star2 = 3  a = 0.0243207  e = 0.965155
star1 = 0  star2 = 21  a = 0.0311286  e = 0.764542
star1 = 2  star2 = 19  a = 0.00233747  e = 0.506845
star1 = 3  star2 = 21  a = 0.0458811  e = 0.746161
|gravity>
```

Bob: I'm glad to see that only at time 0 we have those skinny binaries with eccentricity 0. And hey, look at that very tight binary, with a semi-major axis of only 0.002, formed by stars 2 and 19!

Carol: It is clearly a persistent binary. It was already there at time 6, with pretty much the same short semi-major axis.

Alice: Once a binary is so tight, it is less likely that any other star will come close to that close pair, so it will take a long time before there is another strong interaction.

Bob: I see what you mean: the same binary was present at time 5, but with a much wider orbit, about ten times as large; and it did not last long at that larger distance, before it got a lot tighter.

Alice: That event must have been a more complex three-body dance. You can see that at time 4, star 2 was simultaneously considered to be bound to start 19 and to star 11. It was far closer to start 11 at that time, so the large binding energy must have come at least partly from that pair.

Bob: Ho! Before we analyze much further, I just noticed something really bad. In our excitement about binaries, we forgot to check whether the energy conservation was reasonable. Look! We have a relative energy error of more than 20%.

Carol: That's no good. We should rerun with a smaller accuracy parameter for the integrator.

Alice: I agree. Still, the main points we made are likely to remain valid qualitatively, even if not quantitatively.

Bob: I'll use the same seed again, but with a three times smaller step size control parameter. With a fourth-order integration scheme, that should reduce the relative energy error for the whole system to much less than 1%.

```
|gravity> ../chap9/sphere1 -s 42 -n 25 | ../chap8/nbody_sh1 -d 0.01 -e 10 -i | find_binaries2 -a 0.1
seed = 42
Starting a Hermite integration for a 25-body system,
  from time t = 0 with time step control parameter dt_param = 0.01  until time 10 ,
  with diagnostics output interval dt_dia = 10,
  and snapshot output interval dt_out = 1.
at time t = 0 , after 0 steps :
  E_kin = 0 , E_pot = -0.61885 , E_tot = -0.61885
                absolute energy error: E_tot - E_init = 0
                relative energy error: (E_tot - E_init) / E_init = -0
time = 0
star1 = 6  star2 = 12  a = 0.0271531  e = 1
star1 = 8  star2 = 23  a = 0.0783721  e = 1
star1 = 13  star2 = 14  a = 0.0386034  e = 1
time = 1.00008
star1 = 5  star2 = 18  a = 0.0804608  e = 0.373533
star1 = 7  star2 = 16  a = 0.0653846  e = 0.980501
star1 = 14  star2 = 16  a = 0.0592383  e = 0.88506
time = 2.00006
star1 = 7  star2 = 15  a = 0.0551175  e = 0.78881
star1 = 8  star2 = 10  a = 0.082922  e = 0.406961
time = 3.00003
star1 = 8  star2 = 13  a = 0.00555818  e = 0.849134
star1 = 10  star2 = 16  a = 0.0420807  e = 0.772984
star1 = 10  star2 = 19  a = 0.0345081  e = 0.967946
time = 4
star1 = 3  star2 = 10  a = 0.0645721  e = 0.651257
star1 = 7  star2 = 23  a = 0.0778391  e = 0.475478
star1 = 8  star2 = 13  a = 0.00554977  e = 0.677847
time = 5
star1 = 3  star2 = 8  a = 0.002126  e = 0.920892
star1 = 4  star2 = 22  a = 0.0423633  e = 0.638268
time = 6
star1 = 3  star2 = 8  a = 0.00212601  e = 0.952222
star1 = 4  star2 = 22  a = 0.0403322  e = 0.827227
time = 7
star1 = 3  star2 = 8  a = 0.00220527  e = 0.777804
time = 8
star1 = 3  star2 = 8  a = 0.00220456  e = 0.732745
time = 9
star1 = 3  star2 = 8  a = 0.00197343  e = 0.801175
```

