An Introduction to Computer Simulation Methods
Applications to Physical System

Harvey Gould, Jan Tobochnik, and Wolfgang Christian

August 27, 2016

Contents

Preface

Computer simulations are now an integral part of contemporary basic and applied physics, and computation has become as important as theory and experiment. The ability to compute is now part of the essential repertoire of research scientists.

Since writing the first two editions of our text, more courses devoted to the study of physics using computers have been introduced into the physics curriculum, and many more traditional courses are incorporating numerical examples. We are gratified to see that our text has helped shape these innovations. The purpose of our book includes the following:

1. To provide a means for students to *do* physics.
2. To give students an opportunity to gain a deeper understanding of the physics they have learned in other courses.
3. To encourage students to "discover" physics in a way similar to how physicists learn in the context of research.
4. To introduce numerical methods and new areas of physics that can be studied with these methods.
5. To give examples of how physics can be applied in a much broader context than is discussed in the traditional physics undergraduate curriculum.
6. To teach object-oriented programming in the context of doing science.

Our overall goal is to encourage students to learn about science through experience and by asking questions. Our objective always is understanding, not the generation of numbers.

The major change in this edition is the use of the Java programming language instead of True Basic, which was used in the first two editions. We chose Java for some of the same reasons we originally chose True Basic. Java is available for all popular operating systems, and is platform independent, contains built-in graphics capabilities, is freely available, and has all the features needed to write powerful computer simulations. There is an abundance of free open source tools available for Java programmers, including the *Eclipse* integrated development environment. Because Java is popular, it continues to evolve, and its speed is now comparable to other languages used in scientific programming. In addition, Java is object oriented, which has become the dominant paradigm in computer science and software engineering, and therefore learning Java is excellent preparation for students with interests in physics and computer science. Java programs can be easily adapted for delivery over the Web. Finally, as for True Basic, the nongraphical parts of our programs can easily be converted to other languages such as C/C++, whose syntax is similar to Java.

When we chose True Basic for our first edition, introductory computer science courses were teaching Pascal. When we continued with True Basic in the second edition, computer science departments were experimenting with teaching C/C++. Finally, we are able to choose a language that is commonly taught and used in many contexts. Thus, it is likely that some of the students reading our text will already know Java and can contribute much to a class that uses our text.

Java provides many powerful libraries for building a graphical user interface and incorporating audio, video, and other media. If we were to discuss these libraries, students would become absorbed in programming tasks that have little or nothing to do with physics. For this reason our text uses the Open Source Physics library which makes it easy to write programs that are simpler and more graphically oriented than those that we wrote in True Basic. In addition, the Open Source Physics library is useful for other computational physics projects which are not discussed in this text, as well as general programming tasks. This library provides for easy graphical input of parameters, tabular output of data, plots, visualizations and animations, and the numerical solution of ordinary differential equations. It also provides several useful data structures. The Open Source Physics library was developed by Wolfgang Christian, with the contributions and assistance of many others. The book *Open Source Physics: A User's Guide with Examples* by Wolfgang Christian is available separately and discusses the Open Source Physics library in much more detail. A CD that comes with the User's Guide contains the source code for the Open Source Physics library, the programs in this book, as well as ready-to-run versions of these programs. The source code and the library can also be downloaded freely from `<www.opensourcephysics.org/sip>`.

The ease of doing visualizations is a new and important aspect of Java and the Open Source Physics library, giving Java an advantage over other languages such as C++ and Fortran, which do not have built-in graphics capabilities. For example, when debugging a program, it is frequently much quicker to detect when the program is not working by looking at a visual representation of the data rather than by scanning the data as lists of numbers. Also, it is easier to choose the appropriate values of the parameters by varying them and visualizing the results. Finally, more insight is likely to be gained by looking at a visualization than a list of numbers. Because animations and the continuous plotting of data usually cause a program to run more slowly, we have designed our programs so that the graphical output can be turned off or implemented infrequently during a simulation.

Java provides support for interacting with a program during runtime. The Open Source Physics library makes this interaction even easier, so that we can write programs that use a mouse to input data, such as the location of a charge, or toggle the value of a cell in a lattice. We also do not need to input how long a simulation should run and can stop the program at any time to change parameters.

As with our previous editions, we assume no background in computer programming. Much of the text can be understood by students with only a semester each of physics and calculus. Chapter 2 introduces Java and the Open Source Physics library. In Chapter 3 we discuss the concept of interfaces and how to use some of the important interfaces in the Open Source Physics library. Later chapters introduce more Java and Open Source Physics constructs as needed, but essentially all of the chapters after Chapter 3 can be studied independently and in any order. We include many topics that are sometimes considered too advanced for undergraduates, such as random walks, chaos, fractals, percolation, simulations of many particle systems, and topics in the theory of complexity, but we introduce these topics so that very little background is required. Other chapters discuss optics, electrodynamics, relativity, rigid body motion, and quantum mechanics, which require knowledge of the physics found in the corresponding standard undergraduate courses.

This text is written so that the physics drives the choice of algorithms and the programming syntax that we discuss. We believe that students can learn how to program more quickly with this approach because they have an immediate context, namely doing simulations, in which to hone their skills. In the beginning most of the programming tasks involve modifying the programs in the text. Students should then be given some assignments that require them to write their own programs by following the format of those in the text. The students may later develop their own style as they work on their projects.

Our text is most appropriately used in a project-oriented course that lets students with a wide variety of backgrounds and abilities work at their own pace. The courses that we have taught using this text have a laboratory component. From our experience we believe that active learning where students are directly grappling with the material in this text is the most efficient. In a laboratory context students who already know a programming language can help those who do not. Also, students can further contribute to a course by sharing their knowledge from various backgrounds in physics, chemistry, computer science, mathematics, biology, economics, and other subjects.

Although most of our text is at the undergraduate level, many of the topics are considered to be graduate level and thus would be of interest to graduate students. One of us regularly teaches a laboratory-based course on computer simulation with both undergraduate and graduate students. Because the course is project oriented, students can go at their own pace and work on different problems. In this context, graduate and undergraduate students can learn much from each other.

Some instructors who might consider using our text in a graduate-level context might think that our text is not sufficiently rigorous. For example, in the suggested problems we usually do not explicitly ask students to do an extensive data analysis. However, we do discuss how to estimate errors in Chapter 11. We encourage instructors to ask for a careful data analysis on at least one assignment, but we believe that it is more important for students to spend most of their time in an exploratory mode where the focus is on gaining physical insight and obtaining numerical results that are qualitatively correct.

There are four types of suggested student activities. The exercises, which are primarily found in the beginning of the text, are designed to help students learn specific programming techniques. The problems, which are scattered throughout each chapter, are open ended and require students to run, analyze, and modify programs given in the text, or write new, but similar programs. Students will soon learn that the format for most of the programs is very similar. Starred problems require either significantly more background or work and may require the writing of a program from scratch. However, the programs for these problems still follow a similar format. The projects at the end of most of the chapters are usually more time consuming and would be appropriate for term projects or independent student research. Many new problems and projects have been added to this edition, while others have been improved or eliminated. Instructors and students should view the problem descriptions and questions as starting points for thinking about the system of interest. It is important that students read the problems even if they do not plan to do them.

We encourage instructors to ask students to write laboratory reports for at least some of the problems. The Appendix to Chapter 1 provides guidance on what these reports should include. Part of the beauty and fun of doing computer simulations is that one is forced to think about the choice of algorithm, its implementation, the choice of parameters, what to measure, and the results. Do the results make sense? What happens if you change a parameter? What if you change the algorithm? Much physics can be learned in this way.

Although all of the programs discussed in the text can be downloaded freely, most are listed in the text to encourage students to read them carefully. Students might find some useful techniques that they can use elsewhere, and the discussion in the text frequently refers to the listings.

A casual perusal of the text might suggest that the text is bereft of figures. One reason that we have not included more figures is that most of the programs in the text have an important visual component in color. Black and white figures pale in comparison. Much of the text is meant to be read while working on the programs. Thus, students can easily see the plots and animations produced by the programs while they are reading the text.

As new technologies become available and the backgrounds and expectations of students change, the question of what is worth knowing needs to be reconsidered. Today, calculators not only do arithmetic and numerical operations, but most can do algebra, calculus, and plotting. Students have lost the sense of number and most can only do the simplest mathematical manipulations in their head. On the other hand, most students feel comfortable using computers and gathering information off the Web. Because there exist programs and applets that can perform many of the simulations in this text, why should students learn how to write their own programs? We have at least two answers. First, most innovative scientific research involves writing programs that do not fit into the domains of existing software. More importantly, we believe that students obtain a deeper understanding of the physics and the algorithms themselves by writing and modifying their own programs. Just as we need to insure that students can carry out basic mathematical operations without a calculator so that they understand what these operations mean, we must do the same when it comes to computational physics.

The recommended readings at the end of each chapter have been selected for their pedagogical value rather than for completeness or for historical accuracy. We apologize to our colleagues whose work has been inadvertently omitted, and we would appreciate suggestions for new and additional references.

Because students come with a different skill set than most of their instructors, it is important that instructors realize that certain aspects of this text might be easier for their students than for them. Some instructors might be surprised that much of the code for organizing the simulations is "hidden" in the Open Source Physics library (although the source code is freely available). Some instructors will initially think that Chapter 2 contains too much material. However, from the student's perspective this material is not that difficult to learn. They are used to downloading files, using various software environments, and learning how to make software do what they want. The difficult parts of the text, where instructor input is most needed, is understanding the physics and the algorithms. Converting algorithms to programs is also difficult for many students, and we spend much time in the text explaining the programs that implement various algorithms. In some cases instructors will find it difficult to set up an environment to use Java and the Open Source Physics library. Because this task depends on the operating system, we have placed instructions on how to set up an environment for Java and Open Source Physics at <`opensourcephysics.org`>. This website also contains links to updates of the evolving Open Source Physics library as well as other resources for this text including the source code for the programs in the text.

We acknowledge generous support from the National Science Foundation which has allowed us to work on many ideas that have found their way into this textbook. We also thank Kipton Barros, Mario Belloni, Doug Brown, Francisco Esquembre, and Joshua Gould for their advice, suggestions, and contributions to the Open Source Physics library and to the text. We thank Anne Cox for suggesting numerous improvements to the narrative and for hosting an Open Source Physics developer's workshop at Eckerd College.

We are especially grateful to Louis Colonna-Romano for drawing almost all of the figures. Lou writes programs in postscript the way others write programs in Java or Fortran.

We are especially thankful to students and faculty at Clark University, Davidson College, and Kalamazoo College who have generously commented on the Open Source Physics project as they class tested early versions of this manuscript. Carlos Ortiz helped prepare the index for this book.

Many individuals reviewed parts of the text and we thank them for their assistance. They include Lowell M. Boone, Roger Cowley, Shamanthi Fernando, Alejandro L. Garcia, Alexander L. Godunov, Rubin Landau, Donald G. Luttermoser, Cristopher Moore, Anders Sandvik, Ross Spencer, Dietrich Stauffer, Jutta Luettmer–Strathmann, Daniel Suson, Matthias Troyer, Slavomir Tuleja, and Michael T. Vaughn. We thank all our friends and colleagues for their encouragement and support.

We are grateful to our wives, Patti Gould, Andrea Moll Tobochnik, and Barbara Christian, and to our children, Joshua, Emily, and Evan Gould, Steven and Howard Tobochnik, and Katherine, Charlie, and Konrad Christian for their encouragement and understanding during the course of this work. It takes a village to raise a child and a community to write a textbook.

No book of this length can be free of typos and errors. We encourage readers to email us about errors that they find and suggestions for improvements. Our plan is to continuously revise our book so that the next edition will be more timely.

Harvey Gould
Clark University
Worcester, MA 01610-1477
hgould@clarku.edu

Jan Tobochnik
Kalamazoo College
Kalamazoo, MI 49006-3295
jant@kzoo.edu

Wolfgang Christian
Davidson College
Davidson, NC 28036-6926
wochristian@davidson.edu

Chapter 1

Introduction

The importance of computers in physics and the nature of computer simulation is discussed. The nature of object-oriented programming and various computer languages is also considered.

1.1 Importance of computers in physics

Computation is now an integral part of contemporary science and is having a profound effect on the way we do physics, on the nature of the important questions, and on the physical systems we choose to study. Developments in computer technology are leading to new ways of thinking about physical systems. Asking "How can I formulate this problem on a computer?" has led to the understanding that it is practical and natural to formulate physical laws as rules for a computer rather than only in terms of differential equations.

For the purposes of discussion, we will divide the use of computers in physics into the following categories: numerical analysis, symbolic manipulation, visualization, simulation, and the collection and analysis of data. *Numerical analysis* refers to the solution of well-defined mathematical problems to produce numerical (in contrast to symbolic) solutions. For example, we know that the solution of many problems in physics can be reduced to the solution of a set of simultaneous linear equations. Consider the equations

$$\begin{aligned} 2x + 3y &= 18 \\ x - y &= 4. \end{aligned}$$

It is easy to find the analytical solution $x = 6$, $y = 2$ using the method of substitution. Suppose we wish to solve a set of four simultaneous equations. We again can find an analytical solution, perhaps using a more sophisticated method. However, if the number of simultaneous equations becomes much larger, we would need to use a computer to find a solution. In this mode the computer is a tool of numerical analysis. Because it is often necessary to compute multidimensional integrals, manipulate large matrices, or solve nonlinear differential equations, this use of the computer is important in physics.

One of the strengths of mathematics is its ability to use the power of abstraction, which allows us to solve many similar problems simultaneously by using symbols. Computers can be used to do much of the *symbolic manipulation*. As an example, suppose we want to know the solution of the quadratic equation, $ax^2 + bx + c = 0$. A symbolic manipulation program can give the solution as $x = [-b \pm \sqrt{b^2 - 4ac}]/2a$. In addition, such a program can give the usual numerical solutions for specific values of a, b, and c. Mathematical operations such as differentiation,

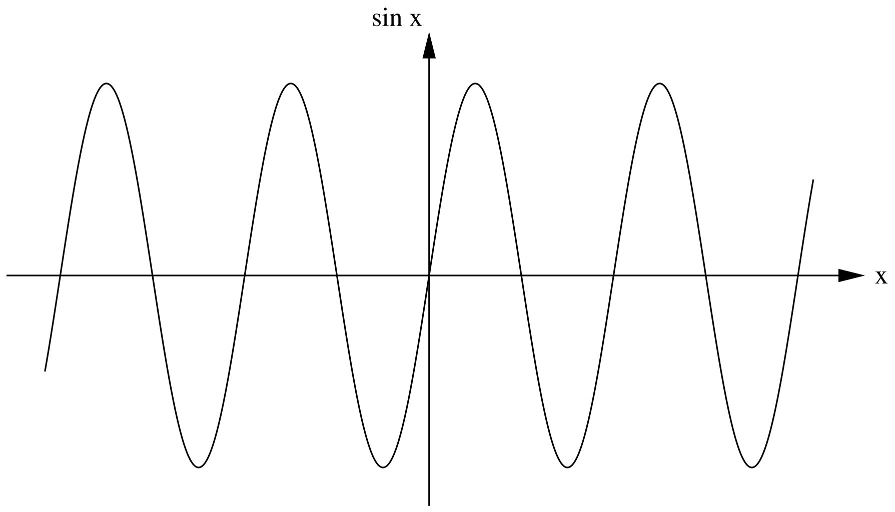


Figure 1.1: What is the meaning of the sine function?

integration, matrix inversion, and power series expansion can be performed using symbolic manipulation programs. The calculation of Feynman diagrams, which represent multidimensional integrals of importance in quantum electrodynamics, has been a major impetus to the development of computer algebra software that can manipulate and simplify symbolic expressions. Maxima, Maple, and Mathematica are examples of software packages that have symbolic manipulation capabilities as well as many tools for numerical analysis. Matlab and Octave are examples of software packages that are convenient for computations involving matrices and related tasks.

As the computer plays an increasing role in our understanding of physical phenomena, the *visual representation* of complex numerical results is becoming even more important. The human eye in conjunction with the visual processing capacity of the brain is a very sophisticated device. Our eyes can determine patterns and trends that might not be evident from tables of data and can observe changes with time that can lead to insight into the important mechanisms underlying a system's behavior. The use of graphics can also increase our understanding of the nature of analytical solutions. For example, what does a sine function mean to you? We suspect that your answer is not the series, $\sin x = x - x^3/3! + x^5/5! + \cdots$, but rather a periodic, constant amplitude curve (see Figure 1.1). What is most important is the mental image gained from a visualization of the form of the function.

Traditional modes of presenting data include two- and three-dimensional plots including contour and field line plots. Frequently, more than three variables are needed to understand the behavior of a system, and new methods of using color and texture are being developed to help researchers gain greater insights into their data.

An essential role of science is to develop models of nature. To know whether a model is consistent with observation, we have to understand the behavior of the model and its predictions. One way to do so is to implement the model on a computer. We call such an implementation a *computer simulation* or simulation for short. For example, suppose a teacher gives \$10 to each student in a class of 100. The teacher, who also begins with \$10 in her pocket, chooses a student at random and flips a coin. If the coin is heads, the teacher gives \$1 to the student; otherwise, the student gives \$1 to the teacher. If either the teacher or the student would go into debt by this transaction, the transaction is not allowed. After many exchanges, what is the probability that a student has s dollars? What is the probability that the teacher has t dollars? Are these two probabilities the same? Although these particular questions can be answered by analytical methods, many problems of this nature cannot be solved in this way (see Problem 1.1).

One way to determine the answers to these questions is to do a classroom experiment. However, such an experiment would be difficult to arrange, and it would be tedious to do a sufficient number of transactions.

A more practical way to proceed is to convert the rules of the model into a computer program, simulate many exchanges, and estimate the quantities of interest. Knowing the results might help us gain more insight into the nature of an analytical solution if one exists. We can also modify the rules and ask "what if?" questions. For example, would the probabilities change if the students could exchange money with one another? What would happen if the teacher was allowed to go into debt?

Simulations frequently use the computational tools of numerical analysis and visualization, and occasionally symbolic manipulation. The difference is one of emphasis. Simulations are usually done with a minimum of analysis. Because simulation emphasizes an exploratory mode of learning, we will stress this approach.

Computers are also involved in all phases of a laboratory experiment, from the design of the apparatus to the *collection and analysis of data*. LabView is an example of a data acquisition program. Some of the roles of the computer in laboratory experiments, such as the varying of parameters and the analysis of data, are similar to those encountered in simulations. However, the tasks involved in *real-time control* and interactive data analysis are qualitatively different and involve the interfacing of computer hardware to various types of instrumentation. We will not discuss this use of the computer.

1.2 The importance of computer simulation

Why is computation becoming so important in physics? One reason is that most of our analytical tools such as differential calculus are best suited to the analysis of *linear* problems. For example, you probably have analyzed the motion of a particle attached to a spring by assuming a linear restoring force and solving Newton's second law of motion. In this case a small change in the displacement of the particle leads to a small change in the force. However, many natural phenomena are *nonlinear*, and a small change in a variable might produce a large change in another. Because relatively few nonlinear problems can be solved by analytical methods, the computer gives us a new tool to explore nonlinear phenomena.

Another reason for the importance of computation is the growing interest in systems with many variables or with many degrees of freedom. The money exchange model described in Section 1.1 is a simple example of a system with many variables. A similar problem is given at the end of this chapter.

Computer simulations are sometimes referred to as *computer experiments* because they share much in common with laboratory experiments. Some of the analogies are shown in Table 1.1. The starting point of a computer simulation is the development of an idealized model of a physical system of interest. We then need to specify a procedure or *algorithm* for implementing the model on a computer and decide what quantities to measure. The results of a computer simulation can serve as a bridge between laboratory experiments and theoretical calculations. In some cases we can obtain essentially exact results by simulating an idealized model that has no

Laboratory Experiment	**Computer Simulation**
sample	model
physical apparatus	computer program
calibration	testing of program
measurement	computation
data analysis	data analysis

Table 1.1: Analogies between a computer simulation and a laboratory experiment.

laboratory counterpart. The results of the idealized model can serve as a stimulus to the development of the theory. On the other hand, we sometimes can do simulations of a more realistic model than can be done theoretically, and hence make a more direct comparison with laboratory experiments. Computation has become a third way of doing physics and complements both theory and experiment.

Computer simulations, like laboratory experiments, are not substitutes for thinking, but are tools that we can use to understand natural phenomena. The goal of all our investigations of fundamental phenomena is to seek explanations of natural phenomena that can be stated concisely.

1.3 Programming languages

There is no single best programming language any more than there is a best natural language. Fortran is the oldest of the more popular scientific programming languages and was developed by John Backus and his colleagues at IBM between 1954 and 1957. Fortran is commonly used in scientific applications and continues to evolve. Fortran 90/95/2000 has many modern features that are similar to C/C++.

The Basic programming language was developed in 1965 by John Kemeny and Thomas Kurtz at Dartmouth College as a language for introductory courses in computer science. In 1983 Kemeny and Kurtz extended the language to include platform independent graphics and advanced control structures necessary for structured programming. The programs in the first two editions of our textbook were written in this version of Basic, known as True Basic.

C was developed by Dennis Ritchie at Bell Laboratories around 1972 in parallel with the Unix operating system. C++ is an extension of C designed by Bjarne Stroustrup at Bell laboratories in the mid-eighties. C++ is considerably more complex than C and has object oriented features, as well as and other extensions. In general, programs written in C/C++ have high performance, but can be difficult to debug. C and C++ are popular choices for developing operating systems and software applications because they provide direct access to memory and other system resources.

Python, like Basic, was designed to be easy to learn and use. Python enthusiasts like to say that C and C++ were written to make life easier for the computer, but Python was designed to be easier for the programmer. Guido van Rossum created Python in the late 80's and early 90's. It is an interpreted, object-oriented, general-purpose programming language that is also good for prototyping. Because Python is interpreted, its performance is significantly less than optimized languages like C or Fortran.

Java is an object-oriented language that was created by James Gosling and others at Sun Microsystems. Since Java was introduced in late 1995, it has rapidly evolved and is the language of choice in most introductory computer science courses. Java borrows much of its syntax from

C++ but has a simpler structure. Although the language contains only fifty keywords, the Java *platform* adds a rich library that enables a Java program to connect to the internet, render images, and perform other high-level tasks.

Most modern languages incorporate *object-oriented* features. The idea of object-oriented programming is that functions and data are grouped together in an *object*, rather than treated separately. A program is a structured collection of objects that communicate with each other causing the internal state within a given object to change. A fundamental goal of object-oriented design is to increase the understandability and reusability of program code by focusing on what an object does and how it is used, rather than how an object is implemented.

Our choice of Java for this text is motivated in part by its platform independence, flexible standard graphics libraries, good performance, and its no cost availability. The popularity of Java ensures that the language will continue to evolve, and that programming experience in Java is a valuable and marketable skill. The Java programmer can leverage a vast collection of third-party libraries, including those for numerical calculations and visualization. Java is also relatively simple to learn, especially the subset of Java that we will need to simulate physical systems.

Java can be thought of as a platform in itself, similar to the Macintosh and Windows, because it has an application programming interface (API) that enables cross-platform graphics and user interfaces. Java programs are compiled to a platform neutral byte code so that they can run on any computer that has a Java Virtual Machine. Despite the high level of abstraction and platform independence, the performance of Java is becoming comparable with native languages. If a project requires more speed, the computationally demanding parts of the program can be converted to C/C++ or Fortran.

Readers who wish to use another programming language should find the algorithmic components of the Java program listings in the text to be easily converted into a language of their choice.

1.4 Object oriented techniques

If you already know how to program, try reading a program that you wrote several years or even several weeks ago. Many of us would not be able to follow the logic of our own program and would have to rewrite it. And your program would probably be of little use to a friend who needs to solve a similar problem. If you are learning programming for the first time, it is important to learn good programming habits to minimize this problem. One way is to employ object-oriented techniques such as encapsulation, inheritance, and polymorphism.

Encapsulation refers to the way that an object's essential information is exposed through a well-documented interface, but unnecessary details of the code are hidden. For example, we can model a particle as an object. Whenever a particle moves, it calculates its acceleration from the total force on it. Someone who wishes to use the trajectory of the particle, for example to animate the particle's trajectory, needs to refer only to the interface and does not need to know how the trajectory is calculated.

Inheritance allows a programmer to add capabilities to existing code without having to rewrite it or even know the details of how the code works. For example, you will write programs that show the evolution of planetary systems, quantum mechanical wave functions, and molecular models. Many of these programs will use (extend) code in the Open Source Physics library known as an `AbstractSimulation`. This code has a timer that periodically executes code in your program and then refreshes the on-screen animation. Using the Open Source Physics library will let you focus your efforts on programming the physics, because it is not necessary to write the code to produce the timer or to refresh the screen. Similarly, we have designed a general purpose graphical user interface (GUI) by extending code written by Sun Microsystems known as a `JFrame`. Our GUI has the features of a standard user interface such as a menu bar, minimize button, and title, even though we did not write the code to implement these features.

Polymorphism helps us to write reusable code. For example, it is easy to imagine many types of objects that are able to evolve over time. In Chapter 15 we will simulate a system of particles using random numbers rather than forces to move the particles. By using polymorphism, we can write general purpose code to do animations with both types of systems.

Science students have a rich context in which to learn programming. The past several decades of doing physics with computers has given us numerous examples that we can use to learn physics, programming, and data analysis. Unlike many programming manuals, the emphasis of this book is on learning by example. We will not discuss all aspects of Java, and this text is not a substitute for a text on Java. Think of how you learned your native language. First you learned by example, and then you learned more systematically.

Although using an object oriented language makes it easier to write well-structured programs, it does not guarantee that your programs will be well written or even correct. The single most important criterion of program quality is readability. If your program is easy to read and follow, it is probably a good program. There are many analogies between a good program and a well-written paper. Few papers and programs come out perfectly on their first draft, regardless of the techniques and rules we use to write them. Rewriting is an important part of programming.

1.5 How to use this book

Most chapters in this text begin with a brief background summary of the nature of a system and the important questions. We then introduce the computer algorithms, new syntax as needed, and discuss a sample program. The programs are meant to be read as text on an equal basis with the discussions and are interspersed throughout the text. It is strongly recommended that all the problems be read, because many concepts are introduced after you have had a chance to think about the result of a simulation.

It is a good idea to maintain a computer-based notebook to record your programs, results, graphical output, and analysis of the data. This practice will help you develop good habits for future research projects, prevent duplication, organize your thoughts, and save you time. After a while you will find that most of your new programs will use parts of your earlier programs. Ideally, you will use your files to write a laboratory report or a paper on your work. Guidelines for writing a laboratory report are given in Appendix 1A.

Many of the problems in the text are open ended and do not lend themselves to simple "back of the book" answers. So how will you know if your results are correct? How will you know when you have done enough? There are no simple answers to either question, but we can give some guidelines. First, you should compare the results of your program to known results whenever possible. The known results might come from an analytical solution that exists in certain limits or from published results. You should also look at your numbers and graphs, and determine if they make sense. Do the numbers have the right sign? Are they the right order of magnitude? Do the trends make sense as you change the parameters? What is the statistical error in the data? What is the systematic error? Some of the problems explicitly ask you to do these checks, but you should make it a habit to do as many as you can whenever possible.

How do you know when you are finished? The main guideline is whether you can tell a coherent story about your system of interest. If you have only a few numbers and do not know their significance, then you need to do more. Let your curiosity lead you to more explorations. Do not let the questions asked in the problems limit what you do. The questions are only starting points, and frequently you will be able to think of your own questions.

The following problem is an example of the kind of problems that will be posed in the following chapters. Note its similarity to the questions posed on page 3. Although most of the simulations that we will do will be on the kind of physical systems that you will encounter in other physics courses, we will consider simulations in related areas, such as traffic flow, small world networks, and economics. Of course, unless you already know how to do simulations, you will have to study the following chapters so that you will able to do problems like the following.

Problem 1.1. Distribution of money

The distribution of income in a society $f(m)$ behaves as $f(m) \propto m^{-1-\alpha}$, where m is the income (money) and the exponent α is between 1 and 2. The quantity $f(m)$ can be taken to be the number of people who have an amount of money between m and $m + \Delta m$. This power law behavior of the income distribution is often referred to as Pareto's law or the 80/20 rule (20% of the people have 80% of the income) and was proposed in the late 1800's by Vilfredo Pareto, an economist and sociologist. In the following, we consider some simple models of a closed economy to determine the relation between the microdynamics and the resulting macroscopic distribution of money.

a. Suppose that N agents (people) can exchange money in pairs. For simplicity, we assume that all the agents are initially assigned the same amount of money m_0, and the agents are then allowed to interact. At each time step, a pair of agents i and j with money m_i and m_j is randomly chosen and a transaction takes place. Again for simplicity, let us assume that $m_i \to m_i'$ and $m_j \to m_j'$ by a random reassignment of their total amount of money, $m_i + m_j$, such that

$$m_i' = \epsilon(m_i + m_j) \tag{1.1a}$$

$$m_j' = (1-\epsilon)(m_i + m_j) \tag{1.1b}$$

where ϵ is a random number between 0 and 1. Note that this reassignment ensures that the agents have no debt after the transaction, that is, they are always left with an amount $m \geq 0$. Simulate this model and determine the distribution of money among the agents after the system has relaxed to an equilibrium state. Choose $N = 100$ and $m_0 = 1000$.

b. Now let us ask what happens if the agents save a fraction λ of their money before the transaction. We write

$$m_i' = m_i + \delta m \tag{1.2a}$$

$$m_j' = m_j - \delta m \tag{1.2b}$$

$$\delta m = (1-\lambda)[\epsilon m_j - (1-\epsilon)m_i]. \tag{1.2c}$$

Modify your program so that this savings model is implemented. Consider $\lambda = 0.25$, 0.50, 0.75, and 0.9. For some of the values of λ, as many as 10^7 transactions will need to be considered. Does the form of $f(m)$ change for $\lambda > 0$?

The form of $f(m)$ for the model in Problem 1.1a can be found analytically and is known to students who have had a course in statistical mechanics. However, the analytical form of $f(m)$

in Problem 1.1b is not known. More information about this model can be found in the article by Patriarca, Chakraborti, and Kaski (see the references at the end of this chapter).

Problem 1.1 illustrates some of the characteristics of simulations that we will consider in the following chapters. Implementing this model on a computer would help you to gain insight into its behavior and might encourage you to explore variations of the model. Note that the model lends itself to asking a relatively simple "what if" question, which in this case leads to qualitatively different behavior. Asking similar questions might require modifying only a few lines of code. However, such a change might convert an analytically tractable problem into one for which the solution is unknown.

Problem 1.2. Questions to consider

a. You are familiar with the fall of various objects near the earth's surface. Suppose that a ball is in the earth's atmosphere long enough for air resistance to be important. How would you simulate the motion of the ball?

b. Suppose that you wish to model a simple liquid such as liquid Argon. Why is such a liquid simpler to simulate than water? What is the maximum number of atoms that can be simulated in a reasonable amount of time using present computer technology? What is the maximum real time that is possible to simulate? That is, if we run our program for a week of computer time, what would be the equivalent time that the liquid has evolved?

c. Discuss some examples of systems that would be interesting to you to simulate. Can these systems be analyzed by analytical methods? Can they be investigated experimentally?

d. An article by Post and Votta (see references) claims that "...(computers) have largely replaced pencil and paper as the theorist's main tool." Do you agree with this statement? Ask some of the theoretical physicists that you know for their opinions.

Appendix 1A: Laboratory reports

Laboratory reports should reflect clear writing style and obey proper rules of grammar and correct spelling. Write in a manner that can be understood by another person who has not done the research. In the following, we give a suggested format for your reports.

Introduction. Briefly summarize the nature of the physical system, the basic numerical method or algorithm, and the interesting or relevant questions.

Method. Describe the algorithm and how it is implemented in the program. In some cases this explanation can be given in the program itself. Give a typical listing of your program. Simple modifications of the program can be included in an appendix if necessary. The program should include your name and date and be annotated in a way that is as self-explanatory as possible. Be sure to discuss any important features of your program.

Verification of program. Confirm that your program is not incorrect by considering special cases and by giving at least one comparison to a hand calculation or known result.

Data. Show the results of some typical runs in graphical or tabular form. Additional runs can be included in an appendix. All runs should be labeled, and all tables and figures must be referred to in the body of the text. Each figure and table should have a caption with complete information, for example, the value of the time step.

Analysis. In general, the analysis of your results will include a determination of qualitative and quantitative relationships between variables and an estimation of numerical accuracy.

Interpretation. Summarize your results and explain them in simple physical terms whenever possible. Specific questions that were raised in the assignment should be addressed here. Also give suggestions for future work or possible extensions. It is not necessary to answer every part of each question in the text.

Critique. Summarize the important physical concepts for which you gained a better understanding and discuss the numerical or computer techniques you learned. Make specific comments on the assignment and suggestions for improvements or alternatives.

Log. Keep a log of the time spent on each assignment and include it with your report.

References and suggestions for further reading

Programming

We list some of our favorite Java programming books here. There are many useful online tutorials.

Joshua Bloch, *Effective Java* (Addison–Wesley, 2001). This excellent book is for advanced Java programmers and should be read after you have become familiar with Java.

Rogers Cadenhead and Laura Lemay *Teach Yourself Java in 21 Days* 4th ed. (Sams, 2004). An inexpensive self-study guide that uses a step by step tutorial approach to cover the basics.

Stephen J. Chapman, *Java for Engineers and Scientists*, 2nd ed. (Prentice Hall, 2004).

Wolfgang Christian, *Open Source Physics: A User's Guide with Examples* (Addison–Wesley, 2006). This guide is a useful supplement to our text.

Bruce Eckel, *Thinking in Java*, 3rd ed. (Prentice Hall. 2003). This text discusses the finer points of object-oriented programming and is recommended after you have become familiar with Java. See also `<www.mindview.net/Books/>`.

David Flanagan, *Java in a Nutshell*, 5th ed. (O'Reilly, 2005) and *Java Examples in a Nutshell*, 3rd ed. (O'Reilly, 2004). A fast-paced Java tutorial for those who already know another programming language.

Brian D. Hahn and Katherine M. Malan, *Essential Java for Scientists and Engineers* (Butterworth-Heinemann, 2002).

Cay S. Horstmann and Gary Cornell, *Core Java 2: Fundamentals* and *Core Java 2: Advanced Features*, both in 7th ed. (Prentice Hall, 2005). A two-volume set that covers all aspects of Java programming.

Patrick Niemeyer and Jonathan Knudsen, *Learning Java*, 2nd ed. (O'Reilly, 2002). A comprehensive introduction to Java that starts with `HelloWorld` and ends with a discussion of XML. The book contains many examples showing how the core Java API is used. This book is one of our favorites for beginning Java programmers. However, it might be intimidating to someone who does not have some familiarity with computers.

Sherry Shavor, Jim D'Anjou, Pat McCarthy, John Kellerman, and Scott Fairbrothe, *The Java Developer's Guide to Eclipse* (Addison–Wesley Professional, 2003). A good reference for the open source Eclipse development environment. Check for new versions because Eclipse is evolving rapidly.

General References on Physics and Computers

Richard E. Crandall, *Projects in Scientific Computation* (Springer–Verlag, 1994).

Paul L. DeVries, *A First Course in Computational Physics* (John Wiley & Sons, 1994).

Alejandro L. Garcia, *Numerical Methods for Physics*, 2nd ed. (Prentice Hall, 2000). Matlab, C++, and Fortran are used.

Neil Gershenfeld, *The Nature of Mathematical Modeling* (Cambridge University Press, 1998).

Nicholas J. Giordano and Hisao Nakanishi, *Computational Physics*. 2nd ed. (Prentice Hall, 2005).

Dieter W. Heermann, *Computer Simulation Methods in Theoretical Physics*, 2nd ed. (Springer–Verlag, 1990). A discussion of molecular dynamics and Monte Carlo methods directed toward advanced undergraduate and beginning graduate students.

David Landau and Kurt Binder, *A Guide to Monte Carlo Simulations in Statistical Physics*, 2nd ed. (Cambridge University Press, 2005). The authors emphasize the complementary nature of simulation to theory and experiment.

Rubin H. Landau, *A First Course in Scientific Computing* (Princeton University Press, 2005).

P. Kevin MacKeown, *Stochastic Simulation in Physics* (Springer, 1997).

Tao Pang, *Computational Physics* (Cambridge University Press, 1997).

Franz J. Vesely, *Computational Physics*, 2nd ed. (Plenum Press, 2002).

Michael M. Woolfson and Geoffrey J. Perl, *Introduction to Computer Simulation* (Oxford University Press, 1999).

Other References

Ruth Chabay and Bruce Sherwood, Matter & Interactions (John Wiley & Sons, 2002). This two-volume text uses computer models written in VPython to present topics not typically discussed in introductory physics courses.

H. Gould, "Computational physics and the undergraduate curriculum," Computer Physics Communications **127** (1), 6–10 (2000).

Brian Hayes, "g-OLOGY," Am. Scientist **92** (3), 212–216 (2004) discusses the g-factor of the electron and the importance of algebraic and numerical calculations.

Problem 1.1 is based on a paper by Marco Patriarca, Anirban Chakraborti, and Kimmo Kaski, "Gibbs versus non-Gibbs distributions in money dynamics," Physica A **340**, 334–339 (2004).

An interesting article on the future of computational science by Douglass E. Post and Lawrence G. Votta, "Computational science demands a new paradigm," Physics Today **58** (1), 35–41 (2005) raises many interesting questions.

Ross L. Spencer, “Teaching computational physics as a laboratory sequence,” Am. J. Phys. 73, 151–153 (2005).

Chapter 2

Tools for Doing Simulations

We introduce some of the core syntax of Java in the context of simulating the motion of falling particles near the Earth's surface. A simple algorithm for solving first-order differential equations numerically is also discussed.

2.1 Introduction

If you were to take a laboratory-based course in physics, you would soon be introduced to the oscilloscope. You would learn the function of many of the knobs, how to read the display, and how to connect various devices so that you could measure various quantities. If you did not know already, you would learn about voltage, current, impedance, and AC and DC signals. Your goal would be to learn how to use the oscilloscope. In contrast, you would learn only a little about the inner workings of the oscilloscope.

The same approach can be easily adopted with an object-oriented language such as Java. If you are new to programming, you will learn how to make Java do what you want, but you will not learn everything about Java. In this chapter, we will present some of the essential syntax of Java and introduce the Open Source Physics library, which will facilitate writing programs with a graphical user interface and visual output such as plots and animations.

One of the ways that science progresses is by making models. If the model is sufficiently detailed, we can determine its behavior and then compare the behavior with experiment. This comparison might lead to verification of the model, changes in the model, and further simulations and experiments. In the context of computer simulation, we usually begin with a set of initial conditions, determine the dynamical behavior of the model numerically, and generate data in the form of tables of numbers, plots, and animations. We begin with a simple example to see how this process works.

Imagine a particle such as a ball near the surface of the Earth subject to a single force, the force of gravity. We assume that air friction is negligible, and the gravitational force is given by

$$F_g = -mg \tag{2.1}$$

where m is the mass of the ball and $g = 9.8\,\mathrm{N/kg}$ is the gravitational field (force per unit mass) near the Earth's surface. To make our example as simple as possible, we first assume that there is only vertical motion. We use Newton's second law to find the motion of the ball,

$$m\frac{d^2y}{dt^2} = F \tag{2.2}$$

where y is the vertical coordinate defined so that up is positive, t is the time, F is the total force on the ball, and m is the inertial mass [which is the same as the gravitational mass in (2.1)]. If we set $F = F_g$, (2.1) and (2.2) lead to

$$\frac{d^2y}{dt^2} = -g. \tag{2.3}$$

Equation (2.3) is a statement of a model for the motion of the ball. In this case the model is in the form of a second-order differential equation.

You are probably familiar with the model summarized in (2.3) and know the analytic solution:

$$y(t) = y(0) + v(0)t - \frac{1}{2}gt^2 \tag{2.4a}$$

$$v(t) = v(0) - gt. \tag{2.4b}$$

Nevertheless, we will determine the motion of a freely falling particle numerically in order to introduce the tools that we will need in a familiar context.

We begin by expressing (2.3) as two first-order differential equations:

$$\frac{dy}{dt} = v \tag{2.5a}$$

$$\frac{dv}{dt} = -g \tag{2.5b}$$

where v is the vertical velocity of the ball. We next *approximate* the derivatives by small (finite) differences:

$$\frac{y(t+\Delta t) - y(t)}{\Delta t} = v(t) \tag{2.6a}$$

$$\frac{v(t+\Delta t) - v(t)}{\Delta t} = -g. \tag{2.6b}$$

Note that in the limit $\Delta t \to 0$, (2.6) reduces to (2.5). We can rewrite (2.6) as

$$y(t+\Delta t) = y(t) + v(t)\Delta t \tag{2.7a}$$

$$v(t+\Delta t) = v(t) - g\Delta t. \tag{2.7b}$$

The finite difference approximation we used to obtain (2.7) is an example of the *Euler* algorithm. Equation (2.7) is an example of a *finite difference* equation, and Δt is the time step.

Now we are ready to follow $y(t)$ and $v(t)$ in time. We begin with an initial value for y and v and then *iterate* (2.7). If Δt is sufficiently small, we will obtain a numerical answer that is close to the solution of the original differential equations in (2.6). In this case we know the answer, and we can test our numerical results directly.

Exercise 2.1. A simple example

Consider the first-order differential equation

$$\frac{dy}{dx} = f(x) \tag{2.8}$$

where $f(x)$ is a function of x. The approximate solution as given by the Euler algorithm is

$$y_{n+1} = y_n + f(x_n)\Delta x. \tag{2.9}$$

Note that the rate of change of y has been approximated by its value at the *beginning* of the interval, $f(x_n)$.

(a) Suppose that $f(x) = 2x$ and $y(x = 0) = 0$. The analytic solution is $y(x) = x^2$, which we can confirm by taking the derivative of $y(x)$. Convert (2.8) into a finite difference equation using the Euler algorithm. For simplicity, choose $\Delta x = 0.1$. It would be a good idea to first use a calculator or pencil and paper to determine y_n for the first several time steps.

(b) Sketch the difference between the exact solution and the approximate solution given by the Euler algorithm. What condition would the rate of change $f(x)$ have to satisfy for the Euler algorithm to give the exact answer? □

Problem 2.2. Invent your own numerical algorithm

As we have mentioned, the Euler algorithm evaluates the rate of change of y by its value at the beginning of the interval, $f(x_n)$. The choice of where to approximate the rate of change of y during the interval from x to $x + \Delta x$ is arbitrary, although we will learn that some choices are better than others. All that is required is that the finite difference equation must reduce to the original differential equation in the limit $\Delta x \to 0$. Think of several other algorithms that are consistent with this condition. □

2.2 Simulating Free Fall

The source code for the *class* `FirstFallingBallApp` shown in Listing 2.1 is defined in a file named FirstFallingBallApp.java. The code consists of a sequence of *statements* that create *variables* and define *methods*. Each statement ends with a semicolon. Each source code file is compiled into *byte code* that can then be executed. The compiler places the byte code in a file with the same name as the Java source code file with the extension `class`. For example, the compiler converts FirstFallingBallApp.java into byte code and produces the FirstFallingBallApp.class file. One of the features of Java is that this byte code can be used by any computer that can run Java programs.

A Java application is a class that contains a *main* method. The following application is an implementation of the Euler algorithm given in (2.7). The program also compares the numerical and analytic results. We will next describe the syntax used in each line of the program.

Listing 2.1: First version of a simulation of a falling particle.

```java
// example of a single line comment statement (ignored by compiler)
package org.opensourcephysics.sip.ch02;      // location of file
// beginning of class definition
public class FirstFallingBallApp {
   // beginning of method definition
   public static void main(String[] args) {
      // braces { } used to group statements.
      // indent statements within a block so that
      // they can be easily identified
      // following statements form the body of main method
       // example of declaration and assignment statement
      double y0 = 10;
      double v0 = 0;    // initial velocity
      double t = 0;     // time
      double dt = 0.01; // time step
      double y = y0;
      double v = v0;
      double g = 9.8;   // gravitational field
19       // beginning of loop, n++ equivalent to n = n + 1
20       for(int n = 0;n<100;n++) {
21          // repeat following three statements 100 times
22          y = y+v*dt;   // indent statements in loop for clarity
23          v = v-g*dt;   // use Euler algorithm
24          t = t+dt;
25       }                // end of for loop
26       System.out.println("Results");
27       System.out.println("final time = "+t);
28       // display numerical result
29       System.out.println("y = "+y+" v = "+v);
30       // display analytic result
31       double yAnalytic = y0+v0*t-0.5*g*t*t;
32       double vAnalytic = v0-g*t;
33       System.out.println("analytic y = "+yAnalytic+" v = "+vAnalytic);
34    } // end of method definition
35 } // end of class definition
```

The first line in Listing 2.1 is an example of a single line comment statement. Comment statements are ignored by the computer but can be very important for the user. Multiple line comments begin with `/*` and end with `*/`. Javadoc comments begin with `/**`, but have been removed from the code listings in the book to save space. Download the source code from comPADRE to view the complete code with documentation.

The second line in Listing 2.1 declares a *package* name, which corresponds to the location (directory) of the source and byte code files. According to the package declaration, the file FirstFallingBallApp.java is in the directory org/opensourcephysics/sip/ch02. The package statement must be the first noncomment statement in the source file. For organizational convenience, it is a good idea to put related files in the same package. When executing a Java program, the Java Virtual Machine (the run-time environment) will search a specific set of directories (called the classpath) for the relevant class files. The documentation for your local development environment will describe how to specify the classpath.

The third line in Listing 2.1 declares the class name, `FirstFallingBallApp`. The Java convention is to begin a class name with an uppercase letter. If a name consists of more than one word, the words are joined together, and each succeeding word begins with an uppercase letter (another Java convention). The keyword `public` means that this class can be used by any other Java class.

Braces are used to delimit a block of code. The left brace {, after the name of the class, begins the body of the class definition, and the corresponding right brace }, inserted at the end of the code listing on line 31 ends the class definition.

The fourth line in Listing 2.1 begins the definition of the `main` method. A method describes a sequence of actions that use the associated data and can be *called* (invoked) within the class or by other classes. The `main` method has a special status in Java. To run a class as a stand-alone program (an application), the class must define the `main` method. (In contrast, a Java applet runs inside a browser and does not require a `main` method; instead, it has methods such as `init` and `start`.) The `main` method is the application's starting point. The argument of the `main` method will always be the same, and understanding its syntax is not necessary here.

Because the code for this book contains hundreds of classes, we will adopt our own convention that classes that define main methods have names that end with `App`. We sometimes refer to an application that we are about to run as the *target* class.

Familiarize yourself with your Java development environment by doing Exercise 2.3.

Exercise 2.3. Our first application

(a) Enter the listing of `FirstFallingBallApp` into a source file named FirstFallingBallApp.java. (Java programs can be written using any text editor that supports standard ASCII characters.) Be sure to pay attention to capitalization because Java is case sensitive. In what directory should you place the source file?

(b) Compile and run `FirstFallingBallApp`. Do the results look reasonable to you? In what directory did the compiler place the byte code? □

Digital computers represent numbers in base 2, that is, sequences of ones and zeros. Each one or zero is called a *bit*. For example, the number 13 is equivalent to 1101 or $(1\times 2^3)+(1\times 2^2)+(0\times 2^1)+(1\times 2^0)$. It would be difficult to write a program if we had to write numbers in base 2. Computer languages allow us to reference memory locations using identifiers or variable names. A valid variable name is a series of characters consisting of letters, digits, underscores, and dollar signs ($) that does not begin with a digit nor contain any spaces. Because Java distinguishes between upper and lowercase characters, `T` and `t` are different variable names. The Java convention is that variable names begin with a lowercase letter, except in special cases, and each succeeding word in a variable name begins with an uppercase letter.

In a purely object-oriented language, all variables would be objects that would be introduced by their class definitions. However, there are certain variable types that are so common that they have a special status and are especially easy to create and access. These types are called *primitive data types* and represent integer, floating point, boolean, and character variables. An example that illustrates that classes are effectively new programmer-defined types is given in Appendix 2A.

An integer variable, a floating point variable, and a boolean variable are created and *initialized* by the following statements:

```
int n = 10;
double y0 = 10.0;
boolean inert = true;
char c = 'A';     // used for single characters
```

There are four types of integers, `byte`, `short`, `int`, and `long`, and two types of floating point numbers; the differences are the range of numbers that these types can store. We will almost always use type `int` because it does not require as much memory as type `long`. There are two types of floating point numbers, but we will always use type `double`, the type with greater precision, to minimize roundoff error and to avoid having to provide multiple versions of various algorithms. A variable must be *declared* before it can be used, and it can be initialized at the same time that its type is declared as is done in Listing 2.1.

Integer arithmetic is exact, in contrast to floating point arithmetic which is limited by the maximum number of decimal places that can be stored. Important uses of integers are as counters in loops and as indices of arrays. An example of the latter is on page 38, where we discuss the motion of many balls.

A subtle and common error is to use integers in division when a floating point number is needed. For example, suppose we flip a coin 100 times and find 53 heads. What is the percentage of heads? In the following we show an unintended side effect of integer division and several ways of obtaining a floating point number from an integer.

```
int heads = 53;
int tosses = 100;
double percentage = heads/tosses;     // percentage will equal 0
percentage = (double)heads/tosses;    // percentage will equal 0.53
percentage = (1.0*heads)/tosses;      // percentage will equal 0.53
```

These statements indicate that if at least one number is a `double`, the result of the division will be a `double`. The expression `(double)heads` is called a *cast* and converts `heads` to a `double`. Because a number with a decimal point is treated as a double, we can also do this conversion by first multiplying heads by 1.0 as is done in the last statement.

Note that we have used the *assignment operator*, which is the equal (=) sign. This operator assigns the value to the memory location that is associated with a variable, such as `y0` and `t`. The following statements illustrate an important difference between the equal sign in mathematics and the assignment operator in most programming languages.

```
int x = 10;
x = x + 1;
```

The equal sign replaces a value in memory and is not a statement of equality. The left and right sides of an assignment operator are usually not equal.

A statement is analogous to a complete sentence, and an expression is similar to a phrase. The simplest expressions are identifiers or variables. More interesting expressions can be created by combining variables using *operators*, such as the following example of the plus (+) operator:

```
x + 3.0
```

Lines twelve through eighteen of Listing 2.1 declare and initialize variables. If a variable is declared but not initialized, for example,

```
double dt;
```

then the default value of the variable is 0 for numbers and `false` for boolean variables. It is a good idea to initialize all variables explicitly and not rely on their default values.

A very useful control structure is the `for` loop in line 15 of Listing 2.1. Loops are blocks of statements that are executed repeatedly until some condition is satisfied. They typically require the initialization of a counter variable, a test to determine if the counter variable has reached its terminal value, and a rule for changing the counter variable. These three parts of the `for` loop are contained within parentheses and are separated by semicolons. It is common in Java to iterate from 0 to 99, as is done in Listing 2.1, rather than from 1 to 100. Note the use of the ++ operator in the loop construct rather than the equivalent statement `n = n + 1`. It is important to indent all the statements within a block so that they can be easily identified. Java ignores these spaces, but they are important visual cues to the structure of the program.

After the program finishes the loop, the result is displayed using the `System.out.println` method. We will explain the meaning of this syntax later. The parameter passed to this method, which appears between the parentheses, is a `String`. A `String` is a sequence of characters and can be created by enclosing text in quotation marks as shown in the first `println` statement in Listing 2.1. We displayed our numerical results by using the + operator. When applied to a `String` and a number, the number is converted to the appropriate `String` and the two strings are concatenated (joined). This use is shown in the next three `println` statements in lines 27, 29, and 33 of Listing 2.1. Note the different outputs produced by the following two statements:

```
System.out.println(("x = " + 2) + 3);     // displays x = 23
System.out.println("x = " + (2 + 3));     // displays x = 5
```

The parentheses in the second line force the compiler to treat the enclosed + operator as the addition operator, but both + operators in the first line are treated as concatenation operators.

Exercise 2.4. Exploring FirstFallingBallApp

(a) Run `FirstFallingBallApp` for various values of the time step Δt. Do the numerical results become closer to the analytic results as Δt is made smaller?

(b) Use an acceptable value for Δt and run the program for various values for the number of iterations. What criteria do you have for acceptable? At approximately what time does the ball hit the ground at $y = 0$?

(c) What happens if you replace the `System.out.println` method by the `System.out.print` method?

(d) What happens if you try to access the value of the counter variable `n` outside the for loop? The *scope* of `n` extends from its declaration to the end of the loop block; `n` is said to have *block scope*. If a loop variable is not needed outside the loop, it should be declared in the initialization expression so that its scope is limited. □

You might have found that doing Exercise 2.4 was a bit tedious and frustrating. To do Exercise 2.4(a) it would be desirable to change the number of iterations at the same time that the value of Δt is changed so that we could compare the results for y and v at the same time. And it is difficult to do Exercise 2.4(b) because we don't know in advance how many iterations are needed to reach the ground. For starters we can improve `FirstFallingBallApp` using a `while` statement instead of the `for` loop.

```
while (y > 0) {
   // statements go here
}
```

In this example the boolean test for the `while` statement is done at the beginning of a loop.

It is also possible to do the test at the end:

```
do {
   // statements go here
}
while (y > 0);
```

Exercise 2.5. Using while statements

Modify `FirstFallingBallApp` so that the `while` statement is used and the program ends when the ball hits the ground at $y = 0$. Then repeat Exercise 2.4(b). □

Exercise 2.6. Summing a series

(a) Write a program to sum the following series for a given value of N:

$$S = \sum_{m=1}^{N} \frac{1}{m^2}. \tag{2.10}$$

The following statements may be useful:

```
double sum = 0;          // sum is equivalent to S in (2.10)
for (int m = 1; m <= N; m++) {
   sum = sum + 1.0/(m*m); // put this statement in for loop
}
```

Note that in this case it is more convenient to start the loop from $m = 1$ instead of $m = 0$. Also note that we have not followed the Java convention, because we have used the variable name `N` instead of `n` so that the Java statements look more like the mathematical equations.

(b) First run your program with $N = 10$. Then run for larger values of N. Does the series converge as $N \to \infty$? What value of N is needed to obtain S to within two decimal places?

(c) Modify your program so that it uses a while loop so that the summation continues until the added term to the sum is less than some value ϵ. Run your program for $\epsilon = 10^{-2}$, 10^{-3}, and 10^{-6}.

(d) Instead of using the = operator in the statement

```
sum = sum + 1.0/(m*m);
```

use the equivalent operator:

```
sum += 1.0/(m*m);
```

Check that you obtain the same results. □

Java provides several shortcut assignment operators that allow you to combine an arithmetic and an assignment operation. Table 2.1 shows the operators that we will use most often.

Operator	**Operand**	**Description**	**Sample Expression**	**Result**
`++, --`	number	increment, decrement	`x++;`	8.0 stored in `x`
`+, -`	numbers	addition, subtraction	`3.5 + x`	11.5
`!`	boolean	logical complement	`!(x == y)`	true
`=`	any	assignment	`y = 3;`	3.0 stored in `y`
`*, /, %`	numbers	multiplication, division, modulus	`7/2`	3.0
`==`	any	test for equality	`x == y`	false
`+=`	numbers	`x += 3;` equivalent to `x = x + 3;`	`x += 3;`	14.5 stored in `x`
`-=`	numbers	`x -= 2;` equivalent to `x = x - 2;`	`x -= 2.3;`	12.2 stored in `x`
`*=`	numbers	`x *= 4;` equivalent to `x = 4*x;`	`x *= 4;`	48.8 stored in `x`
`/=`	numbers	`x /= 2;` equivalent to `x = x/2;`	`x /= 2;`	24.4 stored in `x`
`%=`	numbers	`x %= 5;` equivalent to `x = x % 5;`	`x %= 5;`	4.4 stored in `x`

Table 2.1: Common operators. The result for each row assumes that the statements from previous rows have been executed with `double x = 7, y = 3` declared initially. The *mod* or *modulus* operator `%` computes the remainder after the division by an integer has been performed.

2.3 Getting Started with Object-Oriented Programming

The first step in making our program more object-oriented is to separate the implementation of the model from the implementation of other programming tasks such as producing output. In general, we will do so by creating two classes. The class that defines the model is shown in

Listing 2.2. The `FallingBall` class first declares several (instance) variables and one constant that can be used by any method in the class. To aid reusability, we need to be very careful about the accessibility of these class variables to other classes. For example, if we write `private double dt`, then the value of `dt` would only be available to the methods in `FallingBall`. If we wrote `public double dt`, then `dt` would be available to any class in any package that tried to access it. For our purposes we will use the default package protection, which means that the instance variables can be accessed by classes in the same package.

Listing 2.2: `FallingBall` class.

```
package org.opensourcephysics.sip.ch02;
public class FallingBall {
   double y, v, t;               // instance variables
   double dt;                    // default package protection
   final static double g = 9.8;

   public FallingBall() {        // constructor
      System.out.println("A new FallingBall object is created.");
   }

   public void step() {
      y = y+v*dt; // Euler algorithm for numerical solution
      v = v-g*dt;
      t = t+dt;
   }

   public double analyticPosition(double y0, double v0) {
      return y0+v0*t-0.5*g*t*t;
   }

   public double analyticVelocity(double v0) {
      return v0-g*t;
   }
}
```

As we will see, a class is a blueprint for creating objects, not an object itself. Except for the constant `g`, all the variable declarations in Listing 2.2 are *instance* variables. Each time an object is created or *instantiated* from the class, a separate block of memory is set aside for the instance variables. Thus, two objects created from the same class will, in general, have different values of the instance variables. We can insure that the value of a variable is the same for all objects created from the class by adding the word `static` to the declaration. Such a variable is called a *class* variable and is appropriate for the constant `g`. In addition, you might not want the quantity referred to by an identifier to change. For example, `g` is a constant of nature. We can prevent a change by adding the keyword `final` to the declaration. Thus the statement

```
final static double g = 9.8;
```

means that a single copy of the constant `g` will be created and shared among all the objects instantiated from the class. Without the `final` qualifier, we could change the value of a class variable in every instantiated object by changing it in any one object. Static variables and methods are accessed from another class using the class name without first creating an instance (see page 25).

Another Java convention is that the names of constants should be in upper case. But in physics the meaning of g, the gravitational field, and G, the gravitational constant, have completely different meanings. So we will disregard this convention if doing so makes our programs more readable.

We have used certain words such as `double`, `false`, `main`, `static`, and `final`. These reserved words cannot be used as variable names and are examples of *keywords*.

In addition to the four instance variables `y`, `v`, `t`, and `dt`, and one class variable `g`, the `FallingBall` class has four methods. The first method is `FallingBall` and is a special method known as the *constructor*. A constructor must have the same name as the class and does not have an explicit return type. We will see that constructors allocate memory and initialize instance variables when an object is created.

The second method is `step`, a name that we will frequently use to advance a system's coordinates by one time step. The qualifier `void` means that this method does not return a value.

The next two methods, `analyticPosition` and `analyticVelocity`, each return a double value and have arguments enclosed by parentheses, the parameter list. The list of parameters and their types must be given explicitly and be separated by commas. The parameters can be primitive data types or class types. When the method is invoked, the argument types must match that given in the definition or be convertible into the type given in the definition, but need not have the same names. (Convertible means that the given variable can be unambiguously converted into another data type. For example, an integer can always be converted into a double.) For example, we can write

```
double y0 = 10;       // declaration and assignment
int v0 = 0;           // note v0 is an integer
// v0 becomes a double before method is called
double y = analyticPosition(y0,v0);
double v = analyticVelocity(v0);
```

but the following statements are incorrect:

```
// can't convert String to double automatically
double y = analyticPosition(y0,"0");
// method expects only one argument
double v = analyticVelocity(v0,0);
```

If a method does not receive any parameters, the parentheses are still required as in method `step()`.

The `FallingBall` class in Listing 2.2 cannot be used in isolation because it does not contain a `main` method. Thus, we create a target class which we place in a separate file in the same package. This class will communicate with `FallingBall` and include the output statements. This class is shown in Listing 2.3.

Listing 2.3: `FallingBallApp` class.

```
// package statement appears before beginning of class definition
package org.opensourcephysics.sip.ch02;
// beginning of class definition
public class FallingBallApp {
   // beginning of method definition
   public static void main(String[] args) {
      // declaration and instantiation
      FallingBall ball = new FallingBall();
      // example of declaration and assignment statement
      double y0 = 10;
      double v0 = 0;
      // note use of dot operator to access instance variable
      ball.t = 0;
      ball.dt = 0.01;
      ball.y = y0;
      ball.v = v0;
      while(ball.y>0) {
         ball.step();
      }
      System.out.println("Results");
      System.out.println("final time = "+ball.t);
      // displays numerical results
      System.out.println("y = "+ball.y+" v = "+ball.v);
      // displays analytic results
      System.out.println("analytic y = "+ball.analyticPosition(y0, v0));
      System.out.println("analytic v = "+ball.analyticVelocity(v0));
      System.out.println("acceleration = "+FallingBall.g);
   } // end of method definition
} // end of class definition
```

Note how `FallingBall` is declared and *instantiated* by creating an object called `ball` and how the instance variables and the methods are accessed. The statement

```
FallingBall ball = new FallingBall();  // declaration and instantiation
```

is equivalent to two statements:

```
FallingBall ball;           // declaration
ball = new FallingBall();   // instantiation
```

The declaration statement tells the compiler that the variable `ball` is of type `FallingBall`. It is analogous to the statement `int x` for an integer variable. The `new` operator allocates memory for this object, initializes all the instance variables, and invokes the constructor. We can create two identical balls using the following statements:

```
FallingBall ball1 = new FallingBall();
FallingBall ball2 = new FallingBall();
```

The variables and methods of an object are accessed by using the *dot operator*. For example, the variable `t` of object `ball` is accessed by the expression `ball.t`, and the method `step` is called as `ball.step()`. Because the methods, `analyticPosition` and `analyticVelocity` return values of type `double`, they can appear in any expression in which a double-valued constant or variable can appear. In the present context the values returned by these two methods will be displayed by the `println` statement. Note that the static variable `g` in class `FallingBallApp` is accessed through the class name.

Exercise 2.7. Use of two classes

(a) Enter the listing of `FallingBall` into a file named FallingBall.java and `FallingBallApp` into a file named FallingBallApp.java and put them in the same directory. Run your program and make sure your results are the same as those found in Exercise 2.5.

(b) Modify `FallingBallApp` by adding a second instance variable `ball2` of the same type as `ball`. Add the necessary code to initialize `ball2`, iterate `ball2`, and display the results for both objects. Write your program so that the only difference between the two balls is the value of Δt. How much smaller does Δt have to be to reduce the error in the numerical results by a factor of two for the same final time? What about a factor of four? How does the error depend on Δt?

(c) Add the statement `FallingBall.g = 2.0` to your program from part (b) and use the same value of `dt` for `ball` and `ball2`. What happens when you try to compile the program?

(d) Delete the `final` qualifier for `g` in `FallingBall` and recompile and run your program. Is there any difference between the results for the two balls? Is there a difference between the results compared to what you found for $g = 9.8$?

(e) Remove the qualifier `static`. Now `g` must be accessed using the object name, `ball` or `ball2` instead of `FallingBall`. Recompile your program again, and run your program. How do the results for the two balls compare now?

(f) Explain in your own words the meaning of the qualifiers `static` and `final`. □

It is possible for a class to have more than one constructor. For example, we could have a second constructor defined by

```
public FallingBall(double dt) {
// "this.dt" refers to an instance variable that has the
// same name as the argument
   this.dt = dt;
}
```

Note the possible confusion of the variable name `dt` in the argument of the `FallingBall` constructor and the variable defined near the beginning of the `FallingBall` class. A variable that is passed to a method as an argument (parameter) or that is defined (created) within a method is known as a *local variable*. A variable that is defined outside of a method is known as an *instance variable*. Instance variables are more powerful than local variables because they can be referenced (used) anywhere within an object, and because their values are not lost when the execution of the method is finished. When a variable name conflict occurs, it is necessary to use the keyword `this` to access the instance variable. Otherwise, the program would access the variable in the argument (the local variable) with the same name.

Exercise 2.8. Multiple constructors

(a) Add a second constructor with the argument `double dt` to `FallingBall`, but make no other changes. Run your program. Nothing changed because you didn't use this new constructor.

(b) Now modify `FallingBallApp` to use the new constructor:

```
 // declaration and instantiation
FallingBall ball = new FallingBall(0.01);
```

What statement in `FallingBallApp` can now be removed? Run your program and make sure it works. How can you tell that the new constructor was used?

(c) Show that the number of parameters and their type in the argument list determines which constructor is used in `FallingBall`. For example, show that the statements

```
double tau = 0.01;
// declaration and instantiation
FallingBall ball = new FallingBall(tau);
```

are equivalent to the syntax used in part (b). □

It is easy to create additional models for other kinds of motion. Cut and paste the code in the `FallingBall` into a new file named SHO.java, and change the code to solve the following two first-order differential equations for a ball attached to a spring:

$$\frac{dx}{dt} = v \tag{2.11a}$$

$$\frac{dv}{dt} = -\frac{k}{m}x \tag{2.11b}$$

where x is the displacement from equilibrium and k is the spring constant. Note that the new class shown in Listing 2.4 has a structure similar to that of the class shown in Listing 2.2.

Listing 2.4: SHO class.

```
package org.opensourcephysics.sip.ch02;
public class SHO {
   double x, v, t;
   double dt;
   double k = 1.0;                // spring constant
   double omega0 = Math.sqrt(k); // assume unit mass

   public SHO() {                 // constructor
      System.out.println("A new harmonic oscillator object is created.");
   }

   public void step() {
      // modified Euler algorithm
      v = v-k*x*dt;
      x = x+v*dt; // note that updated v is used
      t = t+dt;
   }

   public double analyticPosition(double y0, double v0) {
      return y0*Math.cos(omega0*t)+v0/omega0*Math.sin(omega0*t);
   }

   public double analyticVelocity(double y0, double v0) {
      return -y0*omega0*Math.sin(omega0*t)+v0*Math.cos(omega0*t);
   }
}
```

Exercise 2.9. Simple harmonic oscillator

(a) Explain how the implementation of the Euler algorithm in the `step` method of class `SHO` differs from what we did previously.

(b) The general form of the analytic solution of (2.11) can be expressed as

$$y(t) = A\cos\omega_0 t + B\sin\omega_0 t \tag{2.12}$$

where $\omega_0^2 = k/m$. What is the form of $v(t)$? Show that (2.12) satisfies (2.11) with $A = y(t = 0)$ and $B = v(t = 0)/\omega_0$. These analytic solutions are used in class SHO.

(c) Write a target class called SHOApp that creates an SHO object and solves (2.11). Start the ball with displacements of $x = 1$, $x = 2$, and $x = 4$. Is the time it takes for the ball to reach $x = 0$ always the same? □

The methods that we have written so far have been nonstatic methods (except for `main`). As we have seen, these methods cannot be used without first creating or instantiating an object. In contrast, *static* methods can be used directly without first creating an object. A class that is included in the core Java distribution and that we will use often is the `Math` class, which provides many common mathematical methods, including trigonometric, logarithmic, exponential, and rounding operations, and predefined constants. Some examples of the use of the `Math` class include:

```
double theta = Math.PI/4;        // constant pi defined in Math class
double u = Math.sin(theta);      // sine of theta
double v = Math.log(0.1);        // natural logarithm of 0.1
double w = Math.pow(10,0.4);     // 10 to the 0.4 power
double x = Math.atan(3.0);       // inverse tangent
```

Note the use of the dot notation in these statements and the Java convention that constants such as the value of π are written in uppercase letters, that is, `Math.PI`. Exercise 2.10 asks you to read the `Math` class documentation to learn about the methods in the `Math` class. To use these methods we need only to know what mathematical functions they compute; we do not need to know about the details of how the methods are implemented.

Exercise 2.10. The Math class

The documentation for Java is a part of most development environments. It can also be downloaded from <`docs.oracle.com/javase/8/docs/api/`>. Look for API docs and a link to the latest standard edition.

(a) Read the documentation of the `Math` class and describe the difference between the two versions of the arctangent method.

(b) Write a program to verify the output of several of the methods in the `Math` class. □

2.4 Inheritance

The falling ball and the simple harmonic oscillator have important features in common. Both are models of physical systems that represent a physical object as if all its mass was concentrated at a single point. Writing two separate classes by cutting and pasting is straightforward and reasonable because the programs are small and easy to understand. But this approach fails when the code becomes more complex. For example, suppose that you wish to simulate a model of a liquid consisting of particles that interact with one another according to some specified force law. Because such simulations are now standard (see Chapter 8), efficient code for such simulations is available. In principle, it would be desirable to use an already written program, assuming that you understood the nature of such simulations. However, in practice, using someone else's program can require much effort if the code is not organized properly. Fortunately, this situation is changing as more programmers learn object- oriented techniques and write their programs so that they can be used by others without needing to know the details of the implementation.

For example, suppose that you decided to modify an already existing program by changing to a different force law. You change the code and save it under a new name. Later you discover that you need a different numerical algorithm to advance the particles' positions and velocities. You again change the code and save the file under yet another name. At the same time the original author discovers a bug in the initialization method and changes her code. Your code is now out of date because it does not contain the bug fix. Although strict documentation and programming standards can minimize these types of difficulties, a better approach is to use object oriented features such as *inheritance*. Inheritance avoids duplication of code and makes it easier to debug a number of classes without needing to change each class separately.

We now write a new class that *encapsulates* the common features of the falling ball and the simple harmonic oscillator. We name this new class `Particle`. The falling ball and harmonic oscillator that we will define later implement their distinguishing features.

Listing 2.5: `Particle` class.

```
package org.opensourcephysics.sip.ch02;
abstract public class Particle {
   double y, v, t;      // instance variables
   double dt;           // time step

   public Particle() { // constructor
      System.out.println("A new Particle is created.");
   }

   abstract protected void step();
   abstract protected double analyticPosition();
   abstract protected double analyticVelocity();
}
```

The `abstract` keyword allows us to define the `Particle` class without knowing how the `step`, `analyticPosition`, and `analyticVelocity` methods will be implemented. Abstract classes are useful in part because they serve as templates for other classes. The abstract class contains some but not all of what a user will need. By making the class abstract, we must express the abstract idea of "particle" explicitly and customize the abstract class to our needs.

By using inheritance we now *extend* the `Particle` class (the superclass) to another class (the subclass). The `FallingParticle` class shown in Listing 2.6 implements the three abstract methods. Note the use of the keyword `extends`. We also have used a constructor with the initial position and velocity as arguments.

Listing 2.6: `FallingParticle` class.

```
package org.opensourcephysics.sip.ch02;
public class FallingParticle extends Particle {
   final static double g = 9.8;                  // constant
   // initial position and velocity
   private double y0 = 0, v0 = 0;

   public FallingParticle(double y, double v) { // constructor
      System.out.println("A new FallingParticle object is created.");
      this.y = y; // instance value set equal to passed value
      this.v = v; // instance value set equal to passed value
      y0 = y;     // no need to use "this" because there is only one y0
      v0 = v;
   }

   public void step() {
      y = y+v*dt; // Euler algorithm
      v = v-g*dt;
      t = t+dt;
   }

   public double analyticPosition() {
      return y0+v0*t-(g*t*t)/2.0;
   }

   public double analyticVelocity() {
      return v0-g*t;
   }
}
```

FallingParticle is a subclass of its superclass Particle. Because the methods and data of the superclass are available to the subclass (except those that are explicitly labeled private), FallingParticle inherits the variables y, v, t, and dt.[1]

We now write a target class to make use of our new abstraction. Note that we create a new FallingParticle, but assign it to a variable of type Particle.

Listing 2.7: FallingParticleApp class.

```
package org.opensourcephysics.sip.ch02;
// beginning of class definition
public class FallingParticleApp {
   // beginning of method definition
   public static void main(String[] args) {
      // declaration and instantiation
      Particle ball = new FallingParticle(10, 0);
      ball.t = 0;
      ball.dt = 0.01;
      while(ball.y>0) {
         ball.step();
      }
      System.out.println("Results");
      System.out.println("final time = "+ball.t);
      // numerical result
      System.out.println("y = "+ball.y+" v = "+ball.v);
      // analytic result
      System.out.println("y analytic = "+ball.analyticPosition());
   }   // end of method definition
} // end of class definition
```

Problem 2.11. Inheritance

(a) Run the FallingParticleApp class. How can you tell that the constructor of the superclass was called?

[1] In this case Particle and FallingParticle must be in the same package. If FallingParticle was in a different package, it would be able to access these variables only if they were declared protected or public.

(b) Rewrite the SHO class so that it is a subclass of Particle. Remove all unnecessary variables and implement the abstract methods.

(c) Write the target class SHOParticleApp to use the new SHOParticle class. Use the analyticPosition and analyticVelocity methods to compare the accuracy of the numerical and analytic answers in both the falling particle and harmonic oscillator models.

(d) Try to instantiate a Particle directly by calling the Particle constructor. Explain what happens when you compile this program. □

If you examine the console output in Problem 2.11a, you should find that whenever an object from the subclass is instantiated, the constructor of the superclass is executed as well as the constructor of the subclass. You will also find that an abstract class cannot be instantiated directly; it must be extended first.

Exercise 2.12. Extending classes

(a) Extend the FallingParticle and SHOParticle classes and give them names such as FallingParticleEC and SHOParticleEC, respectively. These subclasses should redefine the step method so that it first calculates the new velocity and then calculates the new position using the new velocity, that is,

```
public void step() {
   v = v - g*dt;          // falling ball
   y = y + v*dt;f
   t = t + dt;
}

public void step() {
   v = v - k*x*dt;        // harmonic oscillator
   x = x + v*dt;
   t = t + dt;
}
```

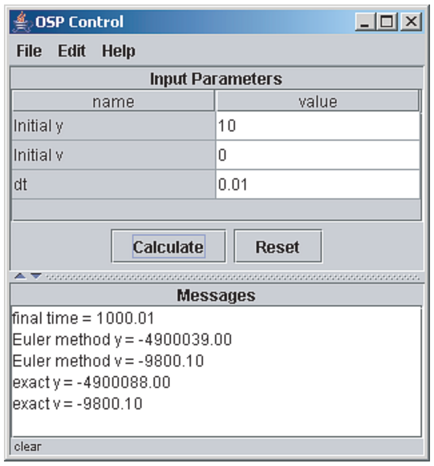
Methods can be redefined (overloaded) in the subclass by writing a new method in the subclass definition with the same name and parameter list as the superclass definition.

(b) Confirm that your new step method is executed instead of the one in the superclass.

(c) The algorithm that is implemented in the redefined step method is known as the *Euler–Cromer* algorithm. Compare the accuracy of this algorithm to the original Euler algorithm for both the falling particle and the harmonic oscillator. We will explore the Euler–Cromer algorithm in more detail in Problem 3.1. □

The falling particle and harmonic oscillator programs are simple, but they demonstrate important object-oriented concepts. However, we typically will not build our models using inheritance because our focus is on the physics and not on producing a software library, and also because readers will not use our programs in the same order. We will find that our main use of inheritance will be to extend abstract classes in the Open Source Physics library to implement calculations and simulations by customizing a small number of methods.

So far our target classes have only included one method, main. We could have used more than one method, but for the short demonstration and test programs we have written so far,



Input Parameters

name	value
Initial y	10
Initial v	0
dt	0.01

Messages

```
final time = 1000.01
Euler method y = -4900039.00
Euler method v = -9800.10
exact y = -4900088.00
exact v = -9800.10
```

Figure 2.1: An Open Source Physics control that is used to input parameter values and display results.

such a practice is unnecessary. When you send a short email to a friend, you are not likely to break up your message into paragraphs. But when you write a paper longer than about a half a page, it makes sense to use more than one paragraph. The same sensitivity to the need for structure should be used in programming. Most of the programs in the following chapters will consist of two classes, each of which will have several instance variables and methods.

2.5 The Open Source Physics Library

For each exercise in this chapter, you have had to change the program, compile it, and then run it. It would be much more convenient to input initial conditions and values for the parameters without having to recompile. However, a discussion of how to make input fields and buttons using Java would distract us from our goal of learning how to simulate physical systems. Moreover, the code we would use for input (and output) would be almost the same in every program. For this reason input and output should be in separate classes so that we can easily use them in all our programs. Our emphasis will be to describe how to use the Open Source Physics library as a tool for writing graphical interfaces, plotting graphs, and doing visualizations. If you are interested, you can read the source code of the many Open Source Physics classes and can modify or subclass them to meet your special needs.

We first introduce the Open Source Physics library in several simple contexts. Download the Open Source Physics library from <www.opensourcephysics.org> and include the library in your development environment. The following program illustrates how to make a simple plot.