```
at time t = 10 , after 2642122 steps :
  E_kin = 0.783526 , E_pot = -0.871684 , E_tot = -0.0881583
                absolute energy error: E_tot - E_init = 0.530692
                relative energy error: (E_tot - E_init) / E_init = -0.857545
time = 10
star1 = 3  star2 = 8  a = 0.00197342  e = 0.806924
|gravity>
```

Bob: What happened? I had expected it to take three times longer, with a three times smaller step size criterion; instead it took five times longer, with more than two and a half million steps instead of the previous half million. And what is worse, the energy conservation is now atrocious!

Alice: The fact that we have embarked on a completely different history is by itself not surprising. The N-body system is exponentially unstable. In other words, even a slight chance anywhere in the position or velocity of any single particle will quickly lead to a completely different behavior of the system. This is what is popularly called the butterfly effect: if a butterfly flaps its wings somewhere in the world, and if that happens in an exponentially unstable part of a weather zone, the whole weather pattern on the planet could become rather different after a few weeks or so from what it otherwise would have been. Whether this is really true for actual butterflies, I don't know, but it is certainly true for stars!

Carol: So we may have just been unlucky, getting not only onto a different trajectory, but on a trajectory with a bad energy behavior to boot. But a few days ago, we had much better luck with energy conservation. How about trying again, but actually degrading the accuracy a bit, with the option `-a 0.02`, in between the default of `-a 0.03` and the `-a 0.01` you just tried? Who knows, we may be more lucky this time.

Bob: Using a Monte Carlo method? I didn't think playing roulette was a very scientific approach! But what the heck, let's give it a try.

Alice: You may not have heard of it, but there is actually something called a Monte Carlo method in science! If we had chosen a rejection technique to construct our homogeneous spherical particle distribution, we would have used a type of Monte Carlo technique.

Bob: What do you know. Well, why don't we call Carol's proposed hit-and-miss approach a Monte Carol method.

Carol: You'd better start typing instead!

```
|gravity> ../chap9/sphere1 -s 42 -n 25 | ../chap8/nbody_sh1 -d 0.01 -e 10 -i | find_binaries2 -a 0.
|gravity> ../chap9/sphere1 -s 42 -n 25 | ../chap8/nbody_sh1 -d 0.02 -e 10 -i | find_binaries2 -a 0.
seed = 42
Starting a Hermite integration for a 25-body system,
  from time t = 0 with time step control parameter dt_param = 0.02  until time 10 ,
  with diagnostics output interval dt_dia = 10,
  and snapshot output interval dt_out = 1.
at time t = 0 , after 0 steps :
  E_kin = 0 , E_pot = -0.61885 , E_tot = -0.61885
                absolute energy error: E_tot - E_init = 0
                relative energy error: (E_tot - E_init) / E_init = -0
time = 0
star1 = 6  star2 = 12  a = 0.0271531  e = 1
star1 = 8  star2 = 23  a = 0.0783721  e = 1
star1 = 13  star2 = 14  a = 0.0386034  e = 1
time = 1.00047
star1 = 5  star2 = 18  a = 0.0810243  e = 0.423949
star1 = 7  star2 = 14  a = 0.0492761  e = 0.914931
time = 2.00002
star1 = 2  star2 = 3  a = 0.084026  e = 0.496036
star1 = 7  star2 = 14  a = 0.068698  e = 0.706102
star1 = 17  star2 = 19  a = 0.0137171  e = 0.782917
time = 3.00001
star1 = 0  star2 = 19  a = 0.0336442  e = 0.813088
star1 = 4  star2 = 17  a = 0.0157056  e = 0.832995
star1 = 7  star2 = 21  a = 0.0377602  e = 0.835347
star1 = 17  star2 = 18  a = 0.0190899  e = 0.891115
time = 4.00003
star1 = 3  star2 = 10  a = 0.0363436  e = 0.795917
star1 = 5  star2 = 19  a = 0.0861304  e = 0.614849
star1 = 7  star2 = 21  a = 0.0377828  e = 0.83908
star1 = 10  star2 = 17  a = 0.0491587  e = 0.319343
time = 5.00006
star1 = 4  star2 = 18  a = 0.049268  e = 0.70185
star1 = 7  star2 = 21  a = 0.0377876  e = 0.839173
star1 = 8  star2 = 24  a = 0.0812205  e = 0.513454
time = 6.00001
star1 = 4  star2 = 17  a = 0.0142633  e = 0.783939
star1 = 7  star2 = 21  a = 0.0377902  e = 0.838337
star1 = 8  star2 = 24  a = 0.0252418  e = 0.941765
star1 = 17  star2 = 24  a = 0.0727696  e = 0.719213
time = 7.00001
star1 = 4  star2 = 13  a = 0.00581595  e = 0.110655
star1 = 7  star2 = 21  a = 0.0377915  e = 0.836602
time = 8.00002
star1 = 4  star2 = 13  a = 0.00627755  e = 0.258843
star1 = 7  star2 = 21  a = 0.0377902  e = 0.831768
time = 9.00014
star1 = 4  star2 = 23  a = 0.0182906  e = 0.961251
star1 = 7  star2 = 21  a = 0.033901  e = 0.86658
star1 = 13  star2 = 15  a = 0.0571123  e = 0.45941
star1 = 13  star2 = 23  a = 0.0455079  e = 0.890129
```

```
at time t = 10 , after 300848 steps :
  E_kin = 0.811683 , E_pot = -0.997429 , E_tot = -0.185746
                absolute energy error: E_tot - E_init = 0.433104
                relative energy error: (E_tot - E_init) / E_init = -0.699853
time = 10
star1 = 4  star2 = 8  a = 0.00255213  e = 0.988263
star1 = 7  star2 = 21  a = 0.0340237  e = 0.880843
star1 = 8  star2 = 24  a = 0.0435369  e = 0.365145
|gravity>
```

Bob: No cigar. Yes, a completely different run, but no, still a bad energy behavior. By now I'm getting pretty curious to see how and when this bad energy error occurs. Let's get some more information, once every unit time increase.

```
|gravity> ../chap9/sphere1 -s 42 -n 25 | ../chap8/nbody_sh1 -d 0.02 > /dev/null
seed = 42
Starting a Hermite integration for a 25-body system,
  from time t = 0 with time step control parameter dt_param = 0.02  until time 10 ,
  with diagnostics output interval dt_dia = 1,
  and snapshot output interval dt_out = 1.
at time t = 0 , after 0 steps :
  E_kin = 0 , E_pot = -0.61885 , E_tot = -0.61885
                absolute energy error: E_tot - E_init = 0
                relative energy error: (E_tot - E_init) / E_init = -0
at time t = 1.00047 , after 12485 steps :
  E_kin = 1.06276 , E_pot = -1.24842 , E_tot = -0.185664
                absolute energy error: E_tot - E_init = 0.433187
                relative energy error: (E_tot - E_init) / E_init = -0.699986
at time t = 2.00002 , after 21245 steps :
  E_kin = 1.14737 , E_pot = -1.33303 , E_tot = -0.185664
                absolute energy error: E_tot - E_init = 0.433187
                relative energy error: (E_tot - E_init) / E_init = -0.699986
at time t = 3.00001 , after 30349 steps :
  E_kin = 1.20407 , E_pot = -1.38973 , E_tot = -0.185664
                absolute energy error: E_tot - E_init = 0.433186
                relative energy error: (E_tot - E_init) / E_init = -0.699986
^d
|gravity>
```

Carol: That was a good move, Bob! So the error started right away. How strange. Could you make the energy output interval even shorter?

Bob: I'll make it a hundred times shorter. See what happens

```
|gravity> ../chap9/sphere1 -s 42 -n 25 | ../chap8/nbody_sh1 -e 0.01 -d 0.02 > /dev/null
seed = 42
Starting a Hermite integration for a 25-body system,
  from time t = 0 with time step control parameter dt_param = 0.02  until time 10 ,
  with diagnostics output interval dt_dia = 0.01,
  and snapshot output interval dt_out = 1.
at time t = 0 , after 0 steps :
  E_kin = 0 , E_pot = -0.61885 , E_tot = -0.61885
                absolute energy error: E_tot - E_init = 0
                relative energy error: (E_tot - E_init) / E_init = -0
at time t = 0.0106043 , after 12 steps :
  E_kin = 0.00111979 , E_pot = -0.61997 , E_tot = -0.61885
                absolute energy error: E_tot - E_init = 4.737e-10
                relative energy error: (E_tot - E_init) / E_init = -7.65452e-10
at time t = 0.0203781 , after 24 steps :
  E_kin = 0.00454037 , E_pot = -0.62339 , E_tot = -0.61885
                absolute energy error: E_tot - E_init = 6.01715e-10
                relative energy error: (E_tot - E_init) / E_init = -9.72311e-10
at time t = 0.0304865 , after 39 steps :
  E_kin = 0.0123603 , E_pot = -0.63121 , E_tot = -0.61885
                absolute energy error: E_tot - E_init = 7.24435e-10
                relative energy error: (E_tot - E_init) / E_init = -1.17062e-09
at time t = 0.0400792 , after 60 steps :
  E_kin = 0.0317015 , E_pot = -0.650552 , E_tot = -0.61885
                absolute energy error: E_tot - E_init = 3.93875e-11
                relative energy error: (E_tot - E_init) / E_init = -6.36463e-11
at time t = 0.0500072 , after 1790 steps :
  E_kin = 0.399428 , E_pot = -1.13502 , E_tot = -0.735588
                absolute energy error: E_tot - E_init = -0.116738
                relative energy error: (E_tot - E_init) / E_init = 0.188638
at time t = 0.0600297 , after 3663 steps :
  E_kin = 0.55247 , E_pot = -0.738132 , E_tot = -0.185662
                absolute energy error: E_tot - E_init = 0.433188
                relative energy error: (E_tot - E_init) / E_init = -0.699989
|gravity>
```

Alice: Ah, a bad seed, so to speak. Something dramatic happened already within the first one tenth of the first time unit!

Carol: It would be interesting to find out more about what happened, and we do have the tools to do so, armed as we are with movie making equipment and a binary detector. We could even write a close encounter detector, similar to the binary detector but then for unbounded hyperbolic orbits. Perhaps they were the culprit.

Bob: All nice and fine, but first things first. Let me just take a different seed, and finally get a decent 25-body evolution, to see what our binary detector does under less pathological circumstances.