Listing 2.8: An example of a simple plot.

```
package org.opensourcephysics.sip.ch02;
import org.opensourcephysics.frames.PlotFrame;

public class PlotFrameApp {
  public static void main(String[] args) {
    PlotFrame frame = new PlotFrame("x", "sin(x)/x", "Plot example");
```

```
      for(int i = 1;i<=100; i++) {
         double x = i*0.2;
         frame.append(0, x, Math.sin(x)/x);
      }
      frame.setVisible(true);
      frame.setDefaultCloseOperation(javax.swing.JFrame.EXIT_ON_CLOSE);
   }
}
```

The `import` statement tells the Java compiler where to find the Open Source Physics classes that are needed. A *frame* is often referred to as a window and can include a title and a menu bar as well as objects such as buttons, graphics, and text information. The Open Source Physics frames package defines several frames that contain data visualization and analysis tools. We will use the `PlotFrame` class to plot x-y data. The constructor for `PlotFrame` has three arguments corresponding to the name of the horizontal axis, the name of the vertical axis, and the title of the plot. To add data to the plot, we use the append method. The first argument of append is an integer that labels a particular set of data points, the second argument is the horizontal (x) value of the data point, and the third argument is the vertical (y) value. The `setVisible(true)` method makes a frame appear on the screen or brings it to the front. The last statement makes the program exit when the frame is closed. What happens when this statement is not included?

The example from the Open Source Physics library in Listing 2.9 illustrates how to control a *calculation* with two buttons, determine the value of an input parameter, and display the result in the text message area.

Listing 2.9: An example of a Calculation.

```
package org.opensourcephysics.sip.ch02;
// * means get all classes in controls subdirectory

import org.opensourcephysics.controls.*;

public class CalculationApp extends AbstractCalculation {
   public void calculate() { // Does a calculation
      control.println("Calculation button pressed.");
      // String must match argument of setValue
      double x = control.getDouble("x value");
      control.println("x*x = "+(x*x));
      control.println("random = "+Math.random());
   }

   public void reset() {
      // describes parameter and sets its value
      control.setValue("x value", 10.0);
   }

   // Creates a calculation control structure using this class
   public static void main(String[] args) {
      CalculationControl.createApp(new CalculationApp());
   }
}
```

`AbstractCalculation` is an *abstract* class, which as we have seen means that it cannot be instantiated directly and must be extended in order to implement the `calculate` method, that is, you must write (implement) the `calculate` method. You can also write an optional `reset` method, which is called whenever the `Reset` button is clicked. Finally, we need to create a graphical user interface that will invoke methods when the `Calculate` and `Reset` buttons are clicked. This user interface is an object of type `CalculationControl`:

```
CalculationControl.createApp(new CalculationApp());
```

The method `createApp` is a static method that instantiates an object of type `CalculationControl` and returns this object. We could have written

```
CalculationControl control = CalculationControl.createApp(new CalculationApp());
```

which shows explicitly the returned object which we gave the name `control`. However, because we do not use the object `control` explicitly in the `main` method, we do not need to actually declare an object name for it.

Exercise 2.13. CalculationApp

Compile and run `CalculationApp`. Describe what the graphical user interface looks like and how it works by clicking the buttons (see Figure 2.1). □

The `reset` method is called automatically when a program is first created and whenever the `Reset` button is clicked. The purpose of this method is to clear old data and recreate the initial state with the default values of the parameters and instance variables. The default values of the parameters are displayed in the control window so that they can be changed by the user. An example of how to show values in a control follows:

```
public void reset () {
   // describes parameter and sets the value
   control.setValue("x value",10.0);
}
```

The string appearing in the `setValue` method must be identical to the one appearing in the `getDouble` method. If you write your own `reset` method, it will override the `reset` method that is already defined in the `AbstractCalculation` superclass.

After the `reset` method stores the parameters in the control, the user can edit the parameters and we can later read these parameters using the `calculate` method:

```
public void calculate () {
   // String must match argument of setValue
   double x = control.getDouble("x value");
   control.println("x*x = " + (x*x));
}
```

Exercise 2.14. Changing parameters

(a) Run `CalculateApp` to see how the control window can be used to change a program's parameters. What happens if the string in the `getDouble` method does not match the string in the `setValue` method?

(b) Incorporate the plot statements in Listing 2.8 into a class that extends the `AbstractCalculation` class and plots the function $\sin kx$ for various values of the input parameter k. □

When you run the modified CalculationApp in Exercise 2.14, you should see a window with two buttons and an input parameter and its default value. Also, there should be a text area below the buttons where messages can appear. When the Calculate button is clicked, the calculate method is executed. The control.getDouble method reads in values from the control window. These values can be changed by the user. Then the calculation is performed and the result displayed in the message area using the control.println method, similar to the way we used System.out.println earlier. If the Reset button is clicked, the message area is cleared and the reset method is called.

We will now use a CalculationControl to change the input parameters for a falling particle. The modified FallingParticleApp is shown in Listing 2.10.

Listing 2.10: FallingParticleCalcApp class.

```
package org.opensourcephysics.sip.ch02;
import org.opensourcephysics.controls.*;

// beginning of class definition
public class FallingParticleCalcApp extends AbstractCalculation {
   public void calculate() {
      // gets initial conditions
      double y0 = control.getDouble("Initial y");
      double v0 = control.getDouble("Initial v");
      // sets initial conditions
      Particle ball = new FallingParticle(y0, v0);
      // reads parameters and sets dt
      ball.dt = control.getDouble("dt");
      while(ball.y>0) {
         ball.step();
      }
      control.println("final time = "+ball.t);
      // displays numerical results
      control.println("y = "+ball.y+" v = "+ball.v);
      // displays analytic position
      control.println("analytic y = "+ball.analyticPosition());
      // displays analytic velocity
      control.println("analytic v = "+ball.analyticVelocity());
   }

   public void reset() {
      control.setValue("Initial y", 10);
      control.setValue("Initial v", 0);
      control.setValue("dt", 0.01);
   }

   // creates a calculation control structure using this class
   public static void main(String[] args) {
      CalculationControl.createApp(new FallingParticleCalcApp());
   }
} // end of class definition
```

Exercise 2.15. Input of parameters and initial conditions

(a) Run FallingParticleCalcApp and make sure you understand how the control works. Try inputting different values of the parameters and the initial conditions.

(b) Vary Δt and find the value of t when $y = 0$ to two decimal places. □

Exercise 2.16. Displaying floating point numbers

Double precision numbers store 16 significant digits and every digit is included when the number is converted to a string. We can reduce the number of digits that are displayed using the `DecimalFormat` class in the `java.text` package. A formatter is created using a pattern, such as `#0.00` or `#0.00E0`, and this format is applied to a number to produce a string.

```java
DecimalFormat decimal2 = new DecimalFormat("#0.00");
double x = 1.0/3.0;
System.out.println("x = "+decimal2.format(x));  // displays 3.33
```

(a) Use the `DecimalFormat` class to modify the output from `FallingParticleCalcApp` so that it matches the output shown in Figure 2.1.

(b) Modify the output so that results are shown using scientific notation with three decimal places.

(c) The Open Source Physics `ControlUtils` class in the controls package contains a static method `f3` that formats a floating point number using three decimal places. Use this method to format the output from `FallingParticleCalcApp`. □

You probably have found that it is difficult to write a program so that it ends exactly when the falling ball is at $y = 0$. We could write the program so that Δt keeps changing near $y = 0$ so that the last value computed is at $y = 0$. Another limitation of our programs that we have written so far is that we have shown the results only at the end of the calculation. We could put `println` statements inside the while loop, but it would be better to plot the results and have a table of the data. An example is shown in Listing 2.11.

Listing 2.11: `FallingParticlePlotApp` class.

```java
package org.opensourcephysics.sip.ch02;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.frames.*;

public class FallingParticlePlotApp extends AbstractCalculation {
   PlotFrame plotFrame = new PlotFrame("t", "y", "Falling Ball");

   public void calculate() {
      // data not cleared at beginning of each calculation
      plotFrame.setAutoclear(false);
      // gets initial conditions
      double y0 = control.getDouble("Initial y");
      double v0 = control.getDouble("Initial v");
      // sets initial conditions
      Particle ball = new FallingParticle(y0, v0);
      // gets parameters
      ball.dt = control.getDouble("dt");
      double t = ball.t; // gets value of time from ball object
      while(ball.y>0) {
         ball.step();
         plotFrame.append(0, ball.t, ball.y);
```

```
            plotFrame.append(1, ball.t, ball.analyticPosition());
         }
      }

      public void reset() {
         control.setValue("Initial y", 10);
         control.setValue("Initial v", 0);
         control.setValue("dt", 0.01);
      }

      // sets up calculation control structure using this class
      public static void main(String[] args) {
         CalculationControl.createApp(new FallingParticlePlotApp());
      }
   }
```

The two data sets, indexed by 0 and 1, correspond to the numerical data and the analytic results, respectively. The default action in the Open Source Physics library is to clear the data and redraw data frames when the `Calculate` button is clicked. This automatic clearing of data can be disabled using the `setAutoclear` method. We have disabled it here to allow the user to compare the results of multiple calculations. Data is automatically cleared when the `Reset` button is clicked.

Exercise 2.17. Data output

(a) Run `FallingParticlePlotApp`. Under the Views menu choose DataTable to see a table of data corresponding to the plot. You can copy this data and use it in another program for further analysis.

(b) Your plotted results probably look like one set of data because the numerical and analytic results are similiar. Let `dt` = 0.1 and click the `Calculate` button. Does the discrepancy between the numerical and analytic results become larger with increasing time? Why?

(c) Run the program for two different values of `dt`. How do the plot and the table of data differ when two runs are done, first separated without clicking Reset, and then done by clicking Reset between calculations? Make sure you look at the entire table to see the difference. When is the data cleared? What happens if you eliminate the `plotFrame.setAutoclear(false)` statement? When is the data cleared now?

(d) Modify your program so that the velocity is shown in a separate window from the position.

□

2.6 Animation and Simulation

The `AbstractCalculation` class provides a structure for doing a single computation for a fixed amount of time. However, frequently we do not know how long we want to run a program, and it would be desirable if the user could intervene at any time. In addition, we would like to be able to visualize the results of a simulation and do an animation. To do so involves a programming construct called a *thread*. Threads enable a program to execute statements independently of each other as if they were run on separate processors (which would be the case on a multiprocessor computer). We will use one thread to update the model and display the results. The other thread, the event thread, will monitor the keyboard and mouse so that we can stop the computation whenever we desire.

The AbstractSimulation class provides a structure for doing simulations by performing a series of computations (steps) that can be started and stopped by the user using a graphical user interface. You will need to know nothing about threads because their use is "hidden" in the AbstractSimulation class. However, it is good to know that the Open Source Physics library is written so that the graphical user interface does not let us change a program's input parameters while the simulation is running. Most of the programs in the text will be done by extending the AbstractSimulation class and implementing the doStep method as shown in Listing 2.12. Just as the AbstractCalculation class uses the graphical user interface of type CalculationControl, the AbstractSimulation class uses one of type SimulationControl. This graphical user interface has three buttons whose labels change depending on the user's actions. As was the case with CalculationControl, the buttons in SimulationControl invoke specific methods.

Listing 2.12: A simple example of the extension of the AbstractSimulation class.

```java
package org.opensourcephysics.sip.ch02;
import org.opensourcephysics.controls.AbstractSimulation;
import org.opensourcephysics.controls.SimulationControl;

public class SimulationApp extends AbstractSimulation {
   int counter = 0;

   public void doStep() { // does a simulation step
      control.println("Counter = "+(counter--));
   }

   public void initialize() {
      counter = control.getInt("counter");
   }

   public void reset() { // invoked when reset button is pressed
      // allows dt to be changed after initializaton
      control.setAdjustableValue("counter", 100);
   }

   public static void main(String[] args) {
      // creates a simulation structure using this class
      SimulationControl.createApp(new SimulationApp());
   }
}
```

Exercise 2.18. AbstractSimulation class

Run SimulationApp and see how it works by clicking the buttons. Explain the role of the various buttons. How many times per second is the doStep method invoked when the simulation is running? □

The buttons in the SimulationControl that were used in SimulationApp in Listing 2.12 invoke methods in the AbstractSimulation class. These methods start and stop threads and perform other housekeeping chores. When the user clicks the Initialize button, the simulation's Initialize method is executed. When the Reset button is clicked, the reset method is executed. If you don't write your own versions of these two methods, their default versions will be used. After the `Initialize` button is clicked, it becomes the `Start` button. After the `Start` button is clicked, it is replaced by a Stop button, and the `doStep` method is invoked continually until the Stop button is clicked. The default is that the frames are redrawn every time `doStep` is executed. Clicking the Step button will cause the `doStep` method to be executed once. The New button changes the `Start` button to an `Initialize` button, which forces the user to initialize a new simulation before restarting. Later we will learn how to add other buttons that give the user even more control over the simulation.

A typical simulation needs to (1) specify the initial state of the system in the `initialize` method, (2) tell the computer what to execute while the thread is running in the `doStep` method, and (3) specify what state the system should return to in the `reset` method.

We could modify the falling particle model to use `AbstractSimulation`, but such a modification would not be very interesting because there is only one particle and all motion takes place in one dimension. Instead, we will define a new class that models a ball moving in two dimensions, and we will allow the ball to bounce off the ground and off of the walls.

Listing 2.13: `BouncingBall` class.

```
package org.opensourcephysics.sip.ch02;
import org.opensourcephysics.display.Circle;

 // Circle is a class that can draw itself
public class BouncingBall extends Circle {
   final static double g = 9.8;
   final static double WALL = 10;
   // initial position and velocity
   private double x, y, vx, vy;

   public BouncingBall(double x, double vx, double y, double vy) {
      this.x = x;   // sets instance value equal to passed value
      this.vx = vx; // sets instance value equal to passed value
      this.y = y;
      this.vy = vy;
      // sets the position using setXY in Circle superclass
      setXY(x, y);
   }

   public void step(double dt) {
      x = x+vx*dt; // Euler algorithm for numerical solution
      y = y+vy*dt;
      vy = vy-g*dt;
      if(x>WALL) {
         vx = -Math.abs(vx); // bounce off right wall
      } else if(x<-WALL) {
         vx = Math.abs(vx);  // bounce off left wall
      }
      if(y<0) {
         vy = Math.abs(vy); // bounce off floor
      }
      setXY(x, y);
   }
}
```

To model the bounce of the ball off a wall, we have added statements such as

```java
if (y < 0) vy = Math.abs(vy);
```

This statement insures that the ball will move up if $y < 0$, and is a crude implementation of an elastic collision. (The `Math.abs` method returns the absolute value of its argument.)

Note our first use of the `if` statement. The general form of an `if` statement is as follows:

```java
if (boolean_expression) {
   // code executed if boolean expression is true
} else {
   // code executed if boolean expression is false
}
```

We can test multiple conditions by chaining `if` statements:

```java
if (boolean_expression) {
   // code goes here
} else if (boolean_expression) {
   // code goes here
} else {
   // code goes here
}
```

If the first boolean expression is true, then only the statements within the first brace will be executed. If the first boolean expression is false, then the second boolean expression in the `else if` expression will be tested, and so forth. If there is an `else` expression, then the statements after it will be executed if all the other boolean expressions are false. If there is only one statement to execute, the braces are optional.

The `BouncingBall` class is similar to the `FallingBall` class except that it extends `Circle`. We inherit from the `Circle` class because this class includes a simple method that allows the object to draw itself in an Open Source Physics frame called `DisplayFrame`, which we will use in `BouncingBallApp`. In the latter we instantiate `BouncingBall` and `DisplayFrame` objects so that the circle will be drawn at its x-y location when the frame is displayed or while a simulation is running.

To make the animation more interesting, we will animate the motion of many noninteracting balls with random initial velocities. `BouncingBallApp` creates an arbitrary number of noninteracting bouncing balls by creating an array of `BouncingBall` objects.

Listing 2.14: `BouncingBallApp` class.

```java
package org.opensourcephysics.sip.ch02;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.frames.*;

public class BouncingBallApp extends AbstractSimulation {
   // declares and instantiates a window to draw balls
   DisplayFrame frame = new DisplayFrame("x", "y", "Bouncing Balls");
   BouncingBall[] ball; // declares an array of BouncingBall objects
   double time, dt;

   public void initialize() {
      // sets boundaries of window in world coordinates
      frame.setPreferredMinMax(-10.0, 10.0, 0, 10);
      time = 0;
      frame.clearDrawables(); // removes old particles
      int n = control.getInt("number of balls");
      int v = control.getInt("speed");
      // instantiates array of n BouncingBall objects
      ball = new BouncingBall[n];
      for(int i = 0;i<n;i++) {
         double theta = Math.PI*Math.random(); // random angle
         // instantiates the ith BouncingBall object
         ball[i] = new BouncingBall(0, v*Math.cos(theta), 0, v*Math.sin(theta));
         // adds ball to frame so that it will be displayed
         frame.addDrawable(ball[i]);
      }
      // decimalFormat instantiated in superclass and used to format numbers conveniently
       // message appears in lower right hand corner
      frame.setMessage("t = "+decimalFormat.format(time));
   }

   // invoked every 1/10 second by timer in AbstractSimulation superclass
   public void doStep() {
      for(int i = 0;i<ball.length;i++) {
         ball[i].step(dt);
      }
      time += dt;
      frame.setMessage("t="+decimalFormat.format(time));
   }

   // invoked when start or step button is pressed
   public void startRunning() {
      dt = control.getDouble("dt");     // gets time step
   }

   public void reset() { // invoked when reset button is pressed
      // allows dt to be changed after initializaton
      control.setAdjustableValue("dt", 0.1);
      control.setValue("number of balls", 40);
      control.setValue("speed", 10);
   }

   public static void main(String[] args) {
      SimulationControl.createApp(new BouncingBallApp());
   }
}
```

Because we will advance the dynamical variables of each ball using a loop, we store them in an *array*. An array such as `ball` is a data structure that holds many objects (or primitive data) of the same type. The elements of an array are accessed using an index in square brackets. The index begins at 0 and ends at the length of the array minus 1. Arrays are created with the `new` operator and have several properties such as `length`. We will discuss arrays in more detail in Section 3.4.

In Listing 2.13 we represent each ball as an object of type `BouncingBall` in an array. This use of objects is appealing, but for better performance, it is usually better to store the positions and the velocities of the balls in an array of doubles. In Chapter 8 we will simulate a system of N mutually interacting particles. Because computational speed will be very important in this case, we will not allocate separate objects for each particle, and instead will treat the system of

N particles as one object.

The `initialize` method in `BouncingBallApp` reads the number of particles and creates an array of the appropriate length. Creating an array sets primitive variables to zero and object values to null. For this reason we next loop to create the balls and add each ball to the frame. We place each ball initially at (0,0) with a random velocity. To produce random angles for the initial velocity, the `Math.random()` method is used. This method returns a random double between 0 and 1, not including the exact value 1. We define the random angle to be between 0 and π so that the initial vertical component of the velocity is positive. Clicking the `Initialize` button removes old objects from the drawing.

Most programming languages, including Java, use pixels to define the location on a window, with the origin at the upper left-hand corner and the vertical coordinate increasing in the downward direction. This choice of coordinates is usually not convenient in physics, and it often is more convenient to choose coordinates such that the vertical coordinate increases upward. The `Circle.setXY` method uses *world* or physical coordinates to set the position of the circle, and its implementation converts these coordinates to pixels so that the Java graphics methods can be used. In `initialize` we set the boundaries for the *world* coordinates using the `setPreferredMinMax` method whose arguments are the minimum x-coordinate, maximum x-coordinate, minimum y-coordinate, and maximum y-coordinate, respectively.

The `doStep` method implements a straightforward loop to advance the dynamical state of each ball in the array. It then advances the time and displays the time in the frame. Frames are automatically redrawn each time the `doStep` method is executed.

Finally, we note that there are two types of input parameters. Some parameters, such as the number of particles, determine properties of the model that should not be changed after the model has been created. We refer to these parameters as *fixed* because their values should be determined when the model is initialized. Other parameters, such as the time step Δt, can be changed between computations, but should not be changed during a computation. For example, if the time step is changed while a differential equation is being solved, one variable might be advanced using the old value of the time step while another variable is advanced using the new value. This type of synchronization error can be avoided by reading the parameters before the `doStep` method is executed. If you wish to allow a parameter to be changed between computations, you can use the optional `startRunning` method. This method is invoked once when the `Step` button is clicked and once when the `Run` button is clicked. In other words this method is called before the thread starts and insures that the simulation has the opportunity to read the most recent values.

In `BouncingBallApp` the time step `dt` is set using the `setAdjustableValue` method rather than the `setValue` method. Parameters that are set using `setAdjustableValue` are editable in the `SimulationControl` after the program has been initialized, whereas those that are set using `setValue` are only editable before the program has been initialized.

Exercise 2.19. Follow the bouncing balls

(a) Run `BouncingBallApp` and try the different buttons and note how they affect the input parameters.

(b) Add the statement `enableStepsPerDisplay(true)` to the `reset` method, and run your program again. You should see a new input in the control window that lets you change the number of simulation steps that are computed between redrawing the frame. Vary this input and note what happens.

(c) What is wrong with the physics of the simulation?

(d) Add a method to the `BouncingBall` class to calculate and return the total energy. Sum the energy of the balls in the program's `doStep` method and display this value in the message box. Does the simple model make sense?

(e) Look at the source code for the `setXY` method. If you are using an Integrated Development Environment (IDE), finding the method and looking at the source code is easy. What would you need to do to change the radius of the circle that is drawn? □

Many of the visualization components in the Open Source Physics library are written using classes provided by others. The goal of this library is to make it easier for you to begin writing your own programs. You are encouraged to look under the hood as you gain experience. The Open Source Physics controls and visualizations will almost always inherit from the `JFrame` class. Drawing is almost always done on a `DrawingPanel` which inherits from the `JPanel` class. Both these superclasses are defined in the `javax.swing` package.

Exercise 2.20. Peeking into Open Source Physics

(a) Look at the source code for `PlotFrame` (in the frames package) and follow its inheritance until you reach the `JFrame` class. How many subclasses are there between `JFrame` and `PlotFrame`? Follow the inheritance from `SimulationControl` (in the controls package) to `JFrame`. Describe in general terms what features are added in each subclass.

(b) Read through the different methods in `PlotFrame`. Don't worry about how the methods are implemented, but try to understand what they do. Which methods have not yet appeared in a program listing? When might you use them?

(c) Look at the source code for `PlottingPanel` (in the display package), which is used in many of the frames. Follow its inheritance until you reach the `JPanel` class. Do you see why we have not described the `PlottingPanel` class in detail? Look through the various methods, and describe in your own words what several of them do and how they might be used.

(d) Find the closest common ancestor (superclass) for `JFrame` and `JPanel` in the core Java library. Note that all objects have `Object` as a common ancestor. □

2.7 Model-View-Controller

Developing large software programs is best viewed as a design process. One criterion for good design is the reuse of data structures and behaviors that can facilitate reuse. Separating the physics (the model) from the user interface (the controller) and the data visualization (the view) facilitate good design. In Open Source Physics, the control object is responsible for handling user initiated events, such as button clicks, and passing them to other objects. The plots that we have constructed present visual representations of the data and are examples of a *view*. By using this design strategy, it is possible to have multiple views of the same data. For example, we can show a plot and a table view of the same data. The physics is expressed in terms of a *model* which contains the data and provides the methods by which the data can change.

At this point we have described a large fraction of the Java syntax and Open Source Physics tools that we will need in the rest of this book. One important topic that we still need to discuss is the use of *interfaces*. There is also much more in the Open Source Physics library that we can

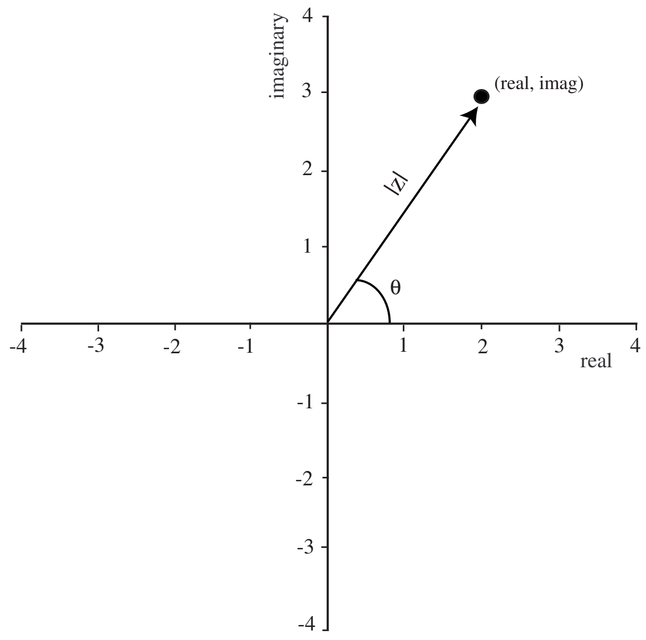


Figure 2.2: A complex number z can be defined by its real and imaginary parts, *real* and *imag*, respectively, or by its magnitude $|z|$ and phase angle θ.

use. For example, there are classes to draw and manipulate lattices as well as classes to iterate differential equations more accurately than the Euler method used in this chapter.

At this stage we hope that you have gained a feel for how Java works, and can focus on the physics in the rest of the text. Additional aspects of Java will be taught by example as they are needed.

Appendix 2A: Complex Numbers

Complex numbers are used in physics to represent quantities such as alternating currents and quantum mechanical wave functions which have an amplitude and phase (see Figure 2.2). Java does not provide a complex number as a primitive data type, so we will write a class that implements some common complex arithmetic operations. This class is an explicit example of the fact that classes are effectively new programmer-defined types.

If our new class is called `Complex`, we could test it by using code such as the following:

```
package org.opensourcephysics.sip.ch02;
public class ComplexApp {
   public static void main(String[] args) {
      Complex a = new Complex(3.0, 2.0);  // complex number 3 + i2
      Complex b = new Complex(1.0, -4.0); // complex number 1 - i4
      System.out.println(a);     // display a using a.toString()
      System.out.println(b);     // display b using b.toString()
      Complex sum = b.add(a);  // add a to b
      System.out.println(sum); // display sum
      Complex product = b.multiply(a); // multiply b by a
      System.out.println(product);     // display product
      a.conjugate();                   // complex conjugate of a
      System.out.println(a);
   }
}
```

Because the methods of class `Complex` are not static, we must first instantiate a `Complex` object with a statement such as

```
Complex a = new Complex(3.0, 2.0);
```

The variable a is an object of class `Complex`. As before, we can think of `new` as creating the instance variables and memory of the object. Compare the form of this statement to the declaration

```
double x = 3.0;
```

A variable of class type `Complex` is literally more complex than a primitive variable because its definition also involves associated methods and instance variables.

Note that we have first written a class that uses the `Complex` class before we have actually written the latter. Although programming is an iterative process, it is usually a good idea to first think about how the objects of a class are to be used. Exercise 2.21 encourages you to do so.

Exercise 2.21. Complex number test

What will be the output when `ComplexApp` is run? Make reasonable assumptions about how the methods of the `Complex` class will perform using your knowledge of Java and complex numbers. □

We need to define methods that add, multiply, and take the conjugate of complex numbers and define a method that prints their values. We next list the code for the `Complex` class.

Listing 2.15: Listing of the `Complex` class.

```
package org.opensourcephysics.sip.ch02;
public class Complex {
   private double real = 0;
   private double imag = 0;

   public Complex() {
      this(0, 0); // invokes second constructor with 0 + i0
   }

   public Complex(double real, double imag) {
      this.real = real;
      this.imag = imag;
   }

   public void conjugate() {
      imag = -imag;
   }

   public Complex add(Complex c) {
      // result also is complex so need to introduce another variable
      // of type Complex
      Complex sum = new Complex();
      sum.real = real+c.real;
      sum.imag = imag+c.imag;
      return sum;
   }

   public Complex multiply(Complex c) {
      Complex product = new Complex();
      product.real = (real*c.real)-(imag*c.imag);
      product.imag = (real*c.imag)+(imag*c.real);
      return product;
   }

   public String toString() {
      // note example of method overriding
      if(imag>=0) {
          return real+" + i"+Math.abs(imag);
      } else {
          return real+" - i"+Math.abs(imag);
      }
   }
}
```

The Complex class defines two constructors that are distinguished by their parameter list. The constructor with two arguments allows us to initialize the values of the instance variables. Notice how the class *encapsulates* (hides) both the data and the methods that characterize a complex number. That is, we can use the Complex class without any knowledge of how its methods are implemented or how its data is stored.

The general features of this class definition are as before. The variables real and imag are the instance variables of class Complex. In contrast, the variable sum in method add is a *local* variable because it can be accessed only within the method in which it is defined.

The most important new feature of the Complex class is that the add and multiply methods return new Complex objects. One reason we need to return a variable of type Complex is that a method returns (at most) a *single* value. For this reason we cannot return both sum.real and sum.imag. More importantly, we want the sum of two complex numbers to be also of type Complex so that we can add a third complex number to the result. Note also that we have defined add and multiply so that they do not change the values of the instance variables of the numbers to be added, but create a new complex number that stores the sum.

Exercise 2.22. Complex numbers

Another way to represent complex numbers is by their magnitude and phase, $|z|e^{\imath\theta}$. If $z = a + \imath b$, then

$$|z| = \sqrt{a^2 + b^2} \tag{2.13a}$$

and

$$\theta = \arctan\frac{b}{a}. \tag{2.13b}$$

(a) Write methods to get the magnitude and phase of a complex number, getMagnitude and `getPhase`, respectively. Add test code to invoke these methods. Be sure to check the phase in all four quadrants.

(b) Create a new class named `ComplexPolar` that stores a complex number as a magnitude and phase. Define methods for this class so that it behaves the same as the `Complex` class. Test this class using the code for `ComplexApp`. □

This example of the `Complex` class illustrates the nature of objects, their limitations, and the tradeoffs that enter into design choices. Because accessing an object requires more computer time than accessing primitive variables, it is faster to represent a complex number by two doubles, corresponding to its real and imaginary parts. Thus N complex data points could be represented by an array of $2N$ doubles, with the first N values corresponding to the real values. Considerations of computational speed are important only if complex data types are used extensively.

References and Suggestions for Further Reading

By using the Open Source Physics library, we have hidden most of the Java code needed to use threads, and have only touched on the graphical capabilities of Java. See the *Open Source Physics: A User's Guide with Examples* for a description of additional details on how threads and the other Open Source Physics tools are implemented and used. The source code for all the programs in the text and the Open Source Physics library can be downloaded from `<www.compadre.org/portal/items/detail.cfm?ID=7147>`.

There are many good books on Java graphics and Java threads. We list a few of our favorites in the following.

David M. Geary, *Graphic Java: Vol. 2, Swing*, 3rd ed. (Prentice Hall, 1999).

Jonathan Knudsen, *Java 2D Graphics* (O'Reilly, 1999).

Scott Oaks and Henry Wong, *Java Threads*, 3rd ed. (O'Reilly, 2004).

Chapter 3

Simulating Particle Motion

We discuss several numerical methods needed to simulate the motion of particles using Newton's laws and introduce *interfaces*, an important Java construct that makes it possible for unrelated objects to declare that they perform the same methods.

3.1 Modified Euler algorithms

To motivate the need for a general differential equation solver, we discuss why the simple Euler algorithm is insufficient for many problems. The Euler algorithm assumes that the velocity and acceleration do not change significantly during the time step Δt. Thus, to achieve an acceptable numerical solution, the time step Δt must be chosen to be sufficiently small. However, if we make Δt too small, we run into several problems. As we do more and more iterations, the round-off error due to the finite precision of any floating point number will accumulate, and eventually the numerical results will become inaccurate. Also, the greater the number of iterations, the greater the computer time required for the program to finish. In addition to these problems, the Euler algorithm is unstable for many systems, which means that the errors accumulate exponentially, and thus the numerical solution becomes inaccurate very quickly. For these reasons more accurate and stable numerical algorithms are necessary.

To illustrate why we need algorithms other than the simple Euler algorithm, we make a very simple change in the Euler algorithm and write

$$v(t+\Delta t) = v(t) + a(t)\Delta t \tag{3.1a}$$

$$y(t+\Delta t) = y(t) + v(t+\Delta t)\Delta t \tag{3.1b}$$

where a is the acceleration. The only difference between this algorithm and the simple Euler algorithm,

$$v(t+\Delta t) = v(t) + a(t)\Delta t \tag{3.2a}$$

$$y(t+\Delta t) = y(t) + v(t)\Delta t \tag{3.2b}$$

is that the computed velocity at the end of the interval, $v(t+\Delta t)$, is used to compute the new position, $y(t+\Delta t)$ in (3.1b). As we found in Problem 2.12 and will see in more detail in Problem 3.1, this modified Euler algorithm is significantly better for oscillating systems. We refer to this algorithm as the Euler–Cromer algorithm.

Problem 3.1. Comparing Euler algorithms

(a) Write a class that extends `Particle` and models a simple harmonic oscillator for which $F = -kx$. For simplicity, choose units such that $k = 1$ and $m = 1$. Determine the numerical error in the position of the simple harmonic oscillator after the particle has evolved for several cycles. Is the original Euler algorithm stable for this system? What happens if you run for longer times?

(b) Repeat part (a) using the Euler–Cromer algorithm. Does this algorithm work better? If so, in what way?

(c) Modify your program so that it computes the total energy, $E_{\text{sho}} = v^2/2 + x^2/2$. How well is the total energy conserved for the two algorithms? Also consider the quantity $\tilde{E} = E_{\text{sho}} + (\Delta t/2)xp$. What is the behavior of this quantity for the Euler–Cromer algorithm? □

Perhaps it has occurred to you that it would be better to compute the velocity at the middle of the interval rather than at the beginning or at the end. The *Euler–Richardson* algorithm is based on this idea. This algorithm is particularly useful for velocity-dependent forces, but does as well as other simple algorithms for forces that do not depend on the velocity. The algorithm consists of using the Euler algorithm to find the intermediate position y_{mid} and velocity v_{mid} at a time $t_{\text{mid}} = t + \Delta t/2$. We then compute the force, $F(y_{\text{mid}}, v_{\text{mid}}, t_{\text{mid}})$ and the acceleration a_{mid} at $t = t_{\text{mid}}$. The new position y_{n+1} and velocity v_{n+1} at time t_{n+1} are found using v_{mid} and a_{mid} and the Euler algorithm. We summarize the Euler–Richardson algorithm as:

$$a_n = F(y_n, v_n, t_n)/m \tag{3.3a}$$

$$v_{\text{mid}} = v_n + \frac{1}{2} a_n \Delta t \tag{3.3b}$$

$$y_{\text{mid}} = y_n + \frac{1}{2} v_n \Delta t \tag{3.3c}$$

$$a_{\text{mid}} = F\Big(y_{\text{mid}}, v_{\text{mid}}, t + \frac{1}{2}\Delta t\Big)/m \tag{3.3d}$$

and

$$v_{n+1} = v_n + a_{\text{mid}} \Delta t \tag{3.4a}$$

$$y_{n+1} = y_n + v_{\text{mid}} \Delta t \qquad \text{(Euler–Richardson algorithm).} \tag{3.4b}$$

Although we need to do twice as many computations per time step, the Euler–Richardson algorithm is much faster than the Euler algorithm because we can make the time step larger and still obtain better accuracy than with either the Euler or Euler–Cromer algorithms. A derivation of the Euler–Richardson algorithm is given in Appendix 3A.

Exercise 3.2. The Euler–Richardson algorithm

(a) Extend `FallingParticle` in Listing 2.6 to a new class that implements the Euler–Richardson algorithm. All you need to do is write a new step method.

(b) Use $\Delta t = 0.08$, 0.04, 0.02, and 0.01 and determine the error in the computed position when the particle hits the ground. How do your results compare with the Euler algorithm? How does the error in the velocity depend on Δt for each algorithm?

(c) Repeat part (b) for the simple harmonic oscillator and compute the error after several cycles. □

As we gain more experience simulating various physical systems, we will learn that no single algorithm for solving Newton's equations of motion numerically is superior under all conditions.

The Open Source Physics library includes classes that can be used to solve systems of coupled first-order differential equations using different algorithms. To understand how to use this library, we first discuss *interfaces* and then *arrays*.

3.2 Interfaces

We have seen how to combine data and methods into a class. A class definition *encapsulates* this information in one place, thereby simplifying the task of the programmer who needs to modify the class and the user who needs to understand or use the class.

Another tool for data abstraction is known as an *interface*. An interface specifies methods that an object performs but does not implement these methods. In other words, an interface describes the behavior or functionality of any class that implements it. Because an interface is not tied to a given class, any class can *implement* any particular interface as long as it defines all the methods specified by the interface. An important reason for interfaces is that a class can inherit from only one superclass, but it can implement more than one interface.

An example of an interface is the `Function` interface in the numerics package:

```
public interface Function {
   public double evaluate (double x);
}
```

The interface contains one method, `evaluate`, with one argument, but no body. Notice that the definition uses the keyword `interface` rather then the keyword `class`.

We can define a class that encapsulates a quadratic polynomial as follows:

```
public class QuadraticPolynomial implements Function {
   double a,b,c;

   public QuadraticPolynomial(double a, double b, double c) {
      this.a = a;
      this.b = b;
      this.c = c;
   }

   public double evaluate (double x) {
      return a*x*x + b*x + c;
   }
}
```

Quadratic polynomials can now be instantiated and used as needed.

```
Function f = new QuadraticPolynomial(1,0,2);
for(int x = 0; x < 10; x++) {
   System.out.println("x = " + x + "  f(x)" + f.evaluate(x));
}
```

By using the `Function` interface, we can write methods that use this mathematical abstraction. For example, we can program a simple plot as follows:

```
public void plotFunction(Function f, double xmin, double xmax) {
   PlotFrame frame = new PlotFrame("x","y", "Function");
   double n = 100;          // number of points in plot
   double x = xmin, dx = (xmax - xmin)/(n-1);
   for (int i = 0; i < 100; i++) {
      frame.append(0,x,f.evaluate(x));
      x += dx;
   }
   frame.setVisible(true);    // display frame on screen
}
```

We can also compute a numerical derivative based on the definition of the derivative found in calculus textbooks.

```
public double derivative(Function f, double x, double dx) {
   return (f.evaluate(x+dx) - f.evaluate(x))/dx;
}
```

This way of approximating a derivative is not optimum, but that is not the point here. (A better approximation is given in Problem 3.8.) The important point is that the interface enables us to define the abstract concept $y = f(x)$ and to write code that uses this abstraction.

Exercise 3.3. Function interface

(a) Define a class that encapsulates the function $f(u) = ae^{-bu^2}$.

(b) Write a test program that plots $f(u)$ with $b = 1$ and $b = 4$. Choose $a = 1$ for simplicity.

(c) Write a test program that plots the derivatives of the functions used in part (b) without using the analytic expression for the derivative. □

Although interfaces are very useful for developing large scale software projects, you will not need to define interfaces to do the problems in this book. However, you will use several interfaces, including the `Function` interface, that are defined in the Open Source Physics library. We describe two of the more important interfaces in the following sections.

3.3 Drawing

An interface that we will use often is the `Drawable` interface:

```
package org.opensourcephysics.display;
import java.awt.*;

public interface Drawable {
   public void draw (DrawingPanel panel, Graphics g);
}
```

Notice that this interface contains only one method, `draw`. Objects that implement this interface are rendered in a `DrawingPanel` after they have been added to a `DisplayFrame`. As we saw in Chapter 2, a `DisplayFrame` consists of components including a title bar, menu, and buttons for minimizing and closing the frame. The DisplayFrame contains a DrawingPanel on which graphical output will be displayed. The Graphics class contains methods for drawing simple geometrical objects such as lines, rectangles, and ovals on the panel. In Listing 3.1 we define a class that draws a rectangle using pixel-based coordinates.

Listing 3.1: PixelRectangle.

```
package org.opensourcephysics.sip.ch03;
import java.awt.*; // uses Abstract Window Toolkit
import org.opensourcephysics.display.*;

public class PixelRectangle implements Drawable {
   int left, top;      // position of rectangle in pixels
   int width, height; // size of rectangle in pixels

   PixelRectangle(int left, int top, int width, int height) {
      this.left = left; // location of left edge
      this.top = top;    // location of top edge
      this.width = width;
      this.height = height;
   }

   public void draw(DrawingPanel panel, Graphics g) {
      // this method implements the Drawable interface
      g.setColor(Color.RED);                    // set drawing color to red
      g.fillRect(left, top, width, height); // draws rectangle
   }
}
```

In method draw we used fillRect, a primitive method in the Graphics class. This method draws a filled rectangle using pixel coordinates with the origin at the top left corner of the panel.

To use PixelRectangle, we instantiate an object and add it to a DisplayFrame as shown in Listing 3.2.

Listing 3.2: Listing of DrawingApp.

```
package org.opensourcephysics.sip.ch03;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.display.*;
import org.opensourcephysics.frames.*;

public class DrawingApp extends AbstractCalculation {
   DisplayFrame frame = new DisplayFrame("x", "y", "Graphics");

   public DrawingApp() {
      frame.setPreferredMinMax(0, 10, 0, 10);
   }

   public void calculate() {
      // gets rectangle location
      int left = control.getInt("xleft");
      int top = control.getInt("ytop");
      // gets rectangle dimensions
      int width = control.getInt("width");
      int height = control.getInt("height");
      Drawable rectangle = new PixelRectangle(left, top, width, height);
      frame.addDrawable(rectangle);
      // frame is automatically rendered after Calculate button
      // is clicked
   }

   public void reset() {
      // removes drawables added by the user
      frame.clearDrawables();
      // sets default input values
      control.setValue("xleft", 60);
      control.setValue("ytop", 70);
      control.setValue("width", 100);
      control.setValue("height", 150);
   }

   // creates a calculation control structure using this class
   public static void main(String[] args) {
      CalculationControl.createApp(new DrawingApp());
   }
}
```

Note that multiple rectangles are drawn in the order that they are added to the drawing panel. Rectangles or portions of rectangles may be hidden because they are outside the drawing panel.

Although it is possible to use pixel-based drawing methods to produce visualizations, creating even a simple graph in such an environment would require much tedious programming. The `DrawingPanel` object passed to the `draw` method simplifies this task by defining a system of *world coordinates* that enable us to specify the location and size of various objects in physical units rather than pixels. In the `WorldRectangle` class in Listing 3.3, methods from the `DrawingPanel` class are used to convert pixel coordinates to world coordinates. The range of the world coordinates in the horizontal and vertical directions is defined in the `frame.setPreferredMinMax` method in `DrawingApp`. (This method is not needed if pixel coordinates are used.)

Listing 3.3: `WorldRectangle` illustrates the use of world coordinates.

```
package org.opensourcephysics.sip.ch03;
import java.awt.*;
import org.opensourcephysics.display.*;

public class WorldRectangle implements Drawable {
   double left, top;      // position of rectangle in world coordinates
   double width, height; // size of rectangle in world units

   public WorldRectangle(double left, double top, double width,
                         double height) {
      this.left = left;       // location of left edge
      this.top = top;         // location of top edge
      this.width = width;
      this.height = height;
   }

   public void draw(DrawingPanel panel, Graphics g) {
      // This method implements the Drawable interface
      g.setColor(Color.RED); // set drawing color to red
      // converts from world to pixel coordinates
      int leftPixels = panel.xToPix(left);
      int topPixels = panel.yToPix(top);
      int widthPixels = (int) (panel.getXPixPerUnit()*width);
      int heightPixels = (int) (panel.getYPixPerUnit()*height);
      // draws rectangle
      g.fillRect(leftPixels, topPixels, widthPixels, heightPixels);
   }
}
```

Exercise 3.4. Simple graphics

(a) Run `DrawingApp` and test how the different inputs change the size and location of the rectangle. Note that the pixel coordinates that are obtained from the control window are not the same as the world coordinates that are displayed.

(b) Read the documentation at `<java.sun.com/reference/api/>` for the `Graphics` class, and modify the `WorldRectangle` class to draw lines, filled ovals, and strings of characters. Also play with different colors.

(c) Modify `DrawingApp` to use the `WorldRectangle` class and repeat part (a). Note that the coordinates that are displayed and the inputs are now consistent.

(d) Define and test a `TextMessage` class to display text messages in a drawing panel using world coordinates to position the text. In the `draw` method use the syntax `g.drawString("string to draw",x,y)`, where `(x,y)` are the pixel coordinates. □

Although simple geometric shapes such as circles and rectangles are often all that are needed to visualize many physical models, Java provides a drawing environment based on the Java 2D Application Programming Interface (API) which can render arbitrary geometric shapes, images, and text using composition and matrix-based transformations. We will use a subset of these features to define the `DrawableShape` and `InteractiveShape` classes in the display package of Open Source Physics, which we will introduce in Chapter 9. (See also the Open Source Physics User's Guide.)

So far we have created rectangles using two different classes. Each implementation of a `Drawable` rectangle defined a different `draw` method. Notice that in the display frame's definition of `addDrawable` in `DrawingApp`, the argument is specified to be the interface `Drawable` rather than a specific class. Any class that implements `Drawable` can be an argument of `addDrawable`. Without the interface construct, we would need to write an `addDrawable` method for each type of class.

3.4 Specifying The State of a System Using Arrays

Imagine writing the code for the numerical solution of the motion of three particles in three dimensions using the Euler–Richardson algorithm. The resulting code would be tedious to write. In addition, for each problem we would need to write and debug new code to implement the numerical algorithm. The complications become worse for better algorithms, most of which are algebraically more complex. Moreover, the numerical solution of simple first-order differential equations is a well- developed part of numerical analysis, and thus there is little reason to worry about the details of these algorithms now that we know how they work. In Section 3.5 we will introduce an interface for solving the differential equations associated with Newton's equations of motion. Before we do so we discuss a few features of arrays that we will need.

As we discussed on page 38, ordered lists of data are most easily stored in arrays. For example, if we have an array variable named x, then we can access its first element as x[0], its second element as x[1], etc. All elements must be of the same data type, but they can be just about anything: primitive data types such as doubles or integers, objects, or even other arrays. The following statements show how arrays of primitive data types are defined and instantiated:

```
// x defined to be an array of doubles
double[] x;
double x[];                              // same meaning as double [] x
// x array created with 32 elements
x = new double[32];
// y array defined and created in one statement
double[] y = new double[32];
int[] num = new int[100];                // array of 100 integers
double [] x,y                            // preferred notation
// same meaning as double [] x,y
double x[], y[]
// array of doubles specified by two indices
double[][] sigma = new double[3][3];
// reference to first row of sigma array
double[] row = sigma[0];
```

We will adopt the syntax double[] x instead of double x[]. The array index starts at zero, and the largest index is one less than the number of elements. Note that Java supports multiple array indices by creating arrays of arrays. Although sigma[0][0] refers to a single value of type double in the sigma object, we can refer to an entire row of values in the sigma object using the syntax sigma[i].

As shown in Chapter 2, arrays can contain objects such as bouncing balls.

```
// array of two BouncingBall objects
BouncingBall [] ball = new BouncingBall[2];
ball[0] = new BouncingBall(0,10.0,0,5.0);      // creates first ball
ball[1] = new BouncingBall(0,-13.0,0,7.0);     // creates second ball
```

The first statement allocates an array of BouncingBall objects, each of which is initialized to null. We need to create each object in the array using the new operator.

The numerical solution of an ordinary differential equation (frequently called an ODE) begins by expressing the equation as several first-order differential equations. If the highest derivative in the ODE is order n (for example, $d^n x/dt^n$), then it can be shown that the ODE can be written equivalently as n first-order differential equations. For example, Newton's equation of motion is a second-order differential equation and can be written as two first-order differential equations for the position and velocity in each spatial dimension. For example, in one dimension we can write

$$\frac{dy}{dt} = v(t) \tag{3.5a}$$

$$\frac{dv}{dt} = a(t) = F(t)/m. \tag{3.5b}$$

If we have more than one particle, there are additional first-order differential equations for each particle. It is convenient to have a standard way of handling all these cases.

Let us assume that each differential equation is of the form

$$\frac{dx_i}{dt} = r_i(x_0, x_i, x_2, \ldots, x_{n-1}, t) \tag{3.6}$$

where x_i is a dynamical variable such as a position or a velocity. The rate function r_i can depend on any of the dynamical variables including the time t. We will store the values of the dynamical variables in the `state` array and the values of the corresponding rates in the `rate` array. In the following we show some examples:

```
// one particle in one dimension:
state[0]  // stores x
state[1]  // stores v
state[2]  // stores t (time)
// one particle in two dimensions:
state[0]  // stores x
state[1]  // stores vx
state[2]  // stores y
state[3]  // stores vy
state[4]  //stores t
// two particles in one dimension:
state[0]  // stores x1
state[1]  // stores v1
state[2]  // stores x2
state[3]  // stores v2
state[4]  // stores t
```

Although the Euler algorithm does not assume any special ordering of the state variables, we adopt the convention that a velocity rate follows every position rate in the state array so that we can efficiently code the more sophisticated numerical algorithms that we discuss in Appendix 3A and in later chapters. To solve problems for which the rate contains an explicit time dependence, such as a driven harmonic oscillator (see Section 4.4), we store the time variable in the last element of the state array. Thus, for one particle in one dimension, the time is stored in `state[2]`. In this way we can treat all dynamical variables on an equal footing.

Because arrays can be arguments of methods, we need to understand how Java passes variables from the class that calls a method to the method being called. Consider the following method:

```java
public void example(int r, int s[]) {
   r = 20;
   s[0] = 20;
}
```

What do you expect the output of the following statements to be?

```java
int x = 10;
int[] y = {10};    // array of one element initialized to y[0] = 10
example(x, y);
System.out.println("x = " + x + " y[0] = " + y[0]);
```

The answer is that the output will be `x = 10, y[0] = 20`. Java parameters are "passed-by-value," which means that the values are copied. The method cannot modify the value of the x variable because the method received only a copy of its value. In contrast, when an object or an array is in a method's parameter list, Java passes a copy of the reference to the object or the array. The method can use the reference to read or modify the data in the array or object. For this reason the step method of the ODE solvers, discussed in Section 3.6, does not need to explicitly return an updated state array, but implicity changes the contents of the state array.

Exercise 3.5. Pass by value

As another example of how Java handles primitive variables differently from arrays and objects, consider the statements

```
int x = 10;
int y = x;
x = 20;
```

What is y? Next consider

```
// declares an array of one element initialized to the value 10

int[] x = {10};
int[] y = x;
x[0] = 20;
```

What is y[0]? □

We are now ready to discuss the classes and interfaces from the Open Source Physics library for solving ordinary differential equations.

3.5 The ODE Interface

To introduce the ODE interface, we again consider the equations of motion for a falling particle. We use a state array ordered as $s = (y, v, t)$, so that the dynamical equations can be written as:

$$\dot{s_0} = s_1 \qquad (3.7a)$$
$$\dot{s_1} = -g \qquad (3.7b)$$
$$\dot{s_2} = 1. \qquad (3.7c)$$

The ODE interface enables us to encapsulate (3.7) in a class. The interface contains two methods, getState and getRate, as shown in Listing 3.4.

Listing 3.4: The ODE interface.

```
package org.opensourcephysics.numerics;

public interface ODE {
   public double[] getState();
   public void getRate(double[] state, double[] rate);
}
```

The getState method returns the state array $(s_0, s_1, \ldots, s_n)$. The getRate method evaluates the derivatives using the given state array and stores the result in the rate array, $(\dot{s}_0, \dot{s}_1, \ldots, \dot{s}_n)$.

An example of a Java class that implements the ODE interface for a falling particle is shown in Listing 3.5.

Listing 3.5: Example of the implementation of the ODE interface for a falling particle.

```
package org.opensourcephysics.sip.ch03;
import org.opensourcephysics.numerics.*;

public class FallingParticleODE implements ODE {
   final static double g = 9.8;
   double[] state = new double[3];

   public FallingParticleODE(double y, double v) {
      state[0] = y;
      state[1] = v;
      state[2] = 0;            // initial time
   }

   // required to implement ODE interface
   public double[] getState() {
      return state;
   }

   public void getRate(double[] state, double[] rate) {
      rate[0] = state[1];     // rate of change of y is v
      rate[1] = -g;
      rate[2] = 1;            // rate of change of time is 1
   }
}
```

3.6 The ODESolver Interface

There are many possible numerical algorithms for advancing a system of first-order ODEs from an initial state to a final state. The Open Source Physics library defines ODE solvers such as `Euler` and `EulerRichardson`, as well as `RK4`, a fourth-order algorithm that is discussed in Appendix 3. You can write additional classes for other algorithms if they are needed. Each of these classes implements the `ODESolver` interface, which is defined in Listing 3.6.

Listing 3.6: The ODE solver interface. Note the four methods that must be defined.

```
package org.opensourcephysics.numerics;

public interface ODESolver {
   public void initialize(double stepSize);
   public double step();
   public void setStepSize(double stepSize);
   public double getStepSize();
}
```

A system of first-order differential equations is now solved by creating an object that implements a particular algorithm and repeatedly invoking the `step` method for that solver class. The argument for the solver class constructor must be a class that implements the `ODE` interface. As an example of the use of `ODESolver`, we again consider the dynamics of a falling particle.

Listing 3.7: A falling particle program that uses an `ODESolver`.

```
package org.opensourcephysics.sip.ch03;
```

```
import org.opensourcephysics.controls.*;
import org.opensourcephysics.numerics.*;

public class FallingParticleODEApp extends AbstractCalculation {
   public void calculate() {
      // gets initial conditions
      double y0 = control.getDouble("Initial y");
      double v0 = control.getDouble("Initial v");
      // creates ball with initial conditions
      FallingParticleODE ball = new FallingParticleODE(y0, v0);
      // note how particular algorithm is chosen
      ODESolver solver = new Euler(ball);
      // sets time step dt in the solver
      solver.setStepSize(control.getDouble("dt"));
      while(ball.state[0]>0) {
         solver.step();
      }
      control.println("final time = "+ball.state[2]);
      control.println("y = "+ball.state[0]+" v = "+ball.state[1]);
   }

   public void reset() {
      // sets default input values
      control.setValue("Initial y", 10);
      control.setValue("Initial v", 0);
      control.setValue("dt", 0.01);
   }

   // creates a calculation control structure for this class
   public static void main(String[] args) {
      CalculationControl.createApp(new FallingParticleODEApp());
   }
}
```

The ODE classes are located in the numerics package, and thus we need to import this package as done in the third statement of `FallingParticleODEApp`. We declare and instantiate the variables `ball` and `solver` in the `calculate` method. Note that `ball`, an instance of `FallingParticleODE`, is the argument of the `Euler` constructor. The object `ball` can be an argument because `FallingParticleODE` implements the `ODE` interface.

It would be a good idea to look at the source code of the ODE `Euler` class in the numerics package. The `Euler` class gets the state of the system using `getState` and then sends this state to `getRate` which stores the rates in the `rate` array. The `state` array is then modified using the `rate` array in the Euler algorithm. You don't need to know the details, but you can read the `step` method of the various classes that implement `ODESolver` if you are interested in how the different algorithms are programmed.

Because `FallingParticleODE` appears to be more complicated than `FallingParticle`, you might ask what we have gained. One answer is that it is now much easier to use a different numerical algorithm. The only modification we need to make is to change the statement

```
ODESolver solver = new Euler(ball);
```

to, for example,

```
ODESolver solver = new EulerRichardson(ball);
```

We have separated the physics (in this case a freely falling particle) from the implementation of the numerical method.

Exercise 3.6. ODE solvers

Run `FallingParticleODEApp` and compare your results with our previous implementation of the Euler algorithm in `FallingParticleApp`. How easy is it to use a different algorithm? □

3.7 Effects of Drag Resistance

We have introduced most of the programming concepts that we will use in the remainder of this text. If you are new to programming, you will likely feel a bit confused at this point by all the new concepts and syntax. However, it is not necessary to understand all the details to continue and begin to write your own programs. A prototypical simulation program is given in Listings 3.8 and 3.9. These classes simulate a projectile on the surface of the Earth with no air friction, including a plot of position versus time and an animation of a projectile moving through the air. In the following, we discuss more realistic models that can be simulated by modifying the projectile classes.

Listing 3.8: A simple projectile simulation that is useful as a template for other simulations.

```
package org.opensourcephysics.sip.ch03;
import java.awt.*;
import org.opensourcephysics.display.*;
import org.opensourcephysics.numerics.*;

public class Projectile implements Drawable, ODE {
   static final double g = 9.8;
   double[] state = new double[5]; // {x,vx,y,vy,t}
   / pixel radius for drawing of projectile
   int pixRadius = 6;               /
   EulerRichardson odeSolver = new EulerRichardson(this);

   public void setStepSize(double dt) {
      odeSolver.setStepSize(dt);
   }

   public void step() {
      odeSolver.step(); // do one time step using selected algorithm
   }

   public void setState(double x, double vx, double y, double vy) {
      state[0] = x;
      state[1] = vx;
      state[2] = y;
      state[3] = vy;
      state[4] = 0;
   }

   public double[] getState() {
      return state;
   }
   public void getRate(double[] state, double[] rate) {
      rate[0] = state[1]; // rate of change of x
      rate[1] = 0;        // rate of change of vx
      rate[2] = state[3]; // rate of change of y
      rate[3] = -g;       // rate of change of vy
      rate[4] = 1;        // dt/dt = 1
   }

   public void draw(DrawingPanel drawingPanel, Graphics g) {
      int xpix = drawingPanel.xToPix(state[0]);
      int ypix = drawingPanel.yToPix(state[2]);
      g.setColor(Color.red);
      g.fillOval(xpix-pixRadius, ypix-pixRadius, 2*pixRadius, 2*pixRadius);
      g.setColor(Color.green);
      int xmin = drawingPanel.xToPix(-100);
      int xmax = drawingPanel.xToPix(100);
      int y0 = drawingPanel.yToPix(0);
      // draw a line to represent the ground
      g.drawLine(xmin, y0, xmax, y0);
   }
}
```

Listing 3.9: A target class for projectile motion simulation.

```
package org.opensourcephysics.sip.ch03;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.frames.*;

public class ProjectileApp extends AbstractSimulation {
   PlotFrame plotFrame = new PlotFrame("Time", "x,y", "Position versus time");
   Projectile projectile = new Projectile();
   PlotFrame animationFrame = new PlotFrame("x", "y", "Trajectory");

   public ProjectileApp() {
      animationFrame.addDrawable(projectile);
      plotFrame.setXYColumnNames(0, "t", "x");
      plotFrame.setXYColumnNames(1, "t", "y");
   }

   public void initialize() {
      double dt = control.getDouble("dt");
      double x = control.getDouble("initial x");
      double vx = control.getDouble("initial vx");
      double y = control.getDouble("initial y");
      double vy = control.getDouble("initial vy");
      projectile.setState(x, vx, y, vy);
      projectile.setStepSize(dt);
      // estimate of size needed for display
      double size = (vx*vx+vy*vy)/10;
      animationFrame.setPreferredMinMax(-1, size, -1, size);
   }

   public void doStep() {
        // x vs time data added
```

```
         plotFrame.append(0, projectile.state[4], projectile.state[0]);
         // y vs time data added
         plotFrame.append(1, projectile.state[4], projectile.state[2]);
          // trajectory data added
         animationFrame.append(0, projectile.state[0], projectile.state[2]);
         projectile.step();          // advance the state by one time step
      }

      public void reset() {
         control.setValue("initial x", 0);
         control.setValue("initial vx", 10);
         control.setValue("initial y", 0);
         control.setValue("initial vy", 10);
         control.setValue("dt", 0.01);
         enableStepsPerDisplay(true);
      }

      public static void main(String[] args) {
         SimulationControl.createApp(new ProjectileApp());
      }
   }
```

The analytic solution for free fall near the Earth's surface, (2.4), is well known, and thus finding a numerical solution is useful only as an introduction to numerical methods. It is not difficult to think of more realistic models of motion near the Earth's surface for which the equations of motion do not have simple analytic solutions. For example, if we take into account the variation of the Earth's gravitational field with the distance from the center of the Earth, then the force on a particle is not constant. According to Newton's law of gravitation, the force due to the Earth on a particle of mass m is given by

$$F = \frac{GMm}{(R+y)^2} = \frac{GMm}{R^2(1+y/R)^2} = mg\left(1 - 2\frac{y}{R} + \cdots\right) \tag{3.8}$$

where y is measured from the Earth's surface, R is the radius of the Earth, M is the mass of the Earth, G is the gravitational constant, and $g = GM/R^2$.

Problem 3.7. Position-dependent force

Extend `FallingParticleODE` to simulate the fall of a particle with the position-dependent force law (3.8). Assume that a particle is dropped from a height h with zero initial velocity and compute its impact velocity (speed) when it hits the ground at $y = 0$. Determine the value of h for which the impact velocity differs by one percent from its value with a constant acceleration $g = 9.8\,\text{m/s}^2$. Take $R = 6.37 \times 10^6\,\text{m}$. Make sure that the one percent difference is due to the physics of the force law and not the accuracy of your algorithm. □

For particles near the Earth's surface, a more important modification is to include the drag force due to air resistance. The direction of the drag force $F_d(v)$ is opposite to the velocity of the particle (see Figure 3.1). For a falling body, $F_d(v)$ is upward as shown in Figure 3.1(b). Hence, the total force F on the falling body can be expressed as

$$F = -mg + F_d. \tag{3.9}$$

The velocity dependence of $F_d(v)$ is known theoretically in the limit of very low speeds for small objects. In general, it is necessary to determine the velocity dependence of $F_d(v)$ empirically over a limited range of velocities. One way to obtain the form of $F_d(v)$ is to measure y as

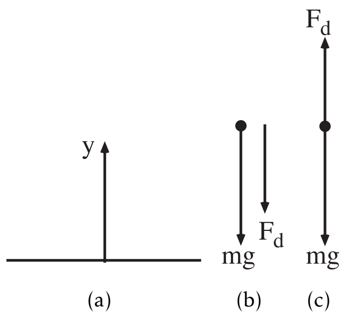


Figure 3.1: (a) Coordinate system with y measured positive upward from the ground. (b) The force diagram for upward motion. (c) The force diagram for downward motion.

a function of t and then compute $v(t)$ by calculating the numerical derivative of $y(t)$. Similarly, we can use $v(t)$ to compute $a(t)$ numerically. From this information, it is possible in principle to find the acceleration as a function of v and to extract $F_d(v)$ from (3.9). However, this procedure introduces errors (see Problem 3.8b) because the accuracy of the derivatives will be less than the accuracy of the measured position. An alternative is to reverse the procedure, that is, assume an explicit form for the v dependence of $F_d(v)$, and use it to solve for $y(t)$. If the calculated values of $y(t)$ are consistent with the experimental values of $y(t)$, then the assumed v dependence of $F_d(v)$ is justified empirically.

The two common assumed forms of the velocity dependence of $F_d(v)$ are

$$F_{1,d}(v) = C_1 v \tag{3.10a}$$

and

$$F_{2,d}(v) = C_2 v^2 \tag{3.10b}$$

where the parameters C_1 and C_2 depend on the properties of the medium and the shape of the object. In general, (3.10a) and (3.10b) are useful *phenomenological* expressions that yield approximate results for $F_d(v)$ over a limited range of v.

Because $F_d(v)$ increases as v increases, there is a limiting or *terminal velocity* (speed) at which the net force on a falling object is zero. This terminal speed can be found from (3.9) and (3.10) by setting $F_d = mg$ and is given by

$$v_{1,t} = \frac{mg}{C_1} \qquad \text{(linear drag)} \tag{3.11a}$$

$$v_{2,t} = \Big(\frac{mg}{C_2}\Big)^{1/2} \qquad \text{(quadratic drag)} \tag{3.11b}$$

for the linear and quadratic cases, respectively. It is often convenient to express velocities in terms of the terminal velocity. We can use (3.10) and (3.11) to write F_d in the linear and quadratic cases as

$$F_{1,d} = C_1 v_{1,t}\Big(\frac{v}{v_{1,t}}\Big) = mg\frac{v}{v_{1,t}} \tag{3.12a}$$

$$F_{2,d} = C_2 {v_{2,t}}^2\Big(\frac{v}{v_{2,t}}\Big)^2 = mg\Big(\frac{v}{v_{2,t}}\Big)^2. \tag{3.12b}$$

t (s)	Position (m)	t (s)	Position (m)	t (s)	Position (m)
0.2055	0.4188	0.4280	0.3609	0.6498	0.2497
0.2302	0.4164	0.4526	0.3505	0.6744	0.2337
0.2550	0.4128	0.4773	0.3400	0.6990	0.2175
0.2797	0.4082	0.5020	0.3297	0.7236	0.2008
0.3045	0.4026	0.5266	0.3181	0.7482	0.1846
0.3292	0.3958	0.5513	0.3051	0.7728	0.1696
0.3539	0.3878	0.5759	0.2913	0.7974	0.1566
0.3786	0.3802	0.6005	0.2788	0.8220	0.1393
0.4033	0.3708	0.6252	0.2667	0.8466	0.1263

Table 3.1: Results for the vertical fall of a coffee filter. Note that the initial time is not zero. The time difference is ≈ 0.0247. This data is also available in the `falling.txt` file in the ch03 package.

Hence, we can write the net force (per unit mass) on a falling object in the convenient forms

$$F_1(v)/m = -g\left(1 - \frac{v}{v_{1,t}}\right), \tag{3.13a}$$

$$F_2(v)/m = -g\left(1 - \frac{v^2}{{v_{2,t}}^2}\right). \tag{3.13b}$$

To determine if the effects of air resistance are important during the fall of ordinary objects, consider the fall of a pebble of mass $m = 10^{-2}$ kg. To a good approximation, the drag force is proportional to v^2. For a spherical pebble of radius 0.01 m, C_2 is found empirically to be approximately 10^{-2} kg/m. From (3.11b) we find the terminal velocity to be about 30 m/s. Because this speed would be achieved by a freely falling body in a vertical fall of approximately 50 m in a time of about 3 s, we expect that the effects of air resistance would be appreciable for comparable times and distances.

Data often is stored in text files, and it is convenient to be able to read this data into a program for analysis. The `ResourceLoader` class in the Open Source Physics `tools` package makes reading these files easy. This class can read many different data types including images and sound. An example of how to use the `ResourceLoader` class to read string data is given in `DataLoaderApp`.

Listing 3.10: Example of the use of the `ResourceLoader` class to read data into a program.

```
package org.opensourcephysics.sip.ch03;
import org.opensourcephysics.tools.*;

public class DataLoaderApp {
   public static void main(String[] args) {
      // reads from directory where DataLoaderApp is located
      String fileName = "falling.txt";
      // gets the data file
      Resource res = ResourceLoader.getResource(fileName,
                           DataLoaderApp.class);
      String data = res.getString();
      // split string on newline character
      String[] lines = data.split("\n");
      // extract x-y data from every line
```

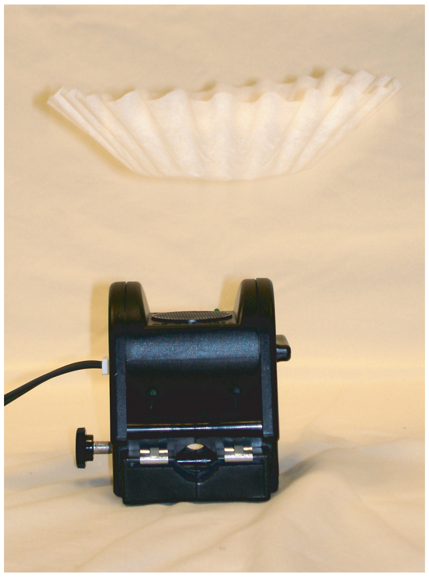


Figure 3.2: A falling coffee filter does not fall with constant acceleration due to the effects of air resistance. The motion sensor below the filter is connected to a computer which records position data and stores it in a text file.

```
         for(int i = 0, n = lines.length;i<n;i++) {
            if(lines[i].trim().startsWith("//")) {
               continue;
            }
            // split on any white space
            String[] numbers = lines[i].trim().split("\\s");
            System.out.print("t = "+numbers[0]);
            System.out.println("  y = "+numbers[1]);
         }
      }
   }
```

Problem 3.8. The fall of a coffee filter

(a) Use the empirical data for the height $y(t)$ of a coffee filter in the `falling.txt` data file to determine the velocity $v(t)$ using the central difference approximation given by

$$v(t) \approx \frac{y(t+\Delta t) - y(t-\Delta t)}{2\Delta t} \qquad \text{(central difference approximation).} \tag{3.14}$$

Show that if we write the acceleration as $a(t) \approx [v(t+\Delta t) - v(t)]/\Delta t$ and use the backward difference approximation for the velocity,

$$v(t) \approx \frac{y(t) - y(t-\Delta t)}{\Delta t} \qquad \text{(backward difference approximation),} \tag{3.15}$$

we can express the acceleration as

$$a(t) \approx \frac{y(t+\Delta t) - 2y(t) + y(t-\Delta t)}{(\Delta t)^2}. \tag{3.16}$$

Use (3.16) to determine the acceleration.

(b) Determine the terminal velocity from the data given in the `falling.txt` file. This determination is difficult, in part because the terminal velocity has not been reached during the time that the fall of the coffee filter was observed. Use your approximate results for $v(t)$ and $a(t)$ to plot a as a function of v and, if possible, determine the nature of the velocity dependence of a. Discuss the accuracy of your results for the acceleration.

(c) Choose one of the numerical algorithms that we have discussed and write a class that encapsulates this algorithm for the motion of a particle with quadratic drag resistance.

(d) Choose the terminal velocity as an input parameter and take as your first guess for the terminal velocity the value you found in part (b). Make sure that your computed results for the height of the particle do not depend on Δt to the necessary accuracy. Compare your plot of the computed values of $y(t)$ for different choices of the terminal velocity with the empirical values of $y(t)$ in `falling.txt`.

(e) Repeat parts (c) and (d) assuming linear drag resistance. What are the qualitative differences between the two computed forms of $y(t)$ for the same terminal velocity?

(f) Visually determine which form of the drag force yields the best overall fit to the data. If the fit is not perfect, what is your criteria for which fit is better? Is it better to match your results to the experimental data at early times or at later times? Or did you adopt another criterion? What can you conclude about the velocity dependence of the drag resistance on a coffee filter? □

Problem 3.9. Effect of air resistance on the ascent and descent of a pebble

(a) Verify the claim made in Section 3.7 that the effects of air resistance on a falling pebble can be appreciable. Compute the speed at which a pebble reaches the ground if it is dropped from rest at a height of 50 m. Compare this speed to that of a freely falling object under the same conditions. Assume that the drag force is proportional to v^2 and that the terminal velocity is 30 m/s.

(b) Suppose a pebble is thrown vertically upward with an initial velocity v_0. In the absence of air resistance, we know that the maximum height reached by the pebble is $v_0^2/2g$, its velocity upon return to the Earth equals v_0, the time of ascent equals the time of descent, and the total time in the air is $2v_0/g$. Before doing a simulation, give a simple qualitative explanation of how you think these quantities will be affected by air resistance. In particular, how will the time of ascent compare with the time of descent?

(c) Do a simulation to determine if your qualitative answers in part (b) are correct. Assume that the drag force is proportional to v^2. Choose the coordinate system shown in Figure 3.1 with y positive upward. What is the net force for $v > 0$ and $v < 0$? We can characterize the magnitude of the drag force by a terminal velocity even if the motion of the pebble is upward and even if the pebble never attains this velocity. Choose the terminal velocity $v_t = 30$ m/s, corresponding to a drag coefficient of $C_2 \approx 0.01089$. It is a good idea to choose an initial velocity that allows the pebble to remain in the air for a time sufficiently long so that the effect of the drag force is appreciable. A reasonable choice is $v(t = 0) = 50$ m/s. You might find it convenient to express the drag force in the form $F_d \propto$ `-v*Math.abs(v)`. One way to determine the maximum height of the pebble is to use the statement

```
if (v*vold < 0) {
   control.println("maximum height = " + y);
}
```

where `v` $= v_{n+1}$ and `vold` $= v_n$. Why is this criterion preferable to other criteria that you might imagine using? □

3.8 Two-Dimensional Trajectories

You are probably familiar with two-dimensional trajectory problems in the absence of air resistance. For example, if a ball is thrown in the air with an initial velocity v_0 at an angle θ_0 with respect to the ground, how far will the ball travel in the horizontal direction, and what is its maximum height and time of flight? Suppose that a ball is released at a nonzero height h above the ground. What is the launch angle for the maximum range? Are your answers still applicable if air resistance is taken into account? We consider these and similar questions in the following.

Consider an object of mass m whose initial velocity $\mathbf{v}_0$ is directed at an angle θ_0 above the horizontal [see Figure 3.3(a)]. The particle is subjected to gravitational and drag forces of magnitude mg and F_d; the direction of the drag force is opposite to $\mathbf{v}$ (see Figure 3.33.3(b)). Newton's equations of motion for the x and y components of the motion can be written as

$$m\frac{dv_x}{dt} = -F_d \cos\theta \tag{3.17a}$$
$$m\frac{dv_y}{dt} = -mg - F_d \sin\theta. \tag{3.17b}$$

For example, let us maximize the range of a round steel ball of radius 4 cm. A reasonable assumption for a steel ball of this size and typical speed is that $F_d = C_2 v^2$. Because $v_x = v\cos\theta$ and $v_y = v\sin\theta$, we can rewrite (3.17) as

$$m\frac{dv_x}{dt} = -C_2 v v_x \tag{3.18a}$$
$$m\frac{dv_y}{dt} = -mg - C_2 v v_y. \tag{3.18b}$$

Note that $-C_2 v v_x$ and $-C_2 v v_y$ are the x and y components of the drag force $-C_2 v^2$. Because (3.18a) and (3.18b) for the change in v_x and v_y involve the square of the velocity, $v^2 = {v_x}^2 + {v_y}^2$, we cannot calculate the vertical motion of a falling body without reference to the horizontal component, that is, the motion in the x and y direction is *coupled*.

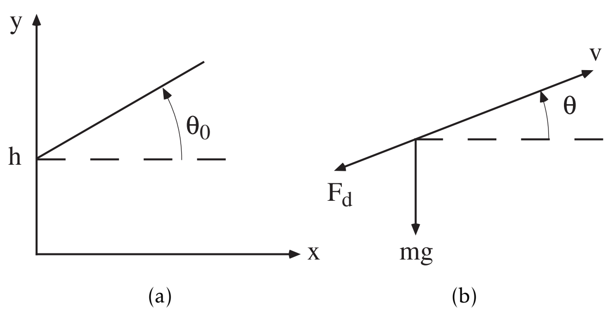


Figure 3.3: (a) A ball is thrown from a height h at a launch angle θ_0 measured with respect to the horizontal. The initial velocity is $\mathbf{v}_0$. (b) The gravitational and drag forces on a particle.

Problem 3.10. Trajectory of a steel ball

(a) Use `Projectile` and `ProjectileApp` to compute the two-dimensional trajectory of a ball moving in air without air friction and plot y as a function of x. Compare your computed results with the exact results. For example, assume that a ball is thrown from ground level at an angle θ_0 above the horizontal with an initial velocity of $v_0 = 15\,\text{m/s}$. Vary θ_0 and show that the maximum range occurs at $\theta_0 = \theta_{\text{max}} = 45°$. What is R_{max}, the maximum range, at this angle? Compare your numerical result to the analytic result $R_{\text{max}} = v_0^2/g$.

(b) Suppose that a steel ball is thrown from a height h at an angle θ_0 above the horizontal with the same initial speed as in part (a). If you neglect air resistance, do you expect θ_{max} to be larger or smaller than 45°? What is θ_{max} for $h = 2\,\text{m}$? By what percent is the range R changed if θ is varied by 2% from θ_{max}?

(c) Consider the effects of air resistance on the range and optimum angle of a steel ball. For a ball of mass 7 kg and cross-sectional area 0.01 m^2, the parameter $C_2 \approx 0.1$. What are the units of C_2? It is convenient to exaggerate the effects of air resistance so that you can more easily determine the qualitative nature of the effects. Hence, compute the optimum angle for $h = 2\,\text{m}$, $v_0 = 30\,\text{m/s}$, and $C_2/m = 0.1$ and compare your answer to the value found in part (b). Is R more or less sensitive to changes in θ_0 from θ_{max} than in part (b)? Determine the optimum launch angle and the corresponding range for the more realistic value of $C_2 = 0.1$. A detailed discussion of the maximum range of the ball has been given by Lichtenberg and Wills. □

Problem 3.11. Comparing the motion of two objects

Consider the motion of two identical objects that both start from a height h. One object is dropped vertically from rest and the other is thrown with a horizontal velocity v_0. Which object reaches the ground first?

(a) Give reasons for your answer assuming that air resistance can be neglected.