```
|gravity> ../chap9/sphere1 -s 43 -n 25 | ../chap8/nbody_sh1 -e 10 -i | find_binaries2 -a 0.1
seed = 43
```

```
Starting a Hermite integration for a 25-body system,
  from time t = 0 with time step control parameter dt_param = 0.03  until time 10 ,
  with diagnostics output interval dt_dia = 10,
  and snapshot output interval dt_out = 1.
at time t = 0 , after 0 steps :
  E_kin = 0 , E_pot = -0.569576 , E_tot = -0.569576
                absolute energy error: E_tot - E_init = 0
                relative energy error: (E_tot - E_init) / E_init = -0
time = 0
star1 = 3  star2 = 6  a = 0.0653857  e = 1
star1 = 10  star2 = 23  a = 0.0388408  e = 1
time = 1.00015
star1 = 6  star2 = 13  a = 0.0631629  e = 0.403642
star1 = 10  star2 = 17  a = 0.0668875  e = 0.452269
time = 2.00015
star1 = 6  star2 = 13  a = 0.0413988  e = 0.918359
star1 = 12  star2 = 22  a = 0.0335088  e = 0.255619
star1 = 17  star2 = 18  a = 0.0773984  e = 0.823559
time = 3.00008
star1 = 0  star2 = 22  a = 0.0906416  e = 0.924285
star1 = 4  star2 = 12  a = 0.0261997  e = 0.596176
star1 = 6  star2 = 13  a = 0.0417364  e = 0.86777
star1 = 10  star2 = 16  a = 0.0706447  e = 0.835167
time = 4.0002
star1 = 4  star2 = 22  a = 0.0858038  e = 0.739406
star1 = 4  star2 = 23  a = 0.0933074  e = 0.730645
star1 = 6  star2 = 13  a = 0.0415368  e = 0.860736
time = 5.00032
star1 = 4  star2 = 22  a = 0.0151778  e = 0.983233
star1 = 18  star2 = 23  a = 0.0982301  e = 0.754695
time = 6.00013
star1 = 3  star2 = 10  a = 0.0345493  e = 0.752254
time = 7.00037
star1 = 3  star2 = 4  a = 0.0726365  e = 0.907422
star1 = 3  star2 = 17  a = 0.0738742  e = 0.856752
star1 = 4  star2 = 17  a = 0.0742944  e = 0.94384
time = 8.00001
star1 = 1  star2 = 8  a = 0.0341744  e = 0.990472
star1 = 22  star2 = 23  a = 0.0467334  e = 0.70404
time = 9.00015
star1 = 0  star2 = 4  a = 0.0548707  e = 0.917466
star1 = 3  star2 = 4  a = 0.0186082  e = 0.237707
star1 = 3  star2 = 8  a = 0.0681921  e = 0.993719
at time t = 10.0001 , after 54838 steps :
  E_kin = 0.490667 , E_pot = -1.06032 , E_tot = -0.569648
                absolute energy error: E_tot - E_init = -7.22701e-05
                relative energy error: (E_tot - E_init) / E_init = 0.000126884
```

```
time = 10.0001
star1 = 6  star2 = 7  a = 0.0980807  e = 0.998395
star1 = 16  star2 = 22  a = 0.0192647  e = 0.694863
|gravity>
```

Bob: Hehe! That's a whole lot better. A relative energy error of about one hundredth of a percent. Perhaps a good time to stop ;>).

Carol: For now, yes, but I have the feeling that we have just only uncovered the tip of an iceberg.

Alice: I'm sure of that, too. I hope we will get back in a while, to delve further into all this.

Exercise 11.1 (Close encounters)
Design and build a piece of code similar to `find_binaries2.C`*, but now for unbound close encounters between two particles. Use that code to analyse in more detail when and how errors become unacceptably large, as we have seen above. Can you suggest ways to improve the error accuracy, other than just setting the overall accuracy parameter to a smaller and smaller value?*