(b) Assume that air resistance cannot be neglected and that the drag force is proportional to v^2. Give reasons for your anticipated answer for this case. Then perform numerical simulations using, for example, $C_2/m = 0.1$, $h = 10\,\text{m}$, and $v_0 = 30\,\text{m/s}$. Are your qualitative results consistent with your anticipated answer? If they are not, the source of the discrepancy might be an error in your program. Or the discrepancy might be due to your failure to anticipate the effects of the coupling between the vertical and horizontal motion.

(c) Suppose that the drag force is proportional to v rather than to v^2. Is your anticipated answer similar to that in part (b)? Do a numerical simulation to test your intuition. □

3.9 Decay Processes

The power of mathematics when applied to physics comes in part from the fact that seemingly unrelated problems frequently have the same mathematical formulation. Hence, if we can solve one problem, we can solve other problems that might appear to be unrelated. For example, the growth of bacteria, the cooling of a cup of hot water, the charging of a capacitor in a RC circuit, and nuclear decay can all be formulated in terms of equivalent differential equations.

Consider a large number of radioactive nuclei. Although the number of nuclei is discrete, we may often treat this number as a continuous variable because the number of nuclei is very large. In this case the law of radioactive decay is that the rate of decay is proportional to the number of nuclei. Thus we can write

$$\frac{dN}{dt} = -\lambda N \tag{3.19}$$

where N is the number of nuclei and λ is the decay constant. Of course, we do not need to use a computer to solve this decay equation, and the analytic solution is

$$N(t) = N_0 e^{-\lambda t} \tag{3.20}$$

where N_0 is the initial number of particles. The quantity λ in (3.19) or (3.20) has dimensions of inverse time.

Problem 3.12. Single nuclear species decay

(a) Write a class that solves and plots the nuclear decay problem. Input the decay constant λ from the control window. For $\lambda = 1$ and $\Delta t = 0.01$, compute the difference between the analytic result and the result of the Euler algorithm for $N(t)/N(0)$ at $t = 1$ and $t = 2$. Assume that time is measured in seconds.

(b) A common time unit for radioactive decay is the half-life $T_{1/2}$, the time it takes for one-half of the original nuclei to decay. Another natural time scale is the time τ it takes for $1/e$ of the original nuclei to decay. Use your modified program to verify that $T_{1/2} = \ln 2/\lambda$. How long does it take for $1/e$ of the original nuclei to decay? How is $T_{1/2}$ related to τ?

(c) Because it is awkward to treat very large or very small numbers on a computer, it is convenient to choose units so that the computed values of the variables are not too far from unity. Determine the decay constant λ in units of s^{-1} for $^{238}\mathrm{U} \to {}^{234}\mathrm{Th}$ if the half-life is 4.5×10^9 years. What units and time step would be appropriate for the numerical solution of (3.19)? How would these values change if the particle being modeled was a muon with a half-life of 2.2×10^{-6} s?

(d) Modify your program so that the time t is expressed in terms of the half-life. That is, at $t = 1$, one half of the particles would have decayed and at $t = 2$, one quarter of the particles would have decayed. Use your program to determine the time for 1000 atoms of $^{238}\mathrm{U}$ to decay to 20% of their original number. What would be the corresponding time for muons? □

Multiple nuclear decays produce systems of first-order differential equations. Problem 3.13 asks you to model such a system using the techniques similar to those that we have already used.

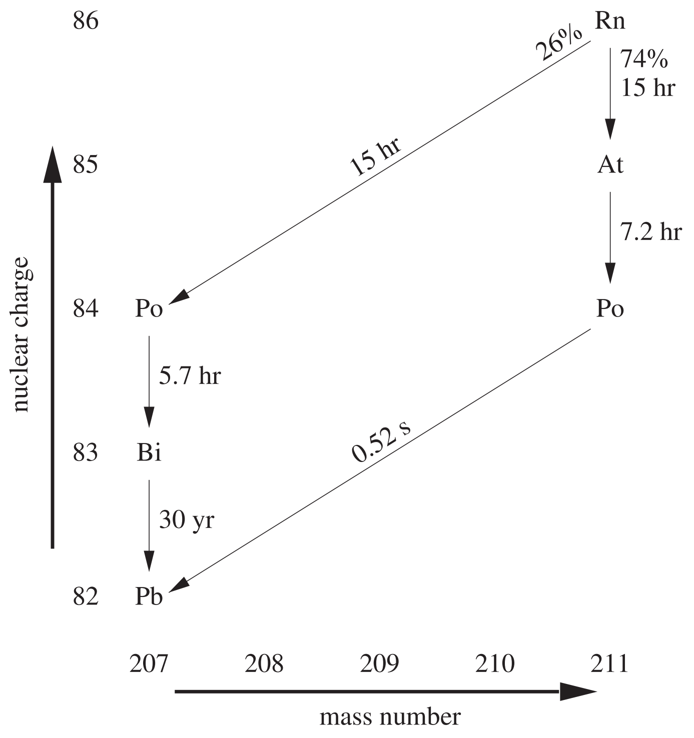


Figure 3.4: The decay scheme of ^{211}Rn. Note that ^{211}Rn decays via two branches, and the final product is the stable isotope ^{207}Pb. All vertical transitions are by electron capture, and all diagonal transitions are by alpha decay. The times represent half-lives.

Problem 3.13. Multiple nuclear decays

(a) ^{76}Kr decays to ^{76}Br via electron capture with a half-life of 14.8 h, and ^{76}Br decays to ^{76}Se via electron capture and positron emission with a half-life of 16.1 h. In this case there are two half-lives, and it is convenient to measure time in units of the smallest half-life. Write a program to compute the time dependence of the amount of ^{76}Kr and ^{76}Se over an interval of one week. Assume that the sample initially contains 1 gm of pure ^{76}Kr.

(b) ^{28}Mn decays via beta emission to ^{28}Al with a half-life of 21 h, and ^{28}Al decays by positron emission to ^{28}Si with a half-life of 2.31 min. If we were to use minutes as the unit of time, our program would have to do many iterations before we would see a significant decay of the ^{28}Mn. What simplifying assumption can you make to speed up the computation?

(c) ^{211}Rn decays via two branches as shown in Figure 3.4. Make any necessary approximations and compute the amount of each isotope as a function of time, assuming that the sample initially consists of 1 μg of ^{211}Rn. □

Problem 3.14. Cooling of a cup of coffee

The nature of the energy transfer from the hot water in a cup of coffee to the surrounding air is complicated and, in general, involves the mechanisms of convection, radiation, evaporation, and conduction. However, if the temperature difference between the water and its surroundings is not too large, the rate of change of the temperature of the water may be assumed to be proportional to the temperature difference. We can formulate this statement more precisely in terms of a differential equation:

$$\frac{dT}{dt} = -r(T - T_s) \tag{3.21}$$

where T is the temperature of the water, T_s is the temperature of its surroundings, and r is the cooling constant. The minus sign in (3.21) implies that if $T > T_s$, the temperature of the water will decrease with time. The value of the cooling constant r depends on the heat transfer mechanism, the contact area with the surroundings, and the thermal properties of the water. The relation (3.21) is sometimes known as Newton's law of cooling, even though the relation is only approximate, and Newton did not express the rate of cooling in this form.

(a) Write a program that computes the numerical solution of (3.21). Test your program by choosing the initial temperature $T_0 = 100°\text{C}$, $T_s = 0°\text{C}$, $r = 1$, and $\Delta t = 0.1$.

(b) Model the cooling of a cup of coffee by choosing $r = 0.03$. What are the units of r? Plot the temperature T as a function of the time using $T_0 = 87\,°\text{C}$ and $T_s = 17\,°\text{C}$. Make sure that your value of Δt is sufficiently small so that it does not affect your results. What is the appropriate unit of time in this case?

(c) Suppose that the initial temperature of a cup of coffee is 87°C, but the coffee can be sipped comfortably only when its temperature is $\leq 75°\text{C}$. Assume that the addition of cream cools the coffee by 5°C. If you are in a hurry and want to wait the shortest possible time, should the cream be added first and the coffee be allowed to cool, or should you wait until the coffee has cooled to 80°C before adding the cream? Use your program to "simulate" these two cases. Choose $r = 0.03$ and $T_s = 17°\text{C}$. What is the appropriate unit of time in this case? Assume that the value of r does not change when the cream is added. □

3.10 *Visualizing Three-Dimensional Motion

The world in which we live is three-dimensional (3D), and it sometimes is necessary to visualize phenomena in three dimensions. There are several 3D visualization packages available, including *Java3D* developed by Oracle. Because we want a three-dimensional visualization framework designed for physics simulations, we have developed our own API.[1]

The Open Source Physics 3D drawing framework is defined in subpackages in the `display3d` package and provides a high level of abstraction for rendering three-dimensional objects. These 3D drawable objects implement the `Element` interface in the `core` package, which enables their position, size, and appearance to be controlled. Elements can be grouped with other elements, can change their visibility, and respond to mouse actions. Listing 3.11 shows that it is not much more difficult to define and manipulate a three-dimensional model than a two-dimensional model. The most significant change is that the program instantiates a `Display3DFrame` and adds `Element` objects such as spheres and boxes to this frame.

[1]A framework consists of several classes and an API that does a particular task. In general, these classes are in different packages.

Listing 3.11: A three-dimensional bouncing ball created using the Open Source Physics `display3D.simple3d` package.

```
package org.opensourcephysics.sip.ch03;
import java.awt.*;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.frames.Display3DFrame;
import org.opensourcephysics.display3d.simple3d.*;
import org.opensourcephysics.display3d.core.Resolution;

public class Ball3DApp extends AbstractSimulation {
   Display3DFrame frame = new Display3DFrame("3D Ball");
   Element ball = new ElementEllipsoid();
   double time = 0, dt = 0.1;
   double vz = 0;

   public Ball3DApp() {
      frame.setPreferredMinMax(-5.0, 5.0, -5.0, 5.0, 0.0, 10.0);
      ball.setXYZ(0, 0, 9);
      // ball displayed in 3D as a planar ellipse of size (dx,dy,dz)
      ball.setSizeXYZ(1, 1, 1);
      frame.addElement(ball);
      Element box = new ElementBox();
      box.setXYZ(0, 0, 0);
      box.setSizeXYZ(4, 4, 1);
      box.getStyle().setFillColor(Color.RED);
      // divide sides of box into smaller rectangles
      box.getStyle().setResolution(new Resolution(5, 5, 2));
      frame.addElement(box);
      frame.setMessage("time = "+ControlUtils.f2(time));
   }

   protected void doStep() {
      time += 0.1;
      double z = ball.getZ()+vz*dt-4.9*dt*dt;
      vz -= 9.8*dt;
      if((vz<0)&&(z<1)) {
         vz = -vz;
      }
      ball.setZ(z);
      frame.setMessage("time = "+ControlUtils.f2(time));
   }

   public static void main(String[] args) {
      SimulationControl.createApp(new Ball3DApp());
   }
}
```

Note that the 3D drawing API is similar to the 2D drawing API described in Section 3.3. The `setPreferredMinMax` method, for example, has a variant that accepts up to six double parameters. You can set the size and location of objects in three dimensions before or after they are added to the frame.

Although the `Display3DFrame` is designed for three-dimensional visualizations, it can also show two-dimensional projections. For example, we can project onto the yz-plane by invoking

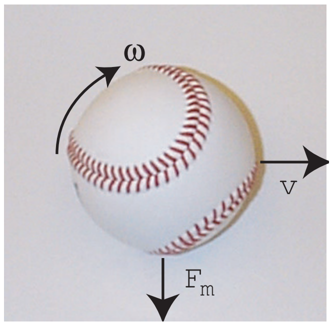


Figure 3.5: The Magnus force on a spinning ball pushes a ball with topspin down.

```
frame.setDisplayMode(VisualizationHints.DISPLAY_PLANAR_YZ);
```

Projections onto various planes are available at runtime using the frame's menu. The full capabilities of Open Source Physics 3D are discussed in the Open Source Physics User's Guide.

We will require only a small subset of the methods of the Open Source Physics 3D framework to create the three-dimensional visualizations in this book and will introduce the necessary objects as needed. Readers may wish to run the demonstration programs in the ch03 directory to obtain an overview of its drawing capabilities.

Of particular interest to baseball fans is the curve of balls in flight due to their rotation. This force was first investigated in 1850 by G. Magnus, and the curvature of the trajectories of spinning objects is now known as the *Magnus effect*. It can be explained qualitatively by observing that the speed of the ball's surface relative to the air is different on opposite edges of the ball. If the drag force has the form $F_{\text{drag}} \sim v^2$, then the unbalanced force due to the difference in the velocity on opposite sides of the ball due to its rotation is given by

$$F_{\text{magnus}} \sim v\Delta v. \tag{3.22}$$

We can express the velocity difference in terms of the ball's angular velocity and radius and write

$$F_{\text{magnus}} \sim vr\omega. \tag{3.23}$$

The direction of the Magnus force is perpendicular to both the velocity and the rotation axis. For example, if we observe a ball moving to the right and rotating clockwise (that is, with topspin), then the velocity of the ball's surface relative to the air at the top, $v+\omega r$, is higher than the velocity at the bottom, $v-\omega r$. Because the larger velocity will produce a larger force, the Magnus effect will contribute a force in the downward direction. These considerations suggest that the Magnus force can be expressed as a vector product:

$$F_{\text{magnus}}/m = C_M(\boldsymbol{\omega} \times \mathbf{v}) \tag{3.24}$$

where m is the mass of the ball. The constant, C_M, depends on the radius of the ball, the viscosity of air, and other factors such as the orientation of the stitching. We will assume that the ball is rotating fast enough so that it can be modeled using an average value. (If the ball does not rotate, the pitcher has thrown a knuckleball.) The total force on the baseball is given by

$$\mathbf{F}/m = \mathbf{g} - C_D|\mathbf{v}|\mathbf{v} + C_M(\boldsymbol{\omega} \times \mathbf{v}). \tag{3.25}$$

Equation (3.25) leads to the following rates for the velocity components:

$$\frac{dv_x}{dt} = -C_D v v_x + C_M(\omega_y v_z - \omega_z v_y) \tag{3.26a}$$

$$\frac{dv_y}{dt} = -C_D v v_y + C_M(\omega_z v_x - \omega_x v_z) \tag{3.26b}$$

$$\frac{dv_z}{dt} = -C_D v v_z + C_M(\omega_x v_y - \omega_y v_x) - g \tag{3.26c}$$

where we will assume that $\boldsymbol{\omega}$ is a constant. The rate for each of the three position variables is the corresponding velocity. Typical parameter values for a 149 gram baseball are $C_D = 6 \times 10^{-3}$ and $C_M = 4 \times 10^{-4}$. See the book by Adair for a more complete discussion.

Problem 3.15. Curveballs

(a) Create a class that implements (3.26). Assume that the initial ball is released at $z = 1.8\,\text{m}$ above and $x = 18\,\text{m}$ from home plate. Set the initial angle above the horizontal and the initial speed using the constructor.

(b) Write a program that plots the vertical and horizontal deflection of the baseball as it travels toward home plate. First set the drag and Magnus forces to zero and test your program using analytic results for a 40 m/s fast ball. What initial angle is required for the pitch to pass over home plate at a height of 1.5 m?

(c) Add the drag force with $C_D = 6 \times 10^{-3}$. What initial angle is required for this pitch to be a strike assuming that the other initial conditions are unchanged? Plot the vertical deflection with and without drag for comparison.

(d) Add topspin to the pitch using a typical spin of $\omega_y = 200\,\text{rad/s}$ and $C_M = 4 \times 10^{-4}$. How much does topspin change the height of the ball as it passes over the plate? What about backspin?

(e) How much does a 35 m/s curve ball deflect if it is pitched with an initial spin of 200 rad/s?

□

Problem 3.16. Visualizing baseball trajectories in three dimensions

Add a 3D visualization of the baseball's trajectory to Problem 3.15 using `ElementTrail` to display the path of the ball. The following code fragment shows how a trail is created and used.

```
ElementTrail trail = new ElementTrail();
trail.setMaximumPoints(500);
trail.getStyle().setLineColor(java.awt.Color.RED);
// frame3D is an OSP3DFrame
frame3D.addElement(trail);
// points are added to a trail to show a trajectory
trail.addPoint(x,y,z);  // adds a point to the trace
```

□

Coupled three-dimensional equations of motion occur in electrodynamics when a charged particle travels through electric and magnetic fields. The equation of motion can be written in vector form as

$$m\dot{\mathbf{v}} = q\mathbf{E} + q(\mathbf{v} \times \mathbf{B}) \tag{3.27}$$

where m is the mass of the particle, q is the charge, and $\mathbf{E}$ and $\mathbf{B}$ represent the electric and magnetic fields, respectively. For the special case of a constant magnetic field, the trajectory of a charged particle is a spiral along the field lines with a cyclotron orbit whose period of revolution is $2\pi m/qB$. The addition of an electric field changes this motion dramatically.

The rates for the velocity components of a charged particle using units such that $m = q = 1$ are

$$\frac{dv_x}{dt} = E_x + v_y B_z - v_z B_y \tag{3.28a}$$
$$\frac{dv_y}{dt} = E_y + v_z B_x - v_x B_z \tag{3.28b}$$
$$\frac{dv_z}{dt} = E_z + v_x B_y - v_y B_x. \tag{3.28c}$$

The rate for each of the three position variables is again the corresponding velocity.

Problem 3.17. Motion in electric and magnetic fields

(a) Write a program to simulate the two-dimensional motion of a charged particle in a constant electric and magnetic field with the magnetic field in the $\hat{z}$ direction and the electric field in the $\hat{y}$ direction. Assume that the initial velocity is in the xy plane.

(b) Why does the trajectory in part (a) remain in the x-y plane?

(c) In what direction does the charged particle drift if there is an electric field in the x direction and a magnetic field in the z direction if it starts at rest from the origin? What type of curve does the charged particle follow?

(d) Create a three-dimensional simulation of the trajectory of a particle in constant electric and magnetic fields. Verify that a charged particle undergoes spiral motion in a constant magnetic field and zero electric field. Predict the trajectory if an electric field is added and compare the results of the simulation to your prediction. Consider electric fields that are parallel to and perpendicular to the magnetic field. □

Although the trajectory of a charged particle in constant electric and magnetic fields can be solved analytically, the trajectories in the presence of dipole fields cannot. A magnetic dipole with dipole moment $\mathbf{p} = |p|\hat{p}$ produces the following magnetic field:

$$\mathbf{B} = \frac{\mu_0 m}{4\pi\epsilon_0 r^3}[3\hat{p}\cdot\hat{r})\hat{r} - \hat{p}]. \tag{3.29}$$

(The distinction between the symbol p for the dipole moment and p for momentum should be clear from the context.)

***Problem 3.18. Motion in a magnetic dipole field**

Model the Earth's Van Allen radiation belt using the following formula for the dipole field:

$$\mathbf{B} = B_0\Big(\frac{R_E}{R}\Big)^3\Big[\big(3\hat{p}\cdot\hat{r}\big)\hat{r} - \hat{p}\Big] \tag{3.30}$$

where R_E is the radius of the Earth, and the magnetic field at the equator is $B_0 = 3.5\times10^{-5}$ tesla. Note that a 1 MeV electron at 2 Earth radii travels in very tight spirals with a cyclotron period that is much smaller than the travel time between the north and south poles. Better visual results can be obtained by raising the electron energies by a factor of ~ 1000. Use classical dynamics, but include the relativistic dependence of the mass on the particle speed.

3.11 Levels of Simulation

So far we have considered models in which the microscopic complexity of the system of interest has been simplified considerably. Consider, for example, the motion of a pebble falling through the air. First we reduced the complexity by representing the pebble as a particle with no internal structure. Then we reduced the number of degrees of freedom even more by representing the collisions of the pebble with the many molecules in the air by a velocity-dependent friction term. The resultant phenomenological model is a fairly accurate representation of realistic physical systems. However, what we gain in simplicity, we lose in range of applicability.

In a more detailed model, the individual physical processes would be represented microscopically. For example, we could imagine doing a simulation in which the effects of the air are represented by a fluid of particles that collide with one another and with the falling body. How accurately do we need to represent the potential energy of interaction between the fluid particles? Clearly the level of detail that is needed depends on the accuracy of the corresponding experimental data and the type of information in which we are interested. For example, we do not need to take into account the influence of the moon on a pebble falling near the Earth's surface. And the level of detail that we can simulate depends in part on the available computer resources.

The terms *simulation* and *modeling* are frequently used interchangeably, and their precise meanings are not important. Many practitioners might say that so far we have solved several mathematical models numerically and have not yet done a simulation. Beginning with the next chapter, we will be able to say that we are doing simulations. The difference is that our models will represent physical systems in more detail, and we will give more attention to what physical quantities we should measure. In other words our simulations will become more analogous to laboratory experiments.

Appendix 3A: Numerical Integration of Newton's Equation of Motion

We summarize several of the common finite difference methods for the solution of Newton's equations of motion with continuous force functions. The number and variety of algorithms currently in use is evidence that no single method is superior under all conditions.

To simplify the notation, we consider the motion of a particle in one dimension and write Newton's equations of motion in the form

$$\frac{dv}{dt} = a(t) \tag{3.31a}$$

$$\frac{dx}{dt} = v(t) \tag{3.31b}$$

where $a(t) \equiv a(x(t), v(t), t)$. The goal of finite difference methods is to determine the values of x_{n+1} and v_{n+1} at time $t_{n+1} = t_n + \Delta t$. We already have seen that Δt must be chosen so that the integration method generates a stable solution. If the system is conservative, Δt must be sufficiently small so that the total energy is conserved to the desired accuracy.

The nature of many of the integration algorithms can be understood by expanding $v_{n+1} =$

$v(t_n + \Delta t)$ and $x_{n+1} = x(t_n + \Delta t)$ in a Taylor series. We write

$$v_{n+1} = v_n + a_n \Delta t + O\big((\Delta t)^2\big), \tag{3.32a}$$

$$x_{n+1} = x_n + v_n \Delta t + \frac{1}{2} a_n (\Delta t)^2 + O\big((\Delta t)^3\big). \tag{3.32b}$$

The familiar Euler algorithm is equivalent to retaining the $O(\Delta t)$ terms in (3.32):

$$v_{n+1} = v_n + a_n \Delta t \tag{3.33a}$$

$$x_{n+1} = x_n + v_n \Delta t \qquad \text{(Euler algorithm).} \tag{3.33b}$$

Because order Δt terms are retained in (3.33), the local truncation error, the error in one time step, is order $(\Delta t)^2$. The global error, the total error over the time of interest, due to the accumulation of errors from step to step is order Δt. This estimate of the global error follows from the fact that the number of steps into which the total time is divided is proportional to $1/\Delta t$. Hence, the order of the global error is reduced by a factor of $1/\Delta t$ relative to the local error. We say that an algorithm is nth order if its global error is order $(\Delta t)^n$. The Euler algorithm is an example of a *first-order* algorithm.

The Euler algorithm is asymmetrical because it advances the solution by a time step Δt, but uses information about the derivative only at the beginning of the interval. We have already found that the accuracy of the Euler algorithm is limited and that frequently its solutions are not stable. We have found also that a simple modification of (3.33) yields solutions that are stable for oscillatory systems. For completeness, we repeat the Euler–Cromer algorithm here:

$$v_{n+1} = v_n + a_n \Delta t \tag{3.34a}$$

$$x_{n+1} = x_n + v_{n+1} \Delta t \qquad \text{(Euler–Cromer algorithm).} \tag{3.34b}$$

An obvious way to improve the Euler algorithm is to use the mean velocity during the interval to obtain the new position. The corresponding *midpoint* algorithm can be written as

$$v_{n+1} = v_n + a_n \Delta t \tag{3.35a}$$

and

$$x_{n+1} = x_n + \frac{1}{2}(v_{n+1} + v_n)\Delta t \qquad \text{(midpoint algorithm).} \tag{3.35b}$$

Note that if we substitute (3.35a) for v_{n+1} into (3.35b), we obtain

$$x_{n+1} = x_n + v_n \Delta t + \frac{1}{2} a_n \, \Delta t^2. \tag{3.36}$$

Hence, the midpoint algorithm yields second-order accuracy for the position and first-order accuracy for the velocity. Although the midpoint approximation yields exact results for constant acceleration, it does not usually yield much better results than the Euler algorithm. In fact, both algorithms are equally poor because the errors increase with each time step.

A higher order algorithm whose error is bounded is the *half-step* algorithm. In this algorithm, the average velocity during an interval is taken to be the velocity in the middle of the interval. The half-step algorithm can be written as

$$v_{n+\frac{1}{2}} = v_{n-\frac{1}{2}} + a_n \Delta t, \tag{3.37a}$$

$$x_{n+1} = x_n + v_{n+\frac{1}{2}} \Delta t. \qquad \text{(half-step algorithm)} \tag{3.37b}$$

Note that the half-step algorithm is not self-starting, that is, (3.37a) does not allow us to calculate $v_{\frac{1}{2}}$. This problem can be overcome by adopting the Euler algorithm for the first half step:

$$v_{\frac{1}{2}} = v_0 + \frac{1}{2}a_0\,\Delta t. \tag{3.37c}$$

Because the half-step algorithm is stable, it is a common textbook algorithm. The Euler–Richardson algorithm can be motivated as follows. We first write $x(t+\Delta t)$ as

$$x_1 \approx x(t+\Delta t) = x(t) + v(t)\Delta t + \frac{1}{2}a(t)(\Delta t)^2. \tag{3.38}$$

The notation x_1 implies that $x(t+\Delta t)$ is related to $x(t)$ by one time step. We may also divide the step Δt into half steps and write the first half step, $x(t+\frac{1}{2}\Delta t)$, as

$$x\Big(t+\frac{1}{2}\Delta t\Big) \approx x(t) + v(t)\frac{\Delta t}{2} + \frac{1}{2}a(t)\Big(\frac{\Delta t}{2}\Big)^2. \tag{3.39}$$

The second half step, $x_2(t+\Delta t)$, may be written as

$$x_2(t+\Delta t) \approx x\Big(t+\frac{1}{2}\Delta t\Big) + v\Big(t+\frac{1}{2}\Delta t\Big)\frac{\Delta t}{2} + \frac{1}{2}a\Big(t+\frac{1}{2}\Delta t\Big)\Big(\frac{\Delta t}{2}\Big)^2. \tag{3.40}$$

We substitute (3.39) into (3.40) and obtain

$$x_2(t+\Delta t) \approx x(t) + \frac{1}{2}\Big[v\Big(t\Big) + v(t+\frac{1}{2}\Delta t\Big)\Big]\Delta t + \frac{1}{2}\Big[a(t) + a\Big(t+\frac{1}{2}\Delta t)\Big]\Big(\frac{1}{2}\Delta t\Big)^2. \tag{3.41}$$

Now $a(t+\frac{1}{2}\Delta t) \approx a(t) + \frac{1}{2}a'(t)\Delta t$. Hence to order $(\Delta t)^2$, (3.41) becomes

$$x_2(t+\Delta t) = x(t) + \frac{1}{2}\Big[v(t) + v\Big(t+\frac{1}{2}\Delta t\Big)\Big]\Delta t + \frac{1}{2}\Big[2a(t)\Big]\Big(\frac{1}{2}\Delta t\Big)^2. \tag{3.42}$$

We can find an approximation that is accurate to order $(\Delta t)^3$ by combining (3.38) and (3.42) so that the terms to order $(\Delta t)^2$ cancel. The combination that works is $2x_2 - x_1$, which gives the Euler–Richardson result:

$$x_{\rm er}(t+\Delta t) \equiv 2x_2(t+\Delta t) - x_1(t+\Delta t) = x(t) + v\Big(t+\frac{1}{2}\Delta t\Big)\Delta t + O(\Delta t)^3. \tag{3.43}$$

The same reasoning leads to an approximation for the velocity accurate to $(\Delta t)^3$ giving

$$v_{\rm er} \equiv 2v_2(t+\Delta t) - v_1(t+\Delta t) = v(t) + a\Big(t+\frac{1}{2}\Delta t\Big)\Delta t + O(\Delta t)^3. \tag{3.44}$$

A bonus of the Euler–Richardson algorithm is that the quantities $|x_2 - x_1|$ and $|v_2 - v_1|$ give an estimate for the error. We can use these estimates to change the time step so that the error is always within some desired level of precision. We will see that the Euler–Richardson algorithm is equivalent to the second-order Runge–Kutta algorithm [see (3.54)].

One of the most common drift-free higher order algorithms is commonly attributed to Verlet. We write the Taylor series expansion for x_{n-1} in a form similar to (3.32b):

$$x_{n-1} = x_n - v_n\Delta t + \frac{1}{2}a_n(\Delta t)^2. \tag{3.45}$$

If we add the forward and reverse forms, (3.32b) and (3.45) respectively, we obtain

$$x_{n+1} + x_{n-1} = 2x_n + a_n(\Delta t)^2 + O\Big((\Delta t)^4\Big) \tag{3.46}$$

or

$$x_{n+1} = 2x_n - x_{n-1} + a_n(\Delta t)^2 \qquad \text{(leapfrog algorithm).} \tag{3.47a}$$

Similarly, the subtraction of the Taylor series for x_{n+1} and x_{n-1} yields

$$v_n = \frac{x_{n+1} - x_{n-1}}{2\Delta t} \qquad \text{(leapfrog algorithm).} \tag{3.47b}$$

Note that the global error associated with the leapfrog algorithm (3.47) is third-order for the position and second-order for the velocity. However, the velocity plays no part in the integration of the equations of motion. The leapfrog algorithm is also known as the explicit central difference algorithm. Because this form of the leapfrog algorithm is not self-starting, another algorithm must be used to obtain the first few terms. An additional problem is that the new velocity (3.47b) is found by computing the difference between two quantities of the same order of magnitude. Such an operation results in a loss of numerical precision and may give rise to roundoff errors.

A mathematically equivalent version of the leapfrog algorithm is given by

$$x_{n+1} = x_n + v_n\Delta t + \frac{1}{2}a_n(\Delta t)^2 \tag{3.48a}$$

$$v_{n+1} = v_n + \frac{1}{2}(a_{n+1} + a_n)\Delta t \qquad \text{(velocity Verlet algorithm).} \tag{3.48b}$$

We see that (3.48), known as the *velocity* form of the Verlet algorithm, is self-starting and minimizes roundoff errors. Because we will not use (3.47) in the text, we will refer to (3.48) as the Verlet algorithm.

We can derive (3.48) from (3.47) by the following considerations. We first solve (3.47b) for x_{n-1} and write $x_{n-1} = x_{n+1} - 2v_n\Delta t$. If we substitute this expression for x_{n-1} into (3.47a) and solve for x_{n+1}, we find the form (3.48a). Then we use (3.47b) to write v_{n+1} as

$$v_{n+1} = \frac{x_{n+2} - x_n}{2\Delta t} \tag{3.49}$$

and use (3.47a) to obtain $x_{n+2} = 2x_{n+1} - x_n + a_{n+1}(\Delta t)^2$. If we substitute this form for x_{n+2} into (3.49), we obtain

$$v_{n+1} = \frac{x_{n+1} - x_n}{\Delta t} + \frac{1}{2}a_{n+1}\Delta t. \tag{3.50}$$

Finally, we use (3.48a) for x_{n+1} to eliminate $x_{n+1} - x_n$ from (3.50); after some algebra we obtain the desired result (3.48b).

Another useful algorithm that avoids the roundoff errors of the leapfrog algorithm is due to Beeman and Schofield. We write the *Beeman* algorithm in the form

$$x_{n+1} = x_n + v_n\Delta t + \frac{1}{6}(4a_n - a_{n-1})(\Delta t)^2 \tag{3.51a}$$

$$v_{n+1} = v_n + \frac{1}{6}(2a_{n+1} + 5a_n - a_{n-1})\Delta t \qquad \text{(Beeman algorithm).} \tag{3.51b}$$

Note that (3.51) does not calculate particle trajectories more accurately than the Verlet algorithm. Its advantage is that it generally does a better job of maintaining energy conservation. However, the Beeman algorithm is not self-starting.

The most common finite difference method for solving ordinary differential equations is the *Runge–Kutta* method. To explain the many algorithms based on this method, we consider the solution of the first-order differential equation

$$\frac{dx}{dt} = f(x,t). \tag{3.52}$$

Runge–Kutta algorithms evaluate the rate $f(x,t)$ multiple times in the interval $[t, t+\Delta t]$. For example, the classic fourth-order Runge–Kutta algorithm, which we will discuss in the following, evaluates $f(x,t)$ at four times t_n, $t_n + a_1\Delta t$, $t_n + a_2\Delta t$, and $t_n + a_3\Delta t$. Each evaluation of $f(x,t)$ produces a slightly different rate r_1, r_2, r_3, and r_4. The idea is to advance the solution using a weighted average of the intermediate rates:

$$y_{n+1} = y_n + (c_1 r_1 + c_2 r_2 + c_3 r_3 + c_4 r_4)\Delta t. \tag{3.53}$$

The various Runge–Kutta algorithms correspond to different choices for the constants a_i and c_i. These algorithms are classified by the number of intermediate rates $\{r_i,\, i = 1,\ldots,N\}$. The determination of the Runge–Kutta coefficients is difficult for all but the lowest order methods, because the coefficients must be chosen to cancel as many terms in the Taylor series expansion of $f(x,t)$ as possible. The first non-zero expansion coefficient determines the order of the Runge–Kutta algorithm. Fortunately, these coefficients are tabulated in most numerical analysis books.

To illustrate how the various sets of Runge–Kutta constants arise, consider the case $N = 2$. The second-order Runge–Kutta solution of (3.52) can be written using standard notation as

$$x_{n+1} = x_n + k_2 + O\Big((\Delta t)^3\Big) \tag{3.54a}$$

where

$$k_2 = f(x_n + \frac{k_1}{2}, t_n + \frac{\Delta t}{2})\Delta t \tag{3.54b}$$

$$k_1 = f(x_n, t_n)\Delta t. \tag{3.54c}$$

Note that the weighted average uses $c_1 = 0$ and $c_2 = 1$. The interpretation of (3.54) is as follows. The Euler algorithm assumes that the slope $f(x_n, t_n)$ at (x_n, t_n) can be used to extrapolate to the next step, that is, $x_{n+1} = x_n + f(x_n, t_n)\Delta t$. A plausible way of making a more accurate determination of the slope is to use the Euler algorithm to extrapolate to the midpoint of the interval and then to use the midpoint slope across the full width of the interval. Hence, the Runge–Kutta estimate for the rate is $f(x^*, t_n + \frac{1}{2}\Delta t)$, where $x^* = x_n + \frac{1}{2} f(x_n, t_n)\Delta t$ [see (3.54c)].

The application of the second-order Runge–Kutta algorithm to Newton's equation of motion (3.31) yields

$$k_{1v} = a_n(x_n, v_n, t_n)\Delta t \tag{3.55a}$$

$$k_{1x} = v_n\Delta t \tag{3.55b}$$

$$k_{2v} = a\Big(x_n + \frac{k_{1x}}{2}, v_n + \frac{k_{1v}}{2}, t + \frac{\Delta t}{2}\Big)\Delta t \tag{3.55c}$$

$$k_{2x} = \Big(v_n + \frac{k_{1v}}{2}\Big)\Delta t \tag{3.55d}$$

and

$$v_{n+1} = v_n + k_{2v} \tag{3.56a}$$

$$x_{n+1} = x_n + k_{2x}. \qquad \text{(second-order Runge Kutta)} \tag{3.56b}$$

Note that the second-order Runge–Kutta algorithm in (3.55) and (3.56) is identical to the Euler–Richardson algorithm.

Other second-order Runge–Kutta type algorithms exist. For example, if we set $c_1 = c_2 = \frac{1}{2}$ we obtain the endpoint method:

$$y_{n+1} = y_n + \frac{1}{2}k_1 + \frac{1}{2}k_2 \tag{3.57a}$$

where

$$k_1 = f(x_n, t_n)\Delta t \tag{3.57b}$$
$$k_2 = f(x_n + k_1, t_n + \Delta t)\Delta t. \tag{3.57c}$$

And if we set $c_1 = \frac{1}{3}$ and $c_2 = \frac{2}{3}$, we obtain Ralston's method:

$$y_{n+1} = y_n + \frac{1}{3}k_1 + \frac{2}{3}k_2 \tag{3.58a}$$

where

$$k_1 = f(x_n, t_n)\Delta t \tag{3.58b}$$
$$k_2 = f\Big(x_n + \frac{3}{4}k_1, t_n + \frac{3}{4}\Delta t\Big)\Delta t. \tag{3.58c}$$

Note that Ralston's method does not calculate the rate at uniformly spaced subintervals of Δt. In general, a Runge–Kutta method adjusts the partition of Δt as well as the constants a_i and c_i so as to minimize the error.

In the *fourth-order* Runge–Kutta algorithm, the derivative is computed at the beginning of the time interval, in two different ways at the middle of the interval, and again at the end of the interval. The two estimates of the derivative at the middle of the interval are given twice the weight of the other two estimates. The algorithm for the solution of (3.52) can be written in standard notation as

$$k_1 = f(x_n, t_n)\Delta t \tag{3.59a}$$
$$k_2 = f\Big(x_n + \frac{k_1}{2}, t_n + \frac{\Delta t}{2}\Big)\Delta t \tag{3.59b}$$
$$k_3 = f\Big(x_n + \frac{k_2}{2}, t_n + \frac{\Delta t}{2}\Big)\Delta t \tag{3.59c}$$
$$k_4 = f(x_n + k_3, t_n + \Delta t)\Delta t \tag{3.59d}$$

and

$$x_{n+1} = x_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4). \tag{3.60}$$

The application of the fourth-order Runge–Kutta algorithm to Newton's equation of motion

(3.31) yields

$$\begin{aligned}
k_{1v} &= a(x_n, v_n, t_n)\Delta t && (3.61a)\\
k_{1x} &= v_n \Delta t && (3.61b)\\
k_{2v} &= a\Big(x_n + \frac{k_{1x}}{2}, v_n + \frac{k_{1v}}{2}, t_n + \frac{\Delta t}{2}\Big)\Delta t && (3.61c)\\
k_{2x} &= \Big(v_n + \frac{k_{1v}}{2}\Big)\Delta t && (3.61d)\\
k_{3v} &= a\Big(x_n + \frac{k_{2x}}{2}, v_n + \frac{k_{2v}}{2}, t_n + \frac{\Delta t}{2}\Big)\Delta t && (3.61e)\\
k_{3x} &= (v_n + \frac{k_{2v}}{2})\Delta t && (3.61f)\\
k_{4v} &= a(x_n + k_{3x}, v_n + k_{3v}, t + \Delta t)\Delta t && (3.61g)\\
k_{4x} &= (v_n + k_{3v})\Delta t, && (3.61h)
\end{aligned}$$

and

$$\begin{aligned}
v_{n+1} &= v_n + \frac{1}{6}(k_{1v} + 2k_{2v} + 2k_{3v} + k_{4v}) && (3.62a)\\
x_{n+1} &= x_n + \frac{1}{6}(k_{1x} + 2k_{2x} + 2k_{3x} + k_{4x}) \quad \text{(fourth-order Runge–Kutta).} && (3.62b)
\end{aligned}$$

Because Runge–Kutta algorithms are self-starting, they are frequently used to obtain the first few iterations for an algorithm that is not self-starting.

As we have discussed, one way to determine the accuracy of a solution is to calculate it twice with two different values of the time step. One way to make this comparison is to choose time steps Δt and $\Delta t/2$ and compare the solution at the desired time. If the difference is small, the error is assumed to be small. This estimate of the error can be used to adjust the value of the time step. If the error is too large, than the time step can be halved. And if the error is much less than the desired value, the time step can be increased so that the program runs faster.

A better way of controlling the step size was developed by Fehlberg who showed that it is possible to evaluate the rate in such a way as to simultaneously obtain two Runge–Kutta approximations with different orders. For example, it is possible to run a fourth-order and fifth-order algorithm in tandem by evaluating five rates. We thus obtain different estimates of the true solution using different weighed averages of these rates:

$$\begin{aligned}
y_{n+1} &= y_n + c_1 k_1 + c_2 k_2 + c_3 k_3 + c_4 k_4 + c_5 k_5 && (3.63a)\\
y^*_{n+1} &= y_n + c^*_1 k_1 + c^*_2 k_2 + c^*_3 k_3 + c^*_4 k_4. && (3.63b)
\end{aligned}$$

Because we can assume that the fifth-order solution is closer to the true solution than the fourth-order algorithm, the difference $|y - y^*|$ provides a good estimate of the error of the fourth-order method. If this estimated error is larger than the desired tolerance, then the step size is decreased. If the error is smaller than the desired tolerance, the step size is increased. The `RK45` ODE solver in the numerics package implements this technique for choosing the optimal step size.

In applications where the accuracy of the numerical solution is important, adaptive time step algorithms should always be used. As stated in *Numerical Recipes*: “Many small steps should tiptoe through treacherous terrain, while a few great strides should speed through uninteresting countryside. The resulting gains in efficiency are not mere tens of percents or factors of two; they can sometimes be factors of ten, a hundred, or more."

Adaptive step size algorithms are not well suited for tabulating functions or for simulation because the intervals between data points are not constant. An easy way to circumvent this problem is to use a method that takes multiple adaptive steps while checking to insure that the last step does not overshoot the desired fixed step size. The `ODEMultistepSolver` implements this technique. The solver acts like a fixed step size solver, even though the solver monitors its internal step size so as to achieve the desired accuracy.

It also is possible to combine the results from a calculation using two different values of the time step to yield a more accurate expression. Consider the Taylor series expansion of $f(t+\Delta t)$ about t:

$$f(t+\Delta t) = f(t) + f'(t)\Delta t + \frac{1}{2!}f''(t)(\Delta t)^2 + \cdots. \tag{3.64}$$

Similarly, we have

$$f(t-\Delta t) = f(t) - f'(t)\Delta t + \frac{1}{2!}f''(t)(\Delta t)^2 + \cdots. \tag{3.65}$$

We subtract (3.65) from (3.64) to find the usual central difference approximation for the derivative

$$f'(t) \approx D_1(\Delta t) = \frac{f(t+\Delta t) - f(t-\Delta t)}{2\Delta t} - \frac{(\Delta t)^2}{6}f'''(t). \tag{3.66}$$

The truncation error is order $(\Delta t)^2$. Next consider the same relation, but for a time step that is twice as large:

$$f'(t) \approx D_1(2\Delta t) = \frac{f(t+2\Delta t) - f(t-2\Delta t)}{4\Delta t} - \frac{4(\Delta t)^2}{6}f'''(t). \tag{3.67}$$

Note that the truncation error is again order $(\Delta t)^2$, but is four times bigger. We can eliminate this error to leading order by dividing (3.67) by 4 and subtracting it from (3.66):

$$f'(t) - \frac{1}{4}f'(t) = \frac{3}{4}f'(t) \approx D_1(\Delta t) - \frac{1}{4}D_1(2\Delta t)$$

or

$$f'(t) \approx \frac{4D_1(\Delta t) - D_1(2\Delta t)}{3}. \tag{3.68}$$

It is easy to show that the error for $f'(t)$ is order $(\Delta t)^4$. Recursive difference formulas for derivatives can be obtained by canceling the truncation error at each order. This method is called *Richardson extrapolation*.

Another class of algorithms are *predictor–corrector* algorithms. The idea is to first *predict* the value of the new position:

$$x_p = x_{n-1} + 2v_n\Delta t \qquad \text{(predictor)}. \tag{3.69}$$

The predicted value of the position allows us to predict the acceleration a_p. Then using a_p, we obtain the *corrected* values of v_{n+1} and x_{n+1}:

$$v_{n+1} = v_n + \frac{1}{2}(a_p + a_n)\Delta t \tag{3.70a}$$

$$x_{n+1} = x_n + \frac{1}{2}(v_{n+1} + v_n)\Delta t \qquad \text{(corrected)}. \tag{3.70b}$$

The corrected values of x_{n+1} and v_{n+1} are used to obtain a new predicted value of a_{n+1}, and, hence, a new predicted value of v_{n+1} and x_{n+1}. This process is repeated until the predicted and corrected values of x_{n+1} differ by less than the desired value.

Note that the predictor–corrector algorithm is not self-starting. The predictor–corrector algorithm gives more accurate positions and velocities than the leapfrog algorithm and is suitable for very accurate calculations. However, it is computationally expensive, needs significant storage (the forces at the last two stages and the coordinates and velocities at the last step), and becomes unstable for large time steps.

As we have emphasized, there is no single algorithm for solving Newton's equations of motion that is superior under all conditions. It is usually a good idea to start with a simple algorithm and then to try a higher order algorithm to see if any real improvement is obtained.

We now discuss an important class of algorithms, known as *symplectic* algorithms, which are particularly suitable for doing long time simulations of Newton's equations of motion when the force is only a function of position. The basic idea of these algorithms derives from the Hamiltonian theory of classical mechanics. We first give some basic results needed from this theory to understand the importance of symplectic algorithms.

In Hamiltonian theory the generalized coordinates q_i and momenta p_i take the place of the usual positions and velocities familiar from Newtonian theory. The index i labels both a particle and a component of the motion. For example, in a two- particle system in two dimensions, i would run from 1 to 4. The Hamiltonian (which for our purposes can be thought of as the total energy) is written as

$$H(q_i, p_i) = T + V \tag{3.71}$$

where T is the kinetic energy and V is the potential energy. Hamilton's theory is most relevant for nondissipative systems, which we consider here. For example, for a two particle system in two dimensions connected by a spring, H would take the form

$$H = \frac{p_1^2}{2m} + \frac{p_2^2}{2m} + \frac{p_3^2}{2m} + \frac{p_4^2}{2m} + \frac{1}{2}k(q_1 - q_3)^2 + \frac{1}{2}k(q_2 - q_4)^2 \tag{3.72}$$

where if the particles are labeled as A and B, we have $p_1 = p_{x,A}$, $p_2 = p_{y,A}$, $p_3 = p_{x,B}$, $p_4 = p_{y,B}$, and similarly for the q_i. The equations of motion are written as first-order differential equations known as Hamilton's equations:

$$\dot{p}_i = -\frac{\partial H}{\partial q_i} \tag{3.73a}$$

$$\dot{q}_i = \frac{\partial H}{\partial p_i} \tag{3.73b}$$

which are equivalent to Newton's second law and an equation relating the velocity to the momentum. The beauty of Hamiltonian theory is that these equations are correct for other coordinate systems such as polar coordinates, and they also describe rotating systems where the momenta become angular momenta, and the position coordinates become angles.

Because the coordinates and momenta are treated on an equal footing, we can consider the properties of flow in phase space where the dimension of phase space includes both the coordinates and momenta. Thus, one particle moving in one dimension corresponds to a two-dimensional phase space. If we imagine a collection of initial conditions in phase space forming a volume in phase space, then one of the results of Hamiltonian theory is that this volume does not change as the system evolves. A slightly different result, called the *symplectic* property, is that the sum of the areas formed by the projection of the phase space volume onto the planes q_i, p_i for each pair of coordinates and momenta also does not change with time. Numerical algorithms that have this property are called symplectic algorithms. These algorithms are built from the following two statements which are repeated M times for each time step.

$$p_i^{(k+1)} = p_i^{(k)} + a_k F_i^{(k)} \delta t \tag{3.74a}$$

$$q_i^{(k+1)} = q_i^{(k)} + b_k p_i^{(k+1)} \delta t \tag{3.74b}$$

where $F_i^{(k)} \equiv -\partial V(q_i^{(k)})/\partial q_i^{(k)}$. The label k runs from 0 to $M-1$ and one time step is given by $\Delta t = M\delta t$. (We will see that δt is the time step of an intermediate calculation that is made during the time step Δt.) Note that in the update for q_i, the already updated p_i is used. For simplicity, we assume that the mass is unity.

Different algorithms correspond to different values of M, a_k, and b_k. For example, $a_0 = b_0 = M = 1$ corresponds to the Euler–Cromer algorithm, and $M = 2$, $a_0 = a_1 = 1$, $b_0 = 2$, and $b_1 = 0$ is equivalent to the Verlet algorithm as we will now show. If we substitute in the appropriate values for a_k and b_k into (3.74), we have

$$p_i^{(1)} = p_i^{(0)} + F_i^{(0)} \delta t \tag{3.75a}$$

$$q_i^{(1)} = q_i^{(0)} + 2p_i^{(1)} \delta t \tag{3.75b}$$

$$p_i^{(2)} = p_i^{(1)} + F_i^{(1)} \delta t \tag{3.75c}$$

$$q_i^{(2)} = q_i^{(1)}. \tag{3.75d}$$

We next combine (3.75a) and (3.75c) for the momentum coordinate and (3.75b) and (3.75d) for the position and obtain

$$p_i^{(2)} = p_i^{(0)} + (F_i^{(0)} + F_i^{(1)})\delta t \tag{3.76a}$$

$$q_i^{(2)} = q_i^{(0)} + 2p_i^{(1)} \delta t. \tag{3.76b}$$

We take $\delta t = \Delta t/2$ and combine (3.76b) with (3.75a) and find

$$p_i^{(2)} = p_i^{(0)} + \frac{1}{2}(F_i^{(0)} + F_i^{(1)})\Delta t \tag{3.77a}$$

$$q_i^{(2)} = q_i^{(0)} + p_i^{(0)}\Delta t + \frac{1}{2}F_i^{(0)}(\Delta t)^2 \tag{3.77b}$$

which is identical to the Verlet algorithm (3.48), because for unit mass the force and acceleration are equal.

Reversing the order of the updates for the coordinates and the momenta also leads to symplectic algorithms:

$$q_i^{(k+1)} = q_i^{(k)} + b_k \delta t p_i^{(k)} \tag{3.78a}$$

$$p_i^{(k+1)} = p_i^{(k)} + a_k \delta t F_i^{(k+1)}. \tag{3.78b}$$

A third variation uses (3.74) and (3.78) for different values of k in one algorithm. Thus, if $M = 2$, which corresponds to two intermediate calculations per time step, we could use (3.74) for the first intermediate calculation and (3.78) for the second.

Why are these algorithms important? Because of the symplectic property, these algorithms will simulate an exact Hamiltonian, although not the one we started with in general (see Problem 3.1c, for example). However, this Hamiltonian will be close to the one we wish to simulate if the a_k and b_k are properly chosen. Second, these algorithms are frequently more accurate and stable than nonsymplectic algorithms. Finally, for even values of M, the algorithms are time-reversible invariant, which is a property of the actual systems we are trying to simulate. Examples and comparisons for various algorithms are given in the paper by Gray et al.

References and Suggestions for Further Reading

F. S. Acton, *Numerical Methods That Work* (The Mathematical Association of America, 1990), Chapter 5.

Robert. K. Adair, *The Physics of Baseball*, 3rd ed. (Harper Collins, 2002).

Byron L. Coulter and Carl G. Adler, "Can a body pass a body falling through the air?," Am. J. Phys. **47**, 841–846 (1979). The authors discuss the limiting conditions for which the drag force is linear or quadratic in the velocity.

Alan Cromer, "Stable solutions using the Euler approximation," Am. J. Phys. **49**, 455–459 (1981). The author shows that a minor modification of the usual Euler approximation yields stable solutions for oscillatory systems including planetary motion and the harmonic oscillator (see Chapter 4).

Paul L. DeVries, *A First Course in Computational Physics* (John Wiley & Sons, 1994).

Denis Donnelly and Edwin Rogers, "Symplectic integrators: An introduction," Am. J. Phys. **73**, 938 (2005).

A. P. French, *Newtonian Mechanics* (W. W. Norton & Company, 1971). Chapter 7 has an excellent discussion of air resistance and a detailed analysis of motion in the presence of drag resistance.

Ian R. Gatland, "Numerical integration of Newton's equations including velocity-dependent forces," Am J. Phys. **62**, 259–265 (1994). The author discusses the Euler–Richardson algorithm.

Stephen K. Gray, Donald W. Noid, and Bobby G. Sumpter, "Symplectic integrators for large scale molecular dynamics simulations: A comparison of several explicit methods," J. Chem. Phys. **101** (5), 4062–4072 (1994).

Margaret Greenwood, Charles Hanna, and John Milton, "Air resistance acting on a sphere: Numerical analysis, strobe photographs, and videotapes," Phys. Teacher **24**, 153–159 (1986). More experimental data and theoretical analysis are given for the fall of ping-pong and styrofoam balls. Also see Mark Peastrel, Rosemary Lynch, and Angelo Armenti, "Terminal velocity of a shuttlecock in vertical fall," Am. J. Phys. **48**, 511–513 (1980).

Michael J. Kallaher, editor, *Revolutions in Differential Equations: Exploring ODEs with Modern Technology* (The Mathematical Association of America, 1999).

K. S. Krane, "The falling raindrop: variations on a theme of Newton," Am. J. Phys. **49**, 113–117 (1981). The author discusses the problem of mass accretion by a drop falling through a cloud of droplets.

D. B. Lichtenberg and J. G. Wills, "Maximizing the range of the shot put," Am. J. Phys. **46** (5), 546–549 (1978).

George C. McGuire, "Using computer algebra to investigate the motion of an electric charge in magnetic and electric dipole fields," Am. J. Phys. **71** (8), 809–812 (2003).

Rabindra Mehta, "Aerodynamics of sports balls," in Ann. Rev. Fluid Mech. **17**, 151–189 (1985).

Neville de Mestre, *The Mathematics of Projectiles in Sport* (Cambridge University Press, 1990). The emphasis of this text is on solving many problems in projectile motion, for example, baseball, basketball, and golf, in the context of mathematical modeling. Many references to the relevant literature are given.

Tao Pang, *Computational Physics* (Cambridge University Press, 1997).

William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery, *Numerical Recipes*, 2nd ed. (Cambridge University Press, 1992). Chapter 16 discusses the integration of ordinary differential equations.

Emilio Segré, *Nuclei and Particles*, 2nd ed. (W. A. Benjamin, 1977). Chapter 5 discusses decay cascades. The decay schemes described briefly in Problem 3.13 are taken from C. M. Lederer, J. M. Hollander, and I. Perlman, *Table of Isotopes*, 6th ed. (John Wiley & Sons, 1967).

Lawrence F. Shampine, *Numerical Solution of Ordinary Differential Equations* (Chapman and Hall, 1994).

Chapter 4

Oscillations

We explore the behavior of oscillatory systems, including the simple harmonic oscillator, a simple pendulum, and electrical circuits, and introduce the concept of phase space.

4.1 Simple Harmonic Motion

There are many physical systems that undergo regular, repeating motion. Motion that repeats itself at definite intervals, for example, the motion of the earth about the sun, is said to be *periodic*. If an object undergoes periodic motion between two limits over the same path, we call the motion *oscillatory*. Examples of oscillatory motion that are familiar to us from our everyday experience include a plucked guitar string and the pendulum in a grandfather clock. Less obvious examples are microscopic phenomena such as the oscillations of the atoms in crystalline solids.

To illustrate the important concepts associated with oscillatory phenomena, consider a block of mass m connected to the free end of a spring. The block slides on a frictionless, horizontal surface (see Figure 4.1). We specify the position of the block by x and take $x = 0$ to be the equilibrium position of the block, that is, the position when the spring is relaxed. If the block is moved from $x = 0$ and then released, the block oscillates along a horizontal line. If the spring is not compressed or stretched too far from $x = 0$, the force on the block at position x is proportional to x:

$$F = -kx. \tag{4.1}$$

The force constant k is a measure of the stiffness of the spring. The negative sign in (4.1) implies that the force acts to restore the block to its equilibrium position. Newton's equation of motion for the block can be written as

$$\frac{d^2x}{dt^2} = -{\omega_0}^2 x \tag{4.2}$$

where the angular frequency ω_0 is defined by

$${\omega_0}^2 = \frac{k}{m}. \tag{4.3}$$

The dynamical behavior described by (4.2) is called *simple harmonic motion* and can be solved analytically in terms of sine and cosine functions. Because the form of the solution will

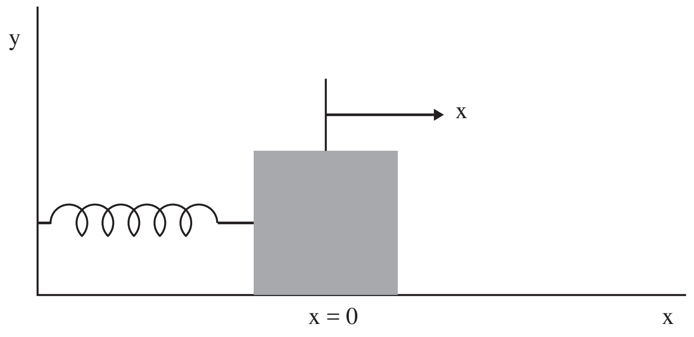


Figure 4.1: A one-dimensional harmonic oscillator. The block slides horizontally on the frictionless surface.

help us introduce some of the terminology needed to discuss oscillatory motion, we include the solution here. One form of the solution is

$$x(t) = A\cos(\omega_0 t + \delta) \tag{4.4}$$

where A and δ are constants and the argument of the cosine is in radians. It is straightforward to check by substitution that (4.4) is a solution of (4.2). The constants A and δ are called the amplitude and the phase, respectively, and are determined by the initial conditions for x and the velocity $v = dx/dt$.

Because the cosine is a periodic function with period 2π, we know that $x(t)$ in (4.4) is also periodic. We define the period T as the smallest time for which the motion repeats itself, that is,

$$x(t+T) = x(t). \tag{4.5}$$

Because $\omega_0 T$ corresponds to one cycle, we have

$$T = \frac{2\pi}{\omega_0} = \frac{2\pi}{\sqrt{k/m}}. \tag{4.6}$$

The frequency ν of the motion is the number of cycles per second and is given by $\nu = 1/T$. Note that T depends on the ratio k/m and not on A and δ. Hence, the period of simple harmonic motion is independent of the amplitude of the motion.

Although the position and velocity of the oscillator are continuously changing, the total energy E remains constant and is given by

$$E = \frac{1}{2}mv^2 + \frac{1}{2}kx^2 = \frac{1}{2}kA^2. \tag{4.7}$$

The two terms in (4.7) are the kinetic and potential energies, respectively.

Problem 4.1. Energy conservation

(a) Use the Euler `ODESolver` to solve the dynamical equations for a simple harmonic oscillator by extending `AbstractSimulation` and implementing the `doStep` method. (See Section 4.2 for an example of such a program for the pendulum.) Have your program plot $\Delta E_n = E_n - E_0$, where E_0 is the initial energy and E_n is the total energy at time $t_n = t_0 + n\Delta t$. (It is necessary to consider only the energy per unit mass.) Plot the difference ΔE_n as a function of t_n for several cycles for a given value of Δt. Choose $x(t=0)=1$, $v(t=0)=0$ and $\omega_0{}^2 = k/m = 9$ and start with $\Delta t = 0.05$. Is the difference ΔE_n uniformly small throughout the cycle? Does ΔE_n drift, that is, become bigger with time? What is the optimum choice of Δt?

(b) Implement the Euler–Cromer algorithm by writing an Euler–Cromer ODESolver and answer the same questions as in part (a).

(c) Modify your program so that the Euler–Richardson or Verlet algorithms are used and answer the same questions as in part (a). (The Verlet algorithm is discussed in Appendix 3A.)

(d) Describe the qualitative differences between the time dependence of ΔE_n using the various algorithms. Which algorithm is most consistent with the requirement of conservation of energy? For fixed Δt, which algorithm yields better results for the position in comparison to the analytic solution (4.4)? Is the requirement of conservation of energy consistent with the relative accuracy of the computed positions?

(e) Choose the best algorithm based on your criteria, and determine the values of Δt that are needed to conserve the total energy to within 0.1% over one cycle for $\omega_0 = 3$ and for $\omega_0 = 12$. Can you use the same value of Δt for both values of ω_0? If not, how do the values of Δt correspond to the relative values of the period in the two cases? □

Problem 4.2. Analysis of simple harmonic motion

a) Use your results from Problem 4.1 to select an appropriate numerical algorithm and value of Δt for the simple harmonic oscillator, and modify your program so that the time dependence of the potential and kinetic energies is plotted. Where in the cycle is the kinetic energy (potential energy) a maximum?

b) Compute the average value of the kinetic energy and the potential energy during a complete cycle. What is the relation between the two averages?

c) Compute $x(t)$ for different values of A and show that the shape of $x(t)$ is independent of A; that is, show that $x(t)/A$ is a *universal* function of t for a fixed value of ω_0. In what units should the time be measured so that the ratio $x(t)/A$ is independent of ω_0?

d) The dynamical behavior of the one-dimensional oscillator is completely specified by $x(t)$ and $p(t)$, where p is the momentum of the oscillator. These quantities may be interpreted as the coordinates of a point in a two-dimensional space known as *phase space*. As the time increases, the point $(x(t), p(t))$ moves along a trajectory in phase space. Modify your program so that the momentum p is plotted as a function of x; that is, choose p and x as the vertical and horizontal axes, respectively. Choose $\omega_0 = 3$ and compute the phase space trajectory for the initial condition $x(t=0)=1, v(t=0)=0$. What is the shape of this trajectory? What is the shape for the initial conditions $x(t=0)=0, v(t=0)=1$ and $x(t=0)=4, v(t=0)=0$? Do you find a different phase trajectory for each initial condition? What physical quantity distinguishes the phase space trajectories? Is the motion of a representative point (x,p) always in the clockwise or counterclockwise direction? □

Problem 4.3. Lissajous figures

A computer display can be used to simulate the output seen on an oscilloscope. Imagine that the vertical and horizontal inputs to an oscilloscope are sinusoidal in time; that is, $x = A_x \sin(\omega_x t + \phi_x)$ and $y = A_y \sin(\omega_y t + \phi_y)$. If the curve that is drawn repeats itself, such a curve is called a *Lissajous figure*. Write a program to plot y versus x, as t advances from $t = 0$. First choose $A_x = A_y = 1$, $\omega_x = 2$, $\omega_y = 3$, $\phi_x = \pi/6$, and $\phi_y = \pi/4$. For what values of the angular frequencies ω_x and ω_y do you obtain a Lissajous figure? How do the phase factors ϕ_x and ϕ_y and the amplitudes A_x and A_y affect the curves? □

Waves are ubiquitous in nature and give rise to important phenomena such as beats and standing waves. We investigate their behavior in Problem 4.4. We will study the behavior of waves more systematically in Chapter 9.

Problem 4.4. Superposition of waves

(a) Write a program to plot $A\sin(kx + \omega t)$ from $x = x_{\min}$ to $x = x_{\max}$ as a function of t. (Implement an `AbstractSimulation` rather than an `AbstractCalculation`.) For simplicity, take $A = 1$, $\omega = 2\pi$, $k = 2\pi/\lambda$, with $\lambda = 2$.

(b) Modify your program so that it plots the sum of $y_1 = \sin(kx - \omega t)$ and $y_2 = \sin(kx + \omega t)$. The quantity $y_1 + y_2$ corresponds to the superposition of two waves. Choose $\lambda = 2$ and $\omega = 2\pi$. What kind of a wave do you obtain?

(c) Use your program to demonstrate beats by plotting $y_1 + y_2$ as a function of time in the range $x_{\min} = -10$ and $x_{\max} = 10$. Determine the beat frequency for each of the following superpositions: $y_1(x,t) = \sin[8.4(x - 1.1t)]$, $y_2(x,t) = \sin[8.0(x - 1.1t)]$; $y_1(x,t) = \sin[8.4(x - 1.2t)]$, $y_2(x,t) = \sin[8.0(x - 1.0t)]$; and $y_1(x,t) = \sin[8.4(x - 1.0t)]$, $y_2(x,t) = \sin[8.0(x - 1.2t)]$. What differences do you observe between these superpositions? □

4.2 The Motion of a Pendulum

A common example of a mechanical system that exhibits oscillatory motion is the simple pendulum (see Figure 4.2). A simple pendulum is an idealized system consisting of a particle or bob of mass m attached to the lower end of a rigid rod of length L and negligible mass; the upper end of the rod pivots without friction. If the bob is pulled to one side from its equilibrium position and released, the pendulum swings in a vertical plane.

Because the bob is constrained to move along the arc of a circle of radius L about the center O, the bob's position is specified by its arc length or by the angle θ (see Figure 4.2). The linear velocity and acceleration of the bob as measured along the arc are given by

$$v = L\frac{d\theta}{dt} \tag{4.8}$$

$$a = L\frac{d^2\theta}{dt^2}. \tag{4.9}$$

In the absence of friction, two forces act on the bob: the force mg vertically downward and the force of the rod which is directed inward to the center if $|\theta| < \pi/2$. Note that the effect of the rigid rod is to constrain the motion of the bob along the arc. From Figure 4.2, we can see that the component of mg along the arc is $mg\sin\theta$ in the direction of decreasing θ. Hence, the equation of motion can be written as

$$mL\frac{d^2\theta}{dt^2} = -mg\sin\theta \tag{4.10}$$

or

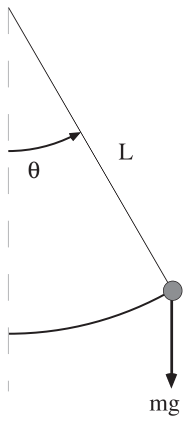


Figure 4.2: Force diagram for a simple pendulum. The angle θ is measured from the vertical direction and is positive if the mass is to the right of the vertical and negative if it is to the left.

$$\frac{d^2\theta}{dt^2} = -\frac{g}{L}\sin\theta. \tag{4.11}$$

Equation (4.11) is an example of a nonlinear equation because $\sin\theta$ rather than θ appears. Most nonlinear equations do not have analytic solutions in terms of well-known functions, and (4.11) is no exception. However, if the amplitude of the pendulum oscillations is sufficiently small, then $\sin\theta \approx \theta$, and (4.11) reduces to

$$\frac{d^2\theta}{dt^2} \approx -\frac{g}{L}\theta \qquad (\theta \ll 1). \tag{4.12}$$

Remember that θ is measured in radians.

Part of the fun of studying physics comes from realizing that equations that appear in different contexts are often similar. An example can be seen by comparing (4.2) and (4.12). If we associate x with θ, we see that the two equations are identical in form, and we can immediately conclude that for $\theta \ll 1$, the period of a pendulum is given by

$$T = 2\pi\sqrt{\frac{L}{g}} \qquad \text{(small amplitude oscillations).} \tag{4.13}$$

One way to understand the motion of a pendulum with large oscillations is to solve (4.11) numerically. Because we know that the numerical solutions must be consistent with conservation of energy, we derive the form of the total energy here. The potential energy can be found from the following considerations. If the rod is deflected by the angle θ, then the bob is raised by the distance $h = L - L\cos\theta$ (see Figure 4.2). Hence, the potential energy of the bob in the gravitational field of the earth is

$$U = mgh = mgL(1-\cos\theta) \tag{4.14}$$

where the zero of the potential energy corresponds to $\theta = 0$. Because the kinetic energy of the pendulum is $\frac{1}{2}mv^2 = \frac{1}{2}mL^2(d\theta/dt)^2$, the total energy E of the pendulum is

$$E = \frac{1}{2}mL^2\Big(\frac{d\theta}{dt}\Big)^2 + mgL(1-\cos\theta). \tag{4.15}$$

We use two classes to simulate and visualize the motion of a pendulum problem, `Pendulum` and `PendulumApp`. The `Pendulum` class implements the `Drawable` and `ODE` interfaces and solves the dynamical equations using the Euler–Richardson algorithm.

Listing 4.1: A `Drawable` class that models the simple pendulum.

```java
package org.opensourcephysics.sip.ch04;
import java.awt.*;
import org.opensourcephysics.display.*;
import org.opensourcephysics.numerics.*;

public class Pendulum implements Drawable, ODE {
   double omega0Squared = 3;                  //  g/L
   double[] state = new double[] {0, 0, 0}; // {theta, dtheta/dt, t}
   Color color = Color.RED;
   int pixRadius = 6;
   EulerRichardson odeSolver = new EulerRichardson(this);

   public void setStepSize(double dt) {
      odeSolver.setStepSize(dt);
   }

   public void step() {
      odeSolver.step(); // execute one Euler-Richardson step
   }

   public void setState(double theta, double thetaDot) {
      state[0] = theta;
      state[1] = thetaDot; // time rate of change of theta
   }

   public double[] getState() {
      return state;
   }

   public void getRate(double[] state, double[] rate) {
      rate[0] = state[1];             // rate of change of angle
      // rate of change of dtheta/dt
      rate[1] = -omega0Squared*Math.sin(state[0]);
      rate[2] = 1;                    // rate of change of time dt/dt = 1
   }

   public void draw(DrawingPanel drawingPanel, Graphics g) {
      int xpivot = drawingPanel.xToPix(0);
      int ypivot = drawingPanel.yToPix(0);
      int xpix = drawingPanel.xToPix(Math.sin(state[0]));
      int ypix = drawingPanel.yToPix(-Math.cos(state[0]));
      g.setColor(Color.black);
      g.drawLine(xpivot, ypivot, xpix, ypix);          // the string
      g.setColor(color);
      g.fillOval(xpix-pixRadius, ypix-pixRadius, 2*pixRadius,
         2*pixRadius); // the bob
   }
}
```

Note that `Pendulum` implements the `draw` method as required by the `Drawable` interface.

The target class, `PendulumApp`, is shown in Listing 4.2. The angle θ is plotted as a function of time and an animation of the motion is drawn.

Listing 4.2: Visualization of the motion of a pendulum.

```java
package org.opensourcephysics.sip.ch04;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.frames.*;

public class PendulumApp extends AbstractSimulation {
   PlotFrame plotFrame = new PlotFrame("Time", "Theta",
                                  "Theta versus time");
   Pendulum pendulum = new Pendulum();
   DisplayFrame displayFrame = new DisplayFrame("Pendulum");

   public PendulumApp() {
      displayFrame.addDrawable(pendulum);
      displayFrame.setPreferredMinMax(-1.2, 1.2, -1.2, 1.2);
   }

   public void initialize() {
      double dt = control.getDouble("dt");
      double theta = control.getDouble("initial theta");
      double thetaDot = control.getDouble("initial dtheta/dt");
      pendulum.setState(theta, thetaDot);
      pendulum.setStepSize(dt);
   }

   public void doStep() {
      // angle vs time data added
      plotFrame.append(0, pendulum.state[2], pendulum.state[0]);
      pendulum.step();              // advances the state by one time step
   }

   public void reset() {
      pendulum.state[2] = 0; // set time = 0
      control.setValue("initial theta", 0.2);
      control.setValue("initial dtheta/dt", 0);
      control.setValue("dt", 0.1);
   }

   // creates a simulation control structure using this class
   public static void main(String[] args) {
      SimulationControl.createApp(new PendulumApp());
   }
}
```

Problem 4.5. Oscillations of a pendulum

(a) Make the necessary changes so that the analytic solution for small angles is also plotted.

(b) Test the program at sufficiently small amplitudes so that $\sin\theta \approx \theta$. Choose $\omega_0 = \sqrt{g/L} = 3$ and the initial conditions $\theta(t = 0) = 0.2$ and $d\theta(t = 0)/dt = 0$. Determine the period numerically and compare your result to the expected analytic result for small amplitudes. Explain your method for determining the period. Estimate the error due to the small angle approximation for these initial conditions.

(c) Consider larger amplitudes and make plots of $\theta(t)$ and $d\theta(t)/dt$ versus t for the initial conditions $\theta(t=0) = 0.1$, 0.2, 0.4, 0.8, and 1.0 with $d\theta(t=0)/dt = 0$. Choose Δt so that the numerical algorithm generates a stable solution; that is, monitor the total energy and ensure that it does not drift from its initial value. Describe the qualitative behavior of θ and $d\theta/dt$. What is the period T and the amplitude $\theta_{\rm max}$ in each case? Plot T versus $\theta_{\rm max}$ and discuss the qualitative dependence of the period on the amplitude. How do your results for T compare in the linear and nonlinear cases; for example, which period is larger? Explain the relative values of T in terms of the relative magnitudes of the restoring force in the two cases. □

4.3 Damped Harmonic Oscillator

We know from experience that most oscillatory motion in nature gradually decreases until the displacement becomes zero; such motion is said to be *damped* and the system is said to be *dissipative* rather than conservative. As an example of a damped harmonic oscillator, consider the motion of the block in Figure 4.1 when a horizontal drag force is included. For small velocities, it is a reasonable approximation to assume that the drag force is proportional to the first power of the velocity. In this case the equation of motion can be written as

$$\frac{d^2x}{dt^2} = -{\omega_0}^2 x - \gamma \frac{dx}{dt}. \tag{4.16}$$

The *damping coefficient* γ is a measure of the magnitude of the drag term. Note that the drag force in (4.16) opposes the motion. We simulate the behavior of the damped linear oscillator in Problem 4.6.

Problem 4.6. Damped linear oscillator

(a) Incorporate the effects of damping into your harmonic oscillator simulation and plot the time dependence of the position and the velocity. Describe the qualitative behavior of $x(t)$ and $v(t)$ for $\omega_0 = 3$ and $\gamma = 0.5$ with $x(t=0) = 1$, $v(t=0) = 0$.

(b) The period of the motion is the time between successive maxima of $x(t)$. Compute the period and corresponding angular frequency and compare their values to the undamped case. Is the period longer or shorter? Make additional runs for $\gamma = 1$, 2, and 3. Does the period increase or decrease with greater damping? Why?

(c) The amplitude is the maximum value of x during one cycle. Compute the *relaxation time* τ, the time it takes for the amplitude of an oscillation to decrease by $1/e \approx 0.37$ from its maximum value. Is the value of τ constant throughout the motion? Compute τ for the values of γ considered in part (b) and discuss the qualitative dependence of τ on γ.

(d) Plot the total energy as a function of time for the values of γ considered in part (b). If the decrease in energy is not monotonic, explain.

(e) Compute the time dependence of $x(t)$ and $v(t)$ for $\gamma = 4$, 5, 6, 7, and 8. Is the motion oscillatory for all γ? How can you characterize the decay? For fixed ω_0, the oscillator is said to be *critically damped* at the smallest value of γ for which the decay to equilibrium is monotonic. For what value of γ does critical damping occur for $\omega_0 = 4$ and $\omega_0 = 2$? For each value of ω_0, compute the value of γ for which the system approaches equilibrium most quickly.

(f) Compute the phase space diagram for $\omega_0 = 3$ and $\gamma = 0.5$, 2, 4, 6, and 8. Why does the phase space trajectory converge to the origin, $x = 0$, $v = 0$? This point is called an *attractor*. Are these qualitative features of the phase space plot independent of γ? □

Problem 4.7. Damped nonlinear pendulum

Consider a damped pendulum with $\omega_0 = \sqrt{g/L} = 3$ and a damping term equal to $-\gamma d\theta/dt$. Choose $\gamma = 1$ and the initial condition $\theta(t = 0) = 0.2, d\theta(t = 0)/dt = 0$. In what ways is the motion of the damped nonlinear pendulum similar to the damped linear oscillator? In what ways is it different? What is the shape of the phase space trajectory for the initial condition $\theta(t = 0) = 1, \omega(t = 0) = 0$? Do you find a different phase space trajectory for other initial conditions? Remember that θ is restricted to be between $-\pi$ and $+\pi$. □

4.4 Response to External Forces

How can we determine the period of a pendulum that is not already in motion? The obvious way is to disturb the system, for example, to displace the bob and observe its motion. We will find that the nature of the response of the system to a small perturbation tells us something about the nature of the system in the absence of the perturbation.

Consider the driven damped linear oscillator with an external force $F(t)$ in addition to the linear restoring force and linear damping force. The equation of motion can be written as

$$\frac{d^2x}{dt^2} = -{\omega_0}^2 x - \gamma v + \frac{1}{m}F(t). \tag{4.17}$$

It is customary to interpret the response of the system in terms of the displacement x rather than the velocity v.

The time dependence of $F(t)$ in (4.17) is arbitrary. Because many forces in nature are periodic, we first consider the form

$$\frac{1}{m}F(t) = A_0 \cos \omega t \tag{4.18}$$

where ω is the angular frequency of the driving force.

Problem 4.8. Response of a driven damped linear oscillator

(a) Modify your simple harmonic oscillator program so that an external force of the form (4.18) is included. Add this force to the class that encapsulates the equations of motion without changing the target class. The angular frequency of the driving force should be added as an input parameter.

(b) Choose $\omega_0 = 3$, $\gamma = 0.5$, $\omega = 2$ and the amplitude of the external force $A_0 = 1$ for all runs unless otherwise stated. For these values of ω_0 and γ, the dynamical behavior in the absence of an external force corresponds to an underdamped oscillator. Plot $x(t)$ versus t in the presence of the external force with the initial condition $x(t = 0) = 1, v(t = 0) = 0$. How does the qualitative behavior of $x(t)$ differ from the nonperturbed case? What is the period and angular frequency of $x(t)$ after several oscillations? Repeat the same observations for $x(t)$ with $x(t = 0) = 0, v(t = 0) = 1$. Identify a transient part of $x(t)$ that depends on the initial conditions and decays in time and a steady state part that dominates at longer times and is independent of the initial conditions.

(c) Compute $x(t)$ for several combinations of ω_0 and ω. What is the period and angular frequency of the steady state motion in each case? What parameters determine the frequency of the steady state behavior?

(d) A measure of the long-term behavior of the driven harmonic oscillator is the amplitude of the steady state displacement $A(\omega)$, which can be be computed for a given value of ω if the simulation is run until a steady state has been achieved. One way to determine A is to check the position after every time step to see if a new maximum has been reached as is done by the following code:

```
if (x > Math.abs(amplitude)) {
 amplitude = Math.abs(x);
 control.println("new amplitude = " + amplitude);
}
```

(e) Measure the amplitude and phase shift to verify that the steady state behavior of $x(t)$ is given by

$$x(t) = A(\omega)\cos(\omega t + \delta). \tag{4.19}$$

The quantity δ is the phase difference between the applied force and the steady state motion. Compute $A(\omega)$ and $\delta(\omega)$ for $\omega_0 = 3$, $\gamma = 0.5$, and $\omega = 0$, 1.0, 2.0, 2.2, 2.4, 2.6, 2.8, 3.0, 3.2, and 3.4. Choose the initial condition $x(t = 0) = 0, v(t = 0) = 0$. Repeat the simulation for $\gamma = 3.0$, and plot $A(\omega)$ and $\delta(\omega)$ versus ω for the two values of γ. Discuss the qualitative behavior of $A(\omega)$ and $\delta(\omega)$ for the two values of γ. If $A(\omega)$ has a maximum, determine the angular frequency ω_{max} at which the maximum of A occurs. Is the value of ω_{max} close to the natural angular frequency ω_0? Compare ω_{max} to ω_0 and to the frequency of the damped linear oscillator in the absence of an external force.

(f) Compute $x(t)$ and $A(\omega)$ for a damped linear oscillator with the amplitude of the external force $A_0 = 4$. How do the steady state results for $x(t)$ and $A(\omega)$ compare to the case $A_0 = 1$? Does the transient behavior of $x(t)$ satisfy the same relation as the steady state behavior?

(g) What is the shape of the phase space trajectory for the initial condition $x(t = 0) = 1, v(t = 0) = 0$? Do you find a different phase space trajectory for other initial conditions?

(h) Why is $A(\omega = 0) < A(\omega)$ for small ω? Why does $A(\omega) \to 0$ for $\omega \gg \omega_0$?

(i) Does the mean kinetic energy resonate at the same frequency as does the amplitude? Compute the mean kinetic energy over one cycle once steady state conditions have been reached. Choose $\omega_0 = 3$ and $\gamma = 0.5$. □

In Problem 4.8 we found that the response of the damped harmonic oscillator to an external driving force is linear. For example, if the magnitude of the external force is doubled, then the magnitude of the steady state motion is also doubled. This behavior is a consequence of the linear nature of the equation of motion. When a particle is subject to nonlinear forces, the response can be much more complicated (see Section 6.8).

For many problems, the sinusoidal driving force in (4.18) is not realistic. Another example of an external force can be found by observing someone pushing a child on a swing. Because the force is nonzero for only short intervals of time, this type of force is impulsive. In the following problem, we consider the response of a damped linear oscillator to an impulsive force.

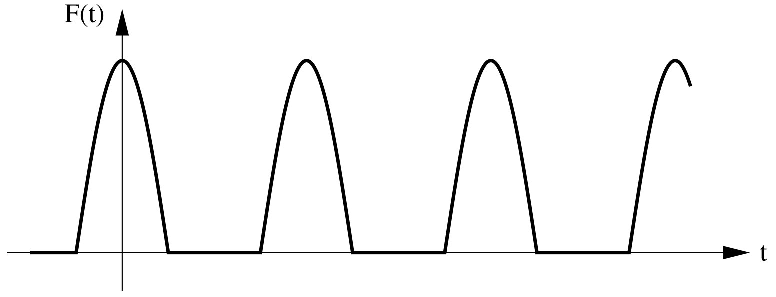


Figure 4.3: A half-wave driving force corresponding to the positive part of a cosine function.

*__Problem 4.9.__ Response of a damped linear oscillator to nonsinusoidal external forces

(a) Assume a swing can be modeled by a dampled linear oscillator. The effect of an impulse is to change the velocity. For simplicity, let the duration of the push equal the time step Δt. Introduce an integer variable for the number of time steps and use the % operator to ensure that the impulse is nonzero only at the time step associated with the period of the external impulse. Determine the steady state amplitude $A(\omega)$ for $\omega = 1.0$, 1.3, 1.4, 1.5, 1.6, 2.5, 3.0, and 3.5. The corresponding period of the impulse is given by $T = 2\pi/\omega$. Choose $\omega_0 = 3$ and $\gamma = 0.5$. Are your results consistent with your experience of pushing a swing and with the comparable results of Problem 4.8?

(b) Consider the response to a half-wave external force consisting of the positive part of a cosine function (see Figure 4.3). Compute $A(\omega)$ for $\omega_0 = 3$ and $\gamma = 0.5$. At what values of ω does $A(\omega)$ have a relative maxima? Is the half-wave cosine driving force equivalent to a sum of cosine functions of different frequencies? For example, does $A(\omega)$ have more than one resonance?

(c) Compute the steady state response $x(t)$ to the external force

$$\frac{1}{m}F(t) = \frac{1}{\pi} + \frac{1}{2}\cos t + \frac{2}{3\pi}\cos 2t - \frac{2}{15\pi}\cos 4t. \tag{4.20}$$

How does a plot of $F(t)$ versus t compare to the half-wave cosine function? Use your results to conjecture a principle of superposition for the solutions to linear equations. □

In many of the problems in this chapter, we have asked you to draw a phase space plot for a single oscillator. This plot provides a convenient representation of both the position and velocity. When we study chaotic phenomena, such plots will become almost indispensable (see Chapter 6). Here we will consider an important feature of phase space trajectories for conservative systems.

If there are no external forces, the undamped simple harmonic oscillator and undamped pendulum are examples of conservative systems; that is, systems for which the total energy is a constant. In Problems 4.10 and 4.11, we will study two general properties of conservative systems, the nonintersecting nature of their trajectories in phase space and the preservation of area in phase space. These concepts will become more important when we study the properties of conservative systems with more than one degree of freedom.

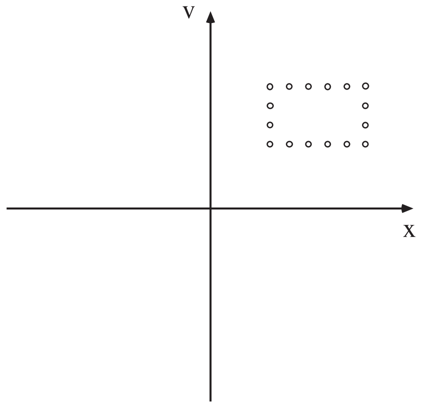


Figure 4.4: What happens to a given area in phase space for conservative systems?

Problem 4.10. Trajectory of a simple harmonic oscillator in phase space

(a) We explore the phase space behavior of a single harmonic oscillator by simulating N initial conditions simultaneously. Write a program to simulate N identical simple harmonic oscillators each of which is represented by a small circle centered at its position and velocity in phase space as shown in Figure 4.4. One way to do so is to adapt the `BouncingBallApp` class introduced in Section 2.6. Choose $N = 16$ and consider random initial positions and velocities. Do the phase space trajectories for different initial conditions ever cross? Explain your answer in terms of the uniqueness of trajectories in a deterministic system.

(b) Choose a set of initial conditions that form a rectangle (see Figure 4.4). Does the shape of this area change with time? □What happens to the total area in comparison to the original area?

Problem 4.11. Trajectory of a pendulum in phase space

(a) Modify your program from Problem 4.10 so that the phase space trajectories (ω versus θ) of $N = 16$ pendula with different initial conditions can be compared. Plot several phase space trajectories for different values of the total energy. Are the phase space trajectories closed? Does the shape of the trajectory depend on the total energy?

(b) Choose a set of initial conditions that form a rectangle in phase space and plot the state of each pendulum as a circle. Does the shape of this area change with time? What happens to the total area? □

4.5 Electrical Circuit Oscillations

In this section we discuss several electrical analogues of the mechanical systems that we have considered. Although the equations of motion are similar in form, it is convenient to consider electrical circuits separately, because the nature of the questions of interest is somewhat different.

The starting point for electrical circuit theory is Kirchhoff's loop rule, which states that the sum of the voltage drops around a closed path of an electrical circuit is zero. This law is a

Element	**Voltage Drop**	**Symbol**	**Units**
resistor	$V_R = IR$	resistance R	ohms (Ω)
capacitor	$V_C = Q/C$	capacitance C	farads (F)
inductor	$V_L = L\,dI/dt$	inductance L	henries (H)

Table 4.1: The voltage drops across the basic electrical circuit elements. Q is the charge (coulombs) on one plate of the capacitor, and I is the current (amperes).

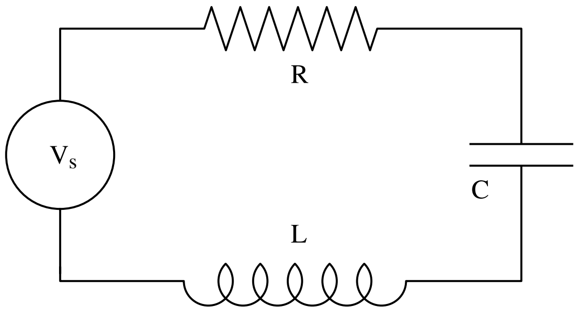


Figure 4.5: A simple series RLC circuit with a voltage source V_s.

consequence of conservation of energy, because a voltage drop represents the amount of energy that is lost or gained when a unit charge passes through a circuit element. The relations for the voltage drops across each circuit element are summarized in Table 4.1.

Imagine an electrical circuit with an alternating voltage source $V_s(t)$ attached in series to a resistor, inductor, and capacitor (see Figure 4.5). The corresponding loop equation is

$$V_L + V_R + V_C = V_s(t). \tag{4.21}$$

The voltage source term V_s in (4.21) is the *emf* and is measured in units of volts. If we substitute the relationships shown in Table 4.1, we find

$$L\frac{d^2Q}{dt^2} + R\frac{dQ}{dt} + \frac{Q}{C} = V_s(t) \tag{4.22}$$

where we have used the definition of current $I = dQ/dt$. We see that (4.22) for the series RLC circuit is identical in form to the damped harmonic oscillator (4.17). The analogies between ideal electrical circuits and mechanical systems are summarized in Table 4.2.

Although we are already familiar with (4.22), we first consider the dynamical behavior of an RC circuit described by

$$RI(t) = R\frac{dQ}{dt} = V_s(t) - \frac{Q}{C}. \tag{4.23}$$

Two RC circuits corresponding to (4.23) are shown in Figure 4.6. Although the loop equation (4.23) is identical regardless of the order of placement of the capacitor and resistor in Figure 4.6, the output voltage measured by the oscilloscope in Figure 4.6 is different. We will see in Problem 4.12 that these circuits act as filters that pass voltage components of certain frequencies while rejecting others.

An advantage of a computer simulation of an electrical circuit is that the measurement of a voltage drop across a circuit element does not affect the properties of the circuit. In fact, digital

Electric Circuit	**Mechanical System**
charge Q	displacement x
current $I = dQ/dt$	velocity $v = dx/dt$
voltage drop	force
inductance L	mass m
inverse capacitance $1/C$	spring constant k
resistance R	damping γ

Table 4.2: Analogies between electrical parameters and mechanical parameters.

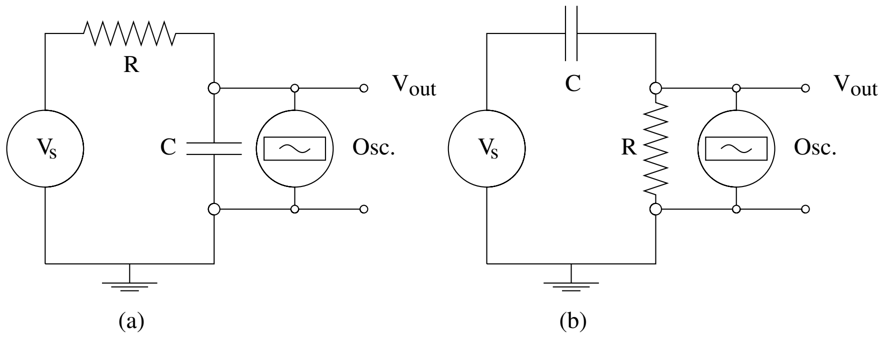


Figure 4.6: Examples of RC circuits used as low and high pass filters. Which circuit is which?

computers are often used to optimize the design of circuits for special applications. The `RCApp` program is not shown here because it is similar to `PendulumApp`, but this program is available in the Chapter 4 package. The `RCApp` program simulates an RC circuit with an alternating current (AC) voltage source of the form $V_s(t) = \cos\omega t$ and plots the time dependence of the charge on the capacitor. You are asked to modify this program in Problem 4.12.

Problem 4.12. Simple filter circuits

(a) Modify the `RCApp` program to simulate the voltages in an RC filter. Your program should plot the voltage across the resistor V_R and the voltage across the source V_s, in addition to the voltage across the capacitor V_C. Run this program with $R = 1000\,\Omega$ and $C = 1.0\,\mu$F (10^{-6} farads). Find the steady state amplitude of the voltage drops across the resistor and across the capacitor as a function of the angular frequency ω of the source voltage $V_s = \cos\omega t$. Consider the frequencies $f = 10, 50, 100, 160, 200, 500, 1000, 5000$, and 10000 Hz. (Remember that $\omega = 2\pi f$.) Choose Δt to be no more than 0.0001 s for $f = 10$ Hz. What is a reasonable value of Δt for $f = 10000$ Hz?

(b) The output voltage depends on where the digital oscilloscope is connected. What is the output voltage of the oscilloscope in Figure 4.6a? Plot the ratio of the amplitude of the output voltage to the amplitude of the input voltage as a function of ω. Use a logarithmic scale for ω. What range of frequencies is passed? Does this circuit act as a high pass or a low pass filter? Answer the same questions for the oscilloscope in Figure 4.6b. Use your results to explain the operation of a high and low pass filter. Compute the value of the cutoff frequency for which the amplitude of the output voltage drops to $1/\sqrt{2}$ (half-power) of the input value. How is the cutoff frequency related to RC?

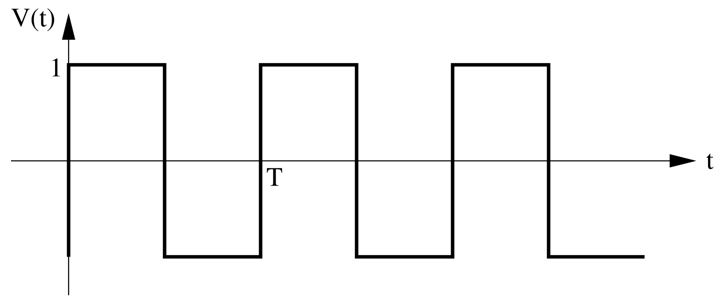


Figure 4.7: Square wave voltage with period T and unit amplitude.

(c) Plot the voltage drops across the capacitor and resistor as a function of time. The phase difference ϕ between each voltage drop and the source voltage can be found by finding the time t_m between the corresponding maxima of the voltages. Because ϕ is usually expressed in radians, we have the relation $\phi/2\pi = t_m/T$, where T is the period of the oscillation. What is the phase difference ϕ_C between the capacitor and the voltage source and the phase difference ϕ_R between the resistor and the voltage source? Do these phase differences depend on ω? Does the current lead or lag the voltage; that is, does the maxima of $V_\text{R}(t)$ come before or after the maxima of $V_s(t)$? What is the phase difference between the capacitor and the resistor? Does the latter difference depend on ω?

(d) Modify your program to find the steady state response of an LR circuit with a source voltage $V_s(t) = \cos \omega t$. Let $R = 100\,\Omega$ and $L = 2 \times 10^{-3}\,$H. Because $L/R = 2 \times 10^{-5}\,$s, it is convenient to measure the time and frequency in units of $T_0 = L/R$. We write $t^* = t/T_0$, $\omega^* = \omega T_0$, and rewrite the equation for an LR circuit as

$$I(t^*) + \frac{dI(t^*)}{dt^*} = \frac{1}{R} \cos \omega^* t^*. \tag{4.24}$$

Because it will be clear from the context, we now simply write t and ω rather than t^* and ω^*. What is a reasonable value of the step size Δt? Compute the steady state amplitude of the voltage drops across the inductor and the resistor for the input frequencies $f = 10$, 20, 30, 35, 50, 100, and 200 Hz. Use these results to explain how an LR circuit can be used as a low pass or a high pass filter. Plot the voltage drops across the inductor and resistor as a function of time and determine the phase differences ϕ_R and ϕ_L between the resistor and the voltage source and the inductor and the voltage source. Do these phase differences depend on ω? Does the current lead or lag the voltage? What is the phase difference between the inductor and the resistor? Does the latter difference depend on ω? □

Problem 4.13. Square wave response of an RC circuit

Modify your program so that the voltage source is a periodic square wave as shown in Figure 4.7. Use a $1.0\,\mu$F capacitor and a $3000\,\Omega$ resistor. Plot the computed voltage drop across the capacitor as a function of time. Make sure the period of the square wave is long enough so that the capacitor is fully charged during one half-cycle. What is the approximate time dependence of $V_\text{C}(t)$ while the capacitor is charging (discharging)? □

We now consider the steady state behavior of the series RLC circuit shown in Figure 4.5 and represented by (4.22). The response of an electrical circuit is the current rather than the charge on the capacitor. Because we have simulated the analogous mechanical system, we already know much about the behavior of driven RLC circuits. Nonetheless, we will find several interesting features of AC electrical circuits in the following two problems.

Problem 4.14. Response of an RLC circuit

(a) Consider an RLC series circuit with $R = 100\,\Omega$, $C = 3.0\,\mu$F, and $L = 2\,$mH. Modify the simple harmonic oscillator program or the RC filter program to simulate an RLC circuit and compute the voltage drops across the three circuit elements. Assume an AC voltage source of the form $V(t) = V_0 \cos\omega t$. Plot the current I as a function of time and determine the maximum steady state current $I_{\rm max}$ for different values of ω. Obtain the *resonance curve* by plotting $I_{\rm max}(\omega)$ as a function of ω and compute the value of ω at which the resonance curve is a maximum. This value of ω is the *resonant frequency*.

(b) The sharpness of the resonance curve of an AC circuit is related to the quality factor or Q value. (Q should not be confused with the charge on the capacitor.) The sharper the resonance, the larger the value of Q. Circuits with high Q (and hence, a sharp resonance) are useful for tuning circuits in a radio so that only one station is heard at a time. We define $Q = \omega_0/\Delta\omega$, where the width $\Delta\omega$ is the frequency interval between points on the resonance curve $I_{\rm max}(\omega)$ that are $\sqrt{2}/2$ of $I_{\rm max}$ at its maximum. Compute Q for the values of R, L, and C given in part (a). Change the value of R by 10% and compute the corresponding percentage change in Q. What is the corresponding change in Q if L or C is changed by 10%?

(c) Compute the time dependence of the voltage drops across each circuit element for approximately fifteen frequencies ranging from 1/10 to 10 times the resonant frequency. Plot the time dependence of the voltage drops.

(d) The ratio of the amplitude of the sinusoidal source voltage to the amplitude of the current is called the *impedance* Z of the circuit; that is, $Z = V_{\rm max}/I_{\rm max}$. This definition of Z is a generalization of the resistance that is defined by the relation $V = IR$ for direct current circuits. Use the plots of part (d) to determine $I_{\rm max}$ and $V_{\rm max}$ for different frequencies and verify that the impedance is given by

$$Z(\omega) = \sqrt{R^2 + (\omega L - 1/\omega C)^2}. \tag{4.25}$$

For what value of ω is Z a minimum? Note that the relation $V = IZ$ holds only for the maximum values of I and V and not for I and V at any time.

(e) Compute the phase difference ϕ_R between the voltage drop across the resistor and the voltage source. Consider $\omega \ll \omega_0$, $\omega = \omega_0$, and $\omega \gg \omega_0$. Does the current lead or lag the voltage in each case; that is, does the current reach a maxima before or after the voltage? Also compute the phase differences ϕ_L and ϕ_C and describe their dependence on ω. Do the relative phase differences between $V_{\rm C}$, $V_{\rm R}$, and $V_{\rm L}$ depend on ω?

(f) Compute the amplitude of the voltage drops across the inductor and the capacitor at the resonant frequency. How do these voltage drops compare to the voltage drop across the resistor and to the source voltage? Also compare the relative phases of $V_{\rm C}$ and $V_{\rm L}$ at resonance. Explain how an RLC circuit can be used to amplify the input voltage. □

4.6 Accuracy and Stability

Now that we have learned how to use numerical methods to find numerical solutions to simple first-order differential equations, we need to develop some practical guidelines to help us estimate the accuracy of the various methods. Because we have replaced a differential equation by a difference equation, our numerical solution is not identically equal to the true solution of the original differential equation, except for special cases. The discrepancy between the two solutions has two causes. One cause is that computers do not store numbers with infinite precision, but rather to a maximum number of digits that is hardware and software dependent. As we have seen, Java allows the programmer to distinguish between *floating point* numbers; that is, numbers with decimal points, and *integer* numbers. Arithmetic with numbers represented by integers is exact, but we cannot solve a differential equation using integer arithmetic. Arithmetic operations involving floating point numbers, such as addition and multiplication, introduce *roundoff error*. For example, if a computer only stored floating point numbers to two significant figures, the product 2.1×3.2 would be stored as 6.7 rather than 6.72. The significance of roundoff errors is that they accumulate as the number of mathematical operations increases. Ideally, we should choose algorithms that do not significantly magnify the roundoff error; for example, we should avoid subtracting numbers that are nearly the same in magnitude.

The other source of the discrepancy between the true answer and the computed answer is the error associated with the choice of algorithm. This error is called the *truncation error*. A truncation error would exist even on an idealized computer that stored floating point numbers with infinite precision and hence had no roundoff error. Because the truncation error depends on the choice of algorithm and can be controlled by the programmer, you should be motivated to learn more about numerical analysis and the estimation of truncation errors. However, there is no general prescription for the best algorithm for obtaining numerical solutions of differential equations. We will find in later chapters that the various algorithms have advantages and disadvantages, and the appropriate selection depends on the nature of the solution, which you might not know in advance, and on your objectives. How accurate must the answer be? Over how large an interval do you need the solution? What kind of computer(s) are you using? How much computer time and personal time do you have?

In practice, we usually can determine the accuracy of a numerical solution by reducing the value of Δt until the numerical solution is unchanged at the desired level of accuracy. Of course, we have to be careful not to make Δt too small, because too many steps would be required and the computation time and roundoff error would increase.

In addition to accuracy, another important consideration is the *stability* of an algorithm. As discussed in Appendix 3A, it might happen that the numerical results are very good for short times, but diverge from the true solution for longer times. This divergence might occur if small errors in the algorithm are multiplied many times, causing the error to grow geometrically. Such an algorithm is said to be *unstable* for the particular problem. We consider the accuracy and the stability of the Euler algorithm in Problems 4.15 and 4.16.

Problem 4.15. Accuracy of the Euler algorithm

(a) Use the Euler algorithm to compute the numerical solution of $dy/dx = 2x$ with $y = 0$ at $x = 0$ and $\Delta x = 0.1$, 0.05, 0.025, 0.01, and 0.005. Make a table showing the difference between the exact solution and the numerical solution. Is the difference between these solutions a decreasing function of Δx? That is, if Δx is decreased by a factor of two, how does the difference change? Plot the difference as a function of Δx. If your points fall approximately on a straight line, then the difference is proportional to Δx (for $\Delta x \ll 1$). The numerical method is called *n*th order if the difference between the analytic solution and the numerical solution is proportional to $(\Delta x)^n$ for a fixed value of x. What is the order of the Euler algorithm?

(b) One way to determine the accuracy of a numerical solution is to repeat the calculation with a smaller step size and compare the results. If the two calculations agree to p decimal places, we can reasonably assume that the results are correct to p decimal places. What value of Δx is necessary for 0.1% accuracy at $x = 2$? What value of Δx is necessary for 0.1% accuracy at $x = 4$? □

Problem 4.16. Stability of the Euler algorithm

(a) Consider the differential equation (4.23) with $Q = 0$ at $t = 0$. This equation represents the charging of a capacitor in an RC circuit with a constant applied voltage V. Choose $R = 2000\,\Omega$, $C = 10^{-6}$ farads, and $V = 10$ volts. Do you expect $Q(t)$ to increase with t? Does $Q(t)$ increase indefinitely, or does it reach a steady-state value? Use a program to solve (4.23) numerically using the Euler algorithm. What value of Δt is necessary to obtain three decimal accuracy at $t = 0.005$?

(b) What is the nature of your numerical solution to (4.23) at $t = 0.05$ for $\Delta t = 0.005$, 0.0025, and 0.001? Does a small change in Δt lead to a large change in the computed value of Q? Is the Euler algorithm stable for reasonable values of Δt? □

4.7 Projects

Project 4.17. Chemical oscillations

The kinetics of chemical reactions can be modeled by a system of coupled first-order differential equations. As an example, consider the following reaction:

$$A + 2B \rightarrow 3B + C \tag{4.26}$$

where A, B, and C represent the concentrations of three different types of molecules. The corresponding rate equations for this reaction are

$$\frac{dA}{dt} = -kAB^2 \tag{4.27a}$$

$$\frac{dB}{dt} = kAB^2 \tag{4.27b}$$

$$\frac{dC}{dt} = kAB^2. \tag{4.27c}$$

The rate at which the reaction proceeds is determined by the reaction constant k. The terms on the right-hand side of (4.27) are positive if the concentration of the molecule increases in (4.26) as it does for B and C, and negative if the concentration decreases as it does for A. Note that the term $2B$ in the reaction (4.26) appears as B^2 in the rate equation (4.27). In (4.27) we have assumed that the reactants are well stirred so that there are no spatial inhomogeneities. In Section 7.8 we will discuss the effects of spatial inhomogeneities due to molecular diffusion.

Most chemical reactions proceed to equilibrium, where the mean concentrations of all molecules are constant. However, if the concentrations of some molecules are replenished, it is possible to observe oscillations and chaotic behavior (see Chapter 6). To obtain oscillations, it is essential to have a series of chemical reactions such that the products of some reactions are the reactants of others. In the following, we consider a simple set of reactions that can lead to oscillations under certain conditions (see Lefever and Nicolis):

$$A \to X \tag{4.28a}$$
$$B + X \to Y + D \tag{4.28b}$$
$$2X + Y \to 3X \tag{4.28c}$$
$$X \to C. \tag{4.28d}$$

If we assume that the reverse reactions are negligible and A and B are held constant by an external source, the corresponding rate equations are

$$\frac{dX}{dt} = A - (B+1)X + X^2Y \tag{4.29a}$$
$$\frac{dY}{dt} = BX - X^2Y. \tag{4.29b}$$

For simplicity, we have chosen the rate constants to be unity.

(a) The steady state solution of (4.29) can be found by setting dX/dt and dY/dt equal to zero. Show that the steady state values for (X, Y) are $(A, B/A)$.

(b) Write a program to solve numerically the rate equations given by (4.29). Your program should input the initial values of X and Y and the fixed concentrations A and B, and plot X versus Y as the reactions evolve.

(c) Systematically vary the initial values of X and Y for given values of A and B. Are their steady state behaviors independent of the initial conditions?

(d) Let the initial value of (X, Y) equal $(A + 0.001, B/A)$ for several different values of A and B, that is, choose initial values close to the steady state values. Classify which initial values result in steady state behavior (stable) and which ones show periodic behavior (unstable). Find the relation between A and B that separates the two types of behavior. □

Project 4.18. Nerve impulses

In 1952 Hodgkin and Huxley developed a model of nerve impulses to understand the nerve membrane potential of a giant squid nerve cell. The equations they developed are known as the Hodgkin-Huxley equations. The idea is that a membrane can be treated as a capacitor where $CV = q$, and thus the time rate of change of the membrane potential V is proportional to the current dq/dt flowing through the membrane. This current is due to the pumping of sodium and potassium ions through the membrane, a leakage current, and an external current stimulus. The model is capable of producing single nerve impulses, trains of nerve impulses, and other effects. The model is described by the following first-order differential equations:

$$C\frac{dV}{dt} = -g_K n^4(V - V_K) - g_{Na}m^3h(V - V_{Na}) - g_L(V - V_L) + I_{\text{ext}}(t) \tag{4.30a}$$
$$\frac{dn}{dt} = \alpha_n(1-n) - \beta_n n \tag{4.30b}$$
$$\frac{dm}{dt} = \alpha_m(1-m) - \beta_m m \tag{4.30c}$$
$$\frac{dh}{dt} = \alpha_h(1-h) - \beta_h h \tag{4.30d}$$

where V is the membrane potential in millivolts (mV), n, m, and h are time dependent functions that describe the gates that pump ions into or out of the cell, C is the membrane capacitance per unit area, the g_i are the conductances per unit area for potassium, sodium, and the leakage current, V_i are the equilibrium potentials for each of the currents, and α_j and β_j are nonlinear functions of V. We use the notation n, m, and h for the gate functions because the notation is universally used in the literature. These gate functions are empirical attempts to describe how the membrane controls the flow of ions into and out of the nerve cell. Hodgkin and Huxley found the following empirical forms for α_j and β_j:

$$\alpha_n = 0.01(V+10)/[e^{(1+V/10)}-1] \tag{4.31a}$$
$$\beta_n = 0.125\,e^{V/80} \tag{4.31b}$$
$$\alpha_m = 0.1(V+25)/[e^{(2.5+V/10)}-1] \tag{4.31c}$$
$$\beta_m = 4\,e^{V/18} \tag{4.31d}$$
$$\alpha_h = 0.07\,e^{V/20} \tag{4.31e}$$
$$\beta_h = 1/[e^{(3+V/10)}+1]. \tag{4.31f}$$

The values of the parameters are $C = 1.0\,\mu\text{F/cm}^2$, $g_K = 36\,\text{mmho/cm}^2$, $g_{Na} = 120\,\text{mmho/cm}^2$, $g_L = 0.3\,\text{mmho/cm}^2$, $V_K = 12\,\text{mV}$, $V_{Na} = -115\,\text{mV}$, and $V_L = 10.6\,\text{mV}$. The unit mho represents ohm^{-1}, and the unit of time is milliseconds (ms). These parameters assume that the resting potential of the nerve cell is zero; however, we now know that the resting potential is about $-70\,\text{mV}$.

We can use the ODE solver to solve (4.30) with the state vector $\{V, n, m, h, t\}$; the rates are given by the right-hand side of (4.30). The following questions ask you to explore the properties of the model.

(a) Write a program to plot n, m, and h as a function of V in the steady state (for which $\dot{n} = \dot{m} = \dot{h} = 0$). Describe how these gates are operating.

(b) Write a program to simulate the nerve cell membrane potential and plot $V(t)$. You can use a simple Euler algorithm with a time step of 0.01 ms. Describe the behavior of the potential when the external current is 0.

(c) Consider a current that is zero except for a one millisecond interval. Try a current spike amplitude of $7\,\mu$A (that is, the external current equals 7 in our units). Describe the resulting nerve impulse $V(t)$. Is there a threshold value for the current below which there is no large spike but only a broad peak?

(d) A constant current should produce a train of spikes. Try different amplitudes for the current and determine if there is a threshold current and how the spacing between spikes depends on the amplitude of the external current.

(e) Consider a situation where there is a steady external current I_1 for 20 ms and then the current increases to $I_2 = I_1 + \Delta I$. There are three types of behavior depending on I_2 and ΔI. Describe the behavior for the following four situations: (1) $I_1 = 2.0\,\mu$A, $\Delta I = 1.5\,\mu$A; (2) $I_1 = 2.0\,\mu$A, $\Delta I = 5.0\,\mu$A; (3) $I_1 = 7.0\,\mu$A, $\Delta I = 1.0\,\mu$A; and (4) $I_1 = 7.0\,\mu$A, $\Delta I = 4.0\,\mu$A. Try other values of I_1 and ΔI as well. In which cases do you obtain a steady spike train? Which cases produce a single spike? What other behavior do you find?

(f) Once a spike is triggered, it is frequently difficult to trigger another spike. Consider a current pulse at $t = 20$ ms of 7 μA that lasts for one millisecond. Then give a second current pulse of the same amplitude and duration at $t = 25$ ms. What happens? What happens if you add a third pulse at 30 ms? □

References and Suggestions for Further Reading

F. S. Acton, *Numerical Methods That Work* (The Mathematical Association of America, 1999), Chapter 5.

G. L. Baker and J. P. Gollub, *Chaotic Dynamics: An Introduction*, 2nd ed. (Cambridge University Press, 1996). A good introduction to the notion of phase space.

Eugene I. Butikov, "Square-wave excitation of a linear oscillator," Am. J. Phys. **72**, 469–476 (2004).

A. Douglas Davis, *Classical Mechanics* (Saunders College Publishing, 1986). The author gives simple numerical solutions of Newton's equations of motion. Much emphasis is given to the harmonic oscillator problem.

S. Eubank, W. Miner, T. Tajima, and J. Wiley, "Interactive computer simulation and analysis of Newtonian dynamics," Am. J. Phys. **57**, 457–463 (1989).

Richard P. Feynman, Robert B. Leighton, and Matthew Sands, *The Feynman Lectures on Physics*, Vol. 1 (Addison–Wesley, 1963). Chapters 21 and 23–25 are devoted to various aspects of harmonic motion.

A. P. French, *Newtonian Mechanics* (W. W. Norton & Company, 1971). An introductory level text with a good discussion of oscillatory motion.

M. Gitterman, "Classical harmonic oscillator with multiplicative noise," Physica A **352**, 309–334 (2005). The analysis is analytical and at the graduate level. However, it would be straightforward to reproduce most of the results after you learn about random processes in Chapter 7.

A. L. Hodgkin and A. F. Huxley, "A quantitative description of ion currents and its applications to conduction and excitation in nerve membranes," J. Physiol. (Lond.) **117**, 500–544 (1952).

Charles Kittel, Walter D. Knight, and Malvin A. Ruderman, *Mechanics*, 2nd ed., revised by A. Carl Helmholz and Burton J. Moyer (McGraw–Hill, 1973).

R. Lefever and G. Nicolis, "Chemical instabilities and sustained oscillations," J. Theor. Biol. **30**, 267 (1971).

Jerry B. Marion and Stephen T. Thornton, *Classical Dynamics*, 5th ed. (Harcourt, 2004). Excellent discussion of linear and nonlinear oscillators.

M. F. McInerney, "Computer-aided experiments with the damped harmonic oscillator," Am. J. Phys. **53**, 991–996 (1985).

William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery, *Numerical Recipes*, 2nd ed. (Cambridge University Press, 1992). Chapter 16 discusses the integration of ordinary differential equations.

Scott Hamilton, *An Analog Electronics Companion* (Cambridge University Press, 2003). A good discussion of the physics and mathematics of basic circuit design including an extensive introduction to circuit simulation using the PSpice simulation program.

S. C. Zilio, "Measurement and analysis of large-angle pendulum motion," Am. J. Phys. **50**, 450–452 (1982).

Chapter 5

Few-Body Problems: The Motion of the Planets

We apply Newton's laws of motion to planetary motion and other systems of a few particles and explore some of the counterintuitive consequences of Newton's laws.

5.1 Planetary Motion

Planetary motion is of special significance because it played an important role in the conceptual history of the mechanical view of the universe. Few theories have affected Western civilization as much as Newton's laws of motion and the law of gravitation, which together relate the motion of the heavens to the motion of terrestrial bodies.

Much of our knowledge of planetary motion is summarized by Kepler's three laws, which can be stated as

1. Each planet moves in an elliptical orbit with the sun located at one of the foci of the ellipse.
2. The speed of a planet increases as its distance from the sun decreases such that the line from the sun to the planet sweeps out equal areas in equal times.
3. The ratio T^2/a^3 is the same for all planets that orbit the sun, where T is the period of the planet and a is the semimajor axis of the ellipse.

Kepler obtained these laws by a careful analysis of the observational data collected over many years by Tycho Brahe.

Kepler's first and third laws describe the shape of the orbit rather than the time dependence of the position and velocity of a planet. Because it is not possible to obtain this time dependence in terms of elementary functions, we will obtain the numerical solution of the equations of motion of planets and satellites in orbit. In addition, we will consider the effects of perturbing forces on the orbit and problems that challenge our intuitive understanding of Newton's laws of motion.

5.2 The Equations of Motion

The motion of the Sun and Earth is an example of a *two-body problem.* We can reduce this problem to a one-body problem in one of two ways. The easiest way is to use the fact that the mass of the Sun is much greater than the mass of the Earth. Hence we can assume that, to a good approximation, the Sun is stationary and is a convenient choice of the origin of our coordinate system.

If you are familiar with the concept of a *reduced mass*, you know that the reduction to a one-body problem is more general. That is, the motion of two objects of mass m and M, whose total potential energy is a function only of their relative separation, can be reduced to an equivalent one-body problem for the motion of an object of reduced mass μ given by

$$\mu = \frac{Mm}{m+M}. \tag{5.1}$$

Because the mass of the Earth, $m = 5.99 \times 10^{24}$ kg, is so much smaller than the mass of the Sun, $M = 1.99 \times 10^{30}$ kg, we find that for most practical purposes, the reduced mass of the Sun and the Earth is that of the Earth alone. In the following, we consider the problem of a single particle of mass m moving about a fixed center of force, which we take as the origin of the coordinate system.

Newton's universal law of gravitation states that a particle of mass M attracts another particle of mass m with a force given by

$$\mathbf{F} = -\frac{GMm}{r^2}\hat{\mathbf{r}} = -\frac{GMm}{r^3}\mathbf{r} \tag{5.2}$$

where the vector $\mathbf{r}$ is directed from M to m (see Figure 5.1). The negative sign in (5.2) implies that the gravitational force is attractive; that is, it tends to decrease the separation r. The gravitational constant G is determined experimentally to be

$$G = 6.67 \times 10^{-11}\,\frac{\mathrm{m}^3}{\mathrm{kg}\cdot\mathrm{s}^2}. \tag{5.3}$$

The force law (5.2) applies to the motion of the center of mass for objects of negligible spatial extent. Newton delayed publication of his law of gravitation for twenty years while he invented integral calculus and showed that (5.2) also applies to any uniform sphere or spherical shell of matter if the distance r is measured from the center of each mass.

The gravitational force has two general properties: its magnitude depends only on the separation of the particles, and its direction is along the line joining the particles. Such a force is called a central force. The assumption of a central force implies that the orbit of the Earth is restricted to a plane (x-y), and the angular momentum $\mathbf{L}$ is conserved and lies in the third (z) direction. We write L_z in the form

$$L_z = (\mathbf{r} \times m\mathbf{v})_z = m(xv_y - yv_x) \tag{5.4}$$

where we have used the cross-product definition $\mathbf{L} = \mathbf{r} \times \mathbf{p}$ and $\mathbf{p} = m\mathbf{v}$. An additional constraint on the motion is that the total energy E is conserved and is given by

$$E = \frac{1}{2}mv^2 - \frac{GMm}{r}. \tag{5.5}$$

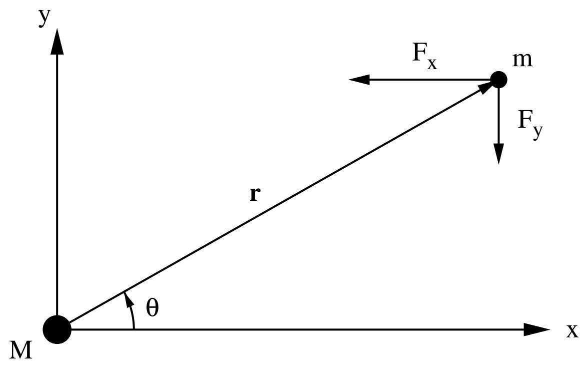


Figure 5.1: An object of mass m moves under the influence of a central force F. Note that $\cos\theta = x/r$ and $\sin\theta = y/r$, which provide useful relations for writing the equations of motion in component form suitable for numerical solutions.

If we fix the coordinate system at the mass M, the equation of motion of the particle of mass m is

$$m\frac{d^2\mathbf{r}}{dt^2} = -\frac{GMm}{r^3}\mathbf{r}. \tag{5.6}$$

It is convenient to write the force in Cartesian coordinates (see Figure 5.1):

$$F_x = -\frac{GMm}{r^2}\cos\theta = -\frac{GMm}{r^3}x \tag{5.7a}$$

$$F_y = -\frac{GMm}{r^2}\sin\theta = -\frac{GMm}{r^3}y. \tag{5.7b}$$

Hence, the equations of motion in Cartesian coordinates are

$$\frac{d^2x}{dt^2} = -\frac{GM}{r^3}x \tag{5.8a}$$

$$\frac{d^2y}{dt^2} = -\frac{GM}{r^3}y \tag{5.8b}$$

where $r^2 = x^2 + y^2$. Equations (5.8a) and (5.8b) are examples of coupled differential equations because each equation contains both x and y.

5.3 Circular and Elliptical Orbits

Because many planetary orbits are nearly circular, it is useful to obtain the condition for a circular orbit. The magnitude of the acceleration a is related to the radius r of the circular orbit by

$$a = \frac{v^2}{r} \tag{5.9}$$

where v is the speed of the object. The acceleration is always directed toward the center and is due to the gravitational force. Hence, we have

$$\frac{mv^2}{r} = \frac{GMm}{r^2} \tag{5.10}$$

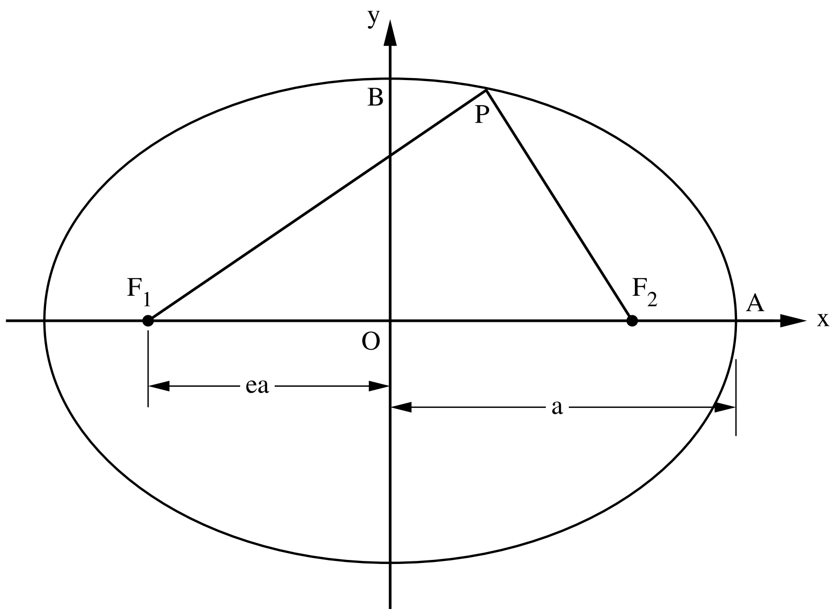


Figure 5.2: The characterization of an ellipse in terms of the semimajor axis a and the eccentricity e. The semiminor axis b is the distance OB. The origin O in Cartesian coordinates is at the center of the ellipse.

and

$$v = \left(\frac{GM}{r}\right)^{1/2}. \tag{5.11}$$

The relation (5.11) between the radius and the speed is the general condition for a circular orbit.

We can also find the dependence of the period T on the radius of a circular orbit using the relation,

$$T = \frac{2\pi r}{v} \tag{5.12}$$

in combination with (5.11) to obtain

$$T^2 = \frac{4\pi^2}{GM} r^3. \tag{5.13}$$

The relation (5.13) is a special case of Kepler's third law with the radius r corresponding to the semimajor axis of an ellipse.

A simple geometrical characterization of an elliptical orbit is shown in Figure 5.2. The two *foci* of an ellipse, F_1 and F_2, have the property that for any point P, the distance $F_1P + F_2P$ is a constant. In general, an ellipse has two perpendicular axes of unequal length. The longer axis is the major axis; half of this axis is the semimajor axis a. The shorter axis is the minor axis; the semiminor axis b is half of this distance. It is common to specify an elliptical orbit by a and by the eccentricity e, where e is the ratio of the distance between the foci to the length of the major axis. Because $F_1P + F_2P = 2a$, it is easy to show that

$$e = \sqrt{1 - \frac{b^2}{a^2}} \tag{5.14}$$

with $0 < e < 1$. (Choose the point P at $x = 0, y = b$.) A special case is $b = a$, for which the ellipse reduces to a circle and $e = 0$.

5.4 Astronomical Units

It is convenient to choose a system of units in which the magnitude of the product GM is not too large and not too small. To describe the Earth's orbit, the convention is to choose the length of the Earth's semimajor axis as the unit of length. This unit of length is called the *astronomical unit* (AU) and is

$$1\,\text{AU} = 1.496 \times 10^{11}\,\text{m}. \tag{5.15}$$

The unit of time is assumed to be one year or 3.15×10^7 s. In these units, the period of the Earth is $T = 1$ years and its semimajor axis is $a = 1$ AU. Hence, from (5.13)

$$GM = \frac{4\pi^2 a^3}{T^2} = 4\pi^2 \text{AU}^3/\text{years}^2 \qquad \text{(astronomical units)}. \tag{5.16}$$

As an example of the use of astronomical units, a program distance of 1.5 would correspond to $1.5 \times (1.496 \times 10^{11}) = 2.244 \times 10^{11}$ m.

5.5 Log-log and Semilog Plots

The values of T and a for our solar system are given in Table 5.1. We first analyze these values and determine if T and a satisfy a simple mathematical relationship.

Suppose we wish to determine whether two variables y and x satisfy a functional relationship, $y = f(x)$. To simplify the analysis, we ignore possible errors in the measurements of y and x. The simplest relation between y and x is linear; that is, $y = mx + b$. The existence of such a relation can be seen by plotting y versus x and finding if the plot is linear. From Table 5.1 we see that T is not a linear function of a. For example, an increase in T from 0.24 to 1, an increase of approximately 4, yields an increase in a of approximately 2.5.

For many problems, it is reasonable to assume an exponential relation

$$y = C\,e^{rx} \tag{5.17}$$

or a power law relation

$$y = C\,x^n \tag{5.18}$$

where C, r, and n are unknown parameters.

If we assume the exponential form (5.17), we can take the natural logarithm of both sides to find

$$\ln y = \ln C + rx. \tag{5.19}$$

Hence, if (5.17) is applicable, a plot of $\ln y$ versus x would yield a straight line with slope r and intercept $\ln C$.

The natural logarithm of both sides of the power law relation (5.18) yields

$$\ln y = \ln C + n \ln x. \tag{5.20}$$

If (5.18) applies, a plot of $\ln y$ versus $\ln x$ yields the exponent n (the slope), which is the usual quantity of physical interest if a power law dependence holds.

Planet	**T (Earth years)**	**a (AU)**
Mercury	0.241	0.387
Venus	0.615	0.723
Earth	1.0	1.0
Mars	1.88	1.523
Jupiter	11.86	5.202
Saturn	29.5	9.539
Uranus	84.0	19.18
Neptune	165	30.06
Pluto	248	39.44

Table 5.1: The period T and semimajor axis a of the planets. The unit of length is the astronomical unit (AU). The unit of time is one (Earth) year.

We illustrate a simple analysis of the data in Table 5.1. Because we expect that the relation between T and a has the power law form $T = Ca^n$, we plot $\ln T$ versus $\ln a$ (see Figure 5.3). A visual inspection of the plot indicates that a linear relationship between $\ln T$ and $\ln a$ is reasonable and that the slope is approximately 1.50 in agreement with Kepler's second law. In Chapter 7 we will discuss the least squares method for fitting a straight line through a number of data points. With a little practice, you can do a visual analysis that is nearly as good.

The `PlotFrame` class contains the axes and titles needed to produce linear, log-log, and semilog plots. It also contains the methods needed to display data in a table format. This table can be displayed programmatically or by right-clicking (control-clicking) at runtime. Listing 5.1 shows a short program that produces the log-log plot of the semimajor axis of the planets versus the orbital period. The arrays a and T contain the semimajor axis of the planets and their periods, respectively. Setting the log scale option causes the `PlotFrame` to transform the data as it is being plotted and causes the axis to change how labels are rendered. Note that the plot automatically adjusts itself to fit the data because the autoscale option is true by default. Also the grid and the tick-labels change as the window is resized.

Listing 5.1: A simple program that producs a log-log plot to demonstrate Kepler's second law.

```
package org.opensourcephysics.sip.ch05;
import org.opensourcephysics.frames.PlotFrame;

public class SecondLawPlotApp {
   public static void main(String[] args) {
      PlotFrame frame = new PlotFrame("ln(a)", "ln(T)",
                              "Kepler's second law");
      frame.setLogScale(true, true);
      frame.setConnected(false);
      double[] period = {
         0.241, 0.615, 1.0, 1.88, 11.86, 29.50, 84.0, 165, 248
      };
      double[] a = {
         0.387, 0.723, 1.0, 1.523, 5.202, 9.539, 19.18, 30.06, 39.44
      };
      frame.append(0, a, period);
      frame.setVisible(true);
      // defines titles of table columns
      frame.setXYColumnNames(0, "T (years)", "a (AU)");
```

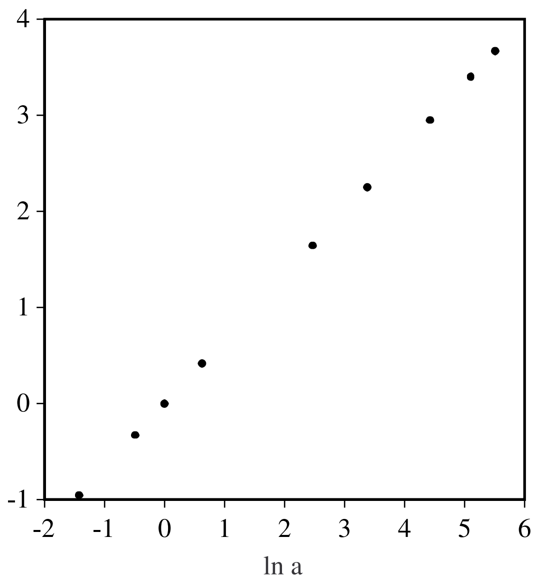


Figure 5.3: Plot of $\ln T$ versus $\ln a$ using the data in Table 5.1. Verify that the slope is 1.50.

x	$y_1(x)$	$y_2(x)$	$y_3(x)$
0	0.00	0.00	2.00
0.5	0.75	1.59	5.44
1.0	3.00	2.00	14.78
1.5	6.75	2.29	40.17
2.0	12.00	2.52	109.20
2.5	18.75	2.71	296.83

Table 5.2: Determine the functional forms of $y(x)$ for the three sets of data. There are no measurement errors, but there are roundoff errors.

```
        // shows data table; can also be done from frame menu
        frame.showDataTable(true);
        frame.setDefaultCloseOperation(javax.swing.JFrame.EXIT_ON_CLOSE);
    }
}
```

Exercise 5.1. Simple functional forms

(a) Run SecondLawPlotApp and convince yourself that you understand the syntax.

(b) Modify SecondLawPlotApp so that the three sets of data shown in Table 5.2 are plotted. Generate linear, semilog, and log-log plots to determine the functional form of $y(x)$ that best fits each data set. □

5.6 Simulation of the Orbit

We now develop a program to simulate the Earth's orbit about the Sun. The `PlanetApp` class shown in Listing 5.2 organizes the startup process and creates the visualization. Because this class extends `AbstractSimulation`, it is sufficient to know that the superclass invokes the `doStep` method periodically when the thread is running or once each time the Step button is clicked. The preferred scale and the aspect ratio for the plot frame are set in the constructor. The statement `frame.setSquareAspect(true)` ensures that a unit of distance will equal the same number of pixels in both the horizontal and vertical directions; the statement `planet.initialize(new double[]{x, vx, y, vy, 0})` in the `initialize` method is used to create an array on the fly as the argument to another method.

Listing 5.2: PlanetApp.

```java
package org.opensourcephysics.sip.ch05;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.frames.*;

public class PlanetApp extends AbstractSimulation {
   PlotFrame frame = new PlotFrame("x (AU)", "y (AU)",
                                   "Planet Simulation");
   Planet planet = new Planet();

   public PlanetApp() {
      frame.addDrawable(planet);
      frame.setPreferredMinMax(-5, 5, -5, 5);
      frame.setSquareAspect(true);
   }

   public void doStep() {
      for(int i = 0;i<5;i++) { // do 5 steps between screen draws
         planet.doStep();      // advances time
      }
      frame.setMessage("t = "+decimalFormat.format(planet.state[4]));
   }

   public void initialize() {
      planet.odeSolver.setStepSize(control.getDouble("dt"));
      double x = control.getDouble("x");
      double vx = control.getDouble("vx");
      double y = control.getDouble("y");
      double vy = control.getDouble("vy");
      // create an array on the fly as the argument to another method
      planet.initialize(new double[] {x, vx, y, vy, 0});
      frame.setMessage("t = 0");
   }

   public void reset() {
      control.setValue("x", 1);
      control.setValue("vx", 0);
      control.setValue("y", 0);
      control.setValue("vy", 6.28);
      control.setValue("dt", 0.01);
      initialize();
   }

   public static void main(String[] args) {
      SimulationControl.createApp(new PlanetApp());
   }
}
```

The Planet class in Listing 5.3 defines the physics and instantiates the numerical method. The latter is the Euler algorithm, which will be replaced in Problem 5.2. Note how the argument to the initialize method is used. The System.arraycopy(array1,index1,array2,index2,length method in the core Java API copies blocks of memory, such as arrays, and is optimized for particular operating systems. This method copies length elements of array1 starting at index1 into array2 starting at index2. In most applications index1 and index2 will be set equal to 0.

Listing 5.3: Class that models the rate equation for a planet acted on by an inverse square law force.

```
package org.opensourcephysics.sip.ch05;
import java.awt.*;
import org.opensourcephysics.display.*;
import org.opensourcephysics.numerics.*;

public class Planet implements Drawable, ODE {
   // GM in units of (AU)^3/(yr)^2
   final static double GM = 4*Math.PI*Math.PI;
   Circle circle = new Circle();
   Trail trail = new Trail();
   double[] state = new double[5];     // {x,vx,y,vy,t}
   Euler odeSolver = new Euler(this); // creates numerical method

   public void doStep() {
      odeSolver.step();                      // advances time
      trail.addPoint(state[0], state[2]); // x,y
   }

   void initialize(double[] initState) {
      System.arraycopy(initState, 0, state, 0, initState.length);
      // reinitializes the solver in case the solver accesses data
      // from previous steps
      odeSolver.initialize(odeSolver.getStepSize());
      trail.clear();
   }

   public void getRate(double[] state, double[] rate) {
      // state[]: x, vx, y, vy, t
      double r2 = (state[0]*state[0])+(state[2]*state[2]); // r squared
      double r3 = r2*Math.sqrt(r2);                        // r cubed
      rate[0] = state[1];          // x rate
      rate[1] = (-GM*state[0])/r3; // vx rate
      rate[2] = state[3];          // y rate
      rate[3] = (-GM*state[2])/r3; // vy rate
      rate[4] = 1;                 // time rate
   }
```

```
    public double[] getState() {
       return state;
    }

    public void draw(DrawingPanel panel, Graphics g) {
       circle.setXY(state[0], state[2]);
       circle.draw(panel, g);
       trail.draw(panel, g);
    }
}
```

The `Planet` class implements the `Drawable` interface and defines the `draw` method as described in Section 3.3. In this case we did not use graphics primitives such as `fillOval` to perform the drawing. Instead, the method calls the methods `circle.draw` and `trail.draw` to draw the planet and its trajectory, respectively.

Invoking a method in another object that has the desired functionality is known as *forwarding* or *delegating* the method. One advantage of forwarding is that we can change the implementation of the drawing within the `Planet` class at any time and still be assured that the `planet` object is drawable. We could, for example, replace the circle by an image of the Earth. Note that we have created a composite object by combining the properties of the simpler `circle` and `trace` objects. These techniques of encapsulation and composition are common in object oriented programming.

Problem 5.2. Verification of `Planet` and `PlanetApp` for circular orbits

(a) Verify `Planet` and `PlanetApp` by considering the special case of a circular orbit. For example, choose (in astronomical units) $x(t = 0) = 1$, $y(t = 0) = 0$, and $v_x(t = 0) = 0$. Use the relation (5.11) to find the value of $v_y(t = 0)$ that yields a circular orbit. How small a value of Δt is needed so that a circular orbit is repeated over many periods? Your answer will depend on your choice of differential equation solver. Find the largest value of Δt that yields an orbit that repeats for many revolutions using the Euler, Euler–Cromer, Verlet, and RK4 algorithms. Is it possible to choose a smaller value of Δt, or are some algorithms, such as the Euler method, simply not stable for this dynamical system?

(b) Write a method to compute the total energy [see (5.5)] and compute it at regular intervals as the system evolves. (It is sufficient to calculate the energy per unit mass, E/m.) For a given value of Δt, which algorithm conserves the total energy best? Is it possible to choose a value of Δt that conserves the energy exactly? What is the significance of the negative sign for the total energy?

(c) Write a separate method to determine the numerical value of the period. (See Problem 3.9c for a discussion of a similar condition.) Choose different sets of values of $x(t = 0)$ and $v_y(t = 0)$, consistent with the condition for a circular orbit. For each orbit determine the radius and the period and verify Kepler's third law. □

Problem 5.3. Verification of Kepler's second and third law

(a) Set $y(t = 0) = 0$ and $v_x(t = 0) = 0$ and find by trial and error several values of $x(t = 0)$ and $v_y(t = 0)$ that yield elliptical orbits of a convenient size. Choose a suitable algorithm and plot the speed of the planet as the orbit evolves. Where is the speed a maximum (minimum)?

(b) Use the same initial conditions as in part (a) and compute the total energy, angular momentum, semimajor and semiminor axes, eccentricity, and period for each orbit. Plot your data for the dependence of the period T on the semimajor axis a and verify Kepler's third law. Given the ratio of T^2/a^3 that you found, determine the numerical value of this ratio in SI units for our solar system.

(c) The force center is at $(x, y) = (0, 0)$ and is one focus. Find the second focus by symmetry. Compute the sum of the distances from each point on the orbit to the two foci and verify that the orbit is an ellipse.

(d) According to Kepler's second law, the orbiting object sweeps out equal areas in equal times. If we use an algorithm with a fixed time step Δt, it is sufficient to compute the area of the triangle swept in each time step. This area equals one-half the base of the triangle times its height, or $\frac{1}{2}\Delta t\,(\mathbf{r}\times\mathbf{v}) = \frac{1}{2}\Delta t(xv_y - yv_x)$. Is this area a constant? This constant corresponds to what physical quantity?

(e)* Show that algorithms with a fixed value of Δt break down if the "planet" is too close to the sun. What is the cause of the failure of the method? What advantage might there be to using a variable time step? What are the possible disadvantages? (See Project 5.19 for an example where a variable time step is very useful.) □

Problem 5.4. Noninverse square forces

(a) Consider the dynamical effects of a small change in the attractive inverse-square force law, for example, let the magnitude of the force equal $Cm/r^{2+\delta}$, where $\delta << 1$. For simplicity, take the numerical value of the constant C to be $4\pi^2$ as before. Consider the initial conditions $x(t = 0) = 1$, $y(t = 0) = 0$, $v_x(t = 0) = 0$, and $v_y(t = 0) = 5$. Choose $\delta = 0.05$ and determine the nature of the orbit. Does the orbit of the planet retrace itself? Verify that your result is not due to your choice of Δt. Does the planet spiral away from or toward the sun? The path of the planet can be described as an elliptical orbit that slowly rotates or *precesses* in the same sense as the motion of the planet. A convenient measure of the precession is the angle between successive orientations of the semimajor axis of the ellipse. This angle is the rate of precession per revolution. Estimate the magnitude of this angle for your choice of δ. What is the effect of decreasing the semimajor axis for fixed δ? What is the effect of changing δ for fixed semimajor axis?

(b) Einstein's theory of gravitation (the general theory of relativity) predicts a correction to the force on a planet that varies as $1/r^4$ due to a weak gravitational field. The result is that the equation of motion for the trajectory of a particle can be written as

$$\frac{d^2\mathbf{r}}{dt^2} = -\frac{GM}{r^2}\left[1 + \alpha\left(\frac{GM}{c^2}\right)^2 \frac{1}{r^2}\right]\hat{\mathbf{r}}, \tag{5.21}$$

where the parameter α is dimensionless. Take $GM = 4\pi^2$ and assume $\alpha = 10^{-3}$. Determine the nature of the orbit for this potential. (For our solar system, the constant α is a maximum for the planet Mercury, but is much smaller than 10^{-3}.)

(c) Suppose that the attractive gravitational force law depends on the inverse cube of the distance, Cm/r^3. What are the units of C? For simplicity, take the numerical value of C to be $4\pi^2$. Consider the initial condition $x(t = 0) = 1$, $y(t = 0) = 0$, $v_x(t = 0) = 0$ and determine analytically the value of $v_y(t = 0)$ required for a circular orbit. How small a value of Δt is needed so that the simulation yields a circular orbit over several periods? How does this value of Δt compare with the value needed for the inverse-square force law?

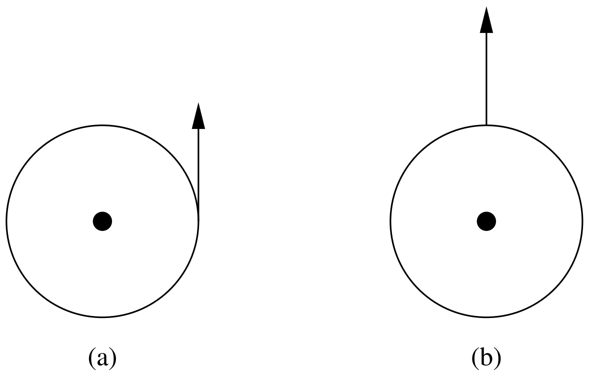


Figure 5.4: (a) An impulse applied in the tangential direction. (b) An impulse applied in the radial direction.

(d) Vary $v_y(t = 0)$ by approximately 2% from the circular orbit condition that you determined in part (c). What is the nature of the new orbit? What is the sign of the total energy? Is the orbit bound? Is it closed? Are all bound orbits closed? □

Problem 5.5. Effect of drag resistance on a satellite orbit

Consider a satellite in orbit about the Earth. In this case it is convenient to measure distances in terms of the radius of the Earth, $R = 6.37 \times 10^6$ m, and the time in terms of hours. Because the force on the satellite is proportional to Gm, where $m = 5.99 \times 10^{24}$ kg is the mass of the Earth, we need to evaluate the product Gm in Earth units (EU). In these units the value of Gm is given by

$$\begin{aligned} Gm &= 6.67 \times 10^{-11} \frac{\mathrm{m}^3}{\mathrm{kg} \cdot \mathrm{s}^2} \Big(\frac{1\,\mathrm{EU}}{6.37 \times 10^6\,\mathrm{m}} \Big)^3 \Big(3.6 \times 10^3\,\mathrm{s/h} \Big)^2 \Big(5.99 \times 10^{24}\,\mathrm{kg} \Big) \\ &= 20.0\,\mathrm{EU}^3/\mathrm{h}^2 \qquad \text{(Earth units).} \end{aligned} \tag{5.22}$$

Modify the `Planet` class to incorporate the effects of drag resistance on the motion of an orbiting Earth satellite. Assume that the drag force is proportional to the square of the speed of the satellite. To be able to observe the effects of air resistance in a reasonable time, take the magnitude of the drag force to be approximately one-tenth of the magnitude of the gravitational force. Choose initial conditions such that a circular orbit would be obtained in the absence of drag resistance and allow at least one revolution before "switching on" the drag resistance. Describe the qualitative change of the orbit due to drag resistance. How does the total energy and the speed of the satellite change with time? □

5.7 Impulsive Forces

What happens to the orbit of an Earth satellite when it is hit by space debris? We now discuss the modifications we need to make in `Planet` and `PlanetApp` so that we can apply an impulsive force (a kick) by a mouse click. If we apply a vertical kick when the position of the satellite is as shown in Figure 5.4a, the impulse would be tangential to the orbit. A radial kick can be applied when the satellite is as shown in Figure 5.4b.

User actions, such as mouse clicks or keyboard entries, are passed from the operating system to Java *event listeners*. Although this standard Java framework is straightforward, we have simplified it to respond to mouse actions within the Open Source Physics panels and frames.[1] In order for an Open Source Physics program to respond to mouse actions, the program implements the `InteractiveMouseHandler` interface and then registers its ability to process mouse actions with the `PlotFrame`. This procedure is demonstrated in the following test program. You can copy the `handleMouseAction` code into your program and replace the print statements with useful methods. Other mouse actions, such as `MOUSE_CLICKED`, `MOUSE_MOVED`, and `MOUSE_ENTERED` are defined in the `InteractivePanel` class.

Listing 5.4: `InteractiveMouseHandler` interface test program.

```
package org.opensourcephysics.sip.ch05;
import java.awt.event.*;
import javax.swing.*;
import org.opensourcephysics.display.*;
import org.opensourcephysics.frames.*;

public class MouseApp implements InteractiveMouseHandler {
   PlotFrame frame = new PlotFrame("x", "y", "Interactive Handler");

   public MouseApp() {
      frame.setInteractiveMouseHandler(this);
      frame.setVisible(true);
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
   }

   public void handleMouseAction(InteractivePanel panel,
            MouseEvent evt) {
      switch(panel.getMouseAction()) {
      case InteractivePanel.MOUSE_DRAGGED :
         panel.setMessage("Dragged");
         break;
      case InteractivePanel.MOUSE_PRESSED :
         panel.setMessage("Pressed");
         break;
      case InteractivePanel.MOUSE_RELEASED :
         panel.setMessage(null);
         break;
      }
   }

   public static void main(String[] args) {
      new MouseApp();
   }
}
```

The `switch` statement is used in Listing 5.4 instead of a chain of `if` statements. The panel's `getMouseAction` method returns an integer. If this integer matches one of the named constants following the case label, then the statements following that constant are executed until a `break` statement is encountered. If a case does not include a `break`, then the execution continues with

[1] See the *Open Source Physics User's Guide* for an extensive discussion of interactive drawing panels.

the next `case`. The equivalent of the `else` construct in an `if` statement is `default` followed by statements that are executed if none of the explicit cases occur.

We now challenge your intuitive understanding of Newton's laws of motion by considering several perturbations of the motion of an orbiting object. Modify your planet program to simulate the effects of the perturbations in Problem 5.6. In each case answer the questions before doing the simulation.

Problem 5.6. Tangential and radial perturbations

(a) Suppose that a small tangential "kick" or impulsive force is applied to a satellite in a circular orbit about the Earth (see Figure 5.4a.) Choose Earth units so that the numerical value of the product Gm is given by (5.22). Apply the impulsive force by stopping the program after the satellite has made several revolutions and click the mouse to apply the force. Recall that the impulse changes the momentum in the desired direction. In what direction does the orbit change? Is the orbit stable, for example, does a small impulse lead to a small change in the orbit? Does the orbit retrace itself indefinitely if no further perturbations are applied? Describe the shape of the perturbed orbit.

(b) How does the change in the orbit depend on the strength of the kick and its duration?

(c) Determine if the angular momentum and the total energy are changed by the perturbation.

(d) Apply a radial kick to the satellite as in Figure 5.4b and answer the same questions as in parts (a)–(c).

(e) Determine the stability of the inverse-cube force law (see Problem 5.4) to radial and tangential perturbations. □

Mouse actions are not the only possible way to affect the simulation. We can also add *custom buttons* to the control. These buttons are added when the program is instantiated in the `main` method.

```
public static void main(String[] args) {
   // OSPControl is a superclass of SimulationControl
   OSPControl control = SimulationControl.createApp(new PlanetApp());
   control.addButton("doRadialKick", "Kick!", "Perform a radial kick");
}
```

Note that `SimulationControl` (and `CalculationControl`) extend the `OSPControl` superclass and therefore support the `addButton` method where this method is defined. We assign the variable returned by the static `createApp` method to a variable of type `OSPControl` to highlight the object-oriented structure of the Open Source Physics library.

The first parameter in the `addButton` method specifies the method that will be invoked when the button is clicked, the second parameter specifies the text label that will appear on the button, and the third parameter specifies the *tool tip* that will appear when the mouse hovers over the button. Custom buttons can be used for just about anything, but the corresponding method must be defined.

Exercise 5.7. Custom buttons

Use a custom button in Problem 5.6 rather than a mouse click to apply an impulsive force to the planet. □

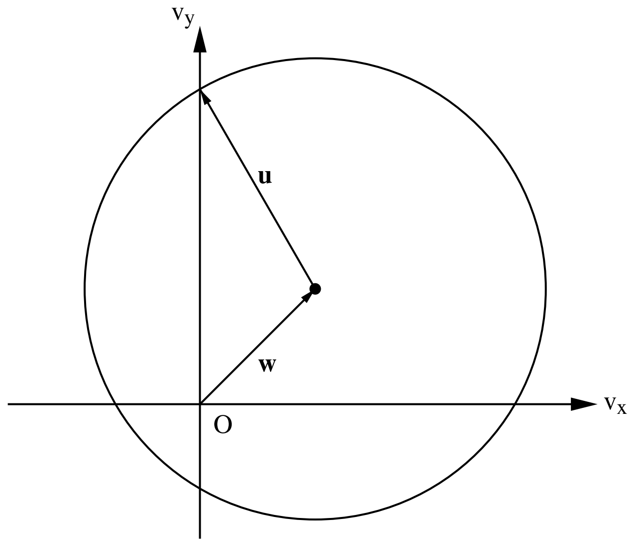


Figure 5.5: The orbit of a particle in velocity space. The vector **w** points from the origin in velocity space to the center of the circular orbit. The vector **u** points from the center of the orbit to the point (v_x, v_y).

5.8 Velocity Space

In Problem 5.6 your intuition might have been incorrect. For example, you might have thought that the orbit would elongate in the direction of the kick. In fact the orbit does elongate but in a direction perpendicular to the kick. Do not worry; you are in good company! Few students have a good qualitative understanding of Newton's law of motion, even after taking an introductory course in physics. A qualitative way of stating Newton's second law is

Forces act on the trajectories of particles by changing velocity, not position.

If we fail to take into account this property of Newton's second law, we will encounter physical situations that appear counterintuitive.

Because force acts to change velocity, it is reasonable to consider both velocity and position on an equal basis. In fact position and momentum are treated in such a manner in advanced formulations of classical mechanics and in quantum mechanics.

In Problem 5.8 we explore some of the properties of orbits in velocity space in the context of the bound motion of a particle in an inverse-square force. Modify your program so that the path in velocity space of the Earth is plotted. That is, plot the point (v_x, v_y) the same way you plotted the point (x, y). The path in velocity space is a series of successive values of the object's velocity vector. If the position space orbit is an ellipse, what is the shape of the orbit in velocity space?

Problem 5.8. Properties of velocity space orbits

(a) Modify your program to display the orbit in position space and in velocity space at the same time. Verify that the velocity space orbit is a circle, even if the orbit in position space is an ellipse. Does the center of this circle coincide with the origin $(v_x, v_y) = (0,0)$ in velocity space? Choose the same initial conditions that you considered in Problems 5.2 and 5.3.

(b)* Let **u** denote the radius vector of a point on the velocity circle and **w** denote the vector from the origin in velocity space to the center of the velocity circle (see Figure 5.5). Then the velocity of the particle can be written as

$$\mathbf{v} = \mathbf{u} + \mathbf{w}. \tag{5.23}$$

Compute **u** and verify that its magnitude is given by

$$u = GMm/L \tag{5.24}$$

where L is the magnitude of the angular momentum. Note that L is proportional to m so that it is not necessary to know the magnitude of m.

(c)* Verify that at each moment in time, the planet's position vector **r** is perpendicular to **u**. Explain why this relation holds. □

Problem 5.9. Effect of impulses in velocity space

How does the velocity space orbit change when an impulsive kick is applied in the tangential or in the radial direction? How do the magnitude and direction of **w** change? From the observed change in the velocity orbit and the above considerations, explain the observed change of the orbit in position space. □

5.9 A Mini-Solar System

So far our study of planetary orbits has been restricted to two-body central forces. However, the solar system is not a two-body system, because the planets exert gravitational forces on one another. Although the interplanetary forces are small in magnitude in comparison to the gravitational force of the sun, they can produce measurable effects. For example, the existence of Neptune was conjectured on the basis of a discrepancy between the experimentally measured orbit of Uranus and the predicted orbit calculated from the known forces.

The presence of other planets implies that the total force on a given planet is not a central force. Furthermore, because the orbits of the planets are not exactly in the same plane, an analysis of the solar system must be extended to three dimensions if accurate calculations are required. However, for simplicity, we will consider a model of a two-dimensional solar system with two planets in orbit about a fixed sun.

The equations of motion of two planets of mass m_1 and mass m_2 can be written in vector form as (see Figure 5.6)

$$m_1 \frac{d^2\mathbf{r}_1}{dt^2} = -\frac{GMm_1}{{r_1}^3}\mathbf{r}_1 + \frac{Gm_1m_2}{{r_{21}}^3}\mathbf{r}_{21} \tag{5.25a}$$

$$m_2 \frac{d^2\mathbf{r}_2}{dt^2} = -\frac{GMm_2}{{r_2}^3}\mathbf{r}_2 - \frac{Gm_1m_2}{{r_{21}}^3}\mathbf{r}_{21} \tag{5.25b}$$

where $\mathbf{r}_1$ and $\mathbf{r}_2$ are directed from the sun to planets 1 and 2 respectively, and $\mathbf{r}_{21} = \mathbf{r}_2 - \mathbf{r}_1$ is the vector from planet 1 to planet 2. It is convenient to divide (5.25a) by m_1 and (5.25b) by m_2 and

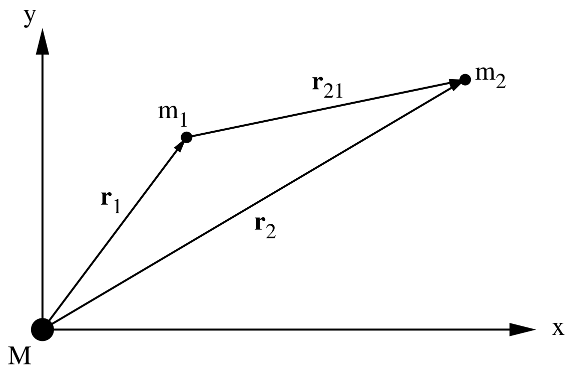


Figure 5.6: The coordinate system used in (5.25). Planets of mass m_1 and m_2 orbit a sun of mass M.

to write the equations of motion as

$$\frac{d^2\mathbf{r}_1}{dt^2} = -\frac{GM}{{r_1}^3}\mathbf{r}_1 + \frac{Gm_2}{{r_{21}}^3}\mathbf{r}_{21} \tag{5.26a}$$

$$\frac{d^2\mathbf{r}_2}{dt^2} = -\frac{GM}{{r_2}^3}\mathbf{r}_2 - \frac{Gm_1}{{r_{21}}^3}\mathbf{r}_{21}. \tag{5.26b}$$

A numerical solution of (5.26) can be obtained by the straightforward extension of the `Planet` class as shown in Listing 5.5. To simplify the drawing of the particle trajectories, the `Planet2` class defines an *inner class*, `Mass`, which extends `Circle` and contains a `Trail`. Whenever a planet moves, a point is added to the trail so that its location and path are shown on the plot. Inner classes are an organizational convenience that save us the trouble of having to create another file, which in this case would be named `Mass.java`. When we compile the `Planet2` class, we will produce a bytecode file named `Planet2$Mass.class` in addition to the file `Planet2.class`. Inner classes are most effective as short helper classes which work in conjuction with the containing class because they have access to all the data (including private variables) in the containing class.

Listing 5.5: A class that implements the rate equation for two interacting planets acted on by an inverse-square law force.

```
package org.opensourcephysics.sip.ch05;
import java.awt.*;
import org.opensourcephysics.display.*;
import org.opensourcephysics.numerics.*;

public class Planet2 implements Drawable, ODE {
   // GM in units of (AU)^3/(yr)^2
   final static double GM = 4*Math.PI*Math.PI;
   final static double GM1 = 0.04*GM;
   final static double GM2 = 0.001*GM;
   double[] state = new double[9];
   ODESolver odeSolver = new RK45MultiStep(this);
   Mass mass1 = new Mass(), mass2 = new Mass();
```

```
public void doStep() {
   odeSolver.step();
   mass1.setXY(state[0], state[2]);
   mass2.setXY(state[4], state[6]);
}

public void draw(DrawingPanel panel, Graphics g) {
   mass1.draw(panel, g);
   mass2.draw(panel, g);
}

void initialize(double[] initState) {
   System.arraycopy(initState, 0, state, 0, initState.length);
   mass1.clear(); // clears data from the old trail
   mass2.clear();
   mass1.setXY(state[0], state[2]);
   mass2.setXY(state[4], state[6]);
}

public void getRate(double[] state, double[] rate) {
   // state[]: x1, vx1, y1, vy1, x2, vx2, y2, vy2, t
   double r1Squared = (state[0]*state[0])+(state[2]*state[2]);
   double r1Cubed = r1Squared*Math.sqrt(r1Squared);
   double r2Squared = (state[4]*state[4])+(state[6]*state[6]);
   double r2Cubed = r2Squared*Math.sqrt(r2Squared);
   double dx = state[4]-state[0];                      // x12 separation
   double dy = state[6]-state[2];                      // y12 separation
   double dr2 = (dx*dx)+(dy*dy);                       // r12 squared
   double dr3 = Math.sqrt(dr2)*dr2;                    // r12 cubed
   rate[0] = state[1];                                 // x1 rate
   rate[2] = state[3];                                 // y1 rate
   rate[4] = state[5];                                 // x2 rate
   rate[6] = state[7];                                 // y2 rate
   rate[1] = ((-GM*state[0])/r1Cubed)+((GM1*dx)/dr3); // vx1 rate
   rate[3] = ((-GM*state[2])/r1Cubed)+((GM1*dy)/dr3); // vy1 rate
   rate[5] = ((-GM*state[4])/r2Cubed)-((GM2*dx)/dr3); // vx2 rate
   rate[7] = ((-GM*state[6])/r2Cubed)-((GM2*dy)/dr3); // vy2 rate
   rate[8] = 1;                                        // time rate
}

public double[] getState() {
   return state;
}

class Mass extends Circle {
   Trail trail = new Trail();

   public void draw(DrawingPanel panel, Graphics g) {
      trail.draw(panel, g);
      super.draw(panel, g);
   }

   void clear() {
```

```
            trail.clear();
        }

        public void setXY(double x, double y) {
            super.setXY(x, y);
            trail.addPoint(x, y);
        }
    }
}
```

The target application, Planet2App, extends AbstractSimulation in the usual way. Because it is almost identical to Listing 5.2, it is not shown here. The complete program is available in the ch05 package.

Problem 5.10. Planetary perturbations

Use Planet2App with the initial conditions given in the program. For illustrative purposes, we have adopted the numerial values $m_1/M = 10^{-3}$ and $m_2/M = 4 \times 10^{-2}$ and hence GM1 = $(m_2/M)GM = 0.04$GM and GM2 = $(m_1/M)GM = 0.001$GM. What would be the shape of the orbits and the periods of the two planets if they did not mutually interact? What is the qualitative effect of their mutual interaction? Describe the shape of the two orbits. Why is one planet affected more by their mutual interaction than the other? Are the angular momentum and the total energy of planet one conserved? Are the total energy and total angular momentum of the two planets conserved? A related but more time consuming problem is given in Project 5.18. □

Problem 5.11. Double stars

Another interesting dynamical system consists of one planet orbiting about two fixed stars of equal mass. In this case there are no closed orbits, but the orbits can be classified as either stable or unstable. Stable orbits may be open loops that encircle both stars, figure eights, or orbits that encircle only one star. Unstable orbits will eventually collide with one of the stars. Modify Planet2 to simulate the double-star system, with the first star located at $(-1,0)$ and the second star of equal mass located at $(1,0)$. Place the planet at $(0.1,1)$ and systematically vary the x and y components of the velocity to obtain different types of orbits. Then try other initial positions. □

5.10 Two-Body Scattering

Much of our understanding of the structure of matter comes from scattering experiments. In this section we explore one of the more difficult concepts in the theory of scattering, the *differential cross section*.

A typical scattering experiment involves a beam with many incident particles all with the same kinetic energy. The coordinate system is shown in Figure 5.7. The incident particles come from the left with an initial velocity $\mathbf{v}$ in the $+x$ direction. We take the center of the beam and the center of the target to be on the x-axis. The *impact parameter b* is the perpendicular distance from the initial trajectory to a parallel line through the center of the target (see Figure 5.7). We assume that the width of the beam is larger than the size of the target. The target contains many scattering centers, but for calculational purposes, we may consider scattering off only one particle if the target is sufficiently thin.

When an incident particle comes close to the target, it is deflected. In a typical experiment, the scattered particles are counted in a detector that is far from the target. The final velocity of the scattered particles is $\mathbf{v}'$, and the angle between $\mathbf{v}$ and $\mathbf{v}'$ is the scattering angle θ.

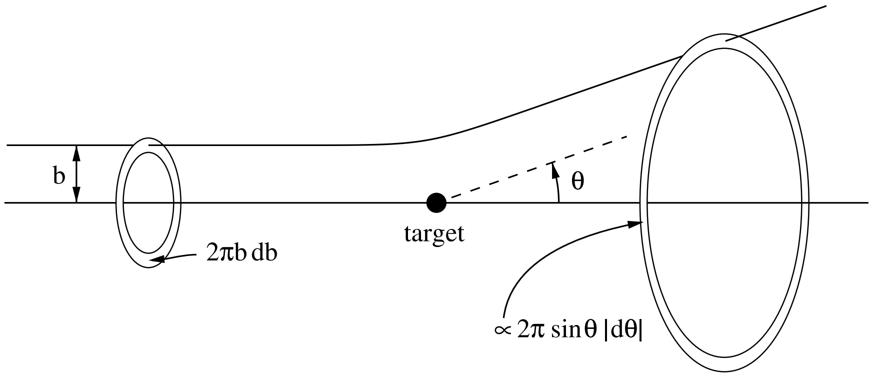


Figure 5.7: The coordinate system used to define the differential scattering cross section. Particles passing through the beam area $2\pi b\,db$ are scattered into the solid angle $d\Omega$.

Let us assume that the scattering is elastic and that the target is much more massive than the beam particles so that the target can be considered to be fixed. (The latter condition can be relaxed by using center of mass coordinates.) We also assume that no incident particle is scattered more than once. These considerations imply that the initial speed and final speed of the incident particles are equal. The functional dependence of θ on b depends on the force on the beam particles due to the target. In a typical experiment, the number of particles in an angular region between θ and $\theta + d\theta$ is detected for many values of θ. These detectors measure the number of particles scattered into the solid angle $d\Omega = \sin\theta\, d\theta\, d\phi$ centered about θ. The *differential cross section* $\sigma(\theta)$ is defined by the relation

$$\frac{dN}{N} = n\sigma(\theta)\, d\Omega \tag{5.27}$$

where dN is the number of particles scattered into the solid angle $d\Omega$ centered about θ and the azimuthal angle ϕ, N is the total number of particles in the beam, and n is the target density defined as the number of targets per unit area.

The interpretation of (5.27) is that the fraction of particles scattered into the solid angle $d\Omega$ is proportional to $d\Omega$ and the density of the target. From (5.27) we see that $\sigma(\theta)$ can be interpreted as the effective area of a target particle for the scattering of an incident particle into the element of solid angle $d\Omega$. Particles that are not scattered are ignored. Another way of thinking about $\sigma(\theta)$ is that it is the ratio of the area $b\,db\,d\phi$ to the solid angle $d\Omega = \sin\theta\, d\theta\, d\phi$, where $b\,db\,d\phi$ is the infinitesimal cross-sectional area of the beam that scatters into the solid angle defined by θ to $\theta + d\theta$ and ϕ to $\phi + d\phi$. The alternative notation for the differential cross section, $d\sigma/d\Omega$, comes from this interpretation.

To do an analytic calculation of $\sigma(\theta)$, we write

$$\sigma(\theta) = \frac{d\sigma}{d\Omega} = \frac{b}{\sin\theta}\left|\frac{db}{d\theta}\right|. \tag{5.28}$$

We see from (5.28) that the analytic calculation of $\sigma(\theta)$ involves b as a function of θ, or more precisely, how b changes to give scattering through an infinitesimally larger angle $\theta + d\theta$.

In a scattering experiment, particles enter from the left (see Figure 5.7) with random values of the impact parameter b and azimuthal angle ϕ, and the number of particles scattered into the various detectors is measured. In our simulation, we know the value of b, and we can integrate

Newton's equations of motion to find the angle at which the incident particle is scattered. Hence, in contrast to the analytic calculation, a simulation naturally yields θ as a function of b.

Because the differential cross section is usually independent of ϕ, we need to consider beam particles only at $\phi = 0$. We have to take into account the fact that in a real beam, there are more particles at some values of b than at others. That is, the number of particles in a real beam is proportional to $2\pi b \Delta b$, the area of the ring between b and $b+\Delta b$, where we have integrated over the values of ϕ to obtain the factor of 2π. Here Δb is the interval between the values of b used in the program. Because there is only one target in the beam, the target density is $n = 1/(\pi R^2)$.

The scattering program requires the `Scatter`, `ScatterAnalysis`, and `ScatterApp` classes. The `ScatterApp` class in Listing 5.6 organizes the startup process and creates the visualizations. As usual, it extends `AbstractSimulation` by overriding the `doStep` method. However, in this case a single step is not a time step. A step calculates a trajectory and scattering angle for the given impact parameter. After a trajectory is calculated, the impact parameter is incremented and the panel is repainted. If necessary, you can eliminate this visualization to increase the computational speed. If the new impact parameter exceeds the beam radius `bmax`, the animation is stopped and the accumulated data is analyzed. Note that the `calculateTrajectory` method returns `true` if the calculation succeeded and that an error message is printed if the calculation fails. Including a failsafe mechanism to stop a computation is good programming practice.

Listing 5.6: A program that calculates the scattering trajectories and computes the differential cross section.

```java
public class ScatterApp extends AbstractSimulation {
  PlotFrame frame = new PlotFrame("x", "y", "Trajectories");
  ScatterAnalysis analysis = new ScatterAnalysis();
  Scatter trajectory = new Scatter();
  double vx;     // speed of the incident particle
  double b, db; // impact parameter and increment
  double bmax;  // maximum impact parameter

  /**
   * Constructs ScatterApp.
   */
  public ScatterApp() {
    frame.setPreferredMinMax(-5, 5, -5, 5);
    frame.setSquareAspect(true);
  }

public void doStep() {
    if(trajectory.calculateTrajectory(frame, b, vx)) {
      analysis.detectParticle(b, trajectory.getAngle());
    } else {
      control.println("Trajectory did not converge at b = "+b);
    }
    frame.setMessage("b = "+decimalFormat.format(b));
    b += db; // increases the impact parameter
    frame.repaint();
    if(b>bmax) {
      control.calculationDone("Maximum impact parameter reached");
      analysis.plotCrossSection(b);
    }
  }
public void initialize() {
    vx = control.getDouble("vx");
    bmax = control.getDouble("bmax");
    db = control.getDouble("db");
    b = db/2; // starts b at average value of first interval 0->db
    // b will increment to 3*db/2, 5*db/2, 7*db/2, ...
    frame.setMessage("b = 0");
    frame.clearDrawables(); // removes old trajectories
    analysis.clear();
  }

public void reset() {
    control.setValue("vx", 3);
    control.setValue("bmax", 0.25);
    control.setValue("db", 0.01);
    initialize();
  }

public static void main(String[] args) {
    SimulationControl.createApp(new ScatterApp());
  }
}
```

The Scatter class shown in Listing 5.7 calculates the trajectories by expressing the equation of motion as a rate equation. The most important method is calculateTrajectory, which calculates a trajectory by stepping the differential equation solver and adding the resulting data to a trail to display the path. Because the beam source is far away, we stop the calculation when the distance of the scattered particle from the target exceeds the initial distance. Note the use of the ternary ?: operator. This very efficient and compact operator uses three expressions. The first expression evaluates to a boolean. If this expression is true, then the statement after the ? is executed. If this expression is false, then the statement after the : is executed. However, because some potentials may trap particles for long periods of time, we also stop the calculation after a predetermined number of time steps.

Listing 5.7: A class that models particle scattering using a central force law.

```
package org.opensourcephysics.sip.ch05;
import java.awt.*;
import org.opensourcephysics.display.*;
import org.opensourcephysics.frames.*;
import org.opensourcephysics.numerics.*;

public class Scatter implements ODE {
  double[] state = new double[5];
  RK4 odeSolver = new RK4(this);

  public Scatter() {
      odeSolver.setStepSize(0.05);
    }

  boolean calculateTrajectory(PlotFrame frame, double b, double vx) {
      state[0] = -5.0; // x
      state[1] = vx;   // vx
      state[2] = b;    // y
      state[3] = 0;     // vy
      state[4] = 0;     // time
      Trail trail = new Trail();
      trail.color = Color.red;
      frame.addDrawable(trail);
      double r2 = (state[0]*state[0])+(state[2]*state[2]);
      double count = 0;
      while((count<=1000)&&((2*r2)>((state[0]*state[0])+(state[2]*state[2])))) {
        trail.addPoint(state[0], state[2]);
        odeSolver.step();
        count++;
      }
      return count<1000;
   }

   private double force(double r) {
     // Coulomb force law
     return(r==0) ? 0 : (1/r/r); // returns 0 if r = 0
   }

   public void getRate(double[] state, double[] rate) {
      double r = Math.sqrt((state[0]*state[0])+(state[2]*state[2]));
      double f = force(r);
      rate[0] = state[1];
      rate[1] = (f*state[0])/r;
      rate[2] = state[3];
      rate[3] = (f*state[2])/r;
      rate[4] = 1;
   }

   public double[] getState() {
      return state;
   }

   double getAngle() {
      return Math.atan2(state[3], state[1]);// /Math.PI;    xx
   }
}
```

The `ScatterAnalysis` class performs the data analysis. This class creates an array of `bins` to sort and accumulate the trajectories according to the scattering angle. The values of the scattering angle between 0° and 180° are divided into bins of width `dtheta`. To compute the number of particles coming from a ring of radius *b*, we accumulate the value of *b* associated with each bin or "detector" and write `bins[index] += b` (see the `detectParticle` method), because the number of particles in a ring of radius *b* is proportional to *b*. The total number of scattered particles is computed in the same way:

```
totalN += b;
```

You might want to increase the number of bins and the range of angles for better resolution.

Listing 5.8: The `ScatterAnalysis` class accumulates the scattering data and plots the differential cross section.

```
public class ScatterAnalysis {
    int numberOfBins = 18;
    PlotFrame frame = new PlotFrame("angle", "sigma",
                        "differential cross section");
    double[] bins = new double[numberOfBins];
    double dtheta = Math.PI/(numberOfBins);
    double totalN = 0;    // total number of scattered particles

  void clear() {
      for(int i = 0;i<numberOfBins;i++) {
        bins[i] = 0;
      }
      totalN = 0;
      frame.clearData();
      frame.repaint();
    }

  void detectParticle(double b, double theta) {
     // treats positive and negative angles equally to get better statistics
      theta = Math.abs(theta);
      int index = (int) (theta/dtheta);
      bins[index] += b;
      totalN += b;
    }

  void plotCrossSection(double radius) {
      double targetDensity = 1/Math.PI/radius/radius;
      double delta = (dtheta*180)/Math.PI; // uses degrees for plot
      frame.clearData();
      for(int i = 0;i<numberOfBins;i++) {
        double domega = 2*Math.PI*Math.sin((i+0.5)*dtheta)*dtheta;
        double sigma = bins[i]/totalN/targetDensity/domega;
        frame.append(0, (i+0.5)*delta, sigma);
      }
      frame.setVisible(true);
    }
  }
```

Problem 5.12. Total cross section

The total cross section σ_T is defined as

$$\sigma_T = \int \sigma(\theta)\, d\Omega. \tag{5.29}$$

Add code to calculate and display the total cross section in the `plotCrossSection` method. Design a test to verify that the ODE solver in the `Scatter` class has sufficient accuracy. □

In Problem 5.13, we consider a model of the hydrogen atom for which the force on a beam particle is zero for $r > a$. Because we do not count the beam particles that are not scattered, we set the beam radius equal to a. For forces that are not identically zero, we need to choose a minimum angle for θ such that particles whose scattering angle is less than this minimum are not counted as scattered (see Problem 5.14).

Problem 5.13. Scattering from a model hydrogen atom

(a) Consider a model of the hydrogen atom for which a positively-charged nucleus of charge $+e$ is surrounded by a uniformly distributed negative charge of equal magnitude. The spherically symmetric negative charge distribution is contained within a sphere of radius a. It is straightforward to show that the force between a positron of charge $+e$ and this model hydrogen atom is given by

$$f(r) = \begin{cases} 1/r^2 - r/a^3 & r \leq a \\ 0 & r > a. \end{cases} \tag{5.30}$$

We have chosen units such that $e^2/(4\pi\epsilon_0) = 1$, and the mass of the positron is unity. What is the ionization energy in these units? Modify the `Scatter` class to incorporate this force. Is the force on the positron from the model hydrogen atom purely repulsive? Choose $a = 1$ and set the beam radius `bmax` = 1. Use $E = 0.125$ and $\Delta t = 0.01$. Compute the trajectories for $b = 0.25$, 0.5, and 0.75 and describe the qualitative nature of the trajectories.

(b) Determine the cross section for $E = 0.125$. Choose nine bins so that the angular width of a detector is `delta` = 20°, and let `db` = 0.1, 0.01, and 0.002. How does the accuracy of your results depend on the number of bins? Determine the differential cross section for different energies and explain its qualitative energy dependence.

(c) What is the value of σ_T for $E = 0.125$? Does σ_T depend on E? The total cross section has units of area, but a point charge does not have an area. To what area does it refer? What would you expect the total cross section to be for scattering from a hard sphere?

(d) Change the sign of the force so that it corresponds to electron scattering. How do the trajectories change? Discuss the change in $\sigma(\theta)$. □

Problem 5.14. Rutherford scattering

(a) One of the most famous scattering experiments was performed by Geiger and Marsden who scattered a beam of alpha particles on a thin gold foil. Based on these experiments, Rutherford deduced that the positive charge of the atom is concentrated in a small region at the center of the atom rather than distributed uniformly over the entire atom. Use a $1/r^2$ force in class `Scatter` and compute the trajectories for $b = 0.25$, 0.5, and 0.75 and describe the trajectories. Choose $E = 5$ and $\Delta t = 0.01$. The default value of x_0, the initial x-coordinate of the beam, is $x_0 = -5$. Is this value reasonable?

(b) For $E = 5$ determine the cross section with `numberOfBins` = 18. Choose the beam width `bmax` = 2. Then vary `db` (or `numberOfBins`) and compare the accuracy of your results to the analytic result for which $\sigma(\theta)$ varies as $[\sin(\theta/2)]^{-4}$. How do your computed results compare with this dependence on θ? If necessary, decrease `db`. Are your results better or worse at small angles, intermediate angles, or large angles near 180°? Explain.

(c) Because the Coulomb force is long range, there is scattering at all impact parameters. Increase the beam radius and determine if your results for $\sigma(\theta)$ change. What happens to the total cross section as you increase the beam width?

(d) Compute $\sigma(\theta)$ for different values of E and estimate the dependence of $\sigma(\theta)$ on E. □

Problem 5.15. Scattering by other potentials

(a) A simple phenomenological form for the effective interaction between electrons in metals is the screened Coulomb (or Thomas–Fermi) potential given by

$$V(r) = \frac{e^2}{4\pi\epsilon_0 r} e^{-r/a}. \tag{5.31}$$

The range of the interaction a depends on the density and temperature of the electrons. The form (5.31) is known as the Yukawa potential in the context of the interaction between nuclear particles and as the Debye potential in the context of classical plasmas. Choose units such that $a = 1$ and $e^2/(4\pi\epsilon_0) = 1$. Recall that the force is given by $f(r) = -dV/dr$. Incorporate this force law into class Scatter and compute the dependence of $\sigma(\theta)$ on the energy of the incident particle. Choose the beam width equal to 3. Compare your results for $\sigma(\theta)$ with your results from the Coulomb potential.

(b) Modify the force law in Scatter so that $f(r) = 24(2/r^{13} - 1/r^7)$. This form for $f(r)$ is used to describe the interactions between simple molecules (see Chapter 8). Describe some typical trajectories and compute the differential cross section for several different energies. Let bmax = 2. What is the total cross section? How do your results change if you vary bmax? Choose a small angle as the minimum scattering angle. How sensitive is the total cross section to this minimum angle? Does the differential cross section vary for any other angles besides the smallest scattering angle? □

5.11 Three-body problems

Poincaré showed that it is impossible to obtain an analytic solution for the unrestricted motion of three or more objects interacting under the influence of gravity. However solutions are known for a few special cases, and it is instructive to study the properties of these solutions.

The ThreeBody class computes the trajectories of three particles of equal mass moving in a plane and interacting under the influence of gravity. Both the physics and the drawing are implemented in the ThreeBody class shown in Listing 5.9. Note that the getRate and computeForce methods compute trajectories for an arbitrary number of masses. Note how the computeForce method uses the arraycopy method to quickly zero the arrays. To simplify the drawing of the particle trajectories, the ThreeBody class uses an inner class that extends a Circle and contains a Trail.

Listing 5.9: A class that models the dynamics of the three-body problem.

```
package org.opensourcephysics.sip.ch05;
import java.awt.*;
import org.opensourcephysics.display.*;
import org.opensourcephysics.numerics.*;

public class ThreeBody implements Drawable, ODE {
   int n = 3; // number of interacting bodies
   // state= {x1, vx1, y1, vy1, x2, vx2, y2, vy2, x3, vx3, y3, vy3, t}
   double[] state = new double[4*n+1];
   double[] force = new double[2*n]
   double[] zeros = new double[2*n];
   ODESolver odeSolver = new RK45MultiStep(this);
   Mass mass1 = new Mass(), mass2 = new Mass(), mass3 = new Mass();
```

```
public void draw(DrawingPanel panel, Graphics g) {
   mass1.draw(panel, g);
   mass2.draw(panel, g);
   mass3.draw(panel, g);
}

public void doStep() {
   odeSolver.step();
   mass1.setXY(state[0], state[2]);
   mass2.setXY(state[4], state[6]);
   mass3.setXY(state[8], state[10]);
}

void initialize(double[] initState) {
   // copies initState to state
   System.arraycopy(initState, 0, state, 0, 13);
   mass1.clear();
   mass2.clear();
   mass3.clear();
   mass1.setXY(state[0], state[2]);
   mass2.setXY(state[4], state[6]);
   mass3.setXY(state[8], state[10]);
}

void computeForce(double[] state) {
    // sets force array elements to 0
   System.arraycopy(zeros, 0, force, 0, force.length);
   for(int i = 0;i<n;i++) {
      for(int j = i+1;j<n;j++) {
         double dx = state[4*i]-state[4*j];
         double dy = state[4*i+2]-state[4*j+2];
         double r2 = dx*dx+dy*dy;
         double r3 = r2*Math.sqrt(r2);
         double fx = dx/r3;
         double fy = dy/r3;
         force[2*i] -= fx;
         force[2*i+1] -= fy;
         force[2*j] += fx;
         force[2*j+1] += fy;
      }
   }
}

public void getRate(double[] state, double[] rate) {
   computeForce(state); // force array alternates fx and fy
   for(int i = 0;i<n;i++) {
      int i4 = 4*i;
      rate[i4] = state[i4+1];     // x rate is vx
      rate[i4+1] = force[2*i];    // vx rate is fx
      rate[i4+2] = state[i4+3];   // y rate is vy
      rate[i4+3] = force[2*i+1];  // vy rate is fy
   }
   rate[state.length-1] = 1; // time rate is last
```

```
      }

      public double[] getState() {
         return state;
      }

      class Mass extends Circle {
         Trail trail = new Trail();
         // Draws the mass
         public void draw(DrawingPanel panel, Graphics g) {
            trail.draw(panel, g);
            super.draw(panel, g);
         }

         // Clears trail
         void clear() {
            trail.clear();
         }

         // Sets postion and adds to trail
         public void setXY(double x, double y) {
            super.setXY(x, y);
            trail.addPoint(x, y);
         }
      }
   }
```

The initial conditions for our examples are contained in the `ThreeBodyInitialConditions` class. This file is available in the ch05 package but is not listed here because it contains mostly numeric data.

In 1765 Euler discovered an analytic solution in which three masses start on a line and rotate so that the central mass stays fixed. The `EULER` array in `ThreeBodyInitialConditions` initializes the model to produce this type of solution. The first mass is placed at the center, and the other two masses are placed on opposite sides with velocities that are equal but opposite. Because of the symmetry, the trajectories are ellipses with a common focus at the center.

A second analytic solution to the unrestricted three-body problem was found by Lagrange in 1772. This solution starts with three masses at the corners of an equilateral triangle. Each mass moves in an ellipse in such a way that the triangle formed by the masses remains equilateral. The `LAGRANGE` array initializes this solution.

A spectacular new solution that adds to the sparse list of analytic three-body solutions was first discovered numerically by Moore and proven to be stable by Chenciner and Montgomery. The `MONTGOMERY` array contains the initial conditions for this solution.

The `ThreeBodyApp` class in Listing 5.10 is the target class for the three-body program. The `doStep` method merely increments the model's differential equations solver and repaints the view.

Listing 5.10: A program that displays the trajectories of three bodies interacting via gravitational forces.

```
package org.opensourcephysics.sip.ch05;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.frames.*;
```

```
public class ThreeBodyApp extends AbstractSimulation {
   PlotFrame frame = new PlotFrame("x", "y", "Three-Body Orbits");
   ThreeBody trajectory = new ThreeBody();

   public ThreeBodyApp() {
      frame.addDrawable(trajectory);
      frame.setSquareAspect(true);
      frame.setSize(450, 450);
   }

   public void initialize() {
      trajectory.odeSolver.setStepSize(control.getDouble("dt"));
      trajectory.initialize(ThreeBodyInitialConditions.MONTGOMERY);
      frame.setPreferredMinMax(-1.5, 1.5, -1.5, 1.5);
   }

   public void reset() {
      control.setValue("dt", 0.1);
      enableStepsPerDisplay(true);
      initialize();
   }

   protected void doStep() {
      trajectory.doStep();
      frame.setMessage("t="+decimalFormat.format(trajectory.state[4]));
   }

   public static void main(String[] args) {
      SimulationControl.createApp(new ThreeBodyApp());
   }
}
```

Problem 5.16. Stability of solutions to the three-body problem

Examine the stability of the three solutions to the three-body problem by slightly varying the initial velocity of one of the masses. Before passing your new initial state to `trajectory.initialize`, calculate the center of mass velocity and subtract this velocity from every object. Show that any instability is due to the physics and not to the numerical differential equation solver. Which of the three analytic solutions is stable? Check conservation of the total energy and angular momentum. □

5.12 Projects

Project 5.17. Effect of a "solar wind"

(a) Assume that a satellite is affected not only by the Earth's gravitational force, but also by a weak uniform "solar wind" of magnitude W acting in the horizontal direction. The equations of motion can be written as

$$\frac{d^2x}{dt^2} = -\frac{GMx}{r^3} + W \tag{5.32a}$$

$$\frac{d^2y}{dt^2} = -\frac{GMy}{r^3}. \tag{5.32b}$$

Choose initial conditions so that a circular orbit would be obtained for $W = 0$. Then choose a value of W whose magnitude is about 3% of the acceleration due to the gravitational field and compute the orbit. How does the orbit change?

(b) Determine the change in the velocity space orbit when the solar wind (5.32) is applied. How does the total angular momentum and energy change? Explain in simple terms the previously observed change in the position space orbit. See Luehrmann for further discussion of this problem. □

Project 5.18. Resonances and the asteroid belt

(a) A histogram of the number of asteroids versus their distance from the sun shows some distinct gaps. These gaps, called the *Kirkwood gaps*, are due to resonance effects. That is, if asteroids were in these gaps, their periods would be simple fractions of the period of Jupiter. Modify class `Planet2` so that planet two has the mass of Jupiter by setting `GM1 = 0.001*GM`. Because the asteroid masses are very small compared to that of Jupiter, the gravitational force on Jupiter due to the asteroids can be neglected. The initial conditions listed in `Planet2` are approximately correct for Jupiter. The initial conditions for the asteroid (planet one in `Planet2`) correspond to the 1/3 resonance (the period of the asteroid is one third that of Jupiter). Run the program with these changes and describe the orbit of the asteroid.

(b) Use Kepler's third law, T^2/a^3 = constant, to determine the values of a, the asteroid's semimajor axis, such that the ratio of its period of revolution about the Sun to that of Jupiter is 1/2, 3/7, 2/5, and 2/3. Set the initial value of `x(1)` equal to a for each of these ratios and choose the initial value of `vy(1)` so that the asteroid would have a circular orbit if Jupiter was not present. Describe the orbits that you obtain.

(c) It is instructive to plot a as a function of time. However, because it is not straightforward to measure a directly in the simulation, it is more convenient to plot the quantity $-2GMm/E$, where E is the total energy of the asteroid and m is the mass of the asteroid. Because E is proportional to m, the quantity $-2GMm/E$ is independent of m. If the interaction of the asteroid with Jupiter is ignored, it can be shown that $a = -2GMm/E$, where E is the asteroid kinetic energy plus the asteroid-sun potential energy. Derive this result for circular orbits. Plot the quantity $-2GMm/E$ versus time for about thirty revolutions for the initial conditions in Problem 5.18b.

(d) Compute the time dependence of $-2GMm/E$ for asteroid orbits whose initial position `x(1)` ranges from 2.0 to 5.0 in steps of 0.2. Choose the initial values of `vy(1)` so that circular orbits would be obtained in the absence of Jupiter. Are there any values of `x(1)` for which the time dependence of a is unusual?

(e) Make a histogram of the number of asteroids versus the value of $-2GMm/E$ at $t = 2000$. (You can use the `HistogramFrame` class described on page 206 if you wish.) Assume that the initial value of `x(1)` ranges from 2.0 to 5.0 in steps of 0.02 and choose the initial values of `vy(1)` as before. Use a histogram bin width of 0.1. If you have time, repeat for $t = 5000$

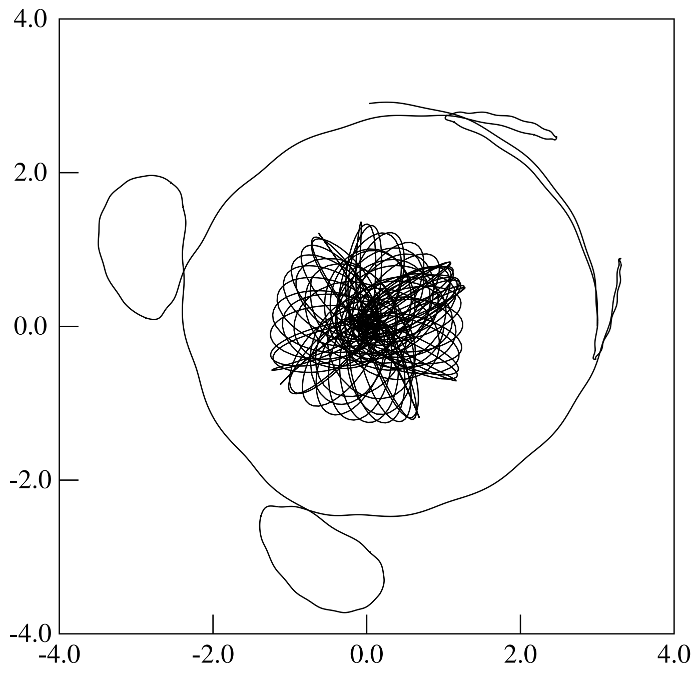


Figure 5.8: Orbits of the two electrons in the classical helium atom with the initial condition $\mathbf{r}_1 = (3, 0), \mathbf{r}_2 = (1, 0), \mathbf{v}_1 = (0, 0.4)$, and $\mathbf{v}_2 = (0, -1)$ (see Project 5.19c).

and compare the histogram with your previous results. Is there any evidence for Kirkwood gaps? A resonance occurs when the periods of the asteroid and Jupiter are related by simple fractions. We expect the number of asteroids with values of a corresponding to resonances to be small.

(f) Repeat part (e) with initial velocities that vary from their values for a circular orbit by 1, 3, and 5%. □

Project 5.19. The classical helium atom

The classical helium atom is a relatively simple example of a three-body problem and is similar to the gravitational three-body problem of a heavy sun and two light planets. The important difference is that the two electrons repel one another, unlike the planetary case where the intraplanetary interaction is attractive. If we ignore the small motion of the heavy nucleus, the equations of motion for the two electrons can be written as

$$\mathbf{a}_1 = -2\frac{\mathbf{r}_1}{r_1^3} + \frac{\mathbf{r}_1 - \mathbf{r}_2}{r_{12}^3} \tag{5.33a}$$

$$\mathbf{a}_2 = -2\frac{\mathbf{r}_2}{r_2^3} + \frac{\mathbf{r}_2 - \mathbf{r}_1}{r_{12}^3} \tag{5.33b}$$

where $\mathbf{r}_1$ and $\mathbf{r}_2$ are measured from the fixed nucleus at the origin, and r_{12} is the distance between the two electrons. We have chosen units such that the mass and charge of the electron are both unity. The charge of the helium nucleus is two in these units. Because the electrons are sometimes very close to the nucleus, their acceleration can become very large, and a very small time step Δt is required. It is not efficient to use the same small time step throughout the simulation, and instead a variable time step or an *adaptive* step size algorithm is suggested. An adaptive step size algorithm can be used with any standard numerical algorithm for solving differential equations. The RK45 algorithm described in Appendix 3A is adaptive and is a good all-around choice for these types of problems.

(a) For simplicity, we restrict our atom to two dimensions. Modify Planet2 to simulate the classical helium atom. Choose units such that the electron mass is one and the other constants are absorbed into the unit of charge so that the force between two electrons is

$$|F| = \frac{1}{r^2}. \tag{5.34}$$

Choose the initial value of the time step to be $\Delta t = 0.001$. Some of the possible orbits are similar to those we have seen in our mini-solar system. For example, try the initial condition $\mathbf{r}_1 = (2,0), \mathbf{r}_2 = (-1,0), \mathbf{v}_1 = (0,0.95)$, and $\mathbf{v}_2 = (0,-1)$.

(b) Most initial conditions result in unstable orbits in which one electron eventually leaves the atom (autoionization). The initial condition $\mathbf{r}_1 = (1.4,0), \mathbf{r}_2 = (-1,0), \mathbf{v}_1 = (0,0.86)$, and $\mathbf{v}_2 = (0,-1)$ gives "braiding" orbits. Make small changes in this initial condition to observe autoionization.

(c) The classical helium atom is capable of very complex orbits (see Figure 5.8). Investigate the motion for the initial condition $\mathbf{r}_1 = (3,0), \mathbf{r}_2 = (1,0), \mathbf{v}_1 = (0,0.4)$, and $\mathbf{v}_2 = (0,-1)$. Does the motion conserve the total angular momentum? Also try $\mathbf{r}_1 = (2.5,0), \mathbf{r}_2 = (1,0), \mathbf{v}_1 = (0,0.4)$, and $\mathbf{v}_2 = (0,-1)$.

(d) Choose the initial condition $\mathbf{r}_1 = (2,0), \mathbf{r}_2 = (-1,0)$, and $\mathbf{v}_2 = (0,-1)$. Then vary the initial value of $\mathbf{v}_1$ from $(0.6,0)$ to $(1.3,0)$ in steps of $\Delta v = 0.02$. For each set of initial conditions, calculate the time it takes for autoionization. Assume that ionization occurs when either electron exceeds a distance of six from the nucleus. Run each simulation for a maximum time of 2000. Plot the ionization time versus v_{1x}. Repeat for a smaller interval of Δv centered about one of the longer ionization times. These calculations require much computer resources. Do the two plots look similar? If so, such behavior is called "self-similar" and is characteristic of chaotic systems and the geometry of fractals (see Chapters 6 and 13). More discussion on the nature of the orbits can be found in Yamamoto and Kaneko. □

References and Suggestions for Further Reading

Harold Abelson, Andrea diSessa, and Lee Rudolph, "Velocity space and the geometry of planetary orbits," Am. J. Phys. **43**, 579–589 (1975). See also Andrea diSessa, "Orbit: a mini-environment for exploring orbital mechanics," in O. Lecarme and R. Lewis, editors, *Computers in Education*, 359 (North–Holland, 1975). Detailed geometrical rather than calculus-based arguments on the origin of closed orbits for inverse-square forces are presented. Sections 5.7 and 5.8 are based on these papers.

Ralph Baierlein, *Newtonian Dynamics* (McGraw–Hill, 1983). An intermediate level text on mechanics. Of particular interest are the discussions on the stability of circular orbits and the effects of an oblate sun.

John J. Brehm and William J. Mullin, *Introduction to the Structure of Matter* (John Wiley & Sons, 1989). See Section 3-4 for a discussion of Rutherford scattering.

Alain Chenciner and Richard Montgomery, "A remarkable periodic solution of the three-body problem in the case of equal masses," Annals of Mathematics **152**, 881–901 (2000).

J. M. A. Danby, *Computer Modeling: From Sports To Spaceflight ... From Order To Chaos* (William-Bell, 1997). See Chapter 11 for a discussion of orbits including an excellent treatment of the Lagrange points.

R. P. Feynman, R. B. Leighton, M. Sands, *The Feynman Lectures in Physics, Vol. 1* (Addison–Wesley, 1963). See Chapter 9.

A. P. French, *Newtonian Mechanics* (W. W. Norton & Company, 1971). An introductory level text with more than a cursory treatment of planetary motion.

Ian R. Gatland, "Numerical integration of Newton's equations including velocity-dependent forces," Am J. Phys. **62**, 259 (1994). The author chooses a variable time step based on the difference in the calculation of the positions rather than the energy as we did in Project 5.19.

Herbert Goldstein, Charles P. Poole, and John L. Safko, *Classical Mechanics*, 3rd ed. (Addison–Wesley, 2002). Chapter 3 has an excellent discussion of the Kepler problem and the conditions for a closed orbit.

Myron Lecar and Fred A. Franklin, "On the original distribution of the asteroids. I," Icarus **20**, 422–436 (1973). The authors use simulations of the motions of asteroids and discuss the Kirkwood gaps.

Arthur W. Luehrmann, "Orbits in the solar wind – a mini-research problem," Am. J. Phys. **42**, 361 (1974). Luehrmann emphasizes the desirability of student problems requiring inductive rather than deductive reasoning.

Jerry B. Marion and Stephen T. Thornton, *Classical Dynamics*, 5th ed. (Harcourt, 2004). Chapter 8 discusses central force motion, the precession of the Mercury, and the stability of circular orbits.

Michael McCloskey, "Intuitive physics," Sci. Am. **248** (4), 122–130 (1983). A discussion of the counterintuitive nature of Newton's laws.

John R. Merrill and Richard A. Morrow, "An introductory scattering experiment by simulation," Am. J. Phys. **38**, 1104–1107 (1970).

C. Moore, "Braids in classical gravity," Phys. Rev. Lett. **70**, 3675 (1993).

Bernard Schutz, *Gravity From the Ground Up* (Cambridge University Press, 2003). The associated website, <www.gravityfromthegroundup.org/>, has many Java programs including a simulation of the orbit of a planet around a black hole or neutron star, using the equation of motion appropriate for general relativity.

Tomomyuki Yamamoto and Kunihiko Kaneko, "Helium atom as a classical three-body problem," Phys. Rev. Lett. **70**, 1928 (1993).

Chapter 6

The Chaotic Motion of Dynamical Systems

We study simple nonlinear deterministic models that exhibit chaotic behavior. We will find that the use of the computer to do numerical experiments will help us gain insight into the nature of chaos.

6.1 Introduction

Most natural phenomena are intrinsically nonlinear. Weather patterns and the turbulent motion of fluids are everyday examples. Although we have explored some of the properties of nonlinear systems in Chapter 4, it is easier to introduce some of the important concepts in the context of ecology. Our first goal will be to motivate and analyze the one-dimensional difference equation

$$x_{n+1} = 4rx_n(1-x_n) \tag{6.1}$$

where x_n is the ratio of the population in the nth generation to a reference population. We shall see that the dynamical properties of (6.1) are surprisingly intricate and have important implications for the development of a more general description of nonlinear phenomena. The significance of the behavior of (6.1) is indicated by the following quote from the ecologist Robert May:

> *"Its study does not involve as much conceptual sophistication as does elementary calculus. Such study would greatly enrich the student's intuition about nonlinear systems. Not only in research but also in the everyday world of politics and economics we would all be better off if more people realized that simple nonlinear systems do not necessarily possess simple dynamical properties."*

The study of chaos is of much current interest, but the phenomena is not new and has been of interest, particularly to astronomers and mathematicians, for over one hundred years. Much of the current interest is due to the use of the computer as a tool for making empirical observations. We will use the computer in this spirit.

6.2 A Simple One-Dimensional Map

Imagine an island with an insect population that breeds in the summer and leaves eggs that hatch the following spring. Because the population growth occurs at discrete times, it is appropriate to model the population growth by a difference equation rather than by a differential equation. A simple model of population growth that relates the population in generation $n+1$ to the population in generation n is given by

$$P_{n+1} = aP_n \tag{6.2}$$

where P_n is the population in generation n and a is a constant. In the following, we will assume that the time interval between generations is unity and will refer to n as the time.

If $a < 1$, the population decreases at each generation, and eventually the population becomes extinct. If $a > 1$, each generation will be a times larger than the previous one. In this case (6.2) leads to geometrical growth and an unbounded population. Although the unbounded nature of geometrical growth is clear, it is remarkable that most of us do not integrate our understanding of geometrical growth into our everyday lives. Can a bank pay 4% interest each year indefinitely? Can the world's human population grow at a constant rate forever?

It is natural to formulate a more realistic model in which the population is bounded by the finite carrying capacity of its environment. A simple model of density-dependent growth is

$$P_{n+1} = P_n(a - bP_n). \tag{6.3}$$

Equation (6.3) is nonlinear due to the presence of the quadratic term in P_n. The linear term represents the natural growth of the population; the quadratic term represents a reduction of this natural growth caused, for example, by overcrowding or by the spread of disease.

It is convenient to rescale the population by letting $P_n = (a/b)x_n$ and rewriting (6.3) as

$$x_{n+1} = ax_n(1 - x_n). \tag{6.4}$$

The replacement of P_n by x_n changes the units used to define the various parameters. To write (6.4) in the standard form (6.1), we define the parameter $r = a/4$ and obtain

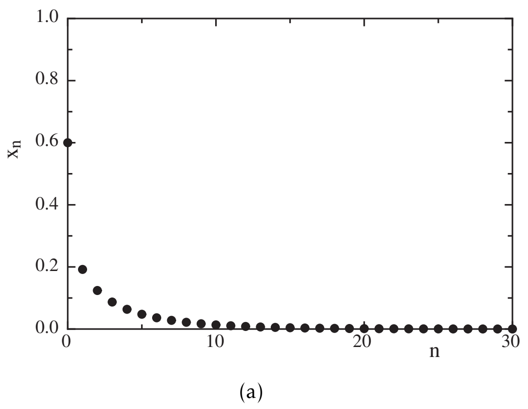
$$x_{n+1} = f(x_n) = 4rx_n(1 - x_n). \tag{6.5}$$

The rescaled form (6.5) has the desirable feature that its dynamics are determined by a single control parameter r instead of the two parameters a and b. Note that if $x_n > 1$, x_{n+1} will be negative. To avoid this nonphysical feature, we impose the conditions that x is restricted to the interval $0 \le x \le 1$ and $0 < r \le 1$, respectively. Because the function $f(x)$ defined in (6.5) transforms any point on the one-dimensional interval $[0, 1]$ into another point in the same interval, the function f is called a *one-dimensional map.*

The form of $f(x)$ in (6.5) is known as the *logistic* map. The logistic map is a simple example of a *dynamical system*; that is, the map is a deterministic, mathematical prescription for finding the future state of a system given its present state.

The sequence of values $x_0, x_1, x_2, \ldots$ is called the *trajectory*. To check your understanding, suppose that the initial value of x_0 or *seed* is $x_0 = 0.5$ and $r = 0.2$. Do a calculation to show that the trajectory is $x_1 = 0.2, x_2 = 0.128, x_3 = 0.089293, \ldots$. The first thirty iterations of (6.5) are shown for two values of r in Figure 6.1.

The class `IterateMapApp` computes the trajectory of the logistic map in (6.5). Note that we have extended the `AbstractCalculation` class, which is appropriate because many of the results of Sections 6.1–6.4 were discovered using a programmable calculator.



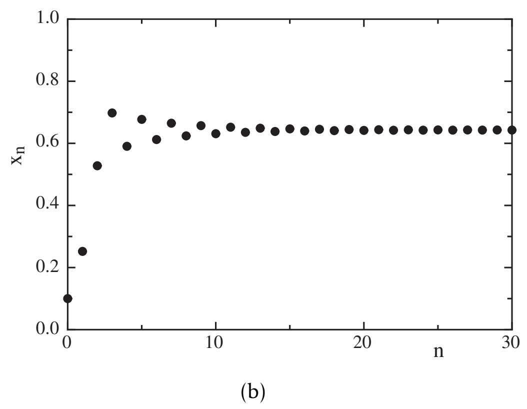


Figure 6.1: (a) The trajectory of x for $r = 0.2$ and $x_0 = 0.6$. The stable fixed point is at $x = 0$. (b) The trajectory for $r = 0.7$ and $x_0 = 0.1$. Note the initial transient behavior.

Listing 6.1: The IterateMapApp class iterates the logistic map and plots the resulting trajectory

```java
package org.opensourcephysics.sip.ch06;
import org.opensourcephysics.frames.*;
import org.opensourcephysics.controls.*;

public class IterateMapApp extends AbstractCalculation {
   int datasetIndex = 0;
   PlotFrame plotFrame = new PlotFrame("iterations", "x",
                                       "trajectory");

   public IterateMapApp() {
      // keep data between calls to calculate
      plotFrame.setAutoclear(false);
   }

   public void reset() {
      control.setValue("r", 0.2);
      control.setValue("x", 0.6);
      control.setValue("iterations", 50);
      datasetIndex = 0;
   }

   public void calculate() {
      double r = control.getDouble("r");
      double x = control.getDouble("x");
      int iterations = control.getInt("iterations");
      for(int i = 0;i<=iterations;i++) {
         plotFrame.append(datasetIndex, i, x);
         x = map(r, x);
      }
      plotFrame.setMarkerSize(datasetIndex, 1);
      plotFrame.setXYColumnNames(datasetIndex, "iteration",
                  "calc #"+datasetIndex);
      datasetIndex++;
   }

   double map(double r, double x) {
      return 4*r*x*(1-x); // iterate map
   }

   public static void main(String[] args) {
      CalculationControl.createApp(new IterateMapApp());
   }
}
```

Problem 6.1. The trajectory of the logistic map

(a) Explore the dynamical behavior of the logistic map in (6.5) with $r = 0.24$ for different values of x_0. Show numerically that $x = 0$ is a *stable fixed point* for this value of r. That is, the iterated values of x converge to $x = 0$ independently of the value of x_0. If x represents the population of insects, describe the qualitative behavior of the population.

(b) Explore the dynamical behavior of (6.5) for $r = 0.26, 0.5, 0.74$, and 0.748. A fixed point is *unstable* if for almost all values of x_0 near the fixed point, the trajectories diverge from it. Verify that $x = 0$ is an unstable fixed point for $r > 0.25$. Show that for the suggested values of r, the iterated values of x do not change after an initial *transient*; that is, the long time dynamical behavior is *period* 1. In Appendix 6A we show that for $r < 3/4$ and for x_0 in the interval $0 < x_0 < 1$, the trajectories approach the *stable attractor* at $x = 1 - 1/4r$. The set of initial points that iterate to the attractor is called the *basin* of the attractor. For the logistic map, the interval $0 < x < 1$ is the basin of attraction of the attractor $x = 1 - 1/4r$.

(c) Explore the dynamical properties of (6.5) for $r = 0.752$, 0.76, 0.8, and 0.862. For $r = 0.752$ and 0.862, approximately 1000 iterations are necessary to obtain convergent results. Show that if r is greater than 0.75, x oscillates between two values after an initial transient behavior. That is, instead of a stable cycle of period 1 corresponding to one fixed point, the system has a stable cycle of period 2. The value of r at which the single fixed point x^* splits or *bifurcates* into two values $x_1{}^*$ and $x_2{}^*$ is $r = b_1 = 3/4$. The pair of x values, $x_1{}^*$ and $x_2{}^*$, form a *stable attractor* of period 2.

(d) What are the stable attractors of (6.5) for $r = 0.863$ and 0.88? What is the corresponding period? What are the stable attractors and corresponding periods for $r = 0.89$, 0.891, and 0.8922? □

Another way to determine the behavior of (6.5) is to plot the values of x as a function of r (see Figure 6.2). The iterated values of x are plotted after the initial transient behavior is discarded. Such a plot is generated by `BifurcateApp`. For each value of r, the first `ntransient` values of x are computed but not plotted. Then the next `nplot` values of x are plotted with the first half with the first half in one color and the second half in another. This process is repeated for a new value of r until the desired range of r values is reached. The magnitude of `nplot` should be at least as large as the longest period that you wish to observe. `BifurcateApp` extends `AbstractSimulation` rather than `AbstractCalculation` because the calculations can be time consuming. For this reason you might want to stop them before they are finished and reset some of the parameters.

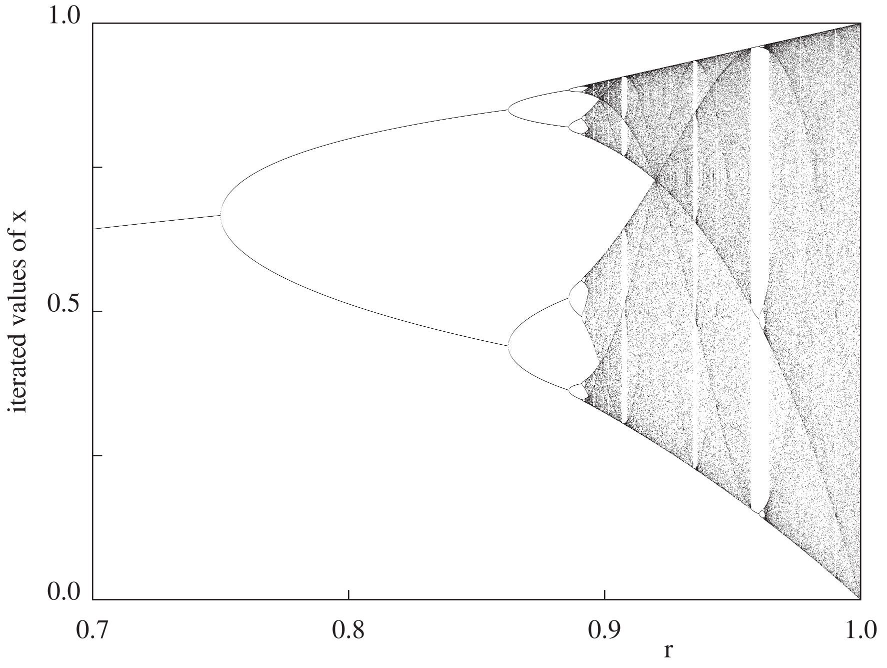


Figure 6.2: Bifurcation diagram of the logistic map. For each value of r, the iterated values of x_n are plotted after the first 1000 iterations are discarded. Note the transition from periodic to chaotic behavior and the narrow windows of periodic behavior within the region of chaos.

Listing 6.2: The `BifurcateApp` program generates a bifurcation plot of the logistic map

```
package org.opensourcephysics.sip.ch06;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.frames.*;

public class BifurcateApp extends AbstractSimulation {
   double r;         // control parameter
   double dr;        // incremental change of r, suggest dr <= 0.01
   int ntransient; // number of iterations not plotted
   int nplot;        // number of iterations plotted
   PlotFrame plotFrame = new PlotFrame("r", "x",
            "Bifurcation diagram");

   public BifurcateApp() {
      // small size gives better resolution
      plotFrame.setMarkerSize(0, 0);
      plotFrame.setMarkerSize(1, 0);
   }

   public void initialize() {
      plotFrame.clearData();
      r = control.getDouble("initial r");
      dr = control.getDouble("dr");
      ntransient = control.getInt("ntransient");
      nplot = control.getInt("nplot");
   }

   public void doStep() {
      if(r<1.0) {
         double x = 0.5;
         for(int i = 0;i<ntransient;i++) {     // x values not plotted
            x = map(x, r);
         }
         // plot half the points in dataset zero
         for(int i = 0;i<nplot/2;i++) {
            x = map(x, r);
            // shows different x values for given value of r
            plotFrame.append(0, r, x);
         }
         // plot remaining points in dataset one
         for(int i = nplot/2+1;i<nplot;i++) {
            x = map(x, r);
            // dataset one has a different color
            plotFrame.append(1, r, x);
            i++;
         }
         r += dr;
      }
   }

   public void reset() {
      control.setValue("initial r", 0.2);
      control.setValue("dr", 0.005);
      control.setValue("ntransient", 200);
      control.setValue("nplot", 50);
   }

   double map(double x, double r) {
      return 4*r*x*(1-x);
   }

   public static void main(String[] args) {
      SimulationControl.createApp(new BifurcateApp());
   }
}
```

Problem 6.2. Qualitative features of the logistic map

(a) Use `BifurcateApp` to identify period 2, period 4, and period 8 behavior as can be seen in Figure 6.2. Choose `ntransient` ≥ 1000. It might be necessary to "zoom in" on a portion of the plot. How many period doublings can you find?

(b) Change the scale so that you can follow the iterations of x from period 4 to period 16 behavior. How does the plot look on this scale in comparison to the original scale?

(c) Describe the shape of the trajectory near the bifurcations from period 2 to period 4, period 4 to period 8, etc. These bifurcations are frequently called *pitchfork bifurcations*. □

The bifurcation diagram in Figure 6.2 indicates that the period doubling behavior ends at $r \approx 0.892$. This value of r is known very precisely and is given by $r = r_\infty = 0.892486417967\ldots$. At $r = r_\infty$, the sequence of period doublings accumulates to a trajectory of infinite period. In Problem 6.3 we explore the behavior of the trajectories for $r > r_\infty$.

Problem 6.3. Chaotic behavior

(a) For $r > r_\infty$, two initial conditions that are very close to one another can yield very different trajectories after a few iterations. As an example, choose $r = 0.91$ and consider $x_0 = 0.5$ and 0.5001. How many iterations are necessary for the iterated values of x to differ by more than ten percent? What happens for $r = 0.88$ for the same choice of seeds?

(b) The accuracy of floating point numbers retained on a digital computer is finite. To test the effect of the finite accuracy of your computer, choose $r = 0.91$ and $x_0 = 0.5$ and compute the trajectory for 200 iterations. Then modify your program so that after each iteration, the operation `x = x/10` is followed by `x = 10*x`. This combination of operations truncates the last digit that your computer retains. Compute the trajectory again and compare your results. Do you find the same discrepancy for $r < r_\infty$?

(c) What are the dynamical properties for $r = 0.958$? Can you find other windows of periodic behavior in the interval $r_\infty < r < 1$? □

6.3 Period Doubling

The results of the numerical experiments that we did in Section 6.2 probably have convinced you that the dynamical properties of a simple, nonlinear deterministic system can be quite complicated.

To gain more insight into how the dynamical behavior depends on r, we introduce a simple graphical method for iterating (6.5). In Figure 6.3 we show a graph of $f(x)$ versus x for $r = 0.7$. A diagonal line corresponding to $y = x$ intersects the curve $y = f(x)$ at the two fixed points $x^* = 0$ and $x^* = 9/14 \approx 0.642857$ [see (6.6b)]. If x_0 is not a fixed point, we can find the trajectory in the following way. Draw a vertical line from $(x = x_0, y = 0)$ to the intersection with the curve $y = f(x)$ at $(x_0, y_0 = f(x_0))$. Next draw a horizontal line from (x_0, y_0) to the intersection with the diagonal line at (y_0, y_0). On this diagonal line $y = x$, and hence the value of x at this intersection is the first iteration $x_1 = y_0$. The second iteration x_2 can be found in the same way. From the point (x_1, y_0), draw a vertical line to the intersection with the curve $y = f(x)$. Keep y fixed at $y = y_1 = f(x_1)$, and draw a horizontal line until it intersects the diagonal line; the value of x at this intersection is x_2. Further iterations can be found by repeating this process.

This graphical method is illustrated in Figure 6.3 for $r = 0.7$ and $x_0 = 0.9$. If we begin with any x_0 (except $x_0 = 0$ and $x_0 = 1$), the iterations will converge to the fixed point $x^* \approx 0.643$. It would be a good idea to repeat the procedure shown in Figure 6.3 by hand. For $r = 0.7$, the fixed point is stable (an attractor of period 1). In contrast, no matter how close x_0 is to the fixed point at $x = 0$, the iterates diverge away from it, and this fixed point is unstable.

How can we explain the qualitative difference between the fixed point at $x = 0$ and at $x^* = 0.642857$ for $r = 0.7$? The local slope of the curve $y = f(x)$ determines the distance moved horizontally each time f is iterated. A slope steeper than 45° leads to a value of x further away from its initial value. Hence, the criterion for the stability of a fixed point is that the magnitude of the slope at the fixed point must be less than 45°. That is, if $|df(x)/dx|_{x=x^*}$ is less than unity, then x^* is stable; conversely, if $|df(x)/dx|_{x=x^*}$ is greater than unity, then x^* is unstable.

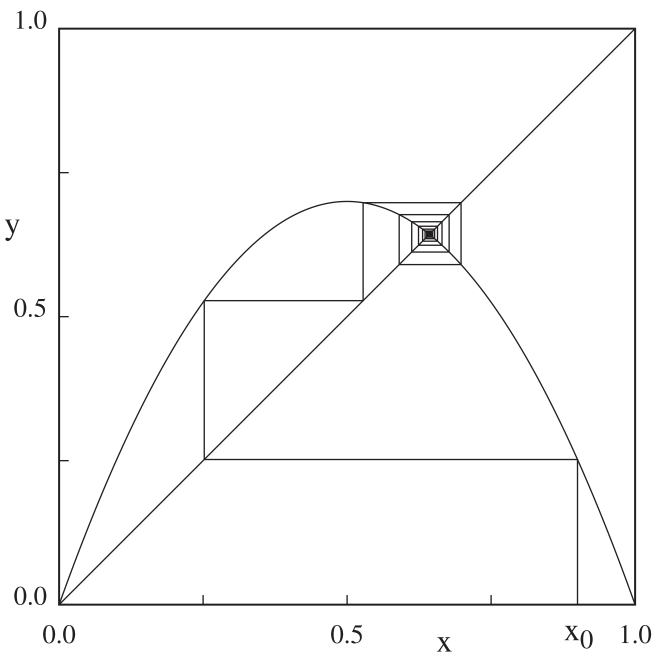


Figure 6.3: Graphical representation of the iteration of the logistic map (6.5) with $r = 0.7$ and $x_0 = 0.9$. Note that the graphical solution converges to the fixed point $x^* \approx 0.643$.

An inspection of $f(x)$ in Figure 6.3 shows that $x = 0$ is unstable because the slope of $f(x)$ at $x = 0$ is greater than unity. In contrast, the magnitude of the slope of $f(x)$ at $x = x^* \approx 0.643$ is less than unity, and this fixed point is stable. In Appendix 6A we show that

$$x^* = 0 \text{ is stable for } 0 < r < 1/4 \tag{6.6a}$$

and

$$x^* = 1 - \frac{1}{4r} \text{ is stable for } 1/4 < r < 3/4. \tag{6.6b}$$

Thus for $0 < r < 3/4$, the behavior after many iterations is known.

What happens if r is greater than 3/4? We found in Section 6.2 that if r is slightly greater than 3/4, the fixed point of f becomes unstable and bifurcates to a cycle of period 2. Now x returns to the same value after every second iteration, and the fixed points of $f\big(f(x)\big)$ are the stable attractors of $f(x)$. In the following, we write $f^{(2)}(x) = f\big(f(x)\big)$ and $f^{(n)}(x)$ for the nth iterate of $f(x)$. (Do not confuse $f^{(n)}(x)$ with the nth derivative of $f(x)$.) For example, the second iterate $f^{(2)}(x)$ is given by the fourth-order polynomial:

$$\begin{aligned} f^{(2)}(x) &= 4r\big[4rx(1-x)\big] - 4r\big[4rx(1-x)\big]^2 \\ &= 4r[4rx(1-x)]\big[1 - 4rx(1-x)\big] \\ &= 16r^2x\big[-4rx^3 + 8rx^2 - (1+4r)x + 1\big]. \end{aligned} \tag{6.7}$$

What happens if we increase r still further? Eventually the magnitude of the slope of the fixed points of $f^{(2)}(x)$ exceeds unity, and the fixed points of $f^{(2)}(x)$ become unstable. Now

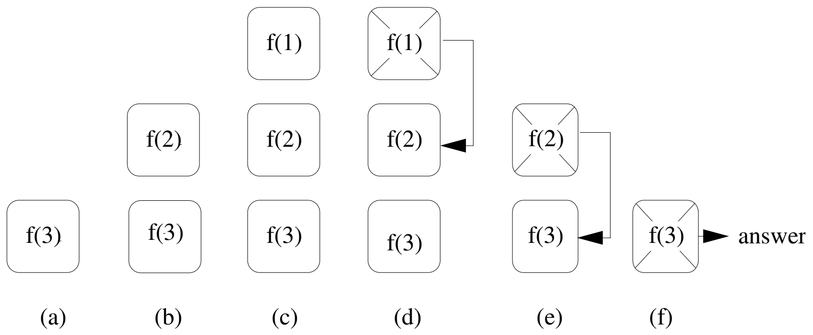


Figure 6.4: Example of the calculation of `f(0.4,0.8,3)` using the recursive function defined in `GraphicalSolutionApp`. The number in each box is the value of the variable `iterate`. The computer executes code from left to right, and each box represents a copy of the function in the computer's memory. The input values $x = 0.4$ and $r = 0.8$, which are the same in each copy, are not shown. The arrows indicate when a copy is finished and its value is returned to one of the other copies. Notice that the first copy of the function $f(3)$ is the last one to finish. The value of `f(x,r,3)` = 0.7842.

the cycle of f is period 4, and the fixed points of the fourth iterate $f^{(4)}(x) = f^{(2)}\big(f^{(2)}(x)\big) = f\Big(f\big(f(f(x))\big)\Big)$ are stable. These fixed points also eventually become unstable, and we are led to the phenomena of *period doubling* that we observed in Problem 6.2.

`GraphicalSolutionApp` implements the graphical analysis of the iterations of $f(x)$. The nth-order iterates are defined in `f(x,r,iterate)`, a *recursive* method. (The parameter `iterate` is 1, 2, and 4 for the functions $f(x)$, $f^{(2)}(x)$, and $f^{(4)}(x)$, respectively.) Recursion is an idea that is simple once you understand it, but it can be difficult to grasp initially. Although the method calls itself, the rules for method calls remain the same. Imagine that a recursive method is called. The computer then starts to execute the code in the method, but comes to another call of the same method as itself. At this point the computer stops executing the code of the original method, and makes an exact copy of the method with possibly different input parameters, and starts executing the code in the copy. There are now two possibilities. One is that the computer comes to the end of the copy without another recursive call. In that case the computer deletes the copy of the method and continues executing the code in the original method. The other possibility is that a recursive call is made in the copy, and a third copy is made of the method, and the code in the third copy is now executed. This process continues until the code in all the copies is executed. Every recursive method must have a possibility of reaching the end of the method; otherwise, the program will eventually crash.

To understand the method `f(x,r,iterate)`, suppose we want to compute `f(0.4,0.8,3)`. First we write `f(0.4,0.8,3)` as in Figure 6.4a. Follow the statements within the method until another call to `f(0.4,0.8,iterate)` occurs. In this case, the call is to `f(0.4,0.8,iterate-1)` which equals `f(0.4,0.8,2)`. Write `f(0.4,0.8,2)` above `f(0.4,0.8,3)` (see Figure 6.4b). When you come to the end of the definition of the method, write down the value of `f` that is actually returned, and remove the method from the stack by crossing it out (see Figure 6.4d). This returned value for `f` equals `y` if `iterate` > 1, or it is the output of the method for `iterate` = 1. Continue deleting copies of `f` as they are finished, until there are no copies left on the paper. The final value of `f` is the value returned by the computer. Write a short program that defines `f(x,r,iterate)` and prints the value of `f(0.4,0.8,3)`. Is the answer the same as your hand calculation?

Listing 6.3: `GraphicalSolutionApp` displays the graphical solution of the logistic map trajectory

```
package org.opensourcephysics.sip.ch06;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.frames.PlotFrame;

public class GraphicalSolutionApp extends AbstractSimulation {
   PlotFrame plotFrame = new PlotFrame("iterations", "x",
                 "graphical solution");
   double r;    // control parameter
   int iterate; // iterate of f(x)
   double x, y;
   double x0, y0;

   public GraphicalSolutionApp() {
      plotFrame.setPreferredMinMax(0, 1, 0, 1);
      plotFrame.setConnected(true);
      plotFrame.setXPointsLinked(true);
      // second argument indicates no marker
      plotFrame.setMarkerShape(2, 0);
   }

   public void reset() {
      control.setValue("r", 0.89);
      control.setValue("x", 0.2);
      plotFrame.setMarkerShape(0, 0);
      control.setAdjustableValue("iterate", 1);
   }

   public void initialize() {
      r = control.getDouble("r");
      x = control.getDouble("x");
      iterate = control.getInt("iterate");
      x0 = x;
      y0 = 0;
      clear();
   }

   public void startRunning() {
      if(iterate!=control.getInt("iterate")) {
         iterate = control.getInt("iterate");
         clear();
      }
      r = control.getDouble("r");
   }

   public void doStep() {
      y = f(x, r, iterate);
      plotFrame.append(1, x0, y0);
      plotFrame.append(1, x0, y);
      plotFrame.append(1, y, y);
      x = x0 = y0 = y;
      control.setValue("x", x);
   }

   void drawFunction() {
      int nplot = 200; // # of points at which function computed
      double delta = 1.0/nplot;
      double x = 0;
      double y = 0;
      for(int i = 0;i<=nplot;i++) {
         y = f(x, r, iterate);
         plotFrame.append(0, x, y);
         x += delta;
      }
   }

   void drawLine() { // draws line y = x
      for(double x = 0;x<1;x += 0.001) {
         plotFrame.append(2, x, x);
      }
   }

   public double f(double x, double r, int iterate) {
      if(iterate>1) {
         double y = f(x, r, iterate-1);
         return 4*r*y*(1-y);
      } else {
         return 4*r*x*(1-x);
      }
   }

   public void clear() {
      plotFrame.clearData();
      drawFunction();
      drawLine();
      plotFrame.repaint();
   }

   public static void main(String[] args) {
      SimulationControl control = SimulationControl.createApp(
                                     new GraphicalSolutionApp());
      control.addButton("clear", "Clear", "Clears the trajectory.");
   }
}
```

Problem 6.4. Qualitative properties of the fixed points

(a) Use `GraphicalSolutionApp` to show graphically that there is a single stable fixed point of $f(x)$ for $r < 3/4$. It would be instructive to modify the program so that the value of the slope $df/dx|_{x=x_n}$ is shown as you step each iteration. At what value of r does the absolute value of this slope exceed unity? Let b_1 denote the value of r at which the fixed point of $f(x)$ bifurcates and becomes unstable. Verify that $b_1 = 0.75$.

(b) Describe the trajectory of $f(x)$ for $r = 0.785$. Is the fixed point given by $x = 1 - 1/4r$ stable or unstable? What is the nature of the trajectory if $x_0 = 1 - 1/4r$? What is the period of $f(x)$ for all other choices of x_0? What are the values of the two-point attractor?

(c) The function $f(x)$ is symmetrical about $x = 1/2$ where $f(x)$ is a maximum. What are the qualitative features of the second iterate $f^{(2)}(x)$ for $r = 0.785$? Is $f^{(2)}(x)$ symmetrical about $x = 1/2$? For what value of x does $f^{(2)}(x)$ have a minimum? Iterate $x_{n+1} = f^{(2)}(x_n)$ for $r = 0.785$ and find its two fixed points ${x_1}^*$ and ${x_2}^*$. (Try $x_0 = 0.1$ and $x_0 = 0.3$.) Are the fixed points of $f^{(2)}(x)$ stable or unstable for this value of r? How do these values of ${x_1}^*$ and ${x_2}^*$ compare with the values of the two-point attractor of $f(x)$? Verify that the slopes of $f^{(2)}(x)$ at ${x_1}^*$ and ${x_2}^*$ are equal.

(d) Verify the following properties of the fixed points of $f^{(2)}(x)$. As r is increased, the fixed points of $f^{(2)}(x)$ move apart, and the slope of $f^{(2)}(x)$ at its fixed points decreases. What is the value of $r = s_2$ at which one of the two fixed points of $f^{(2)}$ equals $1/2$? What is the value of the other fixed point? What is the slope of $f^{(2)}(x)$ at $x = 1/2$? What is the slope at the other fixed point? As r is further increased, the slopes at the fixed points become negative. Finally at $r = b_2 \approx 0.8623$, the slopes at the two fixed points of $f^{(2)}(x)$ equal -1, and the two fixed points of $f^{(2)}$ become unstable. (The exact value of b_2 is $b_2 = (1 + \sqrt{6})/4$.)

(e) Show that for r slightly greater than b_2, for example $r = 0.87$, there are four stable fixed points of $f^{(4)}(x)$. What is the value of $r = s_3$ when one of the fixed points equals $1/2$? What are the values of the three other fixed points at $r = s_3$?

(f) Determine the value of $r = b_3$ at which the four fixed points of $f^{(4)}$ become unstable.

(g) Choose $r = s_3$ and determine the number of iterations that are necessary for the trajectory to converge to period 4 behavior. How does this number of iterations change when neighboring values of r are considered? Choose several values of x_0 so that your results do not depend on the initial conditions. □

Problem 6.5. Periodic windows in the chaotic regime

(a) If you look closely at the bifurcation diagram in Figure 6.2, you will see that the range of chaotic behavior for $r > r_\infty$ is interrupted by intervals of periodic behavior. Magnify your bifurcation diagram so that you can look at the interval $0.957107 \leq r \leq 0.960375$, where a periodic trajectory of period 3 occurs. (Period 3 behavior starts at $r = (1 + \sqrt{8})/4$.) What happens to the trajectory for slightly larger r, for example, $r = 0.9604$?

(b) Plot $f^{(3)}(x)$ versus x at $r = 0.96$, a value of r in the period 3 window. Draw the line $y = x$ and determine the intersections with $f^{(3)}(x)$. The stable fixed points satisfy the condition $x^* = f^{(3)}(x^*)$. Because $f^{(3)}(x)$ is an eighth-order polynomial, there are eight solutions (including $x = 0$). Find the intersections of $f^{(3)}(x)$ with $y = x$ and identify the three stable fixed points. What are the slopes of $f^{(3)}(x)$ at these points? Then decrease r to $r = 0.957107$, the (approximate) value of r below which the system is chaotic. Draw the line $y = x$ and determine the number of intersections with $f^{(3)}(x)$. Note that at this value of r, the curve $y = f^{(3)}(x)$ is tangent to the diagonal line at the three stable fixed points. For this reason, this type of transition is called a *tangent bifurcation*. Note that there is also an unstable point at $x \approx 0.76$.

(c) Plot $x_{n+1} = f^{(3)}(x_n)$ versus n for $r = 0.9571$, a value of r just below the onset of period 3 behavior. How would you describe the behavior of the trajectory? This type of chaotic motion

k	$\mathbf{b_k}$
1	0.750 000
2	0.862 372
3	0.886 023
4	0.891 102
5	0.892 190
6	0.892 423
7	0.892 473
8	0.892 484

Table 6.1: Values of the control parameter $r = b_k$ for the onset of the kth bifurcation. Six decimal places are shown.

is an example of *intermittency*; that is, nearly periodic behavior interrupted by occasional irregular bursts.

(d) To understand the mechanism for the intermittent behavior, we need to "zoom in" on the values of x near the stable fixed points that you found in part (c). To do so change the arguments of the `setPreferredMinMax` method. You will see a narrow channel between the diagonal line $y = x$ and the plot of $f^{(3)}(x)$ near each fixed point. The trajectory can require many iterations to squeeze through the channel, and we see apparent period 3 behavior during this time. Eventually, the trajectory escapes from the channel and bounces around until it is again enters a channel at some unpredictable later time. □

6.4 Universal Properties and Self-Similarity

In Sections 6.2 and 6.3 we found that the trajectory of the logistic map has remarkable properties as a function of the control parameter r. In particular, we found a sequence of period doublings accumulating in a chaotic trajectory of infinite period at $r = r_\infty$. For most values of $r > r_\infty$, the trajectory is very sensitive to the initial conditions. We also found "windows" of period 3, 6, 12, ... embedded in the range of chaotic behavior. How typical is this type of behavior? In the following, we will find further numerical evidence that the general behavior of the logistic map is independent of the details of the form (6.5) of $f(x)$.

You might have noticed that the range of r between successive bifurcations becomes smaller as the period increases (see Table 6.1). For example, $b_2 - b_1 = 0.112398$, $b_3 - b_2 = 0.023624$ and $b_4 - b_3 = 0.00508$. A good guess is that the decrease in $b_k - b_{k-1}$ is geometric; that is, the ratio $(b_k - b_{k-1})/(b_{k+1} - b_k)$ is a constant. You can check that this ratio is not exactly constant, but converges to a constant with increasing k. This behavior suggests that the sequence of values of b_k has a limit and follows a geometrical progression:

$$b_k \approx r_\infty - C\delta^{-k}, \tag{6.8}$$

where δ is known as the *Feigenbaum number* and C is a constant. From (6.8) it is easy to show that δ is given by the ratio

$$\delta = \lim_{k\to\infty} \frac{b_k - b_{k-1}}{b_{k+1} - b_k}. \tag{6.9}$$

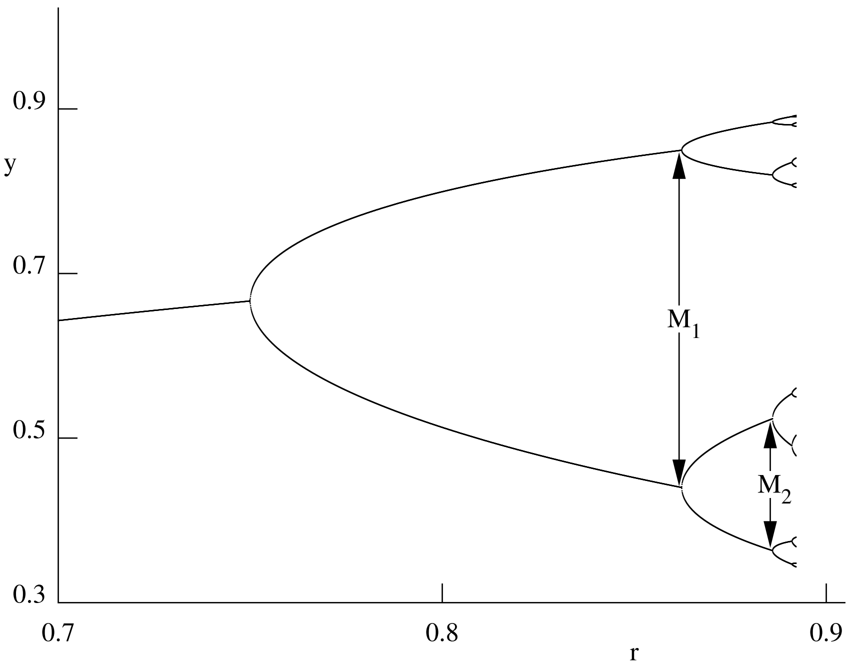


Figure 6.5: The first few bifurcations of the logistic equation showing the scaling of the maximum distance M_k between the asymptotic values of x describing the bifurcation.

Problem 6.6. Estimation of the Feigenbaum constant

(a) Derive the relation (6.9) given (6.8). Plot $\delta_k = (b_k - b_{k-1})/(b_{k+1} - b_k)$ versus k using the values of b_k in Table 6.1 and determine the value of δ. Is the number of decimal places given in Table 6.1 for b_k sufficient for all the values of k shown? The best numerical determination of δ is

$$\delta = 4.669\,201\,609\,102\,991\ldots. \tag{6.10}$$

The number of decimal places in (6.10) is shown to indicate that δ is known precisely. Use (6.8) and (6.10) and the values of b_k to determine the value of r_∞.

(b) In Problem 6.4 we found that one of the four fixed points of $f^{(4)}(x)$ is at $x^* = 1/2$ for $r = s_3 \approx 0.87464$. We also found that the convergence to the fixed points of $f^{(4)}(x)$ for this value of r is more rapid than at nearby values of r. In Appendix 6A we show that these *superstable* trajectories occur whenever one of the fixed points is at $x^* = 1/2$. The values of $r = s_m$ that give superstable trajectories of period 2^{m-1} are much better defined than the points of bifurcation, $r = b_k$. The rapid convergence to the final trajectories also gives better numerical results, and we always know one member of the trajectory, namely $x = 1/2$. Assume that δ can be defined as in (6.9) with b_k replaced by s_m. Use $s_1 = 0.5$, $s_2 \approx 0.809017$, and $s_3 = 0.874640$ to determine δ. The numerical values of s_m are found in Project 6.22 by solving the equation $f^{(m)}(x = 1/2) = 1/2$ numerically; the first eight values of s_m are listed in Table 6.2 in Section 6.11. □

We can associate another number with the series of "pitchfork" bifurcations. From Figures 6.3 and 6.5, we see that each pitchfork bifurcation gives birth to "twins" with the new

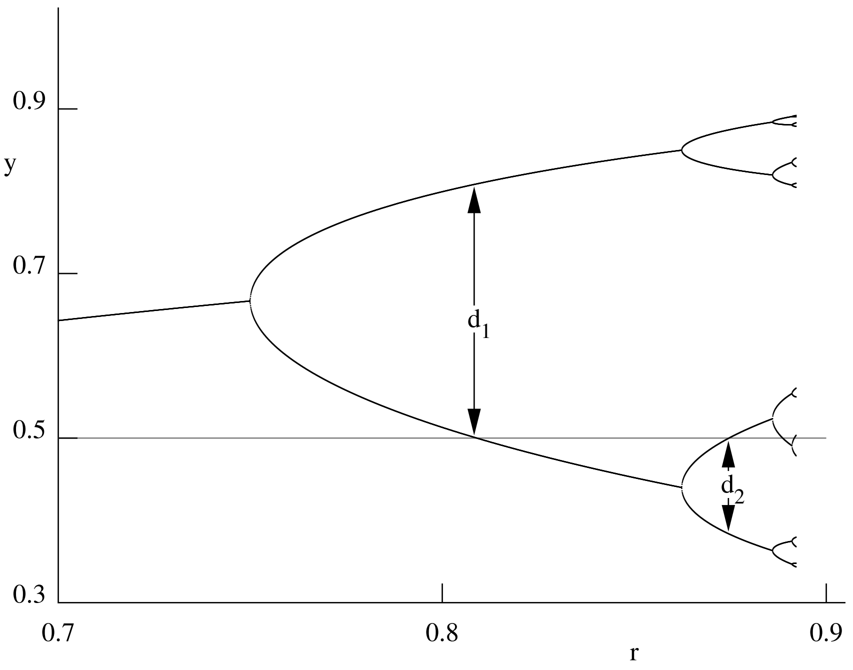


Figure 6.6: The quantity d_k is the distance from $x^* = 1/2$ to the nearest element of the attractor of period 2^k. It is convenient to use this quantity to determine the exponent α.

generation more densely packed than the previous generation. One measure of this density is the maximum distance M_k between the values of x describing the bifurcation (see Figure 6.5). The disadvantage of using M_k is that the transient behavior of the trajectory is very long at the boundary between two different periodic behaviors. A more convenient measure of the distance is the quantity $d_k = x_k{}^* - 1/2$, where $x_k{}^*$ is the value of the fixed point nearest to the fixed point $x^* = 1/2$. The first two values of d_k are shown in Figure 6.6 with $d_1 \approx 0.3090$ and $d_2 \approx -0.1164$. The next value is $d_3 \approx 0.0460$. Note that the fixed point nearest to $x = 1/2$ alternates from one side of $x = 1/2$ to the other. We define the quantity α by the ratio

$$\alpha = \lim_{k\to\infty} -\Big(\frac{d_k}{d_{k+1}}\Big). \tag{6.11}$$

The ratios $\alpha = (0.3090/0.1164) = 2.65$ for $k = 1$ and $\alpha = (0.1164/0.0460) = 2.53$ for $k = 2$ are consistent with the asymptotic value $\alpha = 2.5029078750958928485\ldots$

We now give qualitative arguments that suggest that the general behavior of the logistic map in the period doubling regime is independent of the detailed form of $f(x)$. As we have seen, period doubling is characterized by self-similarities, for example, the period doublings look similar except for a change of scale. We can demonstrate these similarities by comparing $f(x)$ for $r = s_1 = 0.5$ for the superstable trajectory with period 1 to the function $f^{(2)}(x)$ for $r = s_2 \approx 0.809017$ for the superstable trajectory of period 2 (see Figure 6.7). The function $f(x, r = s_1)$ has unstable fixed points at $x = 0$ and $x = 1$ and a stable fixed point at $x = 1/2$. Similarly, the function $f^{(2)}(x, r = s_2)$ has a stable fixed point at $x = 1/2$ and an unstable fixed point at $x \approx 0.69098$. Note the similar shape but different scale of the curves in the square boxes in part (a) and part (b) of Figure 6.7. This similarity is an example of scaling. That is, if we scale

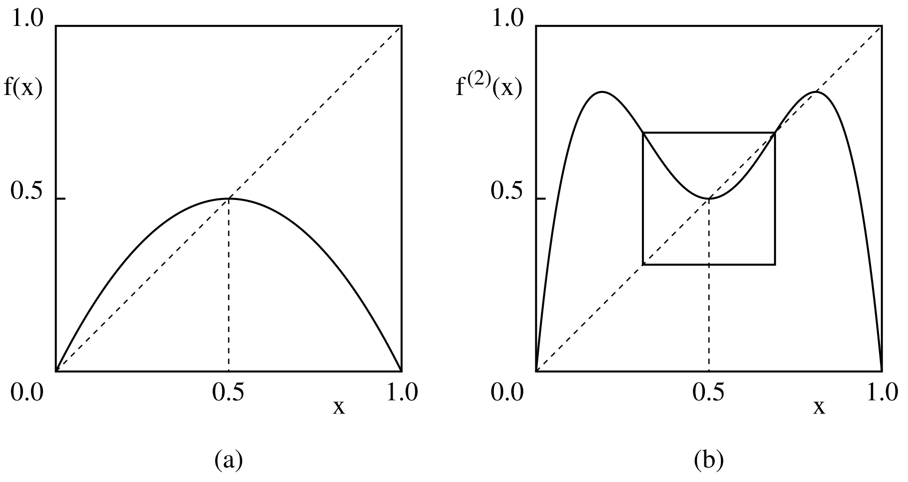


Figure 6.7: Comparison of $f(x,r)$ for $r = s_1$ with the second iterate $f^{(2)}(x)$ for $r = s_2$. (a) The function $f(x, r = s_1)$ has unstable fixed points at $x = 0$ and $x = 1$ and a stable fixed point at $x = 1/2$. (b) The function $f^{(2)}(x, r = s_1)$ has a stable fixed point at $x = 1/2$. The unstable fixed point of $f^{(2)}(x)$ nearest to $x = 1/2$ occurs at $x \approx 0.69098$, where the curve $f^{(2)}(x)$ intersects the line $y = x$. The upper right-hand corner of the square box in (b) is located at this point, and the center of the box is at $(1/2, 1/2)$. Note that if we reflect this square about the point $(1/2, 1/2)$, the shape of the reflected graph in the square box is nearly the same as it is in part (a) but on a smaller scale.

$f^{(2)}$ and change *(renormalize)* the value of r, we can compare $f^{(2)}$ to f. (See Chapter 12 for a discussion of scaling and renormalization in another context.)

This graphical comparison is meant only to be suggestive. A precise approach shows that if we continue the comparison of the higher-order iterates, for example, $f^{(4)}(x)$ to $f^{(2)}(x)$, etc., the superposition of functions converges to a universal function that is independent of the form of the original function $f(x)$.

Problem 6.7. Further determinations of the exponents α and δ

(a) Determine the appropriate scaling factor and superimpose f and the rescaled form of $f^{(2)}$ found in Figure 6.7.

(b) Use arguments similar to those discussed in the text and in Figure 6.7 and compare the behavior of $f^{(4)}(x, r = s_3)$ in the square about $x = 1/2$ with $f^{(2)}(x, r = s_2)$ in its square about $x = 1/2$. The size of the squares are determined by the unstable fixed point nearest to $x = 1/2$. Find the appropriate scaling factor and superimpose $f^{(2)}$ and the rescaled form of $f^{(4)}$. □

***Problem 6.8.** Other one-dimensional maps

It is easy to modify your programs to consider other one-dimensional maps. Determine the qualitative properties of the one-dimensional maps

$$f(x) = xe^{r(1-x)} \tag{6.12}$$

$$f(x) = r \sin \pi x. \tag{6.13}$$

Do they also exhibit the period doubling route to chaos? The map in (6.12) has been used by ecologists (cf. May) to study a population that is limited at high densities by the effect of epidemics. Although it is more complicated than (6.5), its advantage is that the population remains positive no matter what (positive) value is taken for the initial population. There are no restrictions on the maximum value of r, but if r becomes sufficiently large, x eventually becomes effectively zero. What is the behavior of the time series of (6.12) for $r = 1.5, 2$, and 2.7? Describe the qualitative behavior of $f(x)$. Does it have a maximum?

The sine map (6.13) with $0 < r \leq 1$ and $0 \leq x \leq 1$ has no special significance, except that it is nonlinear. If time permits, determine the approximate value of δ for both maps. What limits the accuracy of your determination of δ? □

The above qualitative arguments and numerical results suggest that the quantities α and δ are *universal*; that is, independent of the detailed form of $f(x)$. In contrast, the values of the accumulation point r_∞ and the constant C in (6.8) depend on the detailed form of $f(x)$. Feigenbaum has shown that the period doubling route to chaos and the values of δ and α are universal properties of maps that have a quadratic maximum; that is, $f'(x)_{|x=x_m} = 0$ and $f''(x)_{|x=x_m} < 0$.

Why is the universality of period doubling and the numbers δ and α more than a curiosity? The reason is that because this behavior is independent of the details, there might exist realistic systems whose underlying dynamics yield the same behavior as the logistic map. Of course, most physical systems are described by differential rather than difference equations. Can these systems exhibit period doubling behavior? Several workers (cf. Testa et al.) have constructed nonlinear RLC circuits driven by an oscillatory source voltage. The output voltage shows bifurcations, and the measured values of the exponents δ and α are consistent with the predictions of the logistic map.

Of more general interest is the nature of turbulence in fluid systems. Consider a stream of water flowing past several obstacles. We know that at low flow speeds, the water flows past obstacles in a regular and time-independent fashion called *laminar* flow. As the flow speed is increased (as measured by a dimensionless parameter called the Reynolds number) some swirls develop, but the motion is still time independent. As the flow speed is increased still further, the swirls break away and start moving downstream. The flow pattern as viewed from the bank becomes time-dependent. For still larger flow speeds, the flow pattern becomes very complex and looks random. We say that the flow pattern has made a transition from laminar flow to *turbulent* flow.

This qualitative description of the transition to chaos in fluid systems is superficially similar to the description of the logistic map. Can fluid systems be analyzed in terms of the simple models of the type we have discussed here? In a few instances such as turbulent convection in a heated saucepan, period doubling and other types of transitions to turbulence have been observed. The type of theory and analysis we have discussed has suggested new concepts and approaches, and the study of turbulent flow is a subject of much current interest.

6.5 Measuring Chaos

How do we know if a system is chaotic? The most important characteristic of chaos is *sensitivity to initial conditions*. In Problem 6.3, for example, we found that the trajectories starting from $x_0 = 0.5$ and $x_0 = 0.5001$ for $r = 0.91$ become very different after a small number of iterations. Because computers only store floating numbers to a certain number of digits, the implication of this result is that our numerical predictions of the trajectories of chaotic systems are restricted

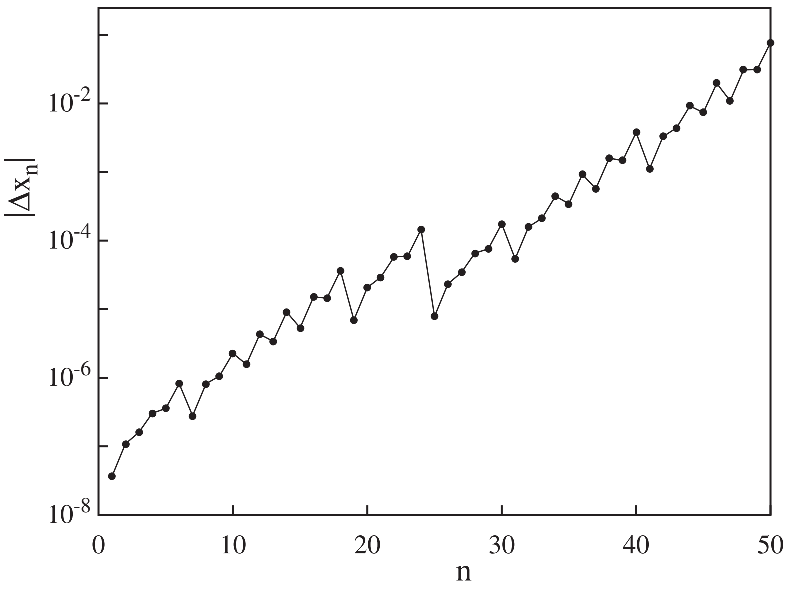


Figure 6.8: The evolution of the difference Δx_n between the trajectories of the logistic map at $r = 0.91$ for $x_0 = 0.5$ and $x_0 = 0.5001$. The separation between the two trajectories increases with n, the number of iterations, if n is not too large. (Note that $|\Delta x_1| \sim 10^{-8}$ and that the trend is not monotonic.)

to small time intervals. That is, sensitivity to initial conditions implies that even though the logistic map is deterministic, our ability to make numerical predictions of its trajectory is limited.

How can we quantify this lack of predictably? In general, if we start two identical dynamical systems from slightly different initial conditions, we expect that the difference between the trajectories will increase as a function of n. In Figure 6.8 we show a plot of the difference $|\Delta x_n|$ versus n for the same conditions as in Problem 6.3a. We see that, roughly speaking, $\ln|\Delta x_n|$ is a linearly increasing function of n. This result indicates that the separation between the trajectories grows exponentially if the system is chaotic. This divergence of the trajectories can be described by the *Lyapunov* exponent λ, which is defined by the relation

$$|\Delta x_n| = |\Delta x_0| e^{\lambda n} \tag{6.14}$$

where Δx_n is the difference between the trajectories at time n. If the Lyapunov exponent λ is positive, then nearby trajectories diverge exponentially. Chaotic behavior is characterized by the *exponential divergence of nearby trajectories*.

A naive way of measuring the Lyapunov exponent λ is to run the same dynamical system twice with slightly different initial conditions and measure the difference of the trajectories as a function of n. We used this method to generate Figure 6.8. Because the rate of separation of the trajectories might depend on the choice of x_0, a better method would be to compute the rate of separation for many values of x_0. This method would be tedious because we would have to fit the separation to (6.14) for each value of x_0 and then determine an average value of λ.

A more important limitation of the naive method is that because the trajectory is restricted to the unit interval, the separation $|\Delta x_n|$ ceases to increase when n becomes sufficiently large. Fortunately, there is a better way of determining λ. We take the natural logarithm of both sides

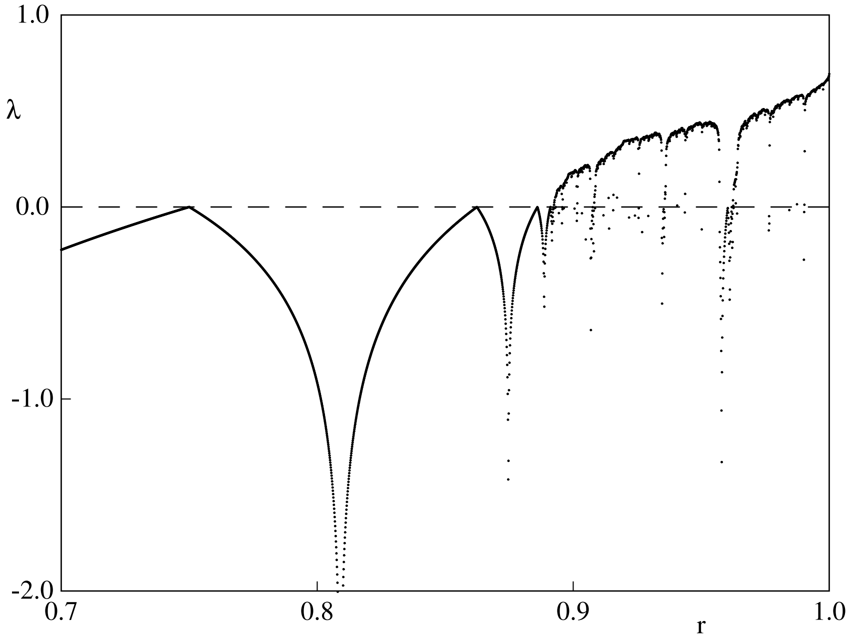


Figure 6.9: The Lyapunov exponent calculated using the method in (6.19) as a function of the control parameter r. Compare the behavior of λ to the bifurcation diagram in Figure 6.2. Note that $\lambda < 0$ for $r < 3/4$ and approaches zero at a period doubling bifurcation. A negative spike corresponds to a superstable trajectory. The onset of chaos is visible near $r = 0.892$, where λ first becomes positive. For $r \gtrsim 0.892$, λ generally increases except for dips below zero whenever a periodic window occurs, for example, the dip due to the period 3 window near $r = 0.96$. For each value of r, the first 1000 iterations were discarded, and 10^5 values of $\ln|f'(x_n)|$ were used to determine λ.

of (6.14), and write λ as

$$\lambda = \frac{1}{n} \ln\left|\frac{\Delta x_n}{\Delta x_0}\right|. \tag{6.15}$$

Because we want to use the data from the entire trajectory after the transient behavior has ended, we use the fact that

$$\frac{\Delta x_n}{\Delta x_0} = \frac{\Delta x_1}{\Delta x_0}\frac{\Delta x_2}{\Delta x_1}\cdots\frac{\Delta x_n}{\Delta x_{n-1}}. \tag{6.16}$$

Hence, we can express λ as

$$\lambda = \frac{1}{n}\sum_{i=0}^{n-1} \ln\left|\frac{\Delta x_{i+1}}{\Delta x_i}\right|. \tag{6.17}$$

The form (6.17) implies that we can interpret x_i for any i as the initial condition.

We see from (6.17) that the problem of computing λ has been reduced to finding the ratio $\Delta x_{i+1}/\Delta x_i$. Because we want to make the initial difference between the two trajectories as small as possible, we are interested in the limit $\Delta x_i \to 0$.

The idea of the more sophisticated procedure is to compute dx_{i+1}/dx_i from the equation of motion at the same time that the equation of motion is being iterated. We use the logistic map as an example. From (6.5) we have

$$\frac{dx_{i+1}}{dx_i} = f'(x_i) = 4r(1-2x_i). \tag{6.18}$$

We can consider x_i for any i as the initial condition and the ratio dx_{i+1}/dx_i as a measure of the rate of change of x_i. Hence, we can iterate the logistic map as before and use the values of x_i and the relation (6.18) to compute $f'(x_i) = dx_{i+1}/dx_i$ at each iteration. The Lyapunov exponent is given by

$$\lambda = \lim_{n\to\infty} \frac{1}{n} \sum_{i=0}^{n-1} \ln\left|f'(x_i)\right| \tag{6.19}$$

where we begin the sum in (6.19) after the transient behavior is finished. We have explicitly included the limit $n \to \infty$ in (6.19) to remind ourselves to choose n sufficiently large. Note that this procedure weights the points on the attractor correctly; that is, if a particular region of the attractor is not visited often by the trajectory, it does not contribute much to the sum in (6.19).

Problem 6.9. Lyapunov exponent for the logistic map

(a) Modify `IterateMapApp` to compute the Lyapunov exponent λ for the logistic map using the naive approach. Choose $r = 0.91$, $x_0 = 0.5$, and $\Delta x_0 = 10^{-6}$, and plot $\ln|\Delta x_n/\Delta x_0|$ versus n. What happens to $\ln|\Delta x_n/\Delta x_0|$ for large n? Determine λ for $r = 0.91$, $r = 0.97$, and $r = 1.0$. Does your result for λ for each value of r depend significantly on your choice of x_0 or Δx_0?

(b) Modify `BifurcateApp` to compute λ using the algorithm discussed in the text for $r = 0.76$ to $r = 1.0$ in steps of $\Delta r = 0.01$. What is the sign of λ if the system is not chaotic? Plot λ versus r and explain your results in terms of behavior of the bifurcation diagram shown in Figure 6.2. Compare your results for λ with those shown in Figure 6.9. How does the sign of λ correlate with the behavior of the system as seen in the bifurcation diagram? For what value of r is λ a maximum?

(c) In Problem 6.3b we saw that roundoff errors in the chaotic regime make the computation of individual trajectories meaningless. That is, if the system's behavior is chaotic, then small roundoff errors are amplified exponentially in time, and the actual numbers we compute for the trajectory starting from a given initial value are not "real." Repeat your calculation of λ for $r = 1$ by changing the roundoff error as you did in Problem 6.3b. Does your computed value of λ change? How meaningful is your computation of the Lyapunov exponent? We will encounter a similar question in Chapter 8 where we compute the trajectories of chaotic systems of many particles. We will find that although the "true" trajectories cannot be computed for long times, averages over the trajectories yield meaningful results. □

We have found that nearby trajectories diverge if $\lambda > 0$. For $\lambda < 0$, the two trajectories converge and the system is not chaotic. What happens for $\lambda = 0$? In this case we will see that the trajectories diverge algebraically; that is, as a power of n. In some cases a dynamical system is at the "edge of chaos" where the Lyapunov exponent vanishes. Such systems are said to exhibit weak chaos to distinguish their behavior from the strongly chaotic behavior ($\lambda > 0$) that we have been discussing.

If we define $z \equiv |\Delta x_n|/|\Delta x_0|$, then z will satisfy the differential equation

$$\frac{dz}{dn} = \lambda z. \tag{6.20}$$

For weak chaos we do not find an exponential divergence, but instead a divergence that is algebraic and is given by

$$\frac{dz}{dn} = \lambda_q z^q \tag{6.21}$$

where q is a parameter that needs to be determined. The solution to (6.21) is

$$z = [1 + (1-q)\lambda_q n]^{1/(1-q)} \tag{6.22}$$

which can be checked by substituting (6.22) into (6.21). In the limit $q \to 1$, we recover the usual exponential dependence.

We can determine the type of chaos using the crude approach of choosing a large number of initial values of x_0 and $x_0 + \Delta x_0$ and plotting the average of $\ln z$ versus n. If we do not obtain a straight line, then the system does not exhibit strong chaos. How can we check for the behavior shown in (6.22)? The easiest way is to plot the quantity

$$\frac{z^{1-q} - 1}{1-q} \tag{6.23}$$

versus n, which will equal $n\lambda_q$ if (6.22) is applicable. We explore these ideas in the following problem.

*__Problem 6.10.__ Measuring weak chaos

(a) Write a program that plots $\ln z$ if $q = 1$ or z_q if $q \neq 1$ as a function of n. Your program should have q, $|\Delta x_0|$, the number of seeds, and the number of iterations as input parameters. To compare with work by Añaños and Tsallis, use a variation of the logistic map given by

$$x_{n+1} = 1 - a x_n^2 \tag{6.24}$$

where $|x_n| \leq 1$ and $0 \leq a \leq 2$. The seeds x_0 should be equally spaced in the interval $|x_0| < 1$.

(b) Consider strong chaos at $a = 2$. Choose $q = 1$, 50 iterations, at least 1000 values of x_0, and $|\Delta x_0| = 10^{-6}$. Do you obtain a straight line for $\ln z$ versus n? Does z_n eventually stop increasing as a function of n? If so why? Try $|\Delta x_0| = 10^{-12}$. How do your results differ and how are they the same? Also iterate Δx directly:

$$\Delta x_{n+1} = x_{n+1} - \tilde{x}_{n+1} = -a(x_n^2 - \tilde{x}_n^2) = -a(x_n - \tilde{x}_n)(x_n + \tilde{x}_n) = -a\Delta x_n(x_n + \tilde{x}_n) \tag{6.25}$$

where x_n is the iterate starting at x_0, and $\tilde{x}_n$ is the iterate starting at $x_0 + \Delta x_0$. Show that straight lines are not obtained for your plot if $q \neq 1$.

(c) The edge of chaos for this map is at $a = 1.401155189$. Repeat part (a) for this value of a and various values of q. Simulations with 10^5 values of x_0 points show that linear behavior is obtained for $q \approx 0.36$. □

A system of fixed energy (and number of particles and volume) has an equal probability of being in any microstate specified by the positions and velocities of the particles (see Sec 15.2). One way of measuring the ability of a system to be in any state is to measure its entropy defined by

$$S = -\sum_i p_i \ln p_i, \tag{6.26}$$

where the sum is over all states, and p_i is the probability or relative frequency of being in the ith state. For example, if the system is always in only one state, then $S = 0$, the smallest possible entropy. If the system explores all states equally, then $S = \ln \Omega$, where Ω is the number of possible states. (You can show this result by letting $p_i = 1/\Omega$.)

*__Problem 6.11.__ Entropy of the logistic map

(a) Write a program to compute S for the logistic map. Divide the interval $[0,1]$ into bins or subintervals of width $\Delta x = 0.01$ and determine the relative number of times the trajectory falls into each bin. At each value of r in the range $0.7 \leq r \leq 1$, the map should be iterated for a fixed number of steps, for example, $n = 1000$. Choose $\Delta x = 0.01$. What happens to the entropy when the trajectory is chaotic?

(b) Repeat part (a) with $n = 10{,}000$. For what values of r does the entropy change significantly? Decrease Δx to 0.001 and repeat. Does this decrease make a difference?

(c) Plot p_i as a function of x for $r = 1$. For what value(s) of x is the plot a maximum? □

We can also measure the (generalized) entropy as a function of time. As we will see in Problem 6.12, $S(n)$ for strong chaos increases linearly with n until all the possible states are visited. However, for weak chaos this behavior is not found. In the latter case we can generalize the entropy to a q-dependent function defined by

$$S_q = \frac{1 - \sum_i p_i^q}{q - 1}. \tag{6.27}$$

In the limit $q \to 1$, $S_q \to S$. The following problem discusses measuring the entropy for the same system as in Problem 6.10.

*__Problem 6.12.__ Entropy of weak and strong chaotic systems

(a) Write a program that iterates the map (6.24) and plots S if $q = 1$, or plots S_q, if $q \neq 1$, as a function of n. The input parameters should be q, the number of bins, the number of random seeds in a single bin, and n, the number of iterations. At each iteration compute the entropy. Then average S over the randomly chosen values of the seeds.

(b) Consider strong chaos at $a = 2$. Choose $q = 1$, $n = 20$, $\Delta x \leq 0.001$, and ten randomly chosen seeds per bin. Do you obtain a straight line for S versus n? Does the curve eventually stop growing? If you decrease Δx, how do your results differ and how are they the same? Show that S is not a linear function of n if $q \neq 1$.

(c) Repeat part (a) with $a = 1.401155189$ and various values of q. Simulations with 10^5 bins show that linear behavior is obtained for $q \approx 0.36$, the same value as for the measurements in Problem 6.10. □

6.6 *Controlling Chaos

The dream of classical physics was that if the initial conditions and all the forces acting on a system were known, then we could predict the future with as much precision as we desire.

The existence of chaos has shattered that dream. However, if a system is chaotic, we can still control its behavior with small, but carefully chosen, perturbations of the system. We will illustrate the method for the logistic map. The application of the method to other one-dimensional systems is straightforward, but the extension to higher-dimensional systems is more complicated (cf. Ott, Lai).

Suppose that we want the trajectory to be periodic even though the parameter r is in the chaotic regime. How can we make the trajectory have periodic behavior without drastically changing r or imposing an external perturbation that is so large that the internal dynamics of the map become irrelevant? The key idea is that for any value of r in the chaotic regime, there is an infinite number of trajectories that have unstable periods. This property of the chaotic regime means that if we choose the value of the seed x_0 to be precisely equal to a point on an unstable trajectory with period p, the subsequent trajectory will have this period. However, if we choose a value of x_0 that differs ever so slightly from this special value, the trajectory will not be periodic. Our goal is to make slight perturbations to the system to keep it on the desired unstable periodic trajectory.

The first step is to find the values of $x(i)$, $i = 1$ to p, that constitute the unstable periodic trajectory. It is an interesting numerical problem to find the values of $x(i)$, and we consider this problem first. To find a fixed point of the map $f^{(p)}$, we need to find the value of x^* such that

$$g^{(p)}(x^*) \equiv f^{(p)}(x^*) - x^* = 0. \tag{6.28}$$

The algorithms for finding the solution to (6.28) are called *root-finding* algorithms. You might have heard of Newton's method, which we describe in Appendix 6B. Here we use the simplest root-finding algorithm, the *bisection* method. The algorithm works as follows:

(i) Choose two values x_{left} and x_{right}, with $x_{\text{left}} < x_{\text{right}}$, such that the product $g^{(p)}(x_{\text{left}})g^{(p)}(x_{\text{right}}) < 0$. Because this product is negative, there must be a value of x such that $g^{(p)}(x) = 0$ in the interval $\left[x_{\text{left}}, x_{\text{right}}\right]$

(ii) Choose the midpoint, $x_{\text{mid}} = x_{\text{left}} + \frac{1}{2}(x_{\text{right}} - x_{\text{left}}) = \frac{1}{2}(x_{\text{left}} + x_{\text{right}})$, as the guess for x^*.

(iii) If $g^{(p)}(x_{\text{mid}})$ has the same sign as $g^{(p)}(x_{\text{left}})$, then replace x_{left} by x_{mid}; otherwise, replace x_{right} by x_{mid}. The interval for the location of the root is now reduced.

(iv) Repeat steps (ii) and (iii) until the desired level of precision is achieved.

The following program implements this algorithm for the logistic map. An alternative implementation named `FixedPointApp` that does not use recursion is not listed, but is available in the ch06 package. One possible problem is that some of the roots of $g^{(p)}(x) = 0$ are also roots of $g^{(p')}(x) = 0$ for p' equal to a factor of p. (For example, if $p = 6$, 2 and 3 are factors.) As p increases, it might become more difficult to find a root that is part of a period p trajectory and not part of a period p' trajectory.

Listing 6.4: The `RecursiveFixedPointApp` program finds stable and unstable periodic trajectories with the given period using the bisection root- finding algorithm

```
package org.opensourcephysics.sip.ch06;
import org.opensourcephysics.controls.*;

public class RecursiveFixedPointApp extends AbstractCalculation {
   double r; // control parameter
```

```
int period;
double xleft, xright;
double gleft, gright;

public void reset() {
   control.setValue("r", 0.8);                 // control parameter r
   control.setValue("period", 2);              // period
   control.setValue("epsilon", 0.0000001); // desired precision
   control.setValue("xleft", 0.01);            // guess for xleft
   control.setValue("xright", 0.99);           // guess for xright
}

public void calculate() {
   double epsilon = control.getDouble("epsilon"); // desired precision
   r = control.getDouble("r");
   period = control.getInt("period");
   xleft = control.getDouble("xleft");
   xright = control.getDouble("xright");
   gleft = map(xleft, r, period)-xleft;
   gright = map(xright, r, period)-xright;
   if(gleft*gright<0) {
      while(Math.abs(xleft-xright)>epsilon) {
         bisection();
      }
      double x = 0.5*(xleft+xright);
      control.println("explicit search for period "+period+" behavior");
      control.println(0+"\t"+x); // result
      for(int i = 1;i<=2*period+1;i++) {
         x = map(x, r, 1);
         control.println(i+"\t"+x);
      }
   } else {
      control.println("range does not enclose a root");
   }
}

public void bisection() {
   // midpoint between xleft and xright
   double xmid = 0.5*(xleft+xright);
   double gmid = map(xmid, r, period)-xmid;
   if(gmid*gleft>0) {
      xleft = xmid;  // change xleft
      gleft = gmid;
   } else {
      xright = xmid; // change xright
      gright = gmid;
   }
}

double map(double x, double r, double period) {
   if(period>1) {
      double y = map(x, r, period-1);
      return 4*r*y*(1-y);
```

```
         } else {
           return 4*r*x*(1-x);
         }
      }

      public static void main(String[] args) {
         CalculationControl.createApp(new RecursiveFixedPointApp());
      }
   }
```

Problem 6.13. Unstable periodic trajectories for the logistic map

(a) Test RecursiveFixedPointApp for values of r for which the logistic map has a stable period with $p = 1$ and $p = 2$. Set the desired precision ϵ equal to 10^{-7}. Initially use $x_{\text{left}} = 0.01$ and $x_{\text{right}} = 0.99$. Calculate the stable attractor analytically and compare the results of your program with the analytic results.

(b) Set $r = 0.95$ and find the periodic trajectories for $p = 1$, 2, 5, 6, 7, 12, 13, and 19.

(c) Modify RecursiveFixedPointApp so that n_b, the number of bisections needed to obtain the unstable trajectory, is listed. Choose three of the cases considered in part (b) and compute n_b for the precision $\epsilon = 0.01$, 0.001, 0.0001, and 0.00001. Determine the functional dependence of n_b on ϵ. □

Now that we know how to find the values of the unstable periodic trajectories, we discuss an algorithm for stabilizing this period. Suppose that we wish to stabilize the unstable trajectory of period p for a choice of $r = r_0$. The idea is to make small adjustments of $r = r_0 + \Delta r$ at each iteration so that the difference between the actual trajectory and the target periodic trajectory is small. If the actual trajectory is x_n and we wish the trajectory to be at $x(i)$, we make the next iterate x_{n+1} equal to $x(i+1)$ by expanding the difference $x_{n+1} - x(i+1)$ in a Taylor series and setting the difference to zero to first order. We have $x_{n+1} - x(i+1) = f(x_n, r) - f(x(i), r_0)$. If we expand $f(x_n, r)$ about $(x(i), r_0)$, we have to first order

$$x_{n+1} - x(i+1) = \frac{\partial f(x,r)}{\partial x}\,[x_n - x(i)] + \frac{\partial f(x,r)}{\partial r}\Delta r = 0. \tag{6.29}$$

The partial derivatives in (6.29) are evaluated at $x = x(i)$ and $r = r_0$. The result is

$$4r_0\big[1 - 2x(i)\big]\big[x_n - x(i)\big] + 4x(i)\big[1 - x(i)\big]\Delta r = 0 \tag{6.30}$$

and the solution of (6.30) for Δr can be written as

$$\Delta r = -r_0\frac{\big[1 - 2x(i)\big]\big[x_n - x(i)\big]}{x(i)\big[1 - x(i)\big]}. \tag{6.31}$$

The procedure is to iterate the logistic map at $r = r_0$ until x_n is sufficiently close to an $x(i)$. The nature of chaotic systems is that the trajectory is guaranteed to eventually come close to the desired unstable trajectory. Then we use (6.31) to change the value of r so that the next iteration is closer to $x(i+1)$. We summarize the algorithm for controlling chaos as follows:

1. Find the unstable periodic trajectory $x(1), x(2) \ldots x(p)$ for the desired value of r_0.

2. Iterate the map with $r = r_0$ until x_n is within ϵ of $x(i)$. Then use (6.31) to determine r.
3. Turn off the control by setting $r = r_0$. □

Problem 6.14. Controlling chaos

(a) Write a program that allows the user to turn the control on and off. The trajectory can be seen by plotting x_n versus n. The program should incorporate as input the desired unstable periodic trajectory $x(i)$, the period p, the value of r_0, and the parameter ϵ.

(b) Test your program with $r_0 = 0.95$ and the periods $p = 1$, 5, and 13. Use $\epsilon = 0.02$.

(c) Modify your program so that the values of r as well as the values of x_n are shown. How does r change if we vary ϵ? Try $\epsilon = 0.05$, 0.01, and 0.005.

(d) Add a method to compute n_ϵ, the number of iterations necessary for the trajectory x_n to be within ϵ of $x(1)$ when the control is on. Find $\langle n_\epsilon \rangle$, the average value of n_ϵ, by starting with 100 random values of x_0. Compute $\langle n_\epsilon \rangle$ as a function of ϵ for $\delta = 0.05$, 0.005, 0.0005, and 0.00005. What is the functional dependence of $\langle n_\epsilon \rangle$ on ϵ? □

6.7 Higher-Dimensional Models

So far we have discussed the logistic map as a mathematical model that has some remarkable properties and produces some interesting computer graphics. In this section we discuss some two- and three-dimensional systems that also might seem to have little to do with realistic physical systems. However, as we will see in Sections 6.8 and 6.9, similar behavior is found in realistic physical systems under the appropriate conditions.

We begin with a two-dimensional map and consider the sequence of points (x_n, y_n) generated by

$$x_{n+1} = y_n + 1 - a{x_n}^2 \tag{6.32a}$$
$$y_{n+1} = bx_n. \tag{6.32b}$$

The map (6.32) was proposed by Hénon who was motivated by the relevance of this dynamical system to the behavior of asteroids and satellites.

Problem 6.15. The Hénon map

(a) Write a program to iterate (6.32) for $a = 1.4$ and $b = 0.3$ and plot 10^4 iterations starting from $x_0 = 0, y_0 = 0$. Make sure you compute the new value of y using the old value of x and not the new value of x. Do not plot the initial transient. Look at the trajectory in the region defined by $|x| \leq 1.5$ and $|y| \leq 0.45$. Make a similar plot beginning from the second initial condition, $x_0 = 0.63135448, y_0 = 0.18940634$. Compare the shape of the two plots. Is the shape of the two curves independent of the initial conditions?

(b) Increase the scale of your plot so that all points in the region $0.50 \leq x \leq 0.75$ and $0.15 \leq y \leq 0.21$ are shown. Begin from the second initial condition and increase the number of computed points to 10^5. Then make another plot showing all points in the region $0.62 \leq x \leq 0.64$ and $0.185 \leq y \leq 0.191$. If time permits, make an additional enlargement and plot all points within the box defined by $0.6305 \leq x \leq 0.6325$ and $0.1889 \leq y \leq 0.1895$. You will have to increase the number of computed points to order 10^6. What is the structure of the curves within each box? Does the attractor appear to have a similar structure on smaller and smaller length scales? The region of points from which the points cannot escape is the basin of the Hénon attractor. The attractor is the set of points to which all points in the basin are attracted. That is, two trajectories that begin from different conditions will eventually lie on the attractor.

(c) Determine if the system is chaotic; that is, sensitive to initial conditions. Start two points very close to each other and watch their trajectories for a fixed time. Choose different colors for the two trajectories.

(d)* It is straightforward in principle to extend the method for computing the Lyapunov exponent that we used for a one-dimensional map to higher-dimensional maps. The idea is to linearize the difference (or differential) equations and replace dx_n by the corresponding vector quantity $d\mathbf{r}_n$. This generalization yields the Lyapunov exponent corresponding to the divergence along the fastest growing direction. If a system has f degrees of freedom, it has a set of f Lyapunov exponents. A method for computing all f exponents is discussed in Project 6.24. □

One of the earliest indications of chaotic behavior was in an atmospheric model developed by Lorenz. His goal was to describe the motion of a fluid layer that is heated from below. The result is convective rolls, where the warm fluid at the bottom rises, cools off at the top, and then falls down later. Lorenz simplified the description by restricting the motion to two spatial dimensions. This situation has been realized experimentally and is known as a Rayleigh–Benard cell. The equations that Lorenz obtained are

$$\frac{dx}{dt} = -\sigma x + \sigma y \tag{6.33a}$$

$$\frac{dy}{dt} = -xz + rx - y \tag{6.33b}$$

$$\frac{dz}{dt} = xy - bz \tag{6.33c}$$

where x is a measure of the fluid flow velocity circulating around the cell, y is a measure of the temperature difference between the rising and falling fluid regions, and z is a measure of the difference in the temperature profile between the bottom and the top from the normal equilibrium temperature profile. The dimensionless parameters σ, r, and b are determined by various fluid properties, the size of the Raleigh-Benard cell, and the temperature difference in the cell. Note that the variables x, y, and z have nothing to do with the spatial coordinates, but are measures of the state of the system. Although it is not expected that you will understand the relation of the Lorenz equations to convection, we have included these equations here to reinforce the idea that simple sets of equations can exhibit chaotic behavior.

LorenzApp displays the solution to (6.33) using the Open Source Physics 3D drawing framework and is available in the ch06 package. To make three-dimensional plots, we use the Display3DFrame class; the only argument of its constructor is the title for the plot. The following code fragment sets up the plot.

```
Display3DFrame frame = new Display3DFrame("Lorenz attractor");
Lorenz lorenz = new Lorenz();
frame.setPreferredMinMax(-15.0, 15.0, -15.0, 15.0, 0.0, 50.0);
frame.setDecorationType(VisualizationHints.DECORATION_AXES);
frame.addElement(lorenz);  // lorenz is a 3D element
```

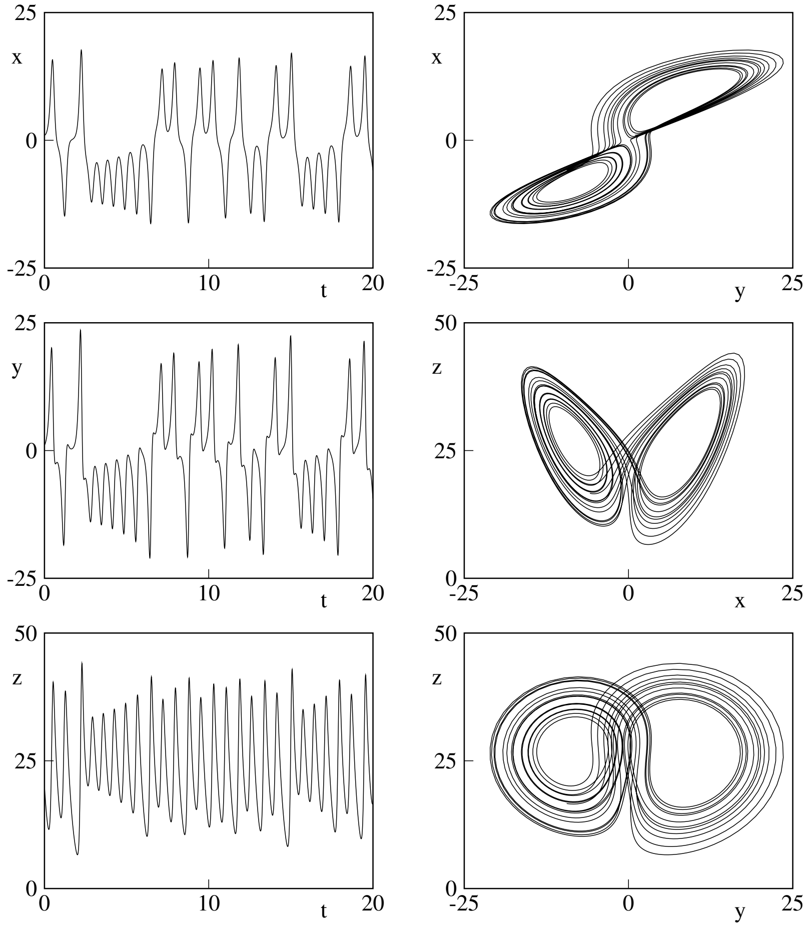


Figure 6.10: A trajectory of the Lorenz model with $\sigma = 10$, $b = 8/3$, and $r = 28$ and the initial condition $x_0 = 1, y_0 = 1, z_0 = 20$. A time interval of $t = 20$ is shown with points plotted at intervals of 0.01. The fourth-order Runge–Kutta algorithm was used with $\Delta t = 0.0025$.

Housekeeping methods such as `reset` and `initialize` are similar to methods in other simulations and are not shown.

The class `Lorenz` draws the attractor in the three-dimensional (x, y, z) space defined by (6.33). The state of the system is shown as a red ball in this 3D space, and the state's trajectory is shown as a trail. An easy way to show the time evolution is to extend the 3D `Group` class and create the ball and the trail inside the group. When points are added to the group, the trail is extended and the position of the ball is set. The `Lorenz` class imports `org.opensourcephysics.display3d.simple3d.*`. The ball and trail are then instantiated and added to the group as follows:

```
public class Lorenz extends Group implements ODE {
   ElementEllipsoid ball = new ElementEllipsoid();
   ElementTrail trail = new ElementTrail();
   addElement(trail);      // adds trace to Lorenz group
   addElement(ball);       // adds ball to Lorenz group
```

```
    ...
}
```

The properties of the `ball` and `trail` objects are set by

```
ball.setSizeXYZ(1, 1, 1);        // sets size of ball in world coordinates
ball.getStyle().setFillColor(java.awt.Color.RED);
```

To plot each part of the trajectory through state space, we use the method `trail.addPoint(x,y,z)` to add to the trail and `ball.setXYZ(x,y,z)` to show the current state. The user can project onto two dimensions using the frame's menu or rotate the three-dimensional plot using the mouse because these capabilities are built into the frame. The `getRate` and `getState` methods model (6.33) by implementing the `ODE` interface.

Problem 6.16. The Lorenz model

(a) Use a Runge–Kutta algorithm such as `RK4` or `RK45` (see Appendix 3A) to obtain a numerical solution of the Lorenz equations (6.33). Generate three-dimensional plots using `Display3DFrame`. Explore the basin of the attractor with $\sigma = 10$, $b = 8/3$, and $r = 28$.

(b) Determine qualitatively the sensitivity to initial conditions. Start two points very close to each other and watch their trajectories for approximately 10^4 time steps.

(c) Let z_m denote the value of z where z is a relative maximum for the mth time. You can determine the value of z_m by finding the average of the two values of z when the right-hand side of (6.33) changes sign. Plot z_{m+1} versus z_m and describe what you find. This procedure is one way that a continuous system can be mapped onto a discrete map. What is the slope of the z_{m+1} versus z_m curve? Is its magnitude always greater than unity? If so, then this behavior is an indication of chaos. Why? □

The application of the Lorenz equations to weather prediction has led to a popular metaphor known as the *butterfly effect*. This metaphor is made even more meaningful by inspection of Figure 6.10. The "butterfly effect" is often ascribed to Lorenz (see Hilborn). In a 1963 paper he remarked that:

> "One meteorologist remarked that if the theory were correct, one flap of a seagull's wings would be enough to alter the course of the weather forever."

By 1972, the seagull had evolved into the more poetic butterfly and the title of his talk was "Predictability: Does the flap of a butterfly's wings in Brazil set off a tornado in Texas?"

6.8 Forced Damped Pendulum

We now consider the dynamics of nonlinear systems described by classical mechanics. The general problem in classical mechanics is the determination of the positions and velocities of a system of particles subjected to certain forces. For example, we considered in Chapter 5 the celestial two-body problem and were able to predict the motion at any time. We will find that we cannot make long-time predictions for the trajectories of nonlinear classical systems when these systems exhibit chaos.

A familiar example of a nonlinear mechanical system is the simple pendulum (see Chapter 3). To make its dynamics more interesting, we assume that there is a linear damping term present and that the pivot is forced to move vertically up and down. Newton's second law for this system is (cf. McLaughlin or Percival and Richards)

$$\frac{d^2\theta}{dt^2} = -\gamma\frac{d\theta}{dt} - [{\omega_0}^2 + 2A\cos\omega t]\sin\theta \tag{6.34}$$

where θ is the angle the pendulum makes with the vertical axis, γ is the damping coefficient, ${\omega_0}^2 = g/L$ is the natural frequency of the pendulum, and ω and A are the frequency and amplitude of the external force. The effect of the vertical acceleration of the pivot is equivalent to a time-dependent gravitational field, because we can write the total vertical force due to gravity, $-mg$ plus the pivot motion $f(t)$ as $-mg(t)$, where $g(t) \equiv g - f(t)/m$.

How do we expect the driven, damped simple pendulum to behave? Because there is damping present, we expect that if there is no external force, the pendulum would come to rest. That is, $(x = 0, v = 0)$ is a stable attractor. As A is increased from zero, this attractor remains stable for sufficiently small A. At a value of A equal to A_c, this attractor becomes unstable. How does the driven nonlinear oscillator behave as we increase A?

It is difficult to determine whether the pendulum has some kind of underlying periodic behavior by plotting only its position or even plotting its trajectory in phase space. We expect that if it does, the period will be related to the period of the external time-dependent force. Thus, we analyze the motion by plotting a point in phase space after every cycle of the external force. Such a phase space plot is called a *Poincaré map*. Hence, we will plot $d\theta/dt$ versus θ for values of t equal to nT for n equal to $1, 2, 3, \ldots$. If the system has a period T, then the Poincaré map consists of a single point. If the period of the system is nT, there will be n points.

PoincareApp uses the fourth-order Runge–Kutta algorithm to compute $\theta(t)$ and the angular velocity $d\theta(t)/dt$ for the pendulum described by (6.34). This equation is modeled in the DampedDrivenPendulum class, but is not shown here because it is similar to other ODE implementations. A phase diagram for $d\theta(t)/dt$ versus $\theta(t)$ is shown in one frame. In the other frame, the Poincaré map is represented by drawing a small box at the point $(\theta, d\theta/dt)$ at time $t = nT$. If the system has period 1; that is, if the same values of $(\theta, d\theta/dt)$ are drawn at $t = nT$, we would see only one box; otherwise, we would see several boxes. Because the first few values of $(\theta, d\theta/dt)$ show the transient behavior, it is desirable to clear the display and draw a new Poincaré map without changing A, θ, or $d\theta/dt$.

Listing 6.5: PoincareApp plots a phase diagram and a Poincaré map for the damped driven pendulum.

```
package org.opensourcephysics.sip.ch06;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.frames.PlotFrame;
import org.opensourcephysics.numerics.RK4;

public class PoincareApp extends AbstractSimulation {
   final static double PI = Math.PI; // defined for brevity
   PlotFrame phaseSpace = new PlotFrame("theta", "angular velocity",
                     "Phase space plot");
   PlotFrame poincare = new PlotFrame("theta", "angular velocity",
                     "Poincare plot");
   int nstep = 100;                 // # iterations between Poincare plot
   DampedDrivenPendulum pendulum = new DampedDrivenPendulum();
   RK4 odeMethod = new RK4(pendulum);

   public PoincareApp() {
      // angular frequency of external force equals two and hence
      // period of external force equals pi
      odeMethod.setStepSize(PI/nstep); // dt = PI/nsteps
      phaseSpace.setMarkerShape(0, 6); // second argument indicates a pixel
      // smaller size gives better resolution
      poincare.setMarkerSize(0, 2);
      poincare.setMarkerColor(0, java.awt.Color.RED);
      phaseSpace.setMessage("t = "+0);
   }

   public void reset() {
      control.setValue("theta", 0.2);
      control.setValue("angular velocity", 0.6);
      control.setValue("gamma", 0.2); // damping constant
      control.setValue("A", 0.85);    // amplitude
   }

   public void doStep() {
      double state[] = pendulum.getState();
      for(int istep = 0;istep<nstep;istep++) {
         odeMethod.step();
         if(state[0]>PI) {
            state[0] = state[0]-2.0*PI;
         } else if(state[0]<-PI) {
            state[0] = state[0]+2*PI;
         }
         phaseSpace.append(0, state[0], state[1]);
      }
      poincare.append(0, state[0], state[1]);
      phaseSpace.setMessage("t = "+decimalFormat.format(state[2]));
      poincare.setMessage("t = "+decimalFormat.format(state[2]));
      if(phaseSpace.isShowing()) {
         phaseSpace.render();
      }
      if(poincare.isShowing()) {
         poincare.render();
      }
   }

   public void initialize() {
      double theta = control.getDouble("theta");          // initial angle
      // initial angular velocity
      double omega = control.getDouble("angular velocity");
      pendulum.gamma = control.getDouble("gamma"); // damping constant
      // amplitude of external force
      pendulum.A = control.getDouble("A");
      pendulum.initializeState(new double[] {theta, omega, 0});
      clear();
   }

   public void clear() {
      phaseSpace.clearData();
      poincare.clearData();
      phaseSpace.render();
      poincare.render();
   }

   public static void main(String[] args) {
      SimulationControl control = SimulationControl.createApp(new PoincareApp());
      control.addButton("clear", "Clear");
   }
}
```

Problem 6.17. Dynamics of a driven, damped simple pendulum

(a) Use PoincareApp to simulate the driven, damped simple pendulum. In the program, $\omega = 2$ so that the period T of the external force equals π. The program also assumes that $\omega_0 = 1$. Use $\gamma = 0.2$ and $A = 0.85$ and compute the phase space trajectory. After the transient, how many points do you see in the Poincaré plot? What is the period of the pendulum? Vary the initial values of θ and $d\theta/dt$. Is the attractor independent of the initial conditions? Remember to ignore the transient behavior.

(b) Modify PoincareApp so that it plots θ and $d\theta/dt$ as a function of t. Describe the qualitative relation between the Poincaré plot, the phase space plot, and the t dependence of θ and $d\theta/dt$.

(c) The amplitude A plays the role of the control parameter for the dynamics of the system. Use the behavior of the Poincaré plot to find the value $A = A_c$ at which the $(0,0)$ attractor becomes unstable. Start with $A = 0.1$ and continue increasing A until the $(0,0)$ attractor becomes unstable.

(d) Find the period for $A = 0.1$, 0.25, 0.5, 0.7, 0.75, 0.85, 0.95, 1.00, 1.02, 1.031, 1.033, 1.036, and 1.05. Note that for small A, the period of the oscillator is twice that of the external force. The steady state period is 2π for $A_c < A < 0.71$, π for $0.72 < A < 0.79$, and then 2π again.

(e) The first period doubling occurs for $A \approx 0.79$. Find the approximate values of A for further period doubling and use these values of A to compute the exponent δ defined by (6.10). Compare your result for δ with the result found for the one-dimensional logistic map. Are your results consistent with those that you found for the logistic map? An analysis of this system can be found in the article by McLaughlin.

(f) Sometimes a trajectory does not approach a steady state even after a very long time, but a slight perturbation causes the trajectory to move quickly onto a steady state attractor. Consider $A = 0.62$ and the initial condition $(\theta = 0.3, d\theta/dt = 0.3)$. Describe the behavior of the trajectory in phase space. During the simulation, change θ by 0.1. Does the trajectory move onto a steady state trajectory? Do similar simulations for other values of A and other initial conditions.

(g) Repeat the calculations of parts (b)–(d) for $\gamma = 0.05$. What can you conclude about the effect of damping?

(h) Replace the fourth-order Runge–Kutta algorithm by the lower-order Euler-Richardson algorithm. Which algorithm gives the better trade-off between accuracy and speed? □

Problem 6.18. The basin of an attractor

(a) For $\gamma = 0.2$ and $A > 0.79$, the pendulum rotates clockwise or counterclockwise in the steady state. Each of these two rotations is an attractor. The set of initial conditions that lead to a particular attractor is called the basin of the attractor. Modify PoincareApp so that the program draws the basin of the attractor with $d\theta/dt > 0$. For example, your program might simulate the motion for about 20 periods and then determine the sign of $d\theta/dt$. If $d\theta/dt > 0$ in the steady state, then the program plots a point in phase space at the coordinates of the initial condition. The program repeats this process for many initial conditions. Describe the basin of attraction for $A = 0.85$ and increments of the initial values of θ and $d\theta/dt$ equal to $\pi/10$.

(b) Repeat part (a) using increments of the initial values of θ and $d\theta/dt$ equal to $\pi/20$ or as small as possible given your computer resources. Does the boundary of the basin of attraction appear smooth or rough? Is the basin of the attractor a single region or is it disconnected into more than one piece?

(c) Repeat parts (a) and (b) for other values of A, including values near the onset of chaos and in the chaotic regime. Is there a qualitative difference between the basins of periodic and chaotic attractors? For example, can you always distinguish the boundaries of the basin? □

6.9 *Hamiltonian Chaos

Hamiltonian systems are a very important class of dynamical systems. The most familiar are mechanical systems without friction, and the most important of these is the solar system. The linear harmonic oscillator and the simple pendulum that we considered in Chapter 3 are two simple examples. Many other systems can be included in the Hamiltonian framework, for example, the motion of charged particles in electric and magnetic fields and ray optics. The Hamiltonian dynamics of charged particles is particularly relevant to confinement issues in particle accelerators, storage rings, and plasmas. In each case a function of all the coordinates and momenta called the Hamiltonian is formed. For many systems this function can be identified with the total energy. The Hamiltonian for a particle in a potential $V(x,y,z)$ is

$$H = \frac{1}{2m}({p_x}^2 + {p_y}^2 + {p_z}^2) + V(x,y,z). \tag{6.35}$$

Typically we write (6.35) using the notation

$$H = \sum_i \frac{p_i^2}{2m} + V(\{q_i\}) \tag{6.36}$$

where $p_1 \equiv p_x$, $q_1 \equiv x$, etc. This notation emphasizes that the p_i and the q_i are generalized coordinates. For example, in some systems p can represent the angular momentum and q can represent an angle. For a system of N particles in three dimensions, the sum in (6.36) runs from 1 to $3N$, where $3N$ is the number of degrees of freedom.

The methods for constructing the generalized momenta and the Hamiltonian are described in standard classical mechanics texts. The time dependence of the generalized momenta and coordinates is given by

$$\dot{p}_i \equiv \frac{dp_i}{dt} = -\frac{\partial H}{\partial q_i} \tag{6.37a}$$

$$\dot{q}_i \equiv \frac{dq_i}{dt} = \frac{\partial H}{\partial p_i}. \tag{6.37b}$$

Check that (6.37) leads to the usual form of Newton's second law by considering the simple example of a single particle in a potential $U(x)$, where $q = x$ and $p = m\dot{x}$.

As we found in Chapter 4, an important property of conservative systems is preservation of areas in phase space. Consider a set of initial conditions of a dynamical system that form a closed surface in phase space. For example, if phase space is two-dimensional, this surface would be a one-dimensional loop. As time evolves, this surface in phase space will typically change its shape. For Hamiltonian systems, the volume (area for a two-dimensional phase space) enclosed by this surface remains constant in time. For dissipative systems, this volume will decrease, and hence dissipative systems are not described by a Hamiltonian. One consequence of the constant phase space volume is that Hamiltonian systems do not have phase space attractors.

In general, the motion of Hamiltonian systems is very complex. In some systems the motion is regular, and there is a constant of the motion (a quantity that does not change with time) for each degree of freedom. Such a system is said to be *integrable*. For time-independent systems, an obvious constant of the motion is the total energy. The total momentum and angular momentum are other examples. There may be others as well. If there are more degrees of freedom than constants of the motion, then the system can be chaotic. When the number of degrees of freedom becomes large, the possibility of chaotic behavior becomes more likely. An important example that we will consider in Chapter 8 is a system of interacting particles. Their chaotic motion is essential for the system to be described by the methods of statistical mechanics.

For regular motion the change in shape of a closed surface in phase space is uninteresting. For chaotic motion, nearby trajectories must exponentially diverge from each other, but are confined to a finite region of phase space. Hence, there will be local stretching of the surface accompanied by repeated folding to ensure confinement. There is another class of systems whose behavior is in between; that is, the system behaves regularly for some initial conditions and chaotically for others. We will study these *mixed* systems in this section.

Consider the Hamiltonian for a system of N particles. If the system is integrable, there are $3N$ constants of the motion. It is natural to identify the generalized momenta with these constants. The coordinates that are associated with each of these constants will vary linearly with time. If the system is confined in phase space, then the coordinates must be periodic. If we have just one coordinate, we can think of the motion as a point moving on a circle in phase space. In two dimensions the motion is a point moving in two circles at once; that is, a point moving on the surface of a torus. In three dimensions we can imagine a generalized torus with three circles, and so on. If the period of motion along each circle is a rational fraction of the period of all the other circles, then the torus is called a resonant torus, and the motion in phase space is periodic. If the periods are not rational fractions of each other, then the torus is called nonresonant.

If we take an integrable Hamiltonian and change it slightly, what happens to these tori? A partial answer is given by a theorem due to Kolmogorov, Arnold, and Moser (KAM), which states that, under certain circumstances, the tori will remain. When the perturbation of the Hamiltonian becomes sufficiently large, these KAM tori are destroyed.

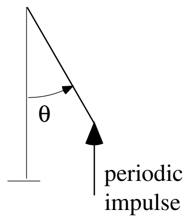


Figure 6.11: Model of a kicked rotor consisting of a rigid rod with moment of inertia I. Gravity and friction at the pivot is ignored. The motion of the rotor is given by the standard map in (6.39).

To understand the basic ideas associated with mixed systems, we consider a simple model of a rotor known as the *standard* map (see Figure 6.11). The rod has a moment of inertia I and length L and is fastened at one end to a frictionless pivot. The other end is subjected to a vertical periodic impulsive force of strength k/L applied at time $t = 0$, τ, 2τ,... Gravity is ignored. The motion of the rotor can be described by the angle θ and the corresponding angular momentum p_θ. The Hamiltonian for this system can be written as

$$H(\theta, p_\theta, t) = \frac{{p_\theta}^2}{2I} + k\cos\theta \sum_n \delta(t - n\tau). \tag{6.38}$$

The term $\delta(t-n\tau)$ is zero everywhere except at $t = n\tau$; its integral over time is unity if $t = n\tau$ is within the limits of integration. If we use (6.37) and (6.38), it is easy to show that the corresponding equations of motion are given by

$$\frac{dp_\theta}{dt} = k\sin\theta \sum_n \delta(t - n\tau) \tag{6.39a}$$

$$\frac{d\theta}{dt} = \frac{p_\theta}{I}. \tag{6.39b}$$

From (6.39) we see that p_θ is constant between kicks (remember that gravity is assumed to be absent), but changes discontinuously at each kick. The angle θ varies linearly with t between kicks and is continuous at each kick.

It is convenient to know the values of θ and p_θ at times just after the kick. We let θ_n and p_n be the values of $\theta(t)$ and $p_\theta(t)$ at times $t = n\tau + 0^+$, where 0^+ is a infinitesimally small positive number. If we integrate (6.39a) from $t = (n+1)\tau - 0^+$ to $t = (n+1)\tau + 0^+$, we obtain

$$p_{n+1} - p_n = k\sin\theta_{n+1}. \tag{6.40a}$$

(Remember that p is constant between kicks and the delta function contributes to the integral only when $t = (n+1)\tau$.) From (6.39b) we have

$$\theta_{n+1} - \theta_n = (\tau/I)p_n. \tag{6.40b}$$

If we choose units such that $\tau/I = 1$, we obtain the standard map

$$\theta_{n+1} = (\theta_n + p_n) \quad \text{modulo } 2\pi \tag{6.41a}$$

$$p_{n+1} = p_n + k\sin\theta_{n+1} \qquad \text{(standard map).} \tag{6.41b}$$

We have added the requirement in (6.41a) that the value of the angle θ is restricted to be between zero and 2π.

Before we iterate (6.41), let us check that (6.41) represents a Hamiltonian system; that is, the area in q-p space is constant as n increases. (Here q corresponds to θ.) Suppose we start with a rectangle of points of length dq_n and dp_n. After one iteration, this rectangle will be deformed into a parallelogram of sides dq_{n+1} and dp_{n+1}. From (6.41) we have

$$dq_{n+1} = dq_n + dp_n \tag{6.42a}$$
$$dp_{n+1} = dp_n + k\cos q_{n+1}\,dq_{n+1}. \tag{6.42b}$$

If we substitute (6.42a) in (6.42b), we obtain

$$dp_{n+1} = (1 + k\cos q_{n+1})\,dp_n + k\cos q_{n+1}\,dq_n. \tag{6.43}$$

To find the area of a parallelogram, we take the magnitude of the cross product of the vectors $d\mathbf{q}_{n+1} = (dq_n, dp_n)$ and $d\mathbf{p}_{n+1} = (1 + k\cos q_n dq_n, k\cos q_n dp_n)$. The result is $dq_n\,dp_n$, and hence the area in phase space has not changed. The standard map is an example of an *area-preserving map*
.

The qualitative properties of the standard map are explored in Problem 6.19. You will find that for $k = 0$, the rod rotates with a fixed angular velocity determined by the momentum $p_n = p_0 =$ constant. If p_0 is a rational number times 2π, then the trajectory in phase space consists of a sequence of isolated points lying on a horizontal line (resonant tori). Can you see why? If p_0 is not a rational number times 2π or if your computer does not have sufficient precision, then after a long time, the trajectory will consist of a horizontal line in phase space. As we increase k, these horizontal lines are deformed into curves that run from $q = 0$ to $q = 2\pi$, and the isolated points of the resonant tori are converted into closed loops. For some initial conditions, the trajectories will become chaotic after the transient behavior has ended.

Problem 6.19. The standard map

(a) Write a program to iterate the standard map and plot its trajectory in phase space. Use different colors so that several trajectories can be shown at the same time for the same value of the parameter k. Choose a set of initial conditions that form a rectangle (see Problem 4.10). Does the shape of this area change with time? What happens to the total area?

(b) Begin with $k = 0$ and choose an initial value of p that is a rational number times 2π. What types of trajectories do you obtain? If you obtain trajectories consisting of isolated points, do these points appear to shift due to numerical roundoff errors? How can you tell? What happens if p_0 is an irrational number times 2π? Remember that a computer can only approximate an irrational number.

(c) Consider $k = 0.2$ and explore the nature of the phase space trajectories. What structures appear that do not appear for $k = 0$? Discuss the motion of the rod corresponding to some of the typical trajectories that you find.

(d) Increase k until you first find several chaotic trajectories. How can you tell that they are chaotic? Do these chaotic trajectories fill all of phase space? If there is one trajectory that is chaotic at a particular value of k, are all trajectories chaotic? What is the approximate value for k_c above which chaotic trajectories appear?

We now discuss a discrete map that models the rings of Saturn (see Fröyland). The model assumes that the rings of Saturn are due to perturbations produced by Mimas. There are two important forces acting on objects near Saturn. The force due to Saturn can be incorporated as follows. We know that each time Mimas completes an orbit, it traverses a total angle of 2π. Hence, the angle θ of any other moon of Saturn relative to Mimas can be expressed as

$$\theta_{n+1} = \theta_n + 2\pi \frac{\sigma^{3/2}}{{r_n}^{3/2}} \tag{6.44}$$

where r_n is the radius of the orbit after n revolutions and $\sigma = 185.7\times10^3$ km is the mean distance of Mimas from Saturn. The other important force is due to Mimas and causes the radial distance r_n to change. A discrete approximation to the radial acceleration dv_r/dt is (see (3.16))

$$\frac{\Delta v_r}{\Delta t} \approx \frac{r(t+\Delta t) - 2r(t) + r(t-\Delta t)}{(\Delta t)^2}. \tag{6.45}$$

The acceleration equals the radial force due to Mimas. If we average over a complete period, then a reasonable approximation for the change in r_n due to Mimas is

$$r_{n+1} - 2r_n + r_{n-1} = f(r_n, \theta_n) \tag{6.46}$$

where $f(r_n, \theta_n)$ is proportional to the radial force. (We have absorbed the factor of $(\Delta t)^2$ and the mass into f.)

In general, the form of $f(r_n, \theta_n)$ is very complicated. We make a major simplifying assumption and take f to be proportional to $-(r_n - \sigma)^{-2}$ and to be periodic in θ_n. This form for the force incorporates the fact that for large r_n, the force has the usual form for the gravitational force. For simplicity, we express this periodicity in the simplest possible way, that is, as $\cos\theta_n$. We also want the map to be area conserving. These considerations lead to the following two-dimensional map:

$$\theta_{n+1} = \theta_n + 2\pi \frac{\sigma^{3/2}}{{r_n}^{3/2}} \tag{6.47a}$$

$$r_{n+1} = 2r_n - r_{n-1} - a\frac{\cos\theta_n}{(r_n - \sigma)^2}. \tag{6.47b}$$

The constant a for Saturn's rings is approximately $2\times10^{12}\,\text{km}^3$. We can show, using a similar technique as before, that the volume in (r, θ) space is preserved, and hence (6.47) is a Hamiltonian map.

The purpose of the above discussion was only to motivate and not to derive the form of the map (6.47). In Problem 6.20 we investigate how the map (6.47) yields the qualitative structure of Saturn's rings. In particular, what happens to the values of r_n if the period of a moon is related to the period of Mimas by the ratio of two integers?

Problem 6.20. A simple model of the rings of Saturn

(a) Write a program to implement the map (6.47). Be sure to save the last two values of r so that the values of r_n are updated correctly. The radius of Saturn is 60.4×10^3 km. Express all lengths in units of 10^3 km. In these units $a = 2000$. Plot the points $(r_n\cos\theta_n, r_n\sin\theta_n)$. Choose initial values for r between the radius of Saturn and σ, the distance of Mimas from Saturn, and find the bands of r_n values where stable trajectories are found.

(b) What is the effect of changing the value of a? Try $a = 200$ and $a = 20{,}000$ and compare your results with part (a).

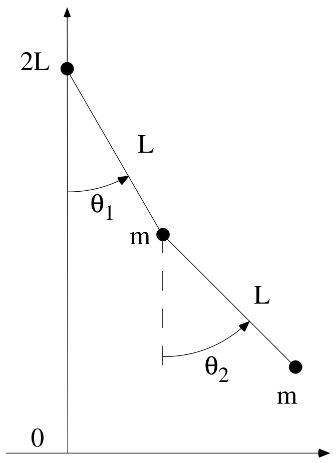


Figure 6.12: The double pendulum.

(c) Vary the force function. Replace $\cos\theta$ by other trigonometric functions. How do your results change? If the changes are small, does that give you some confidence that the model has something to do with Saturn's rings? □

A more realistic dynamical system is the double pendulum, a system that can be demonstrated in the laboratory. This system consists of two equal point masses m, with one suspended from a fixed support by a rigid weightless rod of length L and the other suspended from the first by a similar rod (see Figure 6.12). Because there is no friction, this system is an example of a Hamiltonian system. The four rectangular coordinates x_1, y_1, x_2, and y_2 of the two masses can be expressed in terms of two generalized coordinates θ_1, θ_2:

$$x_1 = L\sin\theta_1 \tag{6.48a}$$

$$y_1 = 2L - L\cos\theta_1 \tag{6.48b}$$

$$x_2 = L\sin\theta_1 + L\sin\theta_2 \tag{6.48c}$$

$$y_2 = 2L - L\cos\theta_1 - L\cos\theta_2. \tag{6.48d}$$

The kinetic energy is given by

$$K = \frac{1}{2}m(\dot{x}_1^2 + \dot{x}_2^2 + \dot{y}_1^2 + \dot{y}_2^2) = \frac{1}{2}mL^2[2\dot{\theta}_1^2 + \dot{\theta}_2^2 + 2\dot{\theta}_1\dot{\theta}_2\cos(\theta_1 - \theta_2)], \tag{6.49}$$

and the potential energy is given by

$$U = mgL\Big[3 - 2\cos\theta_1 - \cos\theta_2\Big]. \tag{6.50}$$

For convenience, U has been defined so that its minimum value is zero.

To use Hamilton's equations of motion (6.37), we need to express the sum of the kinetic energy and potential energy in terms of the generalized momenta and coordinates. In rectangular coordinates the momenta are equal to $p_i = \partial K/\partial\dot{q}_i$, where, for example, $q_i = x_1$ and p_i is the x-component of $m\mathbf{v}_1$. This relation works for generalized momenta as well, and the generalized momentum corresponding to θ_1 is given by $p_1 = \partial K/\partial\dot{\theta}_1$. If we calculate the appropriate

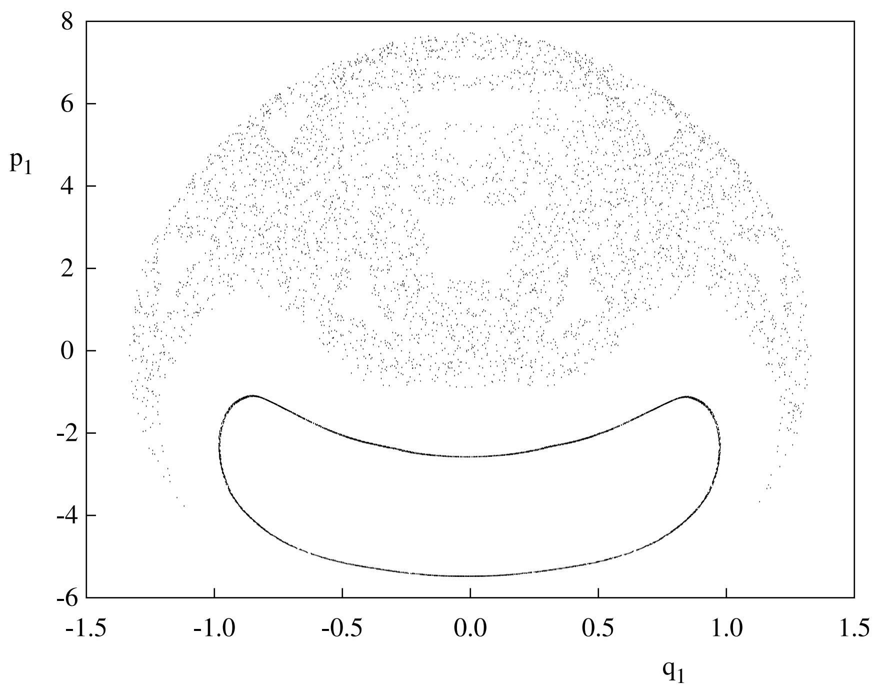


Figure 6.13: Poincaré plot for the double pendulum with p_1 plotted versus q_1 for $q_2 = 0$ and $p_2 > 0$. Two sets of initial conditions, $(q_1, q_2, p_2) = (0, 0, 0)$ and $(1.1, 0, 0)$, respectively, were used to create the plot. The initial value of the coordinate p_2 is found from (6.52) by requiring that $E = 15$.

derivatives, we can show that the generalized momenta can be written as

$$p_1 = mL^2\Big[2\dot{\theta}_1 + \dot{\theta}_2\cos(\theta_1 - \theta_2)\Big] \tag{6.51a}$$

$$p_2 = mL^2\Big[\dot{\theta}_2 + \dot{\theta}_1\cos(\theta_1 - \theta_2)\Big]. \tag{6.51b}$$

The Hamiltonian or total energy becomes

$$\begin{aligned} H &= \frac{1}{2mL^2}\frac{{p_1}^2 + 2{p_2}^2 - 2p_1p_2\cos(q_1 - q_2)}{1 + \sin^2(q_1 - q_2)} \\ &\quad + mgL(3 - 2\cos q_1 - \cos q_2) \end{aligned} \tag{6.52}$$

where $q_1 = \theta_1$ and $q_2 = \theta_2$. The equations of motion can be found by using (6.52) and (6.37).

Figure 6.13 shows a Poincaré map for the double pendulum. The coordinate p_1 is plotted versus q_1 for the same total energy $E = 15$, but for two different initial conditions. The map includes the points in the trajectory for which $q_2 = 0$ and $p_2 > 0$. Note the resemblance between Figure 6.13 and plots for the standard map above the critical value of k; that is, there is a regular trajectory and a chaotic trajectory for the same parameters, but different initial conditions.

Problem 6.21. Double pendulum

(a) Use either the fourth-order Runge–Kutta algorithm (with $\Delta t = 0.003$) or the second-order Euler–Richardson algorithm (with $\Delta t = 0.001$) to simulate the double pendulum. Choose

$m = 1$, $L = 1$, and $g = 9.8$. The input parameter is the total energy E. The initial values of q_1 and q_2 can be chosen either randomly within the interval $|q_i| < \pi$ or by the user. Then set the initial $p_1 = 0$ and solve for p_2 using (6.52) with $H = E$. First explore the pendulum's behavior by plotting the generalized coordinates and momenta as a function of time in four windows. Consider the energies $E = 1$, 5, 10, 15, and 40. Try a few initial conditions for each value of E. Visually determine whether the steady state behavior is regular or appears to be chaotic. Are there some values of E for which all the trajectories appear regular? Are there values of E for which all trajectories appear chaotic? Are there values of E for which both types of trajectories occur?

(b) Repeat part (a) but plot the phase space diagrams p_1 versus q_1 and p_2 versus q_2. Are these plots more useful for determining the nature of the trajectories than those drawn in part (a)?

(c) Draw the Poincaré plot with p_1 plotted versus q_1 only when $q_2 = 0$ and $p_2 > 0$. Overlay trajectories from different initial conditions but with the same total energy on the same plot. Duplicate the plot shown in Figure 6.13. Then produce Poincaré plots for the values of E given in part (a) with at least five different initial conditions for each energy. Describe the different types of behavior.

(d) Is there a critical value of the total energy at which some chaotic trajectories first occur?

(e) Animate the double pendulum, showing the two masses moving back and forth. Describe how the motion of the pendulum is related to the behavior of the Poincaré plot. □

Hamiltonian chaos has important applications in physical systems such as the solar system, the motion of the galaxies, and plasmas. It also has helped us understand the foundation for statistical mechanics. One of the most fascinating applications has been to quantum mechanics, which has its roots in the Hamiltonian formulation of classical mechanics. A current area of interest is the quantum analogue of classical Hamiltonian chaos. The meaning of this analogue is not obvious because well-defined trajectories do not exist in quantum mechanics. Moreover, Schrödinger's equation is linear and can be shown to have only periodic and quasiperiodic solutions.

6.10 Perspective

As the many books and review articles on chaos can attest, it is impossible to discuss all aspects of chaos in a single chapter. We will revisit chaotic systems in Chapter 13 where we introduce the concept of fractals. We will find that one of the characteristics of chaotic dynamics is that the resulting attractors often have an intricate geometrical structure.

The most general ideas that we have discussed in this chapter are that *simple systems can exhibit complex behavior* and that chaotic systems exhibit *extreme sensitivity to initial conditions*. We have also learned that computers allow us to explore the behavior of dynamical systems and visualize the numerical output. However, the simulation of a system does not automatically lead to understanding. If you are interested in learning more about the phenomena of chaos and the associated theory, the suggested readings at the end of the chapter are a good place to start. We also invite you to explore chaotic phenomenon in more detail in the following projects.

6.11 Projects

The first several projects are on various aspects of the logistic map. These projects do not exhaust the possible investigations of the properties of the logistic map.

Project 6.22. A more accurate determination of δ and α

We have seen that it is difficult to determine δ accurately by finding the sequence of values of b_k at which the trajectory bifurcates for the kth time. A better way to determine δ is to compute it from the sequence s_m of superstable trajectories of period 2^{m-1}. We already have found that $s_1 = 1/2$, $s_2 \approx 0.80902$, and $s_3 \approx 0.87464$. The parameters s_1, s_2,... can be computed directly from the equation

$$f^{(2^{m-1})}\Big(x = \frac{1}{2}\Big) = \frac{1}{2}. \tag{6.53}$$

For example, s_2 satisfies the relation $f^{(2)}(x = 1/2) = 1/2$. This relation, together with the analytic form for $f^{(2)}(x)$ given in (6.7), yields

$$8r^2(1-r) - 1 = 0. \tag{6.54}$$

If we wish to solve (6.54) numerically for $r = s_2$, we need to be careful not to find the irrelevant solutions corresponding to a lower period. In this case we can factor out the solution $r = 1/2$ and solve the resultant quadratic equation analytically to find $s_2 = (1 + \sqrt{5})/4$. Clearly $r = s_1 = 1/2$ solves (6.54) with period 1, because from (6.53), $f^{(1)}(x = 1/2) = 4r\frac{1}{2}(1 - \frac{1}{2}) = r = 1/2$ only for $r = 1/2$.

(a) It is straightforward to adapt the bisection method discussed in Section 6.6. Adapt the class `RecursiveFixedPointApp` to find the numerical solutions of (6.53). Good starting values for the left-most and right-most values of r are easy to obtain. The left-most value is $r = r_\infty \approx 0.8925$. If we already know the sequence $s_1, s_2, \ldots, s_m$, then we can determine δ by

$$\delta_m = \frac{s_{m-1} - s_{m-2}}{s_m - s_{m-1}}. \tag{6.55}$$

We use this determination for δ_m to find the right-most value of r:

$$r^{(m+1)}_{\text{right}} = \frac{s_m - s_{m-1}}{\delta_m}. \tag{6.56}$$

We choose the desired precision to be 10^{-16}. A summary of our results is given in Table 6.2. Verify these results and determine δ.

(b) Use your values of s_m to obtain a more accurate determination of α and δ. □

Project 6.23. From chaos to order

The bifurcation diagram of the logistic map (see Figure 6.2) has many interesting features that we have not explored. For example, you might have noticed that there are several smooth dark bands in the chaotic region for $r > r_\infty$. Use `BifurcateApp` to generate the bifurcation diagram for $r_\infty \leq r \leq 1$. If we start at $r = 1.0$ and decrease r, we see that there is a band that narrows and eventually splits into two parts at $r \approx 0.9196$. If you look closely, you will see that the band splits into four parts at $r \approx 0.899$. If you look even more closely, you will see many more bands. What type of change occurs near the splitting (merging) of these bands)? Use `IterateMap` to

m	Period	s_m
1	1	0.500 000 000
2	2	0.809 016 994
3	4	0.874 640 425
4	8	00.888 660 216
5	16	0.891 666 845
6	32	0.892 310 883
7	64	0.892 448 823
8	128	0.892 478 091

Table 6.2: Values of the control parameter s_m for the superstable trajectories of period 2^{m-1}. Nine decimal places are shown.

look at the time series of (6.5) for $r = 0.9175$. You will notice that although the trajectory looks random, it oscillates back and forth between two bands. This behavior can be seen more clearly if you look at the time series of $x_{n+1} = f^{(2)}(x_n)$. A detailed discussion of the splitting of the bands can be found in Peitgen et al. □

Project 6.24. Calculation of the Lyapunov spectrum

In Section 6.5 we discussed the calculation of the Lyapunov exponent for the logistic map. If a dynamical system has a multidimensional phase space, for example, the Hénon map and the Lorenz model, there is a set of Lyapunov exponents called the Lyapunov spectrum that characterize the divergence of the trajectory. As an example, consider a set of initial conditions that forms a filled sphere in phase space for the (three-dimensional) Lorenz model. If we iterate the Lorenz equations, then the set of phase space points will deform into another shape. If the system has a fixed point, this shape contracts to a single point. If the system is chaotic, then the sphere will typically diverge in one direction but become smaller in the other two directions. In this case we can define three Lyapunov exponents to measure the deformation in three mutually perpendicular directions. These three directions generally will not correspond to the axes of the original variables. Instead, we must use a Gram–Schmidt orthogonalization procedure.

The algorithm for finding the Lyapunov spectrum is as follows:

(i) Linearize the dynamical equations. If $\mathbf{r}$ is the f-component vector containing the dynamical variables, then define $\Delta\mathbf{r}$ as the linearized difference vector. For example, the linearized Lorenz equations are

$$\frac{d\Delta x}{dt} = -\sigma\Delta x + \sigma\Delta y \tag{6.57a}$$

$$\frac{d\Delta y}{dt} = -x\Delta z - z\Delta x + r\Delta x - \Delta y \tag{6.57b}$$

$$\frac{d\Delta z}{dt} = x\Delta y + y\Delta x - b\Delta z. \tag{6.57c}$$

(ii) Define f orthonormal initial values for $\Delta\mathbf{r}$. For example, $\Delta\mathbf{r}_1(0) = (1,0,0)$, $\Delta\mathbf{r}_2(0) = (0,1,0)$, and $\Delta\mathbf{r}_3(0) = (0,0,1)$. Because these vectors appear in a linearized equation, they do not have to be small in magnitude.

(iii) Iterate the original and linearized equations of motion. One iteration yields a new vector from the original equation of motion and f new vectors $\Delta\mathbf{r}_\alpha$ from the linearized equations.

(iv) Find the orthonormal vectors $\Delta\mathbf{r}'_\alpha$ from the $\Delta\mathbf{r}_\alpha$ using the Gram–Schmidt procedure. That is,

$$\Delta\mathbf{r}'_1 = \frac{\Delta\mathbf{r}_1}{|\Delta\mathbf{r}_1|} \tag{6.58a}$$

$$\Delta\mathbf{r}'_2 = \frac{\Delta\mathbf{r}_2 - (\Delta\mathbf{r}'_1 \cdot \Delta\mathbf{r}_2)\Delta\mathbf{r}'_1}{|\Delta\mathbf{r}_2 - (\Delta\mathbf{r}'_1 \cdot \Delta\mathbf{r}_2)\Delta\mathbf{r}'_1|} \tag{6.58b}$$

$$\Delta\mathbf{r}'_3 = \frac{\Delta\mathbf{r}_3 - (\Delta\mathbf{r}'_1 \cdot \Delta\mathbf{r}_3)\Delta\mathbf{r}'_1 - (\Delta\mathbf{r}'_2 \cdot \Delta\mathbf{r}_3)\Delta\mathbf{r}'_2}{\left|\Delta\mathbf{r}_3 - (\Delta\mathbf{r}'_1 \cdot \Delta\mathbf{r}_3)\Delta\mathbf{r}'_1 - (\Delta\mathbf{r}'_2 \cdot \Delta\mathbf{r}_3)\Delta\mathbf{r}'_2\right|}. \tag{6.58c}$$

It is straightforward to generalize the method to higher-dimensional models.

(v) Set the $\Delta\mathbf{r}_\alpha(t)$ equal to the orthonormal vectors $\Delta\mathbf{r}'_\alpha(t)$.

(vi) Accumulate the running sum, S_α as $S_\alpha \to S_\alpha + \log|\Delta\mathbf{r}_\alpha(t)|$.

(vii) Repeat steps (iii)–(vi) and periodically output the approximate Lyapunov exponents $\lambda_\alpha = (1/n)S_\alpha$, where n is the number of iterations.

To obtain a result for the Lyapunov spectrum that represents the steady state attractor, include only data after the transient behavior has ended.

(a) Compute the Lyapunov spectrum for the Lorenz model for $\sigma = 16$, $b = 4$, and $r = 45.92$. Try other values of the parameters and compare your results.

(b) Linearize the equations for the Hénon map and find the Lyapunov spectrum for $a = 1.4$ and $b = 0.3$ in (6.32). □

Project 6.25. A spinning magnet

Consider a compass needle that is free to rotate in a periodically reversing magnetic field which is perpendicular to the axis of the needle. The equation of motion of the needle is given by

$$\frac{d^2\phi}{dt^2} = -\frac{\mu}{I} B_0 \cos\omega t \sin\phi \tag{6.59}$$

where ϕ is the angle of the needle with respect to a fixed axis along the field, μ is the magnetic moment of the needle, I its moment of inertia, and B_0 and ω are the amplitude and the angular frequency of the magnetic field, respectively. Choose an appropriate numerical method for solving (6.59) and plot the Poincaré map at time $t = 2\pi n/\omega$. Verify that if the parameter $\lambda = \sqrt{2B_0\mu/I/\omega^2} > 1$, then the motion of the needle exhibits chaotic motion. Briggs (see references) discusses how to construct the corresponding laboratory system and other nonlinear physical systems. □

Project 6.26. Billiard models

Consider a two-dimensional planar geometry in which a particle moves with constant velocity along straight line orbits until it elastically reflects off the boundary. This straight line motion occurs in various "billiard" systems. A simple example of such a system is a particle moving with fixed speed within a circle. For this geometry the angle between the particle's momentum and the tangent to the boundary at a reflection is the same for all points.

Suppose that we divide the circle into two equal parts and connect them by straight lines of length L as shown in Figure 6.14a. This geometry is called a *stadium billiard*. How does the

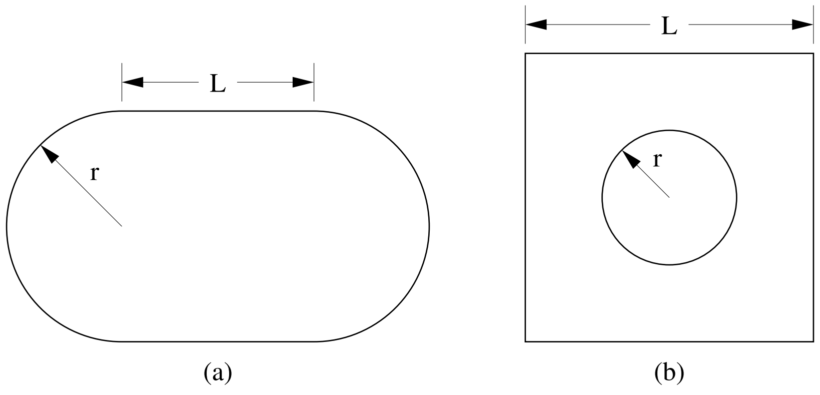


Figure 6.14: (a) Geometry of the stadium billiard model. (b) Geometry of the Sinai billiard model.

motion of a particle in the stadium compare to the motion in the circle? In both cases we can find the trajectory of the particle by geometrical considerations. The stadium billiard model and a similar geometry known as the Sinai billiard model (see Figure 6.14b) have been used as model systems for exploring the foundations of statistical mechanics. There is also much interest in relating the behavior of a classical particle in various billiard models to the solution of Schrödinger's equation for the same geometries.

(a) Write a program to simulate the stadium billiard model. Use the radius r of the semicircles as the unit of length. The algorithm for determining the path of the particle is as follows:

  (i) Begin with an initial position (x_0, y_0) and momentum (p_{x0}, p_{y0}) of the particle such that $|\mathbf{p}_0| = 1$.

  (ii) Determine which of the four sides the particle will hit. The possibilities are the top and bottom line segments and the right and left semicircles.

  (iii) Calculate the next position of the particle from the intersection of the straight line defined by the current position and momentum, and the equation for the segment where the next reflection occurs.

  (iv) Determine the new momentum, (p'_x, p'_y), of the particle after reflection such that the angle of incidence equals the angle of reflection. For reflection off the line segments we have $(p'_x, p'_y) = (p_x, -p_y)$. For reflection off a circle we have

$$p'_x = \Big[y^2 - (x - x_c)^2\Big]p_x - 2(x - x_c)yp_y \tag{6.60a}$$

$$p'_y = -2(x - x_c)yp_x + \Big[(x - x_c)^2 - y^2\Big]p_y \tag{6.60b}$$

  where $(x_c, 0)$ is the center of the circle. (Note that the momentum p_x rather than p'_x is on the right-hand side of (6.60b). Remember that all lengths are scaled by the radius of the circle.)

  (v) Repeat steps (ii)–(iv).

(b) Determine if the particle dynamics is chaotic by estimating the largest Lyapunov exponent. One way to do so is to start two particles with almost identical positions and/or momenta (varying by say 10^{-5}). Compute the difference Δs of the two phase space trajectories as a function of the number of reflections n, where Δs is defined by

$$\Delta s = \sqrt{|\mathbf{r}_1 - \mathbf{r}_2|^2 + |\mathbf{p}_1 - \mathbf{p}_2|^2}. \tag{6.61}$$

Choose $L = 1$ and $r = 1$. The Lyapunov exponent can be found from a semilog plot of Δs versus n. Repeat your calculation for different initial conditions and average your values of Δs before plotting. Repeat the calculation for $L = 0.5$ and 2.0 and determine if your results depend on L.

(c) Another test for the existence of chaos is the reversibility of the motion. Reverse the momentum after the particle has made n reflections, and let the drawing color equal the background color so that the path can be erased. What limitation does roundoff error place on your results? Repeat this simulation for $L = 1$ and $L = 0$.

(d) Place a small hole of diameter d in one of the circular sections of the stadium so that the particle can escape. Choose $L = 1$ and set $d = 0.02$. Give the particle a random position and momentum, and record the time when the particle escapes through the hole. Repeat for at least 10^4 particles and compute the fraction of particles $S(n)$ remaining after a given number of reflections n. The function $S(n)$ will decay with n. Determine the functional dependence of S on n and calculate the characteristic decay time if $S(n)$ decays exponentially. Repeat for $L = 0.1$, 0.5, and 2.0. Is the decay time a function of L? Does $S(n)$ decay exponentially for the circular billiard model ($L = 0$) (see Bauer and Bertsch)?

(e) Choose an arbitrary initial position for the particle in a stadium with $L = 1$ and a small hole as in part (d). Choose at least 5000 values of the initial value p_{x0} uniformly distributed between 0 and 1. Choose p_{y0} so that $|\mathbf{p}| = 1$. Plot the escape time versus p_{x0} and describe the visual pattern of the trajectories. Then choose 5000 values of p_{x0} in a smaller interval centered about the value of p_{x0} for which the escape time was greatest. Plot these values of the escape time versus p_{x0}. Do you see any evidence of self-similarity?

(f) Repeat steps (a)–(e) for the Sinai billiard geometry. □

Project 6.27. The circle map and mode locking

The driven damped pendulum can be approximated by a one-dimensional difference equation for a range of amplitudes and frequencies of the driving force. This difference equation is known as the *circle map* and is given by

$$\theta_{n+1} = \left(\theta_n + \Omega - \frac{K}{2\pi}\sin 2\pi\theta_n\right) \quad \text{(modulo 1)}. \tag{6.62}$$

The variable θ represents an angle, and Ω represents a frequency ratio, the ratio of the natural frequency of the pendulum to the frequency of the periodic driving force. The parameter K is a measure of the strength of the nonlinear coupling of the pendulum to the external force. An important quantity is the winding number which is defined as

$$W = \lim_{m\to\infty} \frac{1}{m} \sum_{n=0}^{m-1} \Delta\theta_n \tag{6.63}$$

where $\Delta\theta_n = \Omega - (K/2\pi)\sin 2\pi\theta_n$.

(a) Consider the linear case $K = 0$. Choose $\Omega = 0.4$ and $\theta_0 = 0.2$ and determine W. Verify that if Ω is a ratio of two integers, then $W = \Omega$ and the trajectory is periodic. What is the value of W if $\Omega = \sqrt{2}/2$, an irrational number? Verify that $W = \Omega$ and that the trajectory comes arbitrarily close to any particular value of θ. Does θ_n ever return exactly to its initial value? This type of behavior of the trajectory is termed *quasiperiodic*.

(b) For $K > 0$, we will find that $W \neq \Omega$ and "locks" into rational frequency ratios for a range of values of K and Ω. This type of behavior is called *mode locking*. For $K < 1$, the trajectory is either periodic or quasiperiodic. Determine the value of W for $K = 1/2$ and values of Ω in the range $O < \Omega \leq 1$. The widths in Ω of the various mode-locked regions where W is fixed increase with K. Consider other values of K, and draw a diagram in the K-Ω plane $(0 \leq K, \Omega \leq 1)$ so that those areas corresponding to frequency locking are shaded. These shaded regions are called *Arnold tongues*.

(c) For $K = 1$, all trajectories are frequency-locked periodic trajectories. Fix K at $K = 1$ and determine the dependence of W on Ω. The plot of W versus Ω for $K = 1$ is called the *Devil's staircase*. □

Project 6.28. Chaotic scattering

In Chapter 5 we discussed the classical scattering of particles off a fixed target, and found that the differential cross section for a variety of interactions is a smoothly varying function of the scattering angle. That is, a small change in the impact parameter b leads to a small change in the scattering angle θ. Here we consider examples where small changes in b lead to large changes in θ. Such a phenomenon is called *chaotic scattering* because of the sensitivity to initial conditions that is characteristic of chaos. The study of chaotic scattering is relevant to the design of electronic nanostructures, because many experimental structures exhibit this type of scattering.

A typical scattering model consists of a target composed of a group of fixed hard disks and a scatterer consisting of a point particle. The goal is to compute the path of the scatterer as it bounces off the disks and measure θ and the time of flight as a function of the impact parameter b. If a particle bounces inside the target region before leaving, the time of flight can be very long. There are even some trajectories for which the particle never leaves the target region.

Because it is difficult to monitor a trajectory that bounces back and forth between the hard disks, we consider instead a two-dimensional map that contains the key features of chaotic scattering (see Yalcinkaya and Lai for further discussion). The map is given by

$$x_{n+1} = a\Big[x_n - \frac{1}{4}(x_n + y_n)^2\Big] \tag{6.64a}$$

$$y_{n+1} = \frac{1}{a}\Big[y_n + \frac{1}{4}(x_n + y_n)^2\Big] \tag{6.64b}$$

where a is a parameter. The target region is centered at the origin. In an actual scattering experiment, the relation between (x_{n+1}, y_{n+1}) and (x_n, y_n) would be much more complicated, but the map (6.64) captures most of the important features of realistic chaotic scattering experiments. The iteration number n is analogous to the number of collisions of the scattered particle off the disks. When x_n or y_n is significantly different from zero, the scatterer has left the target region.

(a) Write a program to iterate the map (6.64). Let $a = 8.0$ and $y_0 = -0.3$. Choose 10^4 initial values of x_0 uniformly distributed in the interval $0 < x_0 < 0.1$. Determine the time $T(x_0)$, the number of iterations for which $x_n \leq -5.0$. After this time, x_n rapidly moves to $-\infty$. Plot

$T(x_0)$ versus x_0. Then choose 10^4 initial values in a smaller interval centered about a value of x_0 for which $T(x_0) > 7$. Plot these values of $T(x_0)$ versus x_0. Do you see any evidence of self-similarity?

(b) A trajectory is said to be *uncertain* if a small change ϵ in x_0 leads to a change in $T(x_0)$. We expect that the number of uncertain trajectories N will depend on a power of ϵ; that is, $N \sim \epsilon^\alpha$. Determine $N(\epsilon)$ for $\epsilon = 10^{-p}$ with $p = 2$ to 7 using the values of x_0 in part (a). Then determine the uncertainty dimension $1 - \alpha$ from a log-log plot of N versus ϵ. Repeat these measurements for other values of a. Does α depend on a?

(c) Choose $4{\times}10^4$ initial conditions in the same interval as in part (a) and determine the number of trajectories $S(n)$ that have not yet reached $x_n = -5$ as a function of the number of iterations n. Plot $\ln S(n)$ versus n and determine if the decay is exponential. It is possible to obtain algebraic decay for values of a less than approximately 6.5.

(d) Let $a = 4.1$ and choose 100 initial conditions uniformly distributed in the region $1.0 < x_0 < 1.05$ and $0.60 < y_0 < 0.65$. Are there any trajectories that are periodic and hence have infinite escape times? Due to the accumulation of roundoff error, it is possible to find only finite, but very long escape times. These periodic trajectories form closed curves, and the regions enclosed by them are called KAM surfaces. □

Project 6.29. Chemical reactions

In Project 4.17 we discussed how chemical oscillations can occur when the reactants are continuously replenished. In this project we introduce a set of chemical reactions that exhibits the period doubling route to chaos. Consider the following reactions (see Peng et al.):

$$P \to A \tag{6.65a}$$
$$P + C \to A + C \tag{6.65b}$$
$$A \to B \tag{6.65c}$$
$$A + 2B \to 3B \tag{6.65d}$$
$$B \to C \tag{6.65e}$$
$$C \to D. \tag{6.65f}$$

Each of the above reactions has an associated rate constant. The time dependence of the concentrations of $A, B,$ and C is given by:

$$\frac{dA}{dt} = k_1 P + k_2 PC - k_3 A - k_4 AB^2 \tag{6.66a}$$
$$\frac{dB}{dt} = k_3 A + k_4 AB^2 - k_5 B \tag{6.66b}$$
$$\frac{dC}{dt} = k_5 B - k_5 C. \tag{6.66c}$$

We assume that P is held constant by replenishment from an external source. We also assume the chemicals are well mixed so that there is no spatial dependence. In Section 7.8 we discuss the effects of spatial inhomogeneities due to molecular diffusion. Equations (6.65) can be written in a dimensionless form as

$$\frac{dX}{d\tau} = c_1 + c_2 Z - X - XY^2 \tag{6.67a}$$

$$c_3 \frac{dY}{d\tau} = X + XY^2 - Y \tag{6.67b}$$

$$c_4 \frac{dZ}{d\tau} = Y - Z \tag{6.67c}$$

where the c_i are constants, $\tau = k_3 t$, and X, Y, and Z are proportional to A, B, and C, respectively.

(a) Write a program to solve the coupled differential equations in (6.67). Use a fourth-order Runge–Kutta algorithm with an adaptive step size. Plot $\ln Y$ versus the time τ.

(b) Set $c_1 = 10$, $c_3 = 0.005$, and $c_4 = 0.02$. The constant c_2 is the control parameter. Consider $c_2 = 0.10$ to 0.16 in steps of 0.005. What is the period of $\ln Y$ for each value of c_2?

(c) Determine the values of c_2 at which the period doublings occur for as many period doublings as you can determine. Compute the constant δ [see (6.9)] and compare its value to the value of δ for the logistic map.

(d) Make a bifurcation diagram by taking the values of $\ln Y$ from the Poincaré plot at $X = Z$ and plotting them versus the control parameter c_2. Do you see a sequence of period doublings?

(e) Use three-dimensional graphics to plot the trajectory of (6.67) with $\ln X$, $\ln Y$, and $\ln Z$ as the three axes. Describe the attractors for some of the cases considered in part (a). □

Appendix 6A: Stability of the Fixed Points of the Logistic Map

In the following, we derive analytic expressions for the fixed points of the logistic map. The fixed-point condition is given by

$$x^* = f(x^*). \tag{6.68}$$

From (6.5) this condition yields the two fixed points

$$x^* = 0 \quad \text{and} \quad x^* = 1 - \frac{1}{4r}. \tag{6.69}$$

Because x is restricted to be positive, the only fixed point for $r < 1/4$ is $x = 0$. To determine the stability of x^*, we let

$$x_n = x^* + \epsilon_n \tag{6.70a}$$

and

$$x_{n+1} = x^* + \epsilon_{n+1}. \tag{6.70b}$$

Because $|\epsilon_n| \ll 1$, we have

$$x_{n+1} = f(x^* + \epsilon_n) \approx f(x^*) + \epsilon_n f'(x^*) = x^* + \epsilon_n f'(x^*). \tag{6.71}$$

If we compare (6.70b) and (6.71), we obtain

$$\epsilon_{n+1}/\epsilon_n = f'(x^*). \tag{6.72}$$

If $|f'(x^*)| > 1$, the trajectory will diverge from x^* because $|\epsilon_{n+1}| > |\epsilon_n|$. The opposite is true for $|f'(x^*)| < 1$. Hence, the local stability criteria for a fixed point x^* are

1. $|f'(x^*)| < 1$, x^* is stable;
2. $|f'(x^*)| = 1$, x^* is marginally stable;
3. $|f'(x^*)| > 1$, x^* is unstable.

If x^* is marginally stable, the second derivative $f''(x)$ must be considered, and the trajectory approaches x^* with deviations from x^* inversely proportional to the square root of the number of iterations.

For the logistic map, the derivatives at the fixed points are. respectively,

$$f'(x = 0) = \frac{d}{dx}[4rx(1-x)]\Big|_{x=0} = 4r \tag{6.73}$$

and

$$f'(x = x^*) = \frac{d}{dx}[4rx(1-x)]\Big|_{x=1-1/4r} = 2-4r. \tag{6.74}$$

It is straightforward to use (6.73) and (6.74) to find the range of r for which $x^* = 0$ and $x^* = 1-1/4r$ are stable.

If a trajectory has period two, then $f^{(2)}(x) = f(f(x))$ has two fixed points. If you are interested, you can solve for these fixed points analytically. As we found in Problem 6.2, these two fixed points become unstable at the same value of r. We can derive this property of the fixed points using the chain rule of differentiation:

$$\frac{d}{dx}f^{(2)}(x)\Big|_{x=x_0} = \frac{d}{dx}f(f(x))\Big|_{x=x_0} = f'(f(x_0))f'(x)\Big|_{x=x_0}. \tag{6.75}$$

If we substitute $x_1 = f(x_0)$, we can write

$$\frac{d}{dx}f(f(x))\Big|_{x=x_0} = f'(x_1)f'(x_0). \tag{6.76}$$

In the same way, we can show that

$$\frac{d}{dx}f^{(2)}(x)\Big|_{x=x_1} = f'(x_0)f'(x_1). \tag{6.77}$$

We see that if x_0 becomes unstable, then $|f^{(2)\prime}(x_0)| > 1$ as does $|f^{(2)\prime}(x_1)|$. Hence, x_1 is also unstable at the same value of r, and we conclude that both fixed points of $f^{(2)}(x)$ bifurcate at the same value of r, leading to an trajectory of period 4.

From (6.74) we see that $f'(x = x^*) = 0$ when $r = 1/2$ and $x^* = 1/2$. Such a fixed point is said to be superstable, because as we found in Problem 6.4, convergence to the fixed point is relatively rapid. Superstable trajectories occur whenever one of the fixed points is at $x^* = 1/2$.

Appendix 6B: Finding the Roots of a Function

The roots of a function $f(x)$ are the values of the variable x for which the function $f(x)$ is zero. Even an apparently simple equation such as

$$f(x) = \tan x - x - c = 0 \tag{6.78}$$

where c is a constant cannot be solved analytically for x.

Regardless of the function and the approach to root finding, the first step should be to learn as much as possible about the function. For example, plotting the function will help us to determine the approximate locations of the roots.

Newton's (or the Newton–Raphson) method is based on replacing the function by the first two terms of the Taylor expansion of $f(x)$ about the root x. If our initial guess for the root is x_0, we can write $f(x) \approx f(x_0) + (x - x_0)f'(x_0)$. If we set $f(x)$ equal to zero and solve for x, we find $x = x_0 - f(x_0)/f'(x_0)$. If we have made a good choice for x_0, the resultant value of x should be closer to the root than x_0. The general procedure is to calculate successive approximations as follows:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}. \tag{6.79}$$

If this series converges, it converges very quickly. However, if the initial guess is poor or if the function has closely spaced multiple roots, the series may not converge. The successive iterations of Newton's method is an another example of a map. Newton's method also works with complex functions as we will see in the following problem.

Problem 6.30. Cube roots

Consider the function $f(z) = z^3 - 1$, where $z = x + iy$, and $f'(z) = z^2$. Map the range of convergence of (6.79) in the region $[-2 < x < 2, -2 < y < 2]$ in the complex plane. Color the starting z value red, green, or blue depending on the root to which the initial guess converges. If the trajectory does not converge, color the starting point black. For more insight add a mouse handler to your program so that if you click on your plot, the sequence of iterations starting from the point where you clicked will be shown. □

The following problem discusses a situation that typically arises in courses on quantum mechanics.

Problem 6.31. Energy levels in a finite square well

The quantum mechanical energy levels in the one-dimensional finite square well can be found by solving the relation

$$\epsilon \tan \epsilon = \sqrt{\rho^2 - \epsilon^2} \tag{6.80}$$

where $\epsilon = \sqrt{mEa^2/2\hbar}$ and $\rho = \sqrt{mV_0a^2/2\hbar}$ are defined in terms of the particle mass m, the particle energy E, the width of the well a, and the depth of the well V_0. The function $\epsilon \tan \epsilon$ has zeros at $\epsilon = 0, \pi, 2\pi, \ldots$ and asymptotes at $\epsilon = 0, \pi/2, 3\pi/2, 5\pi/2 \ldots$. The function $\sqrt{\rho - \epsilon^2}$ is a quarter circle of radius ρ. Write a program to plot these two functions with $\rho = 3$, and then use Newton's method to determine the roots of (6.80). Find the value of ρ and thus V_0 such that below this value there is only one energy level and above this value there is more than one. At what value of ρ do three energy levels first appear? □

In Section 6.6 we introduced the bisection root-finding algorithm. This algorithm is implemented in the Root class in the numerics package. It can be used with any function.

Listing 6.6: The bisection method defined in the Root class in the numerics package

```
public static double bisection(final Function f, double x1, double x2,
  final double tolerance) {
   int count = 0;
   int maxCount = (int) (Math.log(Math.abs(x2 - x1)/tolerance)/Math.log(2));
```

```
    maxCount = Math.max(MAX_ITERATIONS, maxCount) + 2;
    double y1 = f.evaluate(x1), y2 = f.evaluate(x2);
    if (y1 * y2 > 0) {      // y1 and y2 must have opposite sign
       return Double.NaN; // interval does not contain a root
    }
    while (count < maxCount) {
       double x = (x1 + x2) / 2;
       double y = f.evaluate(x);
       if (Math.abs(y) < tolerance) return x;
       if (y * y1 > 0) { // replace the endpoint that has the same sign
          x1 = x;
          y1 = y;
       }
       else {
          x2 = x;
          y2 = y;
       }
       count++;
    }
    return Double.NaN; // did not converge in max iterations
 }
```

The bisection algorithm is guaranteed to converge if you can find an interval where the function changes sign. However, it is slow. Newton's algorithm is very fast but may not converge. We develop an algorithm in the following problem that combines these two approaches.

Problem 6.32. Finding roots

Modify Newton's algorithm to keep track of the interval between the minimum and the maximum of x while iterating (6.79). If the iterate x_{n+1} jumps outside this interval, interrupt Newton's method and use the bisection algorithm for one iteration. Test the root at the end of the iterative process to check that the algorithm actually found a root. Test your algorithm on the function in (6.78). □

References and Suggestions for Further Reading

Books

Ralph H. Abraham and Christopher D. Shaw, *Dynamics – The Geometry of Behavior*, 2nd ed. (Addison–Wesley, 1992). The authors use an abundance of visual representations.

Hao Bai-Lin, *Chaos II* (World Scientific, 1990). A collection of reprints on chaotic phenomena. The following papers were cited in the text. James P. Crutchfield, J. Doyne Farmer, Norman H. Packhard, and Robert S. Shaw, "Chaos," Sci. Am. **255** (6), 46–57 (1986); Mitchell J. Feigenbaum, "Quantitative universality for a class of nonlinear transformations," J. Stat. Phys. **19**, 25–52 (1978); M. Hénon, "A two-dimensional mapping with a strange attractor," Commun. Math. Phys. **50**, 69–77 (1976); Robert M. May, "Simple mathematical models with very complicated dynamics," Nature **261**, 459–467 (1976); Robert Van Buskirk and Carson Jeffries, "Observation of chaotic dynamics of coupled nonlinear oscillators," Phys. Rev. A **31**, 3332–3357 (1985).

G. L. Baker and J. P. Gollub, *Chaotic Dynamics: An Introduction*, 2nd ed. (Cambridge University Press, 1995). A good introduction to chaos with special emphasis on the forced-damped-nonlinear harmonic oscillator. Several programs are given.

Pedrag Cvitanovic, *Universality in Chaos*, 2nd ed. (Adam-Hilger, 1989). A collection of reprints on chaotic phenomena including the articles by Hénon and May also reprinted in the Bai-Lin collection and the chaos classic, Mitchell J. Feigenbaum, "Universal behavior in nonlinear systems," Los Alamos Sci. **1**, 4–27 (1980).

Robert Devaney, *A First Course in Chaotic Dynamical Systems* Addison–Wesley, 1992). This text is a good introduction to the more mathematical ideas behind chaos and related topics.

Jan Fröyland, *Introduction to Chaos and Coherence* (Institute of Physics Publishing, 1992). See Chapter 7 for a simple model of Saturn's rings.

Martin C. Gutzwiller, *Chaos in Classical and Quantum Mechanics* (Springer–Verlag, 1990). A good introduction to problems in quantum chaos for the more advanced student.

Robert C. Hilborn, *Chaos and Nonlinear Dynamics* (Oxford University Press, 1994). An excellent pedagogically oriented text.

Douglas R. Hofstadter, *Metamagical Themas* (Basic Books, 1985). A shorter version is given in his article, "Metamagical themas," Sci. Am. **245** (11), 22–43 (1981).

E. Atlee Jackson, *Perspectives of Nonlinear Dynamics*, Vols. 1 and 2. (Cambridge University Press, 1989, 1991). An advanced text that is a joy to read.

R. V. Jensen, "Chaotic scattering, unstable periodic orbits, and fluctuations in quantum transport," Chaos **1**, 101–109 (1991). This paper discusses the quantum version of systems similar to those discussed in Projects 6.28 and 6.26.

Francis C. Moon, *Chaotic and Fractal Dynamics, An Introduction for Applied Scientists and Engineers* (Wiley–VCH, 1992). An engineering oriented text with a section on how to build devices that demonstrate chaotic dynamics.

Edward Ott, *Chaos in Dynamical Systems* (Cambridge University Press, 1993). An excellent textbook on chaos at the upper undergraduate to graduate level. See also E. Ott, "Strange attractors and chaotic motions of dynamical systems," Rev. Mod. Phys. **53**, 655–671 (1981).

Edward Ott, Tim Sauer, and James A. Yorke, editors, *Coping with Chaos* (John Wiley & Sons, 1994). A reprint volume emphasizing the analysis of experimental time series from chaotic systems.

Heinz–Otto Peitgen, Hartmut Jürgens, and Dietmar Saupe, *Fractals for the Classroom*, Part II (Springer–Verlag, 1992). A delightful book with many beautiful illustrations. Chapter 11 discusses the nature of the bifurcation diagram of the logistic map.

Ian Percival and Derek Richards, *Introduction to Dynamics* (Cambridge University Press, 1982). An advanced undergraduate text that introduces phase trajectories and the theory of stability. A derivation of the Hamiltonian for the driven damped pendulum considered in Section 6.4 is given in Chapter 5, example 5.7.

Ivars Peterson, *Newton's Clock: Chaos in the Solar System* (W. H. Freeman, 1993). An historical survey of our understanding of the motion of bodies within the solar system with a focus on chaotic motion.

Stuart L. Pimm, *The Balance of Nature* (The University of Chicago Press, 1991). An introductory treatment of ecology with a chapter on applications of chaos to real biological systems. The author contends that much of the difficulty in assessing the importance of chaos is that ecological studies are too short.

William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery, *Numerical Recipes*, 2nd ed. (Cambridge University Press, 1992). Chapter 9 discusses various root-finding methods.

S. Neil Rasband, *Chaotic Dynamics of Nonlinear Systems* (Wiley–Interscience, 1990). Clear presentation of the most important topics in classical chaos theory.

M. Lakshmanan and S. Rajaseekar, *Nonlinear Dynamics* (Springer–Verlag, 2003). Although this text is for advanced students, many parts are accessible.

Robert Shaw, *The Dripping Faucet as a Model Chaotic System* (Aerial Press, 1984).

Steven Strogatz, *Nonlinear Dynamics and Chaos with Applications to Physics, Biology, Chemistry and Engineering* (Addison–Wesley, 1994). Another outstanding text.

Anastasios A. Tsonis, *Chaos: From Theory to Applications* (Plenum Press, 1992). Of particular interest is the discussion of applications to nonlinear time series forecasting.

Nicholas B. Tufillaro, Tyler Abbott, and Jeremiah Reilly, *Nonlinear Dynamics and Chaos* (Addison–Wesley, 1992). See also, N. B. Tufillaro and A. M. Albano, "Chaotic dynamics of a bouncing ball," Am. J. Phys. **54**, 939–944 (1986). The authors describe an undergraduate level experiment on a bouncing ball subject to repeated impacts with a vibrating table. See also the article by Warr et al.

Articles

Garin F. J. Añaños and Constantino Tsallis, "Ensemble averages and nonextensivity of one-dimensional maps," Phys. Rev. Lett. **93**, 020601 (2004).

Gregory L. Baker, "Control of the chaotic driven pendulum," Am. J. Phys. **63** (9), 832–838 (1995).

W. Bauer and G. F. Bertsch, "Decay of ordered and chaotic systems," Phys. Rev. Lett. **65**, 2213 (1990). See also the comment by Olivier Legrand and Didier Sornette, "First return, transient chaos, and decay in chaotic systems," Phys. Rev. Lett. **66**, 2172 (1991), and the reply by Bauer and Bertsch on the following page. The dependence of the decay laws on chaotic behavior is very general and has been considered in various contexts including room acoustics and the chaotic scattering of microwaves in an "elbow" cavity. Chaotic behavior is a sufficient but not necessary condition for exponential decay.

Keith Briggs, "Simple experiments in chaotic dynamics," Am. J. Phys. **55**, 1083–1089 (1987).

S. N. Coppersmith, "A simpler derivation of Feigenbaum's renormalization group equation for the period-doubling bifurcation sequence," Am. J. Phys. **67** (1), 52–54 (1999).

J. P. Crutchfield, J. D. Farmer, and B. A. Huberman, "Fluctuations and simple chaotic dynamics," Phys. Repts. **92**, 45–82 (1982).

Robert DeSerio, "Chaotic pendulum: The complete attractor," Am. J. Phys. **71** (3), 250–257 (2003).

William L. Ditto and Louis M. Pecora, "Mastering chaos," Sci. Am. **262** (8), 78–82 (1993).

J. C. Earnshaw and D. Haughey, "Lyapunov exponents for pedestrians," Am. J. Phys. **61**, 401 (1993).

Daniel J. Gauthier, "Resource letter: CC-1: Controlling chaos," Am. J. Phys. **71** (8), 750–759 (2003). The article includes a bibliography of materials on controlling chaos.

Wayne Hayes, "Computer simulations, exact trajectories, and the gravitational N-body problem," Am. J. Phys. **72** (9), 1251–1257 (2004). The article discusses the concept of shadowing which is used in the simulation of chaotic systems.

Robert C. Hilborn, "Sea gulls, butterflies, and grasshoppers: A brief history of the butterfly effect in nonlinear dynamics," Am. J. Phys. **72** (4), 425–427 (2004).

Robert C. Hilborn and Nicholas B. Tufillaro, "Resource letter: ND-1: Nonlinear dynamics," Am. J. Phys. **65** (9), 822–834 (1997).

Ying–Cheng Lai, "Controlling chaos," Computers in Physics **8**, 62 (1994). Section 6.6 is based on this article.

R. B. Levien and S. M. Tan, "Double pendulum: An experiment in chaos," Am. J. Phys. **61** (11), 1038–1044 (1993).

V. Lopac and V. Danani, "Energy conservation and chaos in the gravitationally driven Fermi oscillator," Am. J. Phys. **66** (10), 892–902 (1998).

J. B. McLaughlin, "Period-doubling bifurcations and chaotic motion for a parametrically forced pendulum," J. Stat. Phys. **24**, 375–388 (1981).

Sergio De Souza–Machado, R. W. Rollins, D. T. Jacobs, and J. L. Hartman, "Studying chaotic systems using microcomputer simulations and Lyapunov exponents," Am. J. Phys. **58** (4), 321–329 (1990).

Bo Peng, Stephen K. Scott, and Kenneth Showalter, "Period doubling and chaos in a three-variable autocatalator," J. Phys. Chem. **94**, 5243–5246 (1990).

Bo Peng, Valery Petrov, and Kenneth Showalter, "Controlling chemical chaos," J. Phys. Chem. 95, 4957–4959 (1991).

Troy Shinbrot, Celso Grebogi, Jack Wisdom, and James A. Yorke, "Chaos in a double pendulum," Am. J. Phys. **60** (6), 491–499 (1992).

Niraj Srivastava, Charles Kaufman, and Gerhard Müller, "Hamiltonian chaos," Computers in Physics **4**, 549–553 (1990); ibid. **5**, 239–243 (1991); ibid. **6**, 84–88 (1992).

Todd Timberlake, "A computational approach to teaching conservative chaos," Am. J. Phys. **72** (8), 1002–1007 (2004).

Jan Tobochnik and Harvey Gould, "Quantifying chaos," Computers in Physics **3** (6), 86 (1989). There is a typographical error in this paper in the equations for step (3) of the algorithm for computing the Lyapunov spectrum. The correct equations are given in Project 6.24.

S. Warr, W. Cooke, R. C. Ball, and J. M. Huntley, "Probability distribution functions for a single particle vibrating in one dimension: Experimental study and theoretical analysis," Physica A **231**, 551–574 (1996). This paper and the book by Tufillaro, Abbott, and Reilly consider the motion of a ball bouncing on a periodically vibrating table. This nonlinear dynamical system exhibits fixed points, periodic and strange attractors, and period-doubling bifurcations to chaos, similar to the logistic map. Simulations of this system are very interesting, but not straightforward.

Tolga Yalcinkaya and Ying–Cheng Lai, "Chaotic scattering," Computers in Physics **9**, 511–518 (1995). Project 6.28 is based on a draft of this article. The map (6.64) is discussed in more detail in Yun–Tung Lau, John M. Finn, and Edward Ott, "Fractal dimension in nonhyperbolic chaotic scattering," Phys. Rev. Lett. **66**, 978 (1991).

Chapter 7

Random Processes

Random processes are introduced in the context of several simple physical systems, including random walks on a lattice, polymers, and diffusion–controlled chemical reactions. The generation of random number sequences is also discussed.

7.1 Order to Disorder

In Chapter 6 we saw several examples of how, under certain conditions, the behavior of a nonlinear deterministic system can appear to be random. In this chapter we will see some examples of how chance can generate statistically predictable outcomes. For example, we know that if we bet often on the outcome of a game for which the probability of winning is less than 50%, we will lose money eventually.

We first discuss an example that illustrates the tendency of systems of many particles to evolve to a well-defined state. Imagine a closed box that is divided into two parts of equal volume (see Figure 7.1). The left half contains a gas of N identical particles and the right half is initially empty. We then make a small hole in the partition between the two halves. What happens? We know that after some time, the average number of particles in each half of the box will become $N/2$, and we say that the system has reached equilibrium.

How can we simulate this process? One way is to give each particle an initial velocity and position and adopt a deterministic model of the motion of the particles. For example, we could assume that each particle moves in a straight line until it hits a wall of the box or another particle and undergoes an elastic collision. We will consider similar deterministic models in Chapter 8. Instead, we first simulate a probabilistic model based on a *random process*.

The basic assumptions of this model are that the motion of the particles is random and the particles do not interact with one another. Hence, the probability per unit time that a particle goes through the hole in the partition is the same for all N particles regardless of the number of particles in either half. We also assume that the size of the hole is such that only one particle can pass through at a time. We first model the motion of a particle passing through the hole by choosing one of the N particles at random and moving it to the other side. For visualization purposes, we will use arrays to specify the position of each particle. We then randomly generate an integer `i` between 0 and $N-1$ and change the arrays appropriately. A more efficient Monte Carlo algorithm is discussed in Problem 7.2b. The tool we need to simulate this random process is a random number generator.

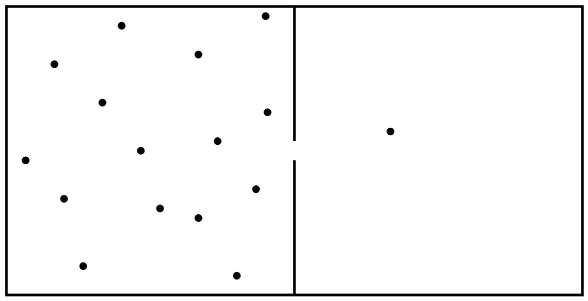


Figure 7.1: A box is divided into two equal halves by a partition. After a small hole is opened in the partition, one particle can pass through the hole per unit time.

It is counterintuitive that we can use a deterministic computer to generate sequences of random numbers. In Section 7.9 we discuss some of the methods for computing a set of numbers that appear statistically random but are in fact generated by a deterministic algorithm. These algorithms are sometimes called *pseudorandom number generators* to distinguish their output from intrinsically random physical processes, such as the time between clicks in a Geiger counter near a radioactive sample.

For the present we will be content to use the random number generator supplied with Java, although the random number generators included with various programming languages vary in quality. The method `Math.random()` produces a random number r that is uniformly distributed in the interval $0 \le r < 1$. To generate a random integer i between 0 and $N-1$, we write:

```
int i = (int)(N*Math.random());
```

The effect of the `(int)` cast is to eliminate the decimal digits from a floating point number. For example, `(int)(5.7) = 5`.

The algorithm for simulating the evolution of the model can be summarized by the following steps:

1. Use a random number generator to choose a particle at random.
2. Move this particle to the other side of the box.
3. Give the particle a random position on the new side of the box. This step is for visualization purposes only.
4. Increase the "time" by unity.

Note that this definition of time is arbitrary. Class `Box` implements this algorithm and class `BoxApp` plots the evolution of the number of particles on the left half of the box.

Listing 7.1: Class Box for the simulation of the approach to equilibrium.

```
package org.opensourcephysics.sip.ch07;
import java.awt.*;
import org.opensourcephysics.display.*;

public class Box implements Drawable {
   public double x[], y[];
   public int N, nleft, time;

   public void initialize() {
      // location of particles (for visualization purposes only)
      x = new double[N];
      y = new double[N];
      nleft = N;          // start with all particles on the left
      time = 0;
      for(int i = 0;i<N;i++) {
         x[i] = 0.5*Math.random(); //  needed only for visualization
         y[i] = Math.random();
      }
   }

   public void step() {
      int i = (int) (Math.random()*N);
      if(x[i]<0.5) {
         nleft--;    // move to right
         x[i] = 0.5*(1+Math.random());
         y[i] = Math.random();
      } else {
         nleft++;    // move to left
         x[i] = 0.5*Math.random();
         y[i] = Math.random();
      }
      time++;
   }

   public void draw(DrawingPanel panel, Graphics g) {
      if(x==null) {
         return;
      }
      int size = 2;
      // position of partition in middle of box
      int xMiddle = panel.xToPix(0.5);
      g.setColor(Color.black);
      g.drawLine(xMiddle, panel.yToPix(0), xMiddle, panel.yToPix(0.45));
      g.drawLine(xMiddle, panel.yToPix(0.55), xMiddle, panel.yToPix(1.0));
      g.setColor(Color.red);
      for(int i = 0;i<N;i++) {
         int xpix = panel.xToPix(x[i]);
         int ypix = panel.yToPix(y[i]);
         g.fillOval(xpix, ypix, size, size);
      }
   }
}
```

Listing 7.2: Target class for plotting the approach to equilibrium.

```
package org.opensourcephysics.sip.ch07;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.frames.*;

public class BoxApp extends AbstractSimulation {
   Box box = new Box();
   PlotFrame plotFrame = new PlotFrame("time", "number on left",
                            "Box data");
   DisplayFrame displayFrame = new DisplayFrame("Partitioned box");

   public void initialize() {
      displayFrame.clearDrawables();
      displayFrame.addDrawable(box);
      box.N = control.getInt("Number of particles");
      box.initialize();
      plotFrame.clearData();
      displayFrame.setPreferredMinMax(0, 1, 0, 1);
   }

   public void doStep() {
      box.step();
      plotFrame.append(0, box.time, box.nleft);
   }

   public void reset() {
      // clicking resets erase positions of particles
      control.setValue("Number of particles", 64);
      plotFrame.clearData();
      enableStepsPerDisplay(true);
      setStepsPerDisplay(10);
   }

   public static void main(String[] args) {
      SimulationControl.createApp(new BoxApp());
   }
}
```

How long does it take for the system to reach equilibrium? How does this time depend on the number of particles? After the system reaches equilibrium, what is the magnitude of the fluctuations? How do the fluctuations depend on the number of particles? Problems 7.2 and 7.3 explore such questions.

Exercise 7.1. Simple tests of operators and methods

(a) It is frequently quicker to write a short program to test how the operators and methods of a computer language work than to look them up in a manual or online. Write a test class to determine the values of `3/2`, `3.0/2.0`, `(int) (3.0/2.0)`, `2/3`, and `(int) (-3/2)`.

(b) Determine the behavior of `Math.round(arg)`, `Math.ceil(arg)`, and `Math.rint(arg)`.

(c) Write a program to test whether the same sequence of random numbers appears each time the program is run if we use the method `Math.random()` to generate the sequence.

(d) Create an object from class `Random` and use the methods `setSeed(long seed)` and `nextDouble()`. Show that you obtain the same sequence of random numbers if the same seed is used. One reason to specify the seed rather than to choose it at random from the time (as is the default) is that it is convenient to use the same random number sequence when testing a program. Suppose that your program gives a strange result for a particular run. If you notice a error in the program and change the program, you would want to use the same random number sequence to test whether your changes corrected the error. Another reason for specifying the seed is that another user could obtain the same results if you tell them the seed that you used. □

Problem 7.2. Approach to equilibrium

(a) Use BoxApp and Box and describe the nature of the evolution of n, the number of particles on the left side of the box. Choose the total number of particles N to be $N = 8$, 16, 64, 400, 800, and 3600. Does the system reach equilibrium? What is your qualitative criterion for equilibrium? Does n, the number of particles on the left-hand side, change when the system is in equilibrium?

(b) The algorithm we have used is needlessly cumbersome, because our only interest is the number of particles on each side. We used the positions only for visualization purposes. Because each particle has the same chance to go through the hole, the probability per unit time that a particle moves from left to right equals the number of particles on the left divided by the total number of particles, that is, $p = n/N$. Modify the program so that the following algorithm is implemented.

   (i) Generate a random number r from a uniformly distributed set of random numbers in the interval $0 \le r < 1$.

   (ii) If $r \le p = n/N$, move a particle from left to right, that is $n \to n-1$; otherwise, $n \to n+1$.

(c) Does the time dependence of n appear to be deterministic for sufficiently large N? What is the qualitative behavior of $n(t)$? Estimate the time for the system to reach equilibrium from the plots. How does this time depend on N? □

Problem 7.3. Equilibrium fluctuations

(a) As a rough measure of the equilibrium fluctuations, visually estimate the deviation of $n(t)$ from $N/2$ for $N = 16$, 64, 400, 800, and 3600? Choose a time interval that is bigger than the time needed to reach equilibrium. How do your results for the deviation depend on N?

(b) A better measure of the equilibrium fluctuations is the mean square fluctuations Δn^2, which is defined as

$$\Delta n^2 = \langle (n - \langle n\rangle)^2\rangle = \langle n^2\rangle - 2\langle n\langle n\rangle\rangle + \langle n\rangle^2 = \langle n^2\rangle - 2\langle n\rangle^2 + \langle n\rangle^2 = \langle n^2\rangle - \langle n\rangle^2. \tag{7.1}$$

The brackets $\langle\cdots\rangle$ denote an average taken after the system has reached equilibrium. The relative magnitude of the fluctuations is $\Delta n/\langle n\rangle$. Modify your program so that averages are taken after equilibrium has been reached. Run for a time that is long enough to obtain meaningful results. Compute the mean square fluctuations Δn^2 for the same values of N considered in part (a). How do the relative fluctuations, $\Delta n/\langle n\rangle$, depend on N? (You might find it helpful to see how averages are computed in Listings 7.3 and 7.4.) □

From Problem 7.2 we see that $n(t)$ decreases in time from its initial value to its equilibrium value in an almost deterministic manner if $N \gg 1$. It is instructive to derive the time dependence of $n(t)$ to show explicitly how chance can generate deterministic behavior. If there are

$n(t)$ particles on the left side after t moves, then the change in $\langle n\rangle(t)$ in the time interval Δt is given by

$$\Delta\langle n\rangle = \Big[\frac{-\langle n\rangle(t)}{N} + \frac{N-\langle n\rangle(t)}{N}\Big]\Delta t. \tag{7.2}$$

(We defined the time so that the time interval $\Delta t = 1$ in our simulations.) What is the meaning of the two terms in (7.2)? If we treat $\langle n\rangle$ and t as continuous variables and take the limit $\Delta t \to 0$, we have

$$\frac{\Delta\langle n\rangle}{\Delta t} \to \frac{d\langle n\rangle}{dt} = 1 - \frac{2\langle n\rangle(t)}{N}. \tag{7.3}$$

The solution of the differential equation (7.3) is

$$\langle n\rangle(t) = \frac{N}{2}[1+e^{-2t/N}] \tag{7.4}$$

where we have used the initial condition $\langle n\rangle(t=0) = N$. Note that $\langle n\rangle(t)$ decays exponentially to its equilibrium value $N/2$. How does this form (7.4) compare to your simulation results for various values of N? We can define a *relaxation time* τ as the time it takes the difference $[\langle n\rangle(t) - N/2]$ to decrease to $1/e$ of its initial value. How does τ depend on N? Does this prediction for τ agree with your results from Problem 7.2?

***Problem 7.4.** A simple modification

Modify your program so that each side of the box is chosen with equal probability. One particle is then moved from the side chosen to the other side. If the side chosen does not have a particle, then no particle is moved during this time interval. Do you expect that the system behaves in the same way as before? Do the simulation starting with all the particles on the left side of the box and choose $N = 800$. Do not keep track of the positions of the particles. Compare the behavior of $n(t)$ with the behavior of $n(t)$ found in Problem 7.3. How do the values of $\langle n\rangle$ and Δn compare? □

The probabilistic method discussed on page 198 for simulating the approach to equilibrium is an example of a *Monte Carlo* algorithm, that is, the random sampling of the most probable outcomes. An alternative method is to use *exact enumeration* and determine all the possibilities at each time interval. For example, suppose that $N = 8$ and $n(t = 0) = 8$. At $t = 1$ the only possibility is $n = 7$ and $n' = 1$. Hence, $P(n = 7, t = 1) = 1$, and all other probabilities are zero. At $t = 2$ one of the seven particles on the left can move to the right, or the one particle on the right can move to the left. Because the first possibility can occur in seven different ways, we have the nonzero probabilities, $P(n = 6, t = 2) = 7/8$ and $P(n = 8, t = 2) = 1/8$. Hence, at $t = 2$ the average number of particles on the left side of the box is

$$\langle n(t=2)\rangle = 6P(n=6,t=2) + 8P(n=8,t=2) = \frac{1}{8}[6\times 7 + 8\times 1] = 6.25. \tag{7.5}$$

Is this exact result consistent with what you found in Problem 7.2? In this example N is small, and we could continue the enumeration of all the possibilities indefinitely. However, for larger N the number of possibilities becomes very large after a few time intervals, and we need to to use Monte Carlo methods to sample the most probable outcomes.

7.2 Random Walks

In Section 7.1 we considered the random motion of many particles in a box, but we did not care about their positions—all we needed to know was the number of particles on each side.

Suppose that we want to characterize the motion of a dust particle in the atmosphere. We know that as a given dust particle collides with molecules in the atmosphere, it changes its direction frequently, and its motion appears to be random. A simple model for the trajectory of a dust particle in the atmosphere is based on the assumption that the particle moves in any direction with equal probability. Such a model is an example of a *random walk*.

The original statement of a random walk was formulated in the context of a drunken sailor. If a drunkard begins at a lamp post and takes N steps of equal length in random directions, how far will the drunkard be from the lamp post? We will find that the mean square displacement of a random walker, for example, a dust particle or a drunkard, grows linearly with time. This result and its relation to diffusion leads to many applications that might seem to be unrelated to the original drunken sailor problem.

We first consider an idealized example of a random walker that can move only along a line. Suppose that the walker begins at $x = 0$ and that each step is of equal length a. At each time interval the walker has a probability p of a step to the right and a probability $q = 1 - p$ of a step to the left. The direction of each step is independent of the preceding one. After N steps the displacement x of the walker from the origin is given by

$$x_N = \sum_{i=1}^{N} s_i \tag{7.6}$$

where $s_i = \pm a$. For $p = 1/2$ we can generate one walk of N steps by flipping a coin N times and increasing x by a each time the coin is heads and decreasing x by a each time the coin is tails.

We expect that if we average over a sufficient number of walks of N steps, then the average of x_N, denoted by $\langle x_N \rangle$, would be $(p-q)Na$. We can derive this result by writing $\langle x \rangle = \sum_{i=1}^{N} \langle s_i \rangle = N\langle s \rangle$, because the average of the sum is the sum of the averages, and the average of each step is the same. We have $\langle s \rangle = p(a) + q(-a) = (p-q)a$, and the result follows. For simplicity, we will frequently drop the subscript N and write $\langle x \rangle$.

Because $\langle x \rangle = 0$ for $p = 1/2$, we need a better measure of the extent of the walk. One measure is the displacement squared:

$$x_N^2 = \Big[\sum_{i=1}^{N} s_i\Big]^2. \tag{7.7}$$

For $p \neq 1/2$ it is convenient to consider the mean square net displacement Δx^2 defined as

$$\Delta x^2 \equiv \Big\langle \big(x - \langle x \rangle\big)^2 \Big\rangle \tag{7.8a}$$

$$= \langle x^2 \rangle - \langle x \rangle^2. \tag{7.8b}$$

(We have written Δx^2 rather than $\langle \Delta x^2 \rangle$ for simplicity.) To determine Δx^2, we write (7.8a) as the sum of two terms:

$$\Delta x^2 = \Big\langle \sum_{i=1}^{N} \Delta_i \sum_{j=1}^{N} \Delta_j \Big\rangle = \Big\langle \sum_{i=1}^{N} \Delta_i^2 \Big\rangle + \Big\langle \sum_{i \neq j=1}^{N} \Delta_i \Delta_j \Big\rangle \tag{7.9}$$

where $\Delta_i = s_i - \langle s \rangle$. The first sum on the right-hand side of (7.9) includes terms for which $i = j$; the second sum is over i and j with $i \neq j$. Because each step is independent, we have $\langle \Delta_i \Delta_j \rangle = \langle \Delta_i \rangle \langle \Delta_j \rangle$. This term is zero because $\langle \Delta_i \rangle = 0$ for any i. The first term in (7.9) equals $N\langle \Delta^2 \rangle = N[\langle s^2 \rangle - \langle s \rangle^2] = N[a^2 - (p-q)^2 a^2] = N4pqa^2$, where we have used the fact that $p + q = 1$. Hence

$$\Delta x^2 = 4pqNa^2 \qquad \text{(analytic result)}. \tag{7.10}$$

We can gain more insight into the nature of random walks by doing a Monte Carlo simulation, that is, by using a computer to "flip coins." The implementation of the random walk algorithm is simple, for example,

```
if (p < Math.random()) {
   x++;
}
else {
   x--;
}
```

Clearly we have to sample many N step walks because, in general, each walk will give a different outcome. We need to do a Monte Carlo simulation many times and average over the results to obtain meaningful averages. Each N-step walk is called a *trial*. How do we know how many trials to use? The simple answer is to average over more and more trials until the average results don't change within the desired accuracy. The more sophisticated answer is to do an error analysis similar to what we do in measurements in the laboratory. Such an analysis is discussed in Section 11.4.

The more difficult parts of a program to simulate random walks are associated with bookkeeping. The walker takes a total of N steps in each trial, and the net displacement x is computed after every step. Our convention will be to use a variable name ending in Accumulator (or Accum) to denote a variable that accumulates the value of some variable. In Listing 7.3 we provide two classes to be used to simulate random walks.

Listing 7.3: Listing of Walker class.

```
package org.opensourcephysics.sip.ch07;
public class Walker {
   // accumulated data on displacement of walkers, index is time
   int xAccum[], xSquaredAccum[];
   int N;                          // maximum number of steps
   double p;                       // probability of step to the right
   int position;                   // position of walker

   public void initialize() {
      xAccum = new int[N+1];
      xSquaredAccum = new int[N+1];
   }

   public void step() {
      position = 0;
      for(int t = 0;t<N;t++) {
         if(Math.random()<p) {
            position++;
         } else {
            position--;
         }
         // determine displacement of walker after each step
         xAccum[t+1] += position;
         xSquaredAccum[t+1] += position*position;
      }
   }
}
```

Listing 7.4: Target class for random walk simulation.

```
package org.opensourcephysics.sip.ch07;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.frames.*;

public class WalkerApp extends AbstractSimulation {

   Walker walker = new Walker();
   PlotFrame plotFrame = new PlotFrame("time", "<x>,<x^2>", "Averages");
   HistogramFrame distribution =
        new HistogramFrame("x", "H(x)", "Histogram");
   int trials; // number of trials

   public WalkerApp() {
      plotFrame.setXYColumnNames(0, "t", "<x>");
      plotFrame.setXYColumnNames(1, "t", "<x^2>");
   }

   public void initialize() {
      walker.p = control.getDouble("Probability p of step to right");
      walker.N = control.getInt("Number of steps N");
      walker.initialize();
      trials = 0;
   }


   public void doStep() {
      trials++;
      walker.step();
      distribution.append(walker.position);
      distribution.setMessage("trials = "+trials);
   }

   public void stopRunning() {
      plotFrame.clearData();
      for(int t = 0; t<=walker.N; t++) {
         double xbar = walker.xAccum[t]*1.0/trials;
         double x2bar = walker.xSquaredAccum[t]*1.0/trials;
         plotFrame.append(0, 1.0*t, xbar);
         plotFrame.append(1, 1.0*t, x2bar - xbar*xbar);
      }
      plotFrame.repaint();
   }

   public void reset() {
      control.setValue("Probability p of step to right", 0.5);
      control.setValue("Number of steps N", 100);
   }

   public static void main(String[] args) {
      SimulationControl.createApp(new WalkerApp());
   }
}
```

Problem 7.5. Random walks in one dimension

(a) In class Walker the steps are of unit length so that $a = 1$. Use Walker and WalkerApp to estimate the number of trials needed to obtain Δx^2 for $N = 20$ and $p = 1/2$ with an accuracy of approximately 5%. Compare your result for Δx^2 to the exact answer in (7.10). Approximately how many trials do you need to obtain the same relative accuracy for $N = 100$?

(b) Is $\langle x \rangle$ exactly zero in your simulations? Explain the difference between the analytic result and the results of your simulations. Note that we have used the same notation $\langle \ldots \rangle$ to denote the exact average calculated analytically and the approximate average computed by averaging over many trials. The distinction between the two averages should be clear from the context.

(c) How do your results for $\langle x \rangle$ and Δx^2 change for $p \neq q$? Choose $p = 0.7$ and determine the N dependence of $\langle x \rangle$ and Δx^2.

(d)* Determine Δx^2 for $N = 1$ to $N = 5$ by enumerating all the possible walks. For simplicity, choose $p = 1/2$ so that $\langle x \rangle = 0$. For $N = 1$ there are two possible walks: one step to the right and one step to the left. In both cases $x^2 = 1$, and hence $\langle {x_1}^2 \rangle = 1$. For $N = 2$ there are four possible walks with the same probability: (i) two steps to the right, (ii) two steps to the left, (iii) first step to the right and second step to the left, and (iv) first step to the left and second step to the right. The value of ${x_2}^2$ for these walks is 4, 4, 0, and 0, respectively, and hence $\langle {x_2}^2 \rangle = (4 + 4 + 0 + 0)/4 = 2$. Write a program that enumerates all the possible walks of a given number of steps and compute the various averages of interest exactly. □

The class WalkerApp displays the distribution of values of the displacement x after N steps. One way of determining the number of times that the variable x has a certain value would be to define a one-dimensional array, probability, and let

```
probability[x] += 1;
```

In this case because x takes only integer values, the array index of probability is the same as x itself. However, the above statement does not work in Java because x can be negative as well as positive. What we need is a way of mapping the value x to a bin or index number. The HistogramFrame class, which is part of the Open Source Physics display package, does this mapping automatically using the Java Hashtable class. In simple data structures, data is accessed by an index that indicates the location of the data in the data structure. Hashtable data is accessed by a *key*, which in our case is the value of x. A hashing function converts the key to an index. The append method of the HistogramFrame class takes a value, finds the index using a hashing function, and then increments the data associated with that key. The HistogramFrame class also draws itself.

The HistogramFrame class is very useful for taking a quick look at the distribution of values in a data set. You do not need to know how to group the data into bins or the range of values of the data. The default bin width is unity, but the bin width can be set using the setBinWidth method. See WalkerApp for an example of the use of the HistogramFrame class. Frequently, we wish to use the histogram data to compute other quantities. You can collect the data using the Data Table menu item in HistogramFrame and copy the data to a file. Another option is to include additional code in your program to analyze the data. The following statements assume that a HistogramFrame object called histogram has been created and data entered into it.

```
// creates array entries of data from histogram
java.util.Map.Entry[] entries = histogram.entries();
for (int i = 0, length = entries.length; i < length; i++) {
   // gets bin number
   Integer binNumber = (Integer) entries[i].getKey();
   // gets number of occurrences for bin number i
   Double occurrences = (Double) entries[i].getValue();
   // gets value of left edge of bin
   double value = histogram.getLeftMostBinPosition(binNumber.intValue());
   // sets value to middle of bin
   value += 0.5*histogram.getBinWidth();
   // convert from Double class to double data type
   double number = occurrences.doubleValue();
   // use value and number in your analysis
}
```

Problem 7.6. Nature of the probability distribution

(a) Compute $P_N(x)$, the probability that the displacement of the walker from the origin is x after N steps. What is the difference between the histogram, that is, the number of occurrences, and the probability? Consider $N = 10$ and $N = 40$ and at least 1000 trials. Does the qualitative form of $P_N(x)$ change as the number of trials increases? What is the approximate width of $P_N(x)$ and the value of $P_N(x)$ at its maximum for each value of N?

(b) What is the approximate shape of the envelope of $P_N(x)$? Does the shape change as N is increased?

(c) Fit the envelope $P_N(x)$ for sufficiently large N to the continuous function

$$C\frac{1}{\sqrt{2\pi\Delta x^2}}e^{-(x-\langle x\rangle)^2/2\Delta x^2}. \tag{7.11}$$

The form of (7.11) is the standard form of the Gaussian distribution with $C = 1$. The easiest way to do this fit is to plot your results for $P_N(x)$ and the form (7.11) on the same graph using your results for $\langle x\rangle$ and Δx^2 as input parameters. Visually choose the constant C to obtain a reasonable fit. What are the possible values of x for a given value of N? What is the minimum difference between these values? How does this difference compare to your value for C? □

Problem 7.7. More random walks in one dimension

(a) Suppose that the probability of a step to the right is $p = 0.7$. Compute $\langle x\rangle$ and Δx^2 for $N = 4$, 8, 16, and 32. What is the interpretation of $\langle x\rangle$ in this case? What is the qualitative dependence of Δx^2 on N?

(b) An interesting property of random walks is the mean number $\langle D_N\rangle$ of *distinct* lattice sites visited during the course of an N step walk. Do a Monte Carlo simulation of $\langle D_N\rangle$ and determine its N dependence. □

We can consider either a large number of successive walks as in Problem 7.7 or a large number of noninteracting walkers moving at the same time as in Problem 7.8.

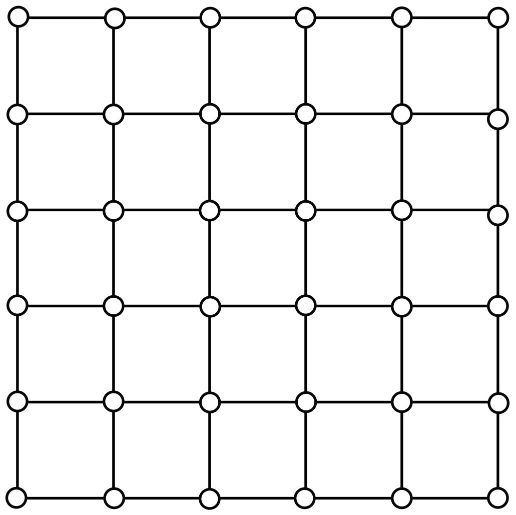


Figure 7.2: An example of a 6×6 square lattice. Note that each site or node has four nearest neighbors.

Problem 7.8. A random walk in two dimensions

(a) Consider a collection of walkers initially at the origin of a square lattice (see Figure 7.2). At each unit of time, each of the walkers moves at random with equal probability in one of the four possible directions. Create a drawable class, `Walker2D`, which contains the positions of M walkers moving in two dimensions and draws their location, and modify `WalkerApp`. Unlike `WalkerApp`, this new class need not specify the maximum number of steps. Instead, the number of walkers should be specified.

(b) Run your application with the number of walkers $M \geq 1000$ and allow the walkers to take at least 500 steps. If each walker represents a bee, what is the qualitative nature of the shape of the swarm of bees? Describe the qualitative nature of the surface of the swarm as a function of the number of steps N. Is the surface jagged or smooth?

(c) Compute the quantities $\langle x \rangle$, $\langle y \rangle$, Δx^2, and Δy^2 as a function of N. The average is over the M walkers. Also compute the mean square displacement R^2 given by

$$R^2 = \langle x^2 \rangle - \langle x \rangle^2 + \langle y^2 \rangle - \langle y \rangle^2 = \Delta x^2 + \Delta y^2. \tag{7.12}$$

What is the dependence of each quantity on N? (As before, we will frequently write R^2 instead of R_N^2.)

(d) Estimate R^2 for $N = 8$, 16, 32, and 64 by averaging over a large number of walkers for each value of N. Assume that $R = \sqrt{R^2}$ has the asymptotic N dependence:

$$R \sim N^{\nu} \qquad (N \gg 1), \tag{7.13}$$

and estimate the exponent ν from a log-log plot of R^2 versus N. We will see in Chapter 13 that the exponent $1/\nu$ is related to how a random walk fills space. If $\nu \approx 1/2$, estimate the magnitude of the self-diffusion coefficient D from the relation $R^2 = 4DN$.

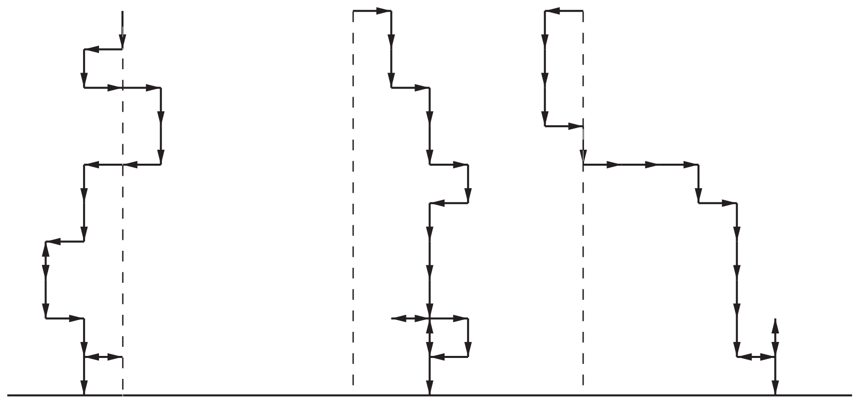


Figure 7.3: Examples of the random path of a raindrop to the ground. The step probabilities are given in Problem 7.9. The walker starts at $x = 0$, $y = h$.

(e) Do a Monte Carlo simulation of R^2 on a triangular lattice (see Figure 8.5) and estimate ν. Can you conclude that the exponent ν is independent of the symmetry of the lattice? Does D depend on the symmetry of the lattice? If so, give a qualitative explanation for this dependence.

(f)* Enumerate all the random walks on a square lattice for $N = 4$ and obtain exact results for $\langle x\rangle$, $\langle y\rangle$, and R^2. Assume that all four directions are equally probable. Verify your program by comparing the Monte Carlo and exact enumeration results. □

Problem 7.9. The fall of a rain drop

Consider a random walk that starts at a site a distance $y = h$ above a horizontal line (see Figure 7.3). If the probability of a step down is greater than the probability of a step up, we expect that the walker will eventually reach a site on the horizontal line. This walk is a simple model of the fall of a rain drop in the presence of a random swirling breeze. Do a Monte Carlo simulation to determine the mean time τ for the walker to reach any site on the line $y = 0$ and find the functional dependence of τ on h. Is it possible to define a velocity in the vertical direction? Because the walker does not always move vertically, it suffers a net displacement x in the horizontal direction. How does Δx^2 depend on h and τ? Reasonable values for the step probabilities are 0.1, 0.6, 0.15, 0.15, corresponding to up, down, right, and left, respectively. □

7.3 Modified Random Walks

So far we have considered random walks on one- and two-dimensional lattices where the walker has no "memory" of the previous step. What happens if the walker remembers the nature of the previous steps? What happens if there are multiple random walkers, with the condition that no double occupancy is allowed? We explore these and other variations of the simple random walk in this section. All these variations have applications to physical systems, but the applications are more difficult to understand than the models themselves.

The fall of a raindrop considered in Problem 7.9 is an example of a *restricted* random walk, that is, a walk in the presence of a boundary. In the following problem, we discuss in a more general context the effects of various types of restrictions or boundaries on random walks. Other examples of a restricted random walk are given in Problems 7.17 and 7.23.

Problem 7.10. Restricted random walks

(a) Consider a one-dimensional lattice with trap sites at $x = 0$ and $x = L\,(L > 0)$. A walker begins at site x_0 $(0 < x_0 < L)$ and takes unit steps to the left and right with equal probability. When the walker arrives at a trap site, it can no longer move. Do a Monte Carlo simulation and verify that the mean number of steps τ for the particle to be trapped (*the mean first passage time*) is given by

$$\tau = (2D)^{-1} x_0(L - x_0) \tag{7.14}$$

where D is the self-diffusion coefficient in the absence of traps [see (7.29)].

(b) Random walk models in the presence of traps have had an important role in condensed matter physics. For example, consider the following idealized model of energy transport in solids. The solid is represented as a lattice with two types of sites: hosts and traps. An incident photon is absorbed at a host site and excites the host molecule or atom. The excitation energy or *exciton* is transferred at random to one of the host's nearest neighbors, and the original excited molecule returns to its ground state. In this way the exciton wanders through the lattice until it reaches a trap site at which a chemical reaction occurs. A simple version of this energy transport model is given by a one-dimensional lattice with traps placed on a periodic sublattice. Because the traps are placed at regular intervals, we can replace the random walk on an infinite lattice by a random walk on a circular ring. Consider a lattice of N host or nontrapping sites and one trap site. If a walker has an equal probability of starting from any host site and an equal probability of a step to each nearest neighbor site, what is the N dependence of the mean survival time τ (the mean number of steps taken before a trap site is reached)? Use the results of part (a) rather than doing a simulation.

(c) Consider a one-dimensional lattice with reflecting sites at $x = -L$ and $x = L$. For example, if a walker reaches the reflecting site at $x = L$, it is reflected at the next step to $x = L - 1$. At $t = 0$ the walker starts at $x = 0$ and steps with equal probability to the left and right. Write a Monte Carlo program to determine $P_N(x)$, the probability that the walker is at site x after N steps. Compare the form of $P_N(x)$ with and without the presence of the reflecting sites. Can you distinguish the two probability distributions if N is the order of L? At what value of N can you first distinguish the two distributions? □

Problem 7.11. A persistent random walk

(a) In a persistent random walk, the *transition* or jump probability depends on the previous step. Consider a walk on a one-dimensional lattice, and suppose that step $N - 1$ has been made. Then step N is made in the same direction with probability α; a step in the opposite direction occurs with probability $1 - \alpha$. Write a program to do a Monte Carlo simulation of the persistent random walk in one dimension. Estimate $\langle x \rangle$, Δx^2, and $P_N(x)$. Note that it is necessary to specify both the initial position and an initial direction of the walker. What is the $\alpha = 1/2$ limit of the persistent random walk?

(b) Consider $\alpha = 0.25$ and $\alpha = 0.75$ and determine Δx^2 for $N = 8$, 64, 256, and 512. Assume that $\Delta x^2 \sim N^{2\nu}$ for large N, and estimate the value of ν from a log-log plot of Δx^2 versus N

for large N. Does ν depend on α? If $\nu \approx 1/2$, determine the self-diffusion coefficient D for $\alpha = 0.25$ and 0.75. In general, D is given by

$$D = \frac{1}{2d} \lim_{N\to\infty} \frac{\Delta x^2}{N} \tag{7.15}$$

where d is the dimension of space. That is, D is given by the asymptotic behavior of the mean square displacement. (For the simple random walk considered in Section 7.2, $\Delta x^2 \propto N$ for all N.) Give a physical argument for why $D(\alpha \neq 0.5)$ is greater (smaller) than $D(\alpha = 0.5)$.

(c) You might have expected that the persistent random walk yields a nonzero value for $\langle x\rangle$. Verify that $\langle x\rangle = 0$, and explain why this result is exact. How does the persistent random walk differ from the biased random walk for which $p \neq q$?

(d) A persistent random walk can be considered as an example of a *multistate* walk in which the state of the walk is defined by the last transition. The walker is in one of two states; at each step the probabilities of remaining in the same state or switching states are α and $1-\alpha$, respectively. One of the earliest applications of a two-state random walk was to the study of diffusion in a chromatographic column. Suppose that a molecule in a chromatographic column can be either in a mobile phase (constant velocity v) or in a trapped phase (zero velocity). Instead of each step changing the position by ± 1, the position at each step changes by $+v$ or 0. A quantity of experimental interest is the probability $P_N(x)$ that a molecule has moved a distance x in N steps. Choose $v = 1$ and $\alpha = 0.75$ and determine the qualitative behavior of $P_N(x)$. □

Problem 7.12. Synchronized random walks

(a) Randomly place two walkers on a one-dimensional lattice of L sites, so that both walkers are not at the same site. At each time step randomly choose whether the walkers move to the left or to the right. Both walkers move in the same direction. If a walker cannot move in the chosen direction because it is at a boundary, then this walker remains at the same site for this time step. A trial ends when both walkers are at the same site. Write a program to determine the mean time and the mean square fluctuations of the time for two walkers to reach the same site. This model is relevant to a method of doing cryptography using neural networks (see Rutter et al.).

(b) Change your program so that you use biased random walkers for which $p \neq q$. How does this change affect your results? □

Problem 7.13. Random walk on a continuum

One of the first continuum models of a random walk was proposed by Rayleigh in 1919. In this model the length a of each step is a random variable and the direction of each step is uniformly random. In this case the variable of interest is R, the distance of the walker from the origin after N steps. The model is known as the freely jointed chain in polymer physics (see Section 7.7) in which case R is the end-to-end distance of the polymer. For simplicity, we first consider a walker in two dimensions with steps of equal (unit) length at a random angle.

(a) Write a Monte Carlo program to compute $\langle R\rangle$ and determine its dependence on N.

(b) Because R is a continuous variable, we need to compute $p_N(R)\Delta R$, the probability that R is between R and $R + \Delta R$ after N steps. The quantity $p_N(R)$ is the *probability density*. Because the area of the ring between R and $R+\Delta R$ is $\pi(R+\Delta R)^2 - \pi R^2 = 2\pi R\Delta R + \pi(\Delta R)^2 \approx 2\pi R\Delta R$, we see that $p_N(R)\Delta R$ is proportional to $R\Delta R$. Verify that for sufficiently large N, $p_N(R)\Delta R$ has the form

$$p_N(R)\Delta R \propto 2\pi R\Delta R\, e^{-(R-\langle R\rangle)^2/2\Delta R^2} \tag{7.16}$$

where $\Delta R^2 = \langle R^2\rangle - \langle R\rangle^2$. □

Problem 7.14. Random walks with steps of variable length

(a) Consider a random walk in one dimension with jumps of all lengths. The probability that the length of a single step is between a and $a+\Delta a$ is $f(a)\Delta a$, where $f(a)$ is the probability density. If the form of $f(a)$ is given by $f(a) = C\,e^{-a}$ for $a>0$ with the normalization condition $\int_0^\infty f(a)\,da = 1$, the code needed to generate step lengths according to this probability density is given by (see Section 11.5)

```
stepLength = -Math.log(1 - Math.random());
```

Modify `Walker` and `WalkerApp` to simulate walks of variable length with this probability density. Consider $N \geq 100$ and visualize the motion of the walker. Generate many walks of N steps and determine $p(x)\Delta x$, the probability that the displacement is between x and $x+\Delta x$ after N steps. Plot $p(x)$ versus x and confirm that the form of $p(x)$ is consistent with a Gaussian distribution. Note that the bin width Δa is one of the input parameters.

(b) Assume that the probability density $f(a)$ is given by $f(a) = C/a^2$ for $a \geq 1$. Determine the normalization constant C using the condition $C\int_1^\infty a^{-2}\,da = 1$. In this case we will learn in Section 11.5 that the statement

```
stepLength = 1.0/(1.0 - Math.random());
```

generates values of a according to this form of $f(a)$. Do a Monte Carlo simulation as in part (a) and determine $p(x)\Delta x$. Is the form of $p(x)$ a Gaussian? This type of random walk, for which $f(a)$ decreases as a power law $a^{-1-\alpha}$, is known as a *Levy flight* for $\alpha \leq 2$. □

Problem 7.15. Exploring the central limit theorem

Consider a continuous random variable x with probability density $f(x)$. That is, $f(x)\Delta x$ is the probability that x has a value between x and $x+\Delta x$. The mth *moment* of $f(x)$ is defined as

$$\langle x^m\rangle = \int x^m f(x)\,dx. \tag{7.17}$$

The mean value $\langle x\rangle$ is given by (7.17) with $m=1$. The *variance* σ_x^2 of $f(x)$ is defined as

$$\sigma_x^2 = \langle x^2\rangle - \langle x\rangle^2. \tag{7.18}$$

Consider the sum y_n corresponding to the average of n values of x:

$$y = y_n = \frac{1}{n}(x_1 + x_2 + \cdots + x_n). \tag{7.19}$$

Suppose that we make many measurements of y. We know that the values of y will not be identical but will be distributed according to a probability density $p(y)$, where $p(y)\Delta y$ is the probability that the measured value of y is in the range y to $y+\Delta y$. The main quantities of interest are $\langle y\rangle$, $p(y)$, and an estimate of the probable variability of y in a series of measurements.

(a) Suppose that $f(x)$ is uniform in the interval $[-1,1]$. Calculate $\langle x\rangle$, $\langle x^2\rangle$, and σ_x analytically.

(b) Write a program to make a sufficient number of measurements of y and determine $\langle y\rangle$ and $p(y)\Delta y$. Use the `HistogramFrame` class to determine and plot $p(y)\Delta y$. Choose at least 10^4 measurements of y for $n = 4$, 16, 32, and 64. What is the qualitative form of $p(y)$? Does the qualitative form of $p(y)$ change as the number of measurements of y is increased for a given value of n? Does the qualitative form of $p(y)$ change as n is increased?

(c) Each value of y can be considered to be a measurement. How much does the value of y vary (on the average) from one measurement to another? Make a rough estimate of this variability by comparing several measurements of y for a given value of n. Increase n by a factor of four and estimate the variability of y again. Does the variability from one measurement to another decrease (on the average) as n is increased?

(d) The *sample variance* $\tilde{\sigma}^2$ is given by

$$\tilde{\sigma}^2 = \frac{\sum_{i=1}^{n}[y_i - \langle y\rangle]^2}{n-1}. \tag{7.20}$$

The reason for the factor of $n-1$ rather than n in (7.20) is that to compute $\tilde{\sigma}^2$, we need to use the n values of x to compute the mean y, and thus, loosely speaking, we have only $n-1$ independent values of x remaining to calculate $\tilde{\sigma}^2$. Show that if $n \gg 1$, then $\tilde{\sigma}^2 \approx \sigma_y^2$, where σ_y^2 is given by

$$\sigma_y^2 = \langle y^2\rangle - \langle y\rangle^2. \tag{7.21}$$

(e) The quantity $\tilde{\sigma}$ is known as the *standard deviation of the mean*. That is, $\tilde{\sigma}$ gives a measure of how much variation we expect to find if we make repeated measurements of y. How does the value of $\tilde{\sigma}$ compare with your estimate of the variability in part (b)?

(f) What is the qualitative shape of the probability density $p(y)$ that you obtained in part (b)? What is the order of magnitude of the width of the probability?

(g) Verify from your results that $\tilde{\sigma} \approx \sigma_y \approx \sigma_x/\sqrt{n-1} \approx \sigma_x/\sqrt{n}$.

(h) To test the generality of your results, consider the exponential probability density

$$f(x) = \begin{cases} e^{-x} & x \geq 0 \\ 0 & x < 0. \end{cases} \tag{7.22}$$

Calculate $\langle x\rangle$ and σ_x analytically. Modify your Monte Carlo program and estimate $\langle y\rangle$, $\tilde{\sigma}$, σ_y, and $p(y)$. How are $\tilde{\sigma}$, σ_y, and σ_x related for a given value of n? Plot $p(y)$ and discuss its qualitative form and its dependence on n and on the number of measurements of y. □

Problem 7.15 illustrates the *central limit theorem*, which states that the probability distribution of a sum of random variables, the random variable y, is a Gaussian centered at $\langle y\rangle$ with a standard deviation approximately given by $1/\sqrt{n}$ times the standard deviation of $f(x)$. The requirements are that $f(x)$ has finite first and second moments, that the measurements of y are statistically independent, and that n is large. What is the relation of the central limit theorem to the calculations of the probability distribution in the random walk models that we have considered?

Problem 7.16. Generation of the Gaussian distribution
Consider the sum

$$y = \sum_{i=1}^{12} r_i \tag{7.23}$$

where r_i is a uniform random number in the unit interval. Make many measurements of y and show that the probability distribution of y approximates the Gaussian distribution with mean value 6 and variance 1. What is the relation of this result to the central limit theorem? Discuss how to use this result to generate a Gaussian distribution with arbitrary mean and variance. This way of generating a Gaussian distribution is particularly useful when a "quick and dirty" approximation is appropriate. A better method for generating a sequence of random numbers distributed according to the Gaussian distribution is discussed in Section 11.5. □

Many of the problems we have considered have revealed the slow convergence of Monte Carlo simulations and the difficulty of obtaining quantitative results for asymptotic quantities. We conclude this section with a cautionary note and consider a "simple" problem for which straightforward Monte Carlo methods give misleading asymptotic results.

***Problem 7.17.** Random walk on lattices containing random traps

(a) In Problem 7.10 we considered the mean survival time of a one-dimensional random walker in the presence of a periodic distribution of traps. Now suppose that the trap sites are distributed at *random* on a one-dimensional lattice with density $\rho = N/L$. For example, if $\rho = 0.01$, the probability that a site is a trap site is 1%. (A site is a trap site if $r \leq \rho$, where, as usual, r is uniformly distributed in the interval $0 \leq r < 1$.) If a walker is placed at random at any nontrapping site, determine its mean survival time τ; that is, the mean number of steps before a trap site is reached. Assume that the walker has an equal probability of moving to a nearest neighbor site at each step and use periodic boundary conditions; that is, the lattice sites are located on a ring. The major complication is that it is necessary to perform *three* averages: the distribution of traps, the origin of the walker, and the different walks for a given trap distribution and origin. Choose reasonable values for the number of trials associated with each average and do a Monte Carlo simulation to estimate the mean survival time τ. If τ exhibits a power law dependence on ρ, for example $\tau \approx \tau_0 \rho^{-z}$, estimate the exponent z.

(b) A seemingly straightforward extension of part (a) is to estimate the survival probability S_N after N steps. Choose $\rho = 0.5$ and do a Monte Carlo simulation of S_N for N as large as possible. (Published results are for $N = 3 \times 10^4$, on lattices large enough that a walker doesn't reach the boundary, with about 54 000 trials.) Assume that the asymptotic form of S_N for large N is given by

$$S_N \sim e^{-bN^\alpha} \tag{7.24}$$

where the exponent α is the quantity of interest, and b is a constant that depends on ρ. Are your results consistent with this form? Is it possible to make a meaningful estimate of the exponent α?

(c) It has been proven that the asymptotic N dependence of S_N has the form (7.24) with $\alpha = 1/3$. Are your Monte Carlo results consistent with this value of α? The object of part (b) is to convince you that it is not possible to use simple Monte Carlo methods directly to obtain the correct asymptotic behavior of S_N. The difficulty is that we are trying to estimate S_N in the asymptotic region where S_N is very small, and the small number of trials in this region prevents us from obtaining meaningful results.

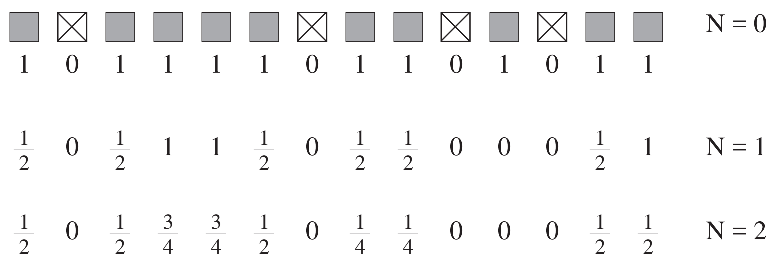


Figure 7.4: Example of the exact enumeration of walks on a given configuration of traps. The filled and empty squares denote regular and trap sites, respectively. At step $N = 0$, a walker is placed at each regular site. The numbers at each site i represent the number of walkers w_i. Periodic boundary conditions are used. The initial number of walkers in this example is $w_0 = 10$. The mean survival probability at step $N = 1$ and $N = 2$ is found to be 0.6 and 0.475, respectively.

(d) One way to reduce the number of required averages is to determine exactly the probability that the walker is at site i after N steps for a given distribution of trap sites. The method is illustrated in Figure 7.4. The first line represents a given configuration of traps distributed randomly on a one-dimensional lattice. One walker is placed at each nontrap site; trap sites are assigned the value 0. Because each walker moves with probability 1/2 to each neighbor, the number of walkers $w_i(N + 1)$ on site i at step $N + 1$ is given by

$$w_i(N+1) = \frac{1}{2}[w_{i+1}(N) + w_{i-1}(N)]. \tag{7.25}$$

(Compare the relation (7.25) to the relation that you found in Problem 7.5d.) The survival probability S_N after N steps for a given configuration of traps is given exactly by

$$S_N = \frac{1}{w_0}\sum_i w_i(N) \tag{7.26}$$

where w_0 is the initial number of walkers and the sum is over all sites in the lattice. Explain the relation (7.26) and write a program that computes S_N using (7.25) and (7.26). Then obtain $\langle S_N\rangle$ by averaging over several configurations of traps. Choose $\rho = 0.5$ and determine S_N for $N = 32$, 64, 128, 512, and 1024. Choose periodic boundary conditions and as large a lattice as possible. How well can you estimate the exponent α? For comparison Havlin et al. consider a lattice of $L = 50,000$ and values of N up to 10^7. □

One reason that random walks are very useful in simulating many physical processes is that they are closely related to solutions of the *diffusion* equation. The one-dimensional diffusion equation can be written as

$$\frac{\partial P(x,t)}{\partial t} = D\frac{\partial^2 P(x,t)}{\partial x^2} \tag{7.27}$$

where D is the self-diffusion coefficient and $P(x,t)\Delta x$ is the probability of a particle being in the interval between x and $x + \Delta x$ at time t. In a typical application $P(x,t)$ might represent the concentration of ink molecules diffusing in a fluid. In three dimensions the second derivative $\partial^2/\partial x^2$ is replaced by the Laplacian ∇^2. In Appendix 7B we show that the solution to the diffusion equation with the boundary condition $P(x=\pm\infty,t)=0$ yields

$$\langle x(t)\rangle = 0 \tag{7.28}$$

and

$$\langle x^2(t)\rangle = 2Dt \qquad \text{(one dimension)}. \tag{7.29}$$

If we compare the form of (7.29) with (7.10), we see that the random walk on a one-dimensional lattice and the diffusion equation give the same time dependence if we identify t with $N\Delta t$ and D with $a^2/\Delta t$.

The relation of discrete random walks to the diffusion equation is an example of how we can approach many problems in several ways. The traditional way to treat diffusion is to formulate the problem as a partial differential equation as in (7.27). The usual method for solving (7.27) numerically is known as the Crank–Nicholson method (see Press et al.). One difficulty with this approach is the treatment of complicated boundary conditions. An alternative is to formulate the problem as a random walk on a lattice for which it is straightforward to incorporate various boundary conditions. We will consider random walks in many contexts (see, for example, Section 10.5 and Chapter 16).

7.4 The Poisson Distribution and Nuclear Decay

As we have seen, we can often change variable names and consider a seemingly different physical problem. Our goal in this section is to discuss the decay of unstable nuclei, but we first discuss a conceptually easier problem related to throwing darts at random. Related physical problems are the distribution of stars in the sky and the distribution of photons on a photographic plate.

Suppose we randomly throw $N=100$ darts at a board that has been divided into $M=1000$ equal size regions. The probability that a dart hits a given region or cell in any one throw is $p=1/M$. If we count the number of darts in the different regions, we would find that most cells are empty, some cells have one dart, and other cells have more than one dart. What is the probability $P(n)$ that a given cell has n darts?

Problem 7.18. Throwing darts

Write a program that simulates the throwing of N darts at random into M cells in a dart board. Throwing a dart at random at the board is equivalent to choosing an integer at random between 1 and M. Determine $H(n)$, the number of cells with n darts. Average $H(n)$ over many trials and then compute the probability distribution

$$P(n) = \frac{H(n)}{M}. \tag{7.30}$$

As an example, choose $N=50$ and $M=500$. Choose the number of trials to be sufficiently large. so that you can determine the qualitative form of $P(n)$. What is $\langle n\rangle$? □

In this case the probability p that a dart lands in a given cell is much less then unity. The conditions $N \gg 1$ and $p \ll 1$ with $\langle n\rangle = Np$ fixed and the independence of the events (the presence of a dart in a particular cell) satisfy the requirements for a *Poisson distribution*. The form of the Poisson distribution is

$$P(n) = \frac{\langle n\rangle^n}{n!} e^{-\langle n\rangle} \tag{7.31}$$

where n is the number of darts in a given cell and $\langle n \rangle$ is the mean number, $\langle n \rangle = \sum_{n=0}^{N} nP(n)$. Because $N \gg 1$, we can take the upper limit of this sum to be ∞ when it is convenient.

Problem 7.19. Darts and the Poisson distribution

(a) Write a program to compute $\sum_{n=0} P(n)$, $\sum_{n=0} nP(n)$, and $\sum_{n=0} n^2 P(n)$ using the form (7.31) for $P(n)$ and reasonable values of p and N. Verify that $P(n)$ in (7.31) is normalized. What is the value of $\sigma_n^2 = \langle n^2 \rangle - \langle n \rangle^2$ for the Poisson distribution?

(b) Modify the program that you developed in Problem 7.18 to compute $\langle n \rangle$ as well as $P(n)$. Choose $N = 50$ and $M = 1000$. How do your computed values of $P(n)$ compare to the Poisson distribution in (7.31) using your measured value of $\langle n \rangle$ as input? If time permits, use larger values of N and M.

(c) Choose $N = 50$ and $M = 100$ and redo part (b). Are your results consistent with a Poisson distribution? What happens if $M = N = 50$? □

Now that we are more familiar with the Poisson distribution, we consider the decay of radioactive nuclei. We know that a collection of radioactive nuclei will decay; however, we cannot know a priori which nucleus will decay next. If all nuclei of a particular type are identical, why do they not all decay at the same time? The answer is based on the uncertainty inherent in the quantum description of matter at the microscopic level. In the following, we will see that a simple model of the decay process leads to exponential decay. This approach complements the continuum approach discussed in Section 3.9.

Because each nucleus is identical, we assume that during any time interval Δt, each nucleus has the same probability per unit time p of decaying. The basic algorithm is simple – choose an unstable nucleus and generate a random number r uniformly distributed in the unit interval $0 \le r < 1$. If $r \le p$, the unstable nucleus decays; otherwise, it does not. Each unstable nucleus is tested once during each time interval. Note that for a system of unstable nuclei, there are many events that can happen during each time interval; for example, $0, 1, 2, \ldots, n$ nuclei can decay. Once a nucleus decays, it is no longer in the group of unstable nuclei that is tested at each time interval. Class `Nuclei` in Listing 7.5 implements the nuclear decay algorithm.

Listing 7.5: The `Nuclei` class.

```
package org.opensourcephysics.sip.ch07;
public class Nuclei {
   int n[];
   // accumulated data on number of unstable nuclei, index is time
   int tmax; // maximum time to record data
   int n0;   // initial number of unstable nuclei
   double p; // decay probability

   public void initialize() {
      n = new int[tmax+1];
   }

   public void step() {
      n[0] += n0;
      int nUnstable = n0;
      for(int t = 0;t<tmax;t++) {
         for(int i = 0;i<nUnstable;i++) {
            if(Math.random()<p) {
               nUnstable--;
            }
         }
         n[t+1] += nUnstable;
      }
   }
}
```

Problem 7.20. Monte Carlo simulation of nuclear decay

(a) Write a target class that extends `AbstractSimulation`, does many trials, and plots the average number of unstable nuclei as a function of time. Assume that the time interval Δt is one second. Choose the initial number of unstable nuclei `n0` $= 10,000$, $p = 0.01$, and `tmax` $= 100$ and average over 100 trials. Is your result for $n(t)$, the mean number of unstable nuclei at time t, consistent with the expected behavior $n(t) = n(0)e^{-\lambda t}$ found in Section 3.9? What is the value of λ for this value of p?

(b) There are a very large number of unstable nuclei in a typical radioactive source. We also know that over any reasonable time interval, only a relatively small number decay. Because $N \gg 1$ and $p \ll 1$, we expect that $P(n)$, the probability that n nuclei decay during a specified time interval, is a Poisson distribution. Modify your target class so that it outputs the probability that n unstable nuclei decay during the first time interval. Choose `n0` $= 1000$, $p = 0.001$, and `tmax` $= 1$ and average over 1000 trials. What is the mean number $\langle n \rangle$ of nuclei that decay during this interval? What is the associated variance? Plot $P(n)$ versus n and compare your results to the Poisson distribution (7.31) with your measured value of $\langle n \rangle$ as input. Then consider $p = 0.02$ and determine if $P(n)$ is a Poisson distribution.

(c) Modify your target class so that it outputs the probability that n unstable nuclei decay during the first two time intervals. Choose `n0` $= 10000$, $p = 0.001$, and `tmax` $= 2$. Average over 1000 trials. Compare the probability you obtain with your results from part (b). How do your results change as the time interval becomes larger?

(d) Increase p for fixed `n0` $= 10,000$ and determine $P(n)$ for a fixed time interval. Estimate the values of p and n for which the Poisson distribution is no longer applicable.

(e) Modify your program so that it flashes a small circle on the screen or makes a sound (like that of a Geiger counter) when a nucleus decays. You can have the computer make a beep by using the method `Toolkit.getDefaultToolkit().beep()` in `java.awt`. Choose the location of the small circle at random. Do a single run and describe the qualitative differences between the visual or audio patterns for the cases in parts (a)–(d)? Choose `n0` ≥ 5000. Such a visualization might be somewhat misleading on a serial computer because only one nuclei can be considered at a time. In contrast, for a real system, the nuclei can decay simultaneously. □

7.5 Problems in Probability

Why have we bothered to simulate many random processes that can be solved by analytic methods? The main reason is that it is simpler to introduce new methods in a familiar context. Another reason is that if we change the nature of many random processes slightly, it is often the

Team	Won	Lost	Percentage
St. Louis Cardinals	105	57	0.648
Houston Astros	92	70	0.568
Chicago Cubs	89	73	0.469
Pittsburgh Pirates	72	89	0.447
Cincinnati Reds	76	86	0.426
Milwaukee Brewers	67	94	0.416

Table 7.1: The National League Central standings for 2004.

case that it would be difficult or impossible to obtain the answers by familiar methods. Still another reason is that writing a program and doing a simulation can aid your intuitive understanding of the system, especially if the questions involve the subtle concept of probability. Probability is an elusive concept in part because it cannot be measured at one time. To reinforce the importance of thinking about how to solve a problem on a computer, we suggest some problems in probability in the following. Does thinking about how to write a program to simulate these problems help you to find a pencil and paper solution?

Problem 7.21. Three boxes: stick or switch?

Suppose that there are three identical boxes, each with a lid. When you leave the room, a friend places a \$10 bill in one of the boxes and closes the lid of each box. The friend knows the location of the \$10 bill, but you do not. You then reenter the room and guess which one of the boxes has the \$10 bill. As soon as you do, your friend opens the lid of a box that is empty. If you have chosen an empty box, your friend will open the lid of the other empty box. If you have chosen the right box, your friend will open the lid of one of the two empty boxes. You now have the opportunity to stay with your original choice or switch to the other unopened box. Suppose that you play this contest many times, and that each time you guess correctly, you keep the money. To maximize your winnings, should you maintain your initial choice or should you switch? Which strategy is better? This contest is known as the Monty Hall problem. Write a program to simulate this game and output the probability of winning for switching and for not switching. It is likely that before you finish your program, the correct strategy will become clear. To make your program more useful, consider an arbitrary number of boxes. □

Problem 7.22. Conditional probability

Suppose that many people in a community are tested at random for HIV. The accuracy of the test is 87%, and the incidence of the disease in the general population, independent of any test, is 1%. If a person tests positive for HIV, what is the probability that this person really has HIV? Write a program to compute the probability. The answer can be found by using Bayes' theorem (cf. Bernardo and Smith). The answer is much less than 87%. □

Problem 7.23. The roll of the dice

Suppose that two gamblers each begin with \$100 in capital, and on each throw of a coin, one gambler must win \$1 and the other must lose \$1. How long can they play on the average until the capital of the loser is exhausted? How long can they play if they each begin with \$1000? Neither gambler is allowed to go into debt. The eventual outcome is known as the gambler's ruin. □

Problem 7.24. The boys of summer

Luck plays a large role in the outcome of any baseball season. The National League Central Division standings for 2004 are given in Table 7.1. Suppose that the teams remain unchanged and their probability of winning a particular game is given by their 2004 winning percentage. Do a simulation to determine the probability that the Cardinals would lead the division for another season. For simplicity, assume that the teams play only each other. □

Much of the present day motivation for the development of probability comes from science, rather than from gambling. The next problem has much to do with statistical physics, even though this application is not apparent.

Problem 7.25. Money exchange

Consider a line that has been subdivided into bins. There can be an indefinite number of coins in each bin. For simplicity, we initially assign one coin to each bin. The money exchange proceeds as follows. Select two bins at random. If there is at least one coin in the first bin, move one coin to the second bin. If the first bin is empty, then do nothing. After many coin exchanges, how is the occupancy of the bins distributed? Are the coins uniformly distributed as in the initial state or are many bins empty? Write a Monte Carlo program to simulate this money exchange and show the state of the bins visually. Consider a system with at least 256 bins. Plot the histogram $H(n)$ versus n, where $H(n)$ is the number of bins with n coins. Do your results change qualitatively if you consider bigger systems or begin with more coins in each bin? □

Problem 7.26. Distribution of cooking times

An industrious physics major finds a job at a local fast food restaurant to help her pay her way through college. Her task is to cook 20 hamburgers on a grill at any one time. When a hamburger is cooked, she is supposed to replace it with an uncooked hamburger. However, our physics major does not pay attention to whether the hamburger is cooked or not. Her method is to choose a hamburger at random and replace it by a uncooked one She does not check if the hamburger that she removes from the grill is ready. What is the distribution of cooking times of the hamburgers that she removes? For simplicity, assume that she replaces a hamburger at regular intervals of thirty seconds and that there is an indefinite supply of uncooked hamburgers. Does the qualitative nature of the distribution change if she cooks 40 hamburgers at any one time? □

7.6 Method of Least Squares

In Problem 7.20 we did a simulation of $N(t)$, the number of unstable nuclei at time t. Given the finite accuracy of our data, how do we know if our simulation results are consistent with the exponential relation between N and t? The approach that we have been using is to plot the computed values of $\log N(t)$ as a function of t and to rely on our eye to help us draw the curve that best fits the data points. Such a visual approach works best when the curve is a straight line; that is, when the relation is linear. The advantages of this approach are that it is straightforward and allows us to see what we are doing. For example, if a data point is far from a straight line or if there is a gap in the data, we will notice it easily. If the analytic relation is not linear, it is likely that we will notice that the data points do not fit a simple straight line but instead show curvature. If we blindly let a computer fit the data to a straight line, we might not notice that the fit is not very good unless we have already had experience fitting data. Finally, the visceral experience of fitting the data manually gives us some feeling for the nature of the data that might otherwise be missed. It is a good idea to plot some data in this way even though a computer can do it much faster.

Although the visual approach is simple, it does not yield precise results; we need to use more systematic fitting methods. The most common method for finding the best straight line fit to a series of measured points is called *linear regression* or *least squares*. Suppose we have n pairs of measurements $(x_1,y_1),(x_2,y_2),\ldots,(x_n,y_n)$ and that the errors are entirely in the values of y. For simplicity, we assume that the uncertainties in $\{y_i\}$ all have the same magnitude. Our goal is to obtain the best fit to the linear function

$$y = mx + b. \tag{7.32}$$

The problem is to calculate the values of the parameters m and b for the best straight line through the n data points. The difference

$$d_i = y_i - mx_i - b \tag{7.33}$$

is a measure of the discrepancy in y_i. It is reasonable to assume that the best pair of values of m and b are those that minimize the quantity

$$\chi^2 = \sum_{i=1}^{n} (y_i - mx_i - b)^2. \tag{7.34}$$

Why should we minimize the sum of the squared differences between the experimental values y_i and the analytic values $mx_i + b$, and not some other function of the differences? The justification is based on the assumption that if we did many simulations or measurements, then the values of d_i would be distributed according to the Gaussian distribution (see Problems 7.5 and 7.15). Based on this assumption, it can be shown that the values of m and b that minimize χ yield a set of values of $mx_i + b$ that are the *most probable* set of measurements that we would find based on the available information. This link to probability is the reason we have discussed least squares fits in this chapter, even though we will not explicitly show that the difference d_i is distributed according to a Gaussian distribution.

To minimize χ, we take the partial derivative of S with respect to b and m:

$$\frac{\partial \chi}{\partial m} = -2 \sum_{i=1}^{n} x_i (y_i - mx_i - b) = 0 \tag{7.35a}$$

$$\frac{\partial \chi}{\partial b} = -2 \sum_{i=1}^{n} (y_i - mx_i - b) = 0. \tag{7.35b}$$

From (7.35) we obtain two simultaneous equations:

$$m \sum_{i=1}^{n} x_i^2 + b \sum_{i=1}^{n} x_i = \sum_{i=1}^{n} x_i y_i \tag{7.36a}$$

$$m \sum_{i=1}^{n} x_i + bn = \sum_{i=1}^{n} y_i. \tag{7.36b}$$

It is convenient to define the quantities

$$\langle x \rangle = \frac{1}{n}\sum_{i=1}^{n} x_i \tag{7.37a}$$

$$\langle y \rangle = \frac{1}{n}\sum_{i=1}^{n} y_i \tag{7.37b}$$

$$\langle xy \rangle = \frac{1}{n}\sum_{i=1}^{n} x_i y_i \tag{7.37c}$$

and rewrite (7.36) as

$$m\langle x^2 \rangle + b\langle x \rangle = \langle xy \rangle \tag{7.38a}$$

$$m\langle x \rangle + b = \langle y \rangle. \tag{7.38b}$$

The solution of (7.38) can be expressed as

$$m = \frac{\langle xy \rangle - \langle x \rangle\langle y \rangle}{\sigma_x^2} \tag{7.39a}$$

$$b = \langle y \rangle - m\langle x \rangle, \tag{7.39b}$$

where

$$\sigma_x^2 = \langle x^2 \rangle - \langle x \rangle^2. \tag{7.39c}$$

Equation (7.39) determines the slope m and the intercept b of the best straight line through the n data points. (Note that the averages for the coefficients m and b are over the data points.)

As an example, consider the data shown in Table 7.2 for a one-dimensional random walk. To make the example more interesting, suppose that the walker takes steps of length 1 or 2 with equal probability. The direction of the step is random and $p = 1/2$. As in Section 7.2, we assume that the mean square displacement Δx^2 obeys the general relation

$$\Delta x^2 = aN^{2\nu} \tag{7.40}$$

with an unknown exponent ν and a a constant. Note that the fitting problem in (7.40) is nonlinear; that is, $\langle x^2 \rangle - \langle x \rangle^2$ depends on N^ν rather than N. Often a problem that looks nonlinear can be turned into a linear problem by a change of variables. In this case we convert the nonlinear relation (7.40) to a linear relation by taking the logarithm of both sides:

$$\ln(\Delta x^2) = \ln a + 2\nu \ln N. \tag{7.41}$$

The values of $y = \ln(\Delta x^2)$ and $x = \ln N$ in Table 7.2 and the least squares fit are shown in Figure 7.5. We use (7.39) and find that $m = 1.02$ and $b = 0.83$. Hence, we conclude from our limited data and the relation $2\nu = m$ that $\nu \approx 0.51$, which is consistent with the expected result $\nu = 1/2$.

The least squares fitting procedure also allows us to estimate the uncertainty or the most probable error in m and b by analyzing the measurements themselves. The result of this analysis is that the most probable error in m and b, σ_m and σ_b, respectively, is given by

$$\sigma_m = \frac{1}{\sqrt{n}}\frac{\Delta}{\sigma_x} \tag{7.42a}$$

$$\sigma_b = \frac{1}{\sqrt{n}}\frac{\big(\langle x^2 \rangle\big)^{1/2}\Delta}{\sigma_x} \tag{7.42b}$$

N	Δx^2
8	19.43
16	37.65
32	76.98
64	160.38

Table 7.2: Computed values of the mean square displacement Δx^2 as a function of the total number of steps N. The mean square displacement was averaged over 1000 trials. The one-dimensional random walker takes steps of length 1 or 2 with equal probability, and the direction of the step is random with $p = 1/2$.

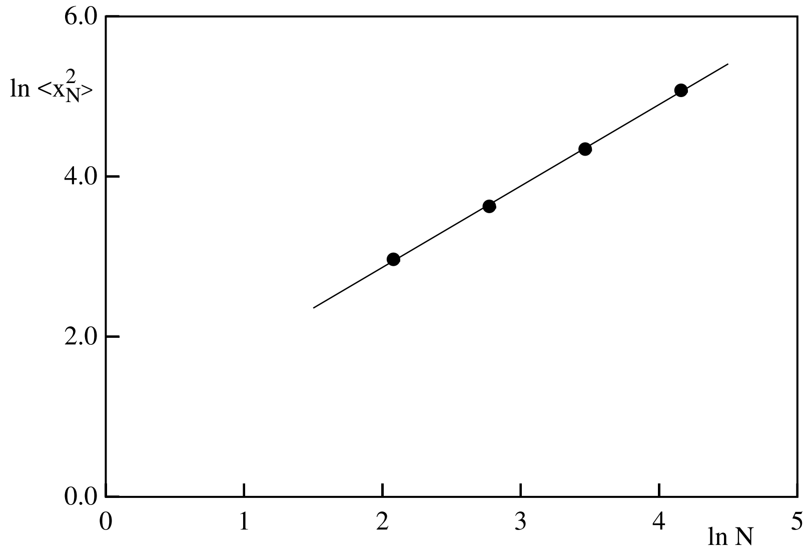


Figure 7.5: Plot of Δx^2 versus $\ln N$ for the data listed in Table 7.2. The straight line $y = 1.02x + 0.83$ through the points is found by minimizing the sum (7.34).

where

$$\Delta^2 = \frac{1}{n-2}\sum_{i=1}^{n} d_i^2 \tag{7.43}$$

and d_i is given by (7.33).

Because there are n data points, we might have guessed that n rather than $n-2$ would be present in the denominator of (7.43). The reason for the factor of $n-2$ is related to the fact that to determine Δ, we first need to calculate *two* quantities m and b, leaving only $n-2$ independent degrees of freedom. To see that the $n-2$ factor is reasonable, consider the special case of $n = 2$. In this case we can find a line that passes exactly through the two data points, but we cannot deduce anything about the reliability of the set of measurements because the fit is exact. If we use (7.43), we see that both the numerator and denominator would be zero, and hence Δ would be undetermined. If a factor of n rather than $n-2$ appeared in (7.43), we would conclude that $\Delta = 0/2 = 0$, an absurd conclusion. Usually $n \gg 1$, and the difference between n and $n-2$ is negligible.

For our example, $\Delta = 0.03$, $\sigma_b = 0.07$, and $\sigma_m = 0.02$. The uncertainties δm and δv are related by $2\delta\nu = \delta m$. Because $\delta m = \sigma_m$, we conclude that our best estimate for ν is $\nu = 0.51 \pm 0.01$.

If the values of y_i have different uncertainties σ_i, then the data points are weighted by the quantity $w_i = 1/\sigma_i^2$. In this case it is reasonable to minimize the quantity

$$\chi^2 = \sum_{i=1}^{n} w_i (y_i - m x_i - b)^2. \tag{7.44}$$

The resulting expressions in (7.39) for m and b are unchanged if we generalize the definition of the averages to be

$$\langle f \rangle = \frac{1}{n\langle w \rangle} \sum_{i=1}^{n} w_i f_i \tag{7.45}$$

where

$$\langle w \rangle = \frac{1}{n} \sum_{i=1}^{n} w_i. \tag{7.46}$$

Problem 7.27. Example of least squares fit

(a) Write a program to find the least squares fit for a set of data. As a check on your program, compute the most probable values of m and b for the data shown in Table 7.2.

(b) Modify the random walk program so that steps of length 1 and 2 are taken with equal probability. Use at least 10,000 trials and do a least squares fit to Δx^2 as done in the text. Is your most probable estimate for ν closer to $\nu = 1/2$? □

For simple random walk problems the relation $\Delta x^2 = aN^\nu$ holds for all N. However, in many random walk problems, a power law relation between Δx^2 and N holds only asymptotically for large N, and hence we should use only the larger values of N to estimate the slope. Also, because we are finding the best fit for the logarithm of the independent variable N, we need to give equal weight to all intervals of $\ln N$. In the above example, we used $N = 8$, 16, 32, and 64, so that the values of $\ln N$ are equally spaced.

7.7 Applications to Polymers

Random walk models play an important role in polymer physics (cf. de Gennes). A polymer consists of N repeat units (monomers) with $N \gg 1$ ($N \sim 10^3 - 10^5$). For example, polyethylene can be represented as $\cdots\text{–CH}_2\text{–CH}_2\text{–}CH_2\text{–}\cdots$. The detailed structure of the polymer is important for many practical applications. For example, if we wish to improve the fabrication of rubber, a good understanding of the local motions of the monomers in the rubber chain is essential. However, if we are interested in the global properties of the polymer, the details of the chain structure can be ignored.

Let us consider a familiar example of a polymer chain in a good solvent: a noodle in warm water. A short time after we place a noodle in warm water, the noodle becomes flexible, and it neither collapses into a little ball or becomes fully stretched. Instead, it adopts a random structure as shown schematically in Figure 7.6. If we do not add too many noodles, we can say that the noodles behave as a dilute solution of polymer chains in a good solvent. The dilute nature of the solution implies that we can ignore entanglement effects of the noodles and consider each

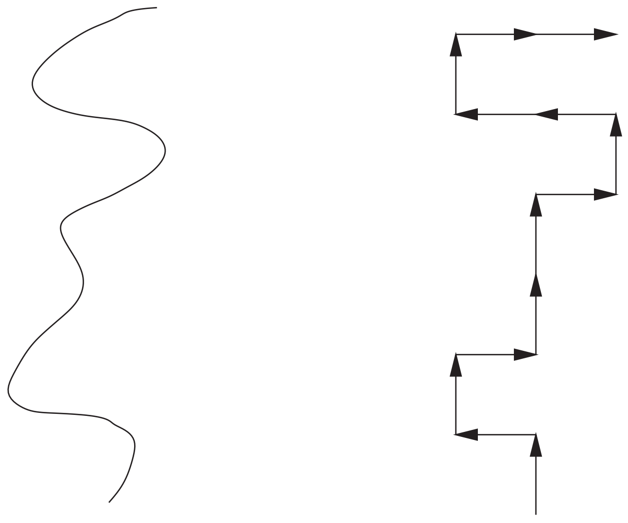


Figure 7.6: (a) Schematic illustration of a linear polymer in a good solvent. (b) Example of the corresponding self-avoiding walk on a square lattice.

noodle individually. The presence of a good solvent implies that the polymers can move freely and adopt many different configurations.

A fundamental geometrical property that characterizes a polymer in a good solvent is the mean square end-to-end distance $\langle R_N^2 \rangle$, where N is the number of monomers. (For simplicity, we will frequently write R^2 in the following.) For a dilute solution of polymer chains in a good solvent, it is known that the asymptotic dependence of R^2 is given by (7.13) with $\nu \approx 0.5874$ in three dimensions. If we were to ignore the interactions of the monomers, the simple random walk model would yield $\nu = 1/2$, independent of the dimension and symmetry of the lattice. Because this result for ν does not agree with experiment, we know that we are overlooking an important physical feature of polymers.

We now discuss a random walk that incorporates the global features of dilute linear polymers in solution. We have already introduced a model of a polymer chain consisting of straight line segments of the same size joined together at random angles (see Problem 7.13). A further idealization is to place the polymer chain on a lattice (see Figure 7.6). A more realistic model of linear polymers accounts for its most important physical feature; that is, two monomers cannot occupy the same spatial position. This constraint is known as the *excluded volume* condition, which is ignored in a simple random walk. A well-known lattice model for a linear polymer chain that incorporates this constraint is known as the *self-avoiding* walk (SAW). This model consists of the set of all N-step walks starting from the origin subject to the global constraint that no lattice site can be visited more than once in each walk; this constraint accounts for the excluded volume condition.

Self-avoiding walks have many applications, such as the physics of magnetic materials and the study of phase transitions, and they are of interest as purely mathematical objects. Many of the obvious questions have resisted rigorous analysis, and exact enumeration and Monte Carlo simulation have played an important role in our current understanding. The result for ν in two dimensions for the self-avoiding walk is known to be exactly $\nu = 3/4$. The proportionality

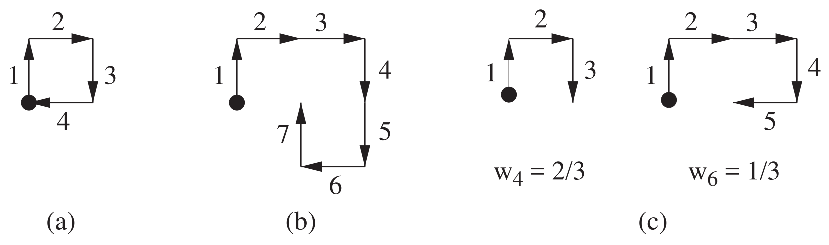


Figure 7.7: Examples of self-avoiding walks on a square lattice. The origin is denoted by a filled circle. (a) An $N = 3$ walk. The fourth step shown is forbidden. (b) An $N = 7$ walk that leads to a self-intersection at the next step; the weight of the $N = 8$ walk is zero. (c) Two examples of the weights of walks in the enrichment method.

constant in (7.13) depends on the structure of the monomers and on the solvent. In contrast, the exponent ν is independent of these details and depends only on the spatial dimension.

We consider Monte Carlo simulations of the self-avoiding walk in two dimensions in Problems 7.28–7.30. Another algorithm for the self-avoiding walk is considered in Project 7.41.

Problem 7.28. The two-dimensional self-avoiding walk

Consider the self-avoiding walk on the square lattice. Choose an arbitrary site as the origin and assume that the first step is "up." The walks generated by the three other possible initial directions only differ by a rotation of the whole lattice and do not have to be considered explicitly. The second step can be in three rather than four possible directions because of the constraint that the walk cannot return to the origin. To obtain unbiased results, we generate a random number to choose one of the three directions. Successive steps are generated in the same way. Unfortunately, the walk will very likely not continue indefinitely. To obtain unbiased results, we must choose at random one of the three steps, even though one or more of these steps might lead to a self-intersection. If the next step does lead to a self-intersection, the walk must be terminated to keep the statistics unbiased. An example of a three-step walk is shown in Figure 7.7a. The next step leads to a self-intersection and violates the constraint. In this case we must start a new walk at the origin.

(a) Write a program that implements this algorithm and record the fraction $f(N)$ of successful attempts at constructing polymer chains with N total monomers. Represent the lattice as a array so that you can record the sites that already have been visited. What is the qualitative dependence of $f(N)$ on N? What is the maximum value of N that you can reasonably consider?

(b) Determine the mean square end-to-end distance $\langle R_N^2 \rangle$ for values of N that you can reasonably consider with this sampling method. □

The disadvantage of the straightforward sampling method in Problem 7.28 is that it becomes very inefficient for long chains; that is, the fraction of successful attempts decreases exponentially. To overcome this attrition, several "enrichment" techniques have been developed. We first discuss a relatively simple algorithm proposed by Rosenbluth and Rosenbluth in which each walk of N steps is associated with a weighting function $w(N)$. Because the first step to the north is always possible, we have $w(1) = 1$. In order that all allowed configurations of a given N are counted equally, the weights $w(N)$ for $N > 1$ are determined according to the following possibilities:

1. All three possible steps violate the self-intersection constraint (see Figure 7.7b). The walk is terminated with a weight $w(N) = 0$, and a new walk is generated at the origin.
2. All three steps are possible and $w(N) = w(N-1)$.
3. Only m steps are possible with $1 \leq m < 3$ (see Figure 7.7c). In this case $w(N) = (m/3)w(N-1)$, and one of the m possible steps is chosen at random.

The desired unbiased value of $\langle R^2 \rangle$ is obtained by weighting R_i^2, the value of R^2 obtained in the ith trial, by the value of $w_i(N)$, the weight found for this trial. Hence, we write

$$\langle R^2 \rangle = \frac{\sum_i w_i(N) R_i^2}{\sum_i w_i(N)} \tag{7.47}$$

where the sum is over all trials.

Problem 7.29. Rosenbluth and Rosenbluth enrichment method
Incorporate the Rosenbluth method into your Monte Carlo program and compute R^2 for $N = 4$, 8, 16, and 32. Estimate the exponent ν from a log-log plot of R^2 versus N. Can you distinguish your estimate for ν from its random walk value $\nu = 1/2$? □

The Rosenbluth and Rosenbluth procedure is not particularly efficient because many walks still terminate, and thus we do not obtain many walkers for large N. Grassberger improved this algorithm by increasing the population of walkers with high weights and reducing the population of walkers with low weights. The idea is that if $w(N)$ for a given trial is above a certain threshold, we add a new walker and give the new and old walker half of the original weight. If $w(N)$ is below a certain threshold, then we eliminate half of the walkers with weights below this threshold (for example, every second walker) and double the weights of the remaining half. It is a good idea to adjust the thresholds as the simulation runs in order to maintain a relatively constant number of walkers.

More recently Prellberg and Krawczyk further improved the Rosenbluth and Rosenbluth enrichment method so that there is no need to provide a threshold value. After each step the average weight of the walkers $\langle w(N) \rangle$ is computed for a given trial, and the ratio $r = w(N)/\langle w(N) \rangle$ is used to determine whether to add walkers (enrichment) or eliminate walkers (pruning). If $r > 1$, then $c = \min(r, m)$ copies of the walker are made each with weight $w(N)/c$. If $r < 1$, then remove this walker with probability $1 - r$. This algorithm leads to an approximately constant number of walkers and is related to the Wang–Landau method which we will discuss in Problem 15.30.

Another enrichment algorithm is the "reptation" method (see Wall and Mandel). For simplicity, consider a model polymer chain in which all bond angles are $\pm 90^\circ$. As an example of this model, the five independent $N = 5$ polymer chains are shown in Figure 7.8. (Other chains differ only by a rotation or a reflection.) The reptation method can be stated as follows:

1. Choose a chain at random and remove the tail link.
2. Attempt to add a link to the head of the chain. There is a maximum of two directions in which the new head link can be added.
3. If the attempt violates the self-intersection constraint, return to the original chain and interchange the head and tail. Include the chain in the statistical sample.

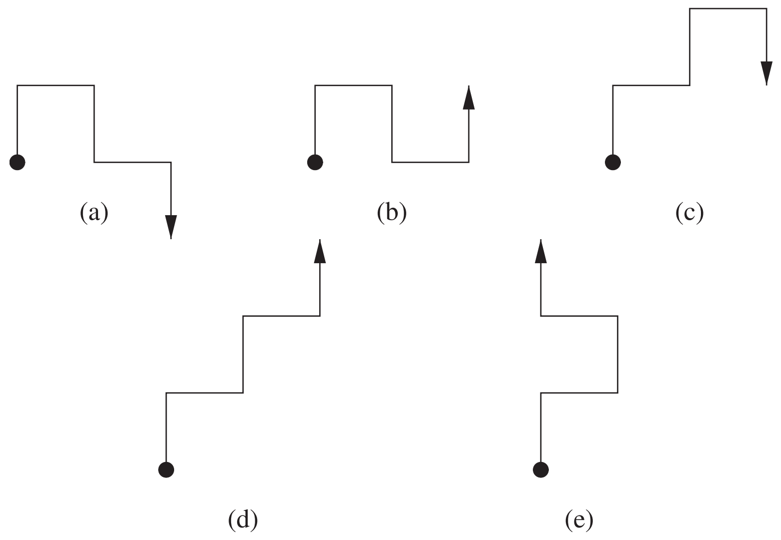


Figure 7.8: The five independent possible walks of $N = 5$ steps on a square lattice with $\pm 90°$ bond angles. The tail and head of each walk are denoted by a circle and arrow, respectively.

The above steps are repeated many times to obtain an estimate of R^2.

As an example of the reptation method, consider chain a of Figure 7.8. A new link can be added in two directions (see Figure 7.9a), so that on the average we find $a \to \frac{1}{2}c + \frac{1}{2}d$. In contrast, a link can be added to chain b in only one direction, and we obtain $b \to \frac{1}{2}e + \frac{1}{2}b$, where the tail and head of chain b have been interchanged (see Figure 7.9b). Confirm that $c \to \frac{1}{2}e + \frac{1}{2}a$, $d \to \frac{1}{2}c + \frac{1}{2}d$, and $e \to \frac{1}{2}a + \frac{1}{2}b$, and that all five chains are equally probable. That is, the transformations in the reptation method preserve the proper statistical weights of the chains without attrition. There is just one problem: unless we begin with a double-ended "cul-de-sac" configuration, such as shown in Figure 7.10, we will never obtain such a configuration using the above transformation. Hence, the reptation method introduces a small statistical bias, and the calculated mean end-to-end distance will be slightly larger than if all configurations were considered. However, the probability of such trapped configurations is very small, and the bias can be neglected for most purposes.

***Problem 7.30.** The reptation method

(a) Adopt the $\pm 90°$ bond angle restriction and calculate by hand the exact value of $\langle R^2 \rangle$ for $N = 5$. Then write a Monte Carlo program that implements the reptation method. Generate one walk of $N = 5$ and use the reptation method to generate a statistical sample of chains. As a check on your program, compute $\langle R^2 \rangle$ for $N = 5$ and compare your result with the exact result. Then extend your Monte Carlo computations of $\langle R^2 \rangle$ to larger N.

(b) Modify the reptation model so that the bond angle can also be $180°$. This modification leads to a maximum of three directions for a new bond. Compare your results with those from part (a). □

In principle, the dynamics of a polymer chain undergoing collisions with solvent molecules can be simulated by using a molecular dynamics method. However, in practice, only relatively small chains can be simulated in this way. An alternative approach is to use a Monte

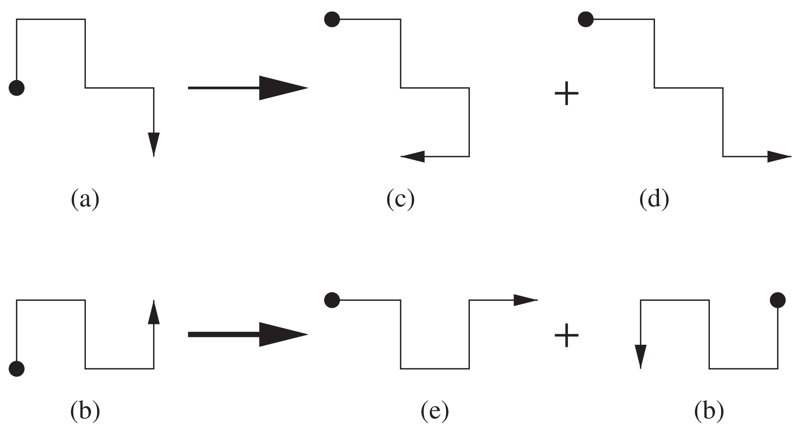


Figure 7.9: The possible transformations of chains a and b. One of the two possible transformations of chain b violates the self-intersection restriction, and the head and tail are interchanged.

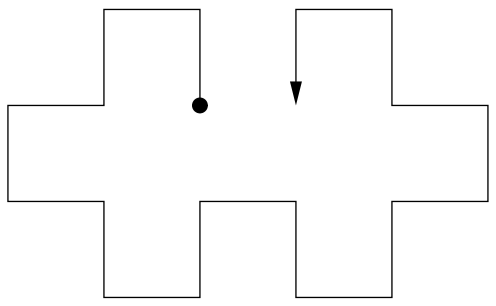


Figure 7.10: Example of a double-cul-de-sac configuration for the self-avoiding walk that cannot be obtained by the reptation method.

Carlo model that simplifies the effect of the random collisions of the solvent molecules with the atoms of the chain. Most of these models (cf. Verdier and Stockmayer) consider the chain to be composed of beads connected by bonds and restrict the positions of the beads to the sites of a lattice. For simplicity, we assume that the bond angles can be either $\pm 90^\circ$ or 180°. The idea is to begin with an allowed configuration of N beads ($N-1$ bonds). A possible starting configuration can be generated by taking successive steps in the positive y direction and positive x directions. The dynamics of the Verdier–Stockmayer algorithm is summarized by the following steps:

1. Select at random a bead (occupied site) on the polymer chain. If the bead is not an end site, then the bead can move to a nearest neighbor site of another bead if this site is empty and if the new angle between adjacent bonds is either $\pm 90^\circ$ or 180°. For example, bead 4 in Figure 7.11 can move to position $4'$, while bead 3 cannot move if selected. That is, a selected bead can move to a diagonally opposite unoccupied site only if the two bonds to which it is attached are mutually perpendicular.

2. If the selected bead is an end site, move it to one of two (maximum) possible unoccupied sites, so that the bond to which it is connected changes its orientation by $\pm 90^\circ$ (see Figure 7.11).

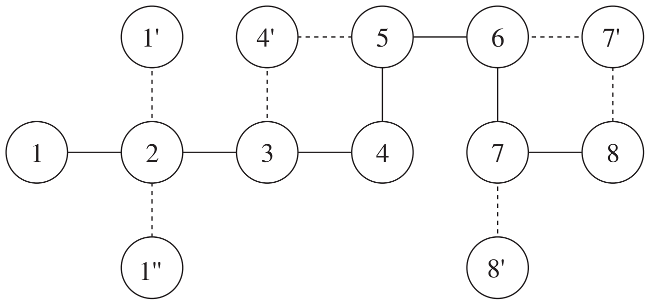


Figure 7.11: Examples of possible moves of the simple polymer dynamics model considered in Problem 7.31. For this configuration, beads 2, 3, 5, and 6 cannot move, while beads 1, 4, 7, and 8 can move to the positions shown if they are selected. Only one bead can move at a time. This figure is adopted from the article by Verdier and Stockmayer.

3. If the selected bead cannot move, retain the previous configuration.

The physical quantities of interest include $\langle R^2 \rangle$ and the mean square displacement of the center of mass of the chain $\langle r^2 \rangle = \langle x^2 \rangle - \langle x \rangle^2 + \langle y^2 \rangle - \langle y \rangle^2$, where x and y are the coordinates of the center of mass. The unit of time is the number of Monte Carlo steps per bead; in one Monte Carlo step per bead, each bead has one chance on the average to move to a different site.

Another efficient method for simulating the dynamics of a polymer chain is the bond fluctuation model (see Carmesin and Kremer).

Problem 7.31. The dynamics of polymers in a dilute solution

(a) Consider a two-dimensional lattice and compute $\langle R^2 \rangle$ and $\langle r^2 \rangle$ for various values of N. How do these quantities depend on N? (The first published results for three dimensions were limited to 32 Monte Carlo steps per bead for $N = 8$, 16, and 32 and only 8 Monte Carlo steps per bead for $N = 64$.) Also compute the probability density $P(R)$ that the end-to-end distance is R. How does this probability compare to a Gaussian distribution?

(b)* Two configurations are strongly correlated if they differ by the position of only one bead. Hence, it would be a waste of computer time to measure the end-to-end distance and the position of the center of mass after every single move. Ideally, we wish to compute these quantities for configurations that are approximately statistically independent. Because we do not know *a priori* the mean number of Monte Carlo steps per bead needed to obtain configurations that are statistically independent, it is a good idea to estimate this time in our preliminary calculations. The correlation time τ is the time needed to obtain statistically independent configurations and can be obtained by computing the equilibrium averaged time-autocorrelation function for a chain of fixed N:

$$C(t) = \frac{\langle R^2(t'+t)R^2(t') \rangle - \langle R^2 \rangle^2}{\langle R^4 \rangle - \langle R^2 \rangle^2}. \tag{7.48}$$

$C(t)$ is defined so that $C(t = 0) = 1$ and $C(t) = 0$ if the configurations are not correlated. Because the configurations will become uncorrelated if the time t between the configurations is sufficiently long, we expect that $C(t) \to 0$ for $t \gg 1$. We expect that $C(t) \sim e^{-t/\tau}$; that is, $C(t)$ decays exponentially with a decay or correlation time τ. Estimate τ from a plot of

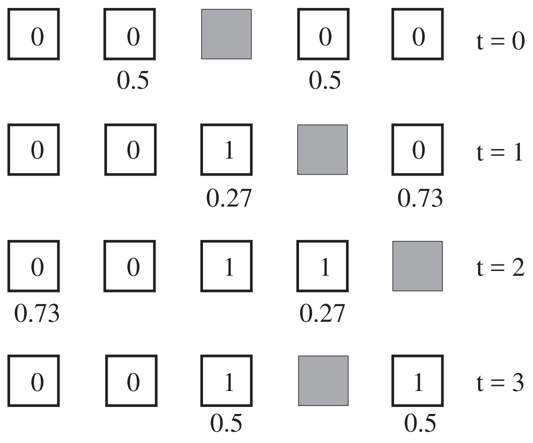


Figure 7.12: Example of the evolution of the true self-avoiding walk with $g = 1$ (see (7.49)). The shaded site represents the location of the walker at time t. The number of visits to each site are given within each site. and the probability of a step to a nearest neighbor site is given below it. Note the use of periodic boundary conditions.

$\ln C(t)$ versus t. Another way of estimating τ is from the integral $\int_0^\infty dt\, C(t)$, where $C(t)$ is normalized so that $C(0) = 1$. (Because we determine $C(t)$ at discrete values of t, this integral is actually a sum.) How do your two estimates of τ compare? A more detailed discussion of the estimation of correlation times can be found in Section 15.7. □

Another type of random walk that is less constrained than the self-avoiding random walk is the "true" self-avoiding walk. This walk describes the path of a random walker that avoids visiting a lattice site with a probability that is a function of the number of times the site has been visited already. This constraint leads to a reduced excluded volume interaction in comparison to the usual self-avoiding walk.

Problem 7.32. The true self-avoiding walk in one dimension

In one dimension the true self-avoiding walk corresponds to a walker that can jump to one of its two nearest neighbors with a probability that depends on the number of times these neighbors have already been visited. Suppose that the walker is at site i at step t. The probability that the walker will jump to site $i + 1$ at time $t + 1$ is given by

$$p_{i+1} = \frac{e^{-gn_{i+1}}}{e^{-gn_{i+1}} + e^{-gn_{i-1}}} \tag{7.49}$$

where $n_{i\pm1}$ is the number of times that the walker has already visited site $i \pm 1$. The probability of a jump to site $i - 1$ is $p_{i-1} = 1 - p_{i+1}$. The parameter g ($g > 0$) is a measure of the "desire" of the path to avoid itself. The first few steps of a typical true self-avoiding walk are shown in Figure 7.12. The main quantity of interest is the exponent ν. We know that $g = 0$ corresponds to the usual random walk with $\nu = 1/2$, and that the limit $g \to \infty$ corresponds to the self-avoiding walk. What is the value of ν for a self-avoiding walk in one dimension? Is the value of ν for any finite value of g different than these two limiting cases?

Write a program to do a Monte Carlo simulation of the true self-avoiding walk in one dimension. Use an array to record the number of visits to every site. At each step calculate the probability p of a jump to the right. Generate a random number r and compare it to p. If $r \leq p$, move the walker to the right; otherwise, move the walker to the left. Compute $\langle \Delta x^2 \rangle$ as a function of the number of steps N, where x is the distance of the walker from the origin. Make a log-log plot of $\langle \Delta x^2 \rangle$ versus N and estimate ν. Can you distinguish ν from its random walk and self-avoiding walk values? Reasonable choices of parameters are $g = 0.1$ and $N \sim 10^3$. Averages over 10^3 trials yield qualitative results. For comparison, published results are for $N \sim 10^4$ and for 10^3 trials; extended results for $g = 2$ are given for $N = 2 \times 10^5$ and 10^4 trials (see Bernasconi and Pietronero). □

7.8 Diffusion-Controlled Chemical Reactions

Imagine a system containing particles of a single species A. The particles diffuse, and when two particles "collide," a reaction occurs such that the two combine to form an inert species which is no longer involved in the reaction. We can represent this chemical reaction as

$$A + A \to 0. \tag{7.50}$$

If we ignore the spatial distribution of the particles, we can describe the kinetics by a simple rate equation:

$$\frac{dA(t)}{dt} = -kA^2(t) \tag{7.51}$$

where A is the concentration of A particles at time t and k is the rate constant. (In the chemical kinetics literature, it is traditional to use the term concentration rather than the number density.) For simplicity, we assume that all reactants are entered into the system at $t = 0$, and that no reactants are added later (the system is closed). It is easy to show that the solution of the first-order differential equation (7.51) is

$$A(t) = \frac{A(0)}{1 + ktA(0)}. \tag{7.52}$$

Hence, $A(t) \sim t^{-1}$ in the limit of long times.

Another interesting case is the bimolecular reaction

$$A + B \to 0. \tag{7.53}$$

If we neglect spatial fluctuations in the concentration as before (this neglect yields what is known as a mean-field approximation), we can write the corresponding rate equation as

$$\frac{dA(t)}{dt} = \frac{dB(t)}{dt} = -kA(t)B(t). \tag{7.54}$$

We have also

$$A(t) - B(t) = \text{constant} \tag{7.55}$$

because each reaction leaves the difference between the concentration of A and B particles unchanged. For the special case of equal initial concentrations, the solution of (7.54) with (7.55) is the same as (7.52). What is the solution for the case $A(0) \neq B(0)$?

This derivation of the time dependence of A for the kinetics of the one and two species annihilation process is straightforward, but is based on the assumption that the particles are distributed uniformly. In the following two problems, we simulate the kinetics of these processes and test this assumption.

Problem 7.33. Diffusion-controlled chemical reactions in one dimension

(a) Assume that N particles do a random walk on a one-dimensional lattice of length L with periodic boundary conditions. Every particle moves once in one unit of time. Use the array `site[j]` to record the label of the particle, if any, at site `j`. Because we are interested in the long time behavior of the system when the concentration $A = N/L$ of particles is small, it is efficient to also maintain an array of particle positions `x[i]` such that `site[x(i] = i`. For example, if particle 5 is located at site 12, then `x[5]` = 12 and `site[12]` = 5. We also need an array, `newSite`, to maintain the new positions of the walkers as they are moved one at a time. After each walker is moved, we check to see if two walkers have landed on the same position `k`. If they have, we set `newSite[k]` = −1 and the value of `x[i]` for these two walkers to −1. The value −1 indicates that no particle exists at the site. After all the walkers have moved, we let `site` = `newSite` for all sites, and remove all the reacting particles in `x` that have values equal to −1. This operation can be accomplished by replacing any reacting particle in `x` by the last particle in the array. Begin with all sites occupied $A(t = 0) = 1$.

(b) Make a log-log plot of the quantity $A(t)^{-1} - A(0)^{-1}$ versus the time t. The times should be separated by exponential intervals, so that your data is equally spaced on a logarithmic plot. For example, you might include data with times equal to 2^p and $p = 1, 2, 3, \ldots$. Does your log-log plot yield a straight line for long times? If so, calculate its slope. Is the mean-field approximation for $A(t)$ valid in one dimension? You can obtain crude results for small lattices of order $L = 100$ and times of order $t = 10^2$. To obtain results to within 10%, you will need lattices of order $L = 10^4$ and times of order $t = 2^{13}$.

(c) More insight into the origin of the time dependence of $A(t)$ can be gained from the behavior of the quantity $P(r,t)$, the probability that the nearest neighbor distance is r at time t. The nearest neighbor distance of a given particle is defined as the minimum distance between it and all other particles. The distribution of these distances changes dramatically as the reaction proceeds, and this change can give information about the reaction mechanism. Place the particles at random on a one-dimensional lattice and verify that the most probable nearest neighbor distance is $r = 1$ (one lattice constant) for all concentrations. (This result is true in any dimension.) Then verify that the distribution of nearest neighbor distances on a one-dimensional lattice is given by

$$P(r,t=0) = 2A\,e^{-2A(r-1)} \qquad \text{(random distribution).} \tag{7.56}$$

Is the form (7.56) properly normalized? Start with $A(t = 0) = 0.1$ and find $P(r,t)$ for $t = 10$, 100, and 1000. Average over all particles. How does $P(r,t)$ change as the reaction proceeds? Does it retain the same form as the concentration decreases?

(d)* Compute the quantity $D(t)$, the number of *distinct* sites visited by an individual walker. How does the time dependence of $D(t)$ compare to the computed time dependence of $A(t)^{-1} - 1$?

(e)* Write a program to simulate the reaction $A + B = 0$. For simplicity, assume that multiple occupancy of the same site is not allowed; for example, an A particle cannot jump to a site already occupied by an A particle. The easiest procedure is to allow a walker to choose one of its nearest neighbor sites at random, but not to move the walker if the chosen site is already occupied by a particle of the same type. If the site is occupied by a walker of another type, then the pair of reacting particles is annihilated. Keep separate arrays for the A and B particles with the value of the array denoting the label of the particle as before.

One way to distinguish A and B walkers is to make the array element `site(k)` positive if the site is occupied by an A particle and negative if the site is occupied by a B particle. Start with equal concentrations of A and B particles and occupy the sites at random. Some of the interesting questions are similar to those that we posed in parts (b)–(d). Color code the particles and observe what happens to the relative positions of the particles. □

*__Problem 7.34.__ Reaction diffusion in two dimensions

(a) Do a similar simulation as in Problem 7.33 on a two-dimensional lattice for the reaction $A + A \to 0$. In this case it is convenient to have one array for each dimension, for example, `siteX` and `siteY`, or to store the lattice as a one-dimensional array (see Section 12.2). Set $A(t = 0) = 1$ and choose $L = 50$. Show the walkers after each Monte Carlo step per walker and describe their distribution as they diffuse. Are the particles uniformly distributed throughout the lattice for all times? Calculate $A(t)$ and compare your results for $A(t)^{-1} - A(0)^{-1}$ to the t-dependence of $D(t)$, the number of distinct lattice sites that are visited in time t. (In two dimensions, $D(t) \sim t/\log t$.) How well do the slopes compare? Do a similar simulation with $A(t = 0) = 0.01$. What slope do you obtain in this case? What can you conclude about the initial density dependence? Is the mean-field approximation valid in this case?

(b) Begin with A and B type random walkers initially segregated on the left and right halves (in the x direction) of a square lattice. The process $A + B \to C$ exhibits a reaction front where the production of particles of type C is nonzero. Some of the quantities of interest are the time dependence of the mean position $\langle x\rangle(t)$ and the width $w(t)$ of the reaction front. The rules of this process are the same as in part (a) except that a particle of type C is added to a site when a reaction occurs. A particular site can be occupied by one particle of type A or type B as well as any number of particles of type C. If $n(x,t)$ is the number of particles of type C at a distance x from the initial boundary of the reactants, then $x(t)$ and $w(t)$ can be written as

$$\langle x\rangle(t) = \frac{\sum_x x\, n(x,t)}{\sum_x n(x,t)} \tag{7.57}$$

$$w(t)^2 = \frac{\sum_x \big[x - \langle x\rangle(t)\big]^2 n(x,t)}{\sum_x n(x,t)}. \tag{7.58}$$

Choose lattice sizes of order 100×100 and average over at least 10 trials. The fluctuations in $x(t)$ and $w(t)$ can be reduced by averaging $n(x,t)$ over the order of 100 time units centered about t. More details can be found in Jiang and Ebner. □

7.9 Random Number Sequences

So far we have used the random number generator supplied with Java to generate the desired random numbers. In principle, we could have generated these numbers from a random physical process, such as the decay of radioactive nuclei or the thermal noise from a semiconductor device. In practice, random number sequences are generated from a physical process only for specific purposes such as a lottery. Although we could store the outcome of a random physical process so that the random number sequence would be both truly random and reproducible, such a method would usually be inconvenient and inefficient in part because we often require very long sequences. In practice, we use a digital computer, a deterministic machine, to generate sequences of *pseudorandom* numbers. Although these sequences cannot be truly random, such a distinction is unimportant if the sequence satisfies all our criteria for randomness. It is common to refer to random number generators even though we really mean pseudorandom number generators.

Most random number generators yield a sequence in which each number is used to find the succeeding one according to a well-defined algorithm. The most important features of a desirable random number generator are that its sequence satisfies the known statistical tests for randomness, which we will explore in the following problems. We also want the generator to be efficient and machine independent and the sequence to be reproducible.

The most widely used random number generator is based on the *linear congruential* method. One advantage of the linear congruential method is that it is very fast. For a given seed x_0, each number in the sequence is determined by the one-dimensional map

$$x_n = (ax_{n-1} + c) \bmod \mathrm{m} \tag{7.59}$$

where a, c, and m as well as x_n are integers. The notation $y = z \bmod \mathrm{m}$ means that m is subtracted from z until $0 \leq y < m$. The map (7.59) is characterized by three parameters, the *multiplier a*, the *increment c*, and the *modulus m*. Because m is the largest integer generated by (7.59), the maximum possible *period* is m.

In general, the period depends on all three parameters. For example, if $a = 3$, $c = 4$, $m = 32$, and $x_0 = 1$, the sequence generated by (7.59) is 1, 7, 25, 15, 17, 23, 9, 31, 1, 7, 25, …, and the period is 8, rather than the maximum possible value of $m = 32$. If we choose a, c, and m carefully such that the maximum period is obtained, then all possible integers between 0 and $m-1$ would occur in the sequence. Because we usually wish to have random numbers r in the unit interval $0 \leq r < 1$, rather than random integers, random number generators usually return the ratio x_n/m which is always less than unity. Several rules have been developed (see Knuth) to obtain the longest period. Some of the properties of the linear congruential method are explored in Problem 7.35.

Another popular random number generator is the *generalized feedback shift register* method which uses bit manipulation (see Sections 14.1 and 14.6). Every integer is represented as a series of 1s and 0s called bits. These bits can be shuffled by using the bitwise `exclusive or` operator $\oplus$ (xor) defined by $a \oplus b = 1$ if the bits $a \neq b$; $a \oplus b = 0$ if $a = b$. The nth member of the sequence is given by

$$x_n = x_{n-p} \oplus x_{n-q} \tag{7.60}$$

where $p > q$, and p, q, and x_n are integers. The first p random integers must be supplied by another random number generator. As an example of how the operator $\oplus$ works, suppose that $n = 6$, $p = 5$, $q = 3$, $x_3 = 11$, and $x_1 = 6$. Then $x_6 = x_1 \oplus x_3 = 0110 \oplus 1011 = 1101 = 2^3 + 2^2 + 2^0 = 8 + 4 + 1 = 13$. Not all values of p and q lead to good results. Some common pairs are $(p, q) = (31, 3), (250, 103)$, and $(521, 168)$.

In Java and C the exclusive or operation on the integers `m` and `n` is written as `m^n`. The algorithm for producing the random numbers after p integers have been produced is shown in the following. Initially the index k can be set to 0.

1. If $k < q$, set $j = k + q$, else set $j = k - p + q$.
2. Set $x_k = x_k \oplus x_j$; x_k is the desired random number for this iteration. If a random number between 0 and 1 is desired, divide x_k by the maximum possible integer that the computer can hold.

3. Increment k to $(k+1) \mod p$.

Because the exclusive or operator and bit manipulation is very fast, this random number generator is very fast. However, the period may not be long enough for some applications, and the correlations between numbers might not be as good as needed. The shuffling algorithm discussed in Problem 7.36 should be used to improve this generator.

These two examples of random number generators illustrate their general nature. That is, numbers in the sequence are used to find the succeeding ones according to a well-defined algorithm. The sequence is determined by the seed, the first number of the sequence, or the first p members of the sequence for the generalized feedback shift register and related generators. Usually, the maximum possible period is related to the size of the computer word, for example, 32 or 64 bits. The choice of the constants and the proper initialization of the sequence is very important, and thus these algorithms must be implemented with care.

There is no necessary and sufficient test for the randomness of a finite sequence of numbers; the most that can be said about any finite sequence of numbers is that it is *apparently* random. Because no single statistical test is a reliable indicator, we need to consider several tests. Some of the best known tests are discussed in Problem 7.35. Many of these tests can be stated in terms of random walks.

Problem 7.35. Statistical tests of randomness

(a) *Period*. An obvious requirement for a random number generator is that its period be much greater than the number of random numbers needed in a specific calculation. One way to visualize the period of the random number generator is to use it to generate a plot of the displacement x of a random walker as a function of the number of steps N. When the period of the random number is reached, the plot will begin to repeat itself. Generate such a plot using (7.59) for $a = 899$, $c = 0$, and $m = 32768$, and for $a = 16807$, $c = 0$, and $m = 2^{31}-1$ with $x_0 = 12$. What are the periods of the corresponding random number generators? Obtain similar plots using different values for the parameters a, c, and m. Why is the seed value $x_0 = 0$ forbidden for the choice $c = 0$? Do some combinations of a, c, and m give longer periods than others?

(b) *Uniformity*. A random number sequence should contain numbers distributed in the unit interval with equal probability. The simplest test of uniformity is to divide this interval into M equal size subintervals or bins. For example, consider the first $N = 10^4$ numbers generated by (7.59) with $a = 106$, $c = 1283$, and $m = 6075$ (see Press et al.). Place each number into one of $M = 100$ bins. Is the number of entries in each bin approximately equal? What happens if you increase N?

(c) *Chi-square test*. Is the distribution of numbers in the bins of part (b) consistent with the laws of statistics? The most common test of this consistency is the *chi-square* or χ^2 test. Let y_i be the observed number in bin i and E_i be the expected value. The chi-square statistic is

$$\chi^2 = \sum_{i=1}^{M} \frac{(y_i - E_i)^2}{E_i}. \tag{7.61}$$

For the example in part (b) with $N = 10^4$ and $M = 100$, we have $E_i = 100$. The magnitude of the number χ^2 is a measure of the agreement between the observed and expected distributions; chi^2 should not be too big or too small. In general, the individual terms in the sum

(7.61) are expected to be order one, and because there are M terms in the sum, we expect $\chi^2 \leq M$. As an example, we did five independent runs of a random number generator with $N = 10^4$ and $M = 100$ and found $\chi^2 \approx 92$, 124, 85, 91, and 99. These values of χ^2 are consistent with this expectation. Although we usually want χ^2 to be as small as possible, we would be suspicious if $\chi^2 \approx 0$, because such a small value suggests that N is a multiple of the period of the generator and that each value in the sequence appears an equal number of times.

(d) *Filling sites.* Although a random number sequence might be distributed in the unit interval with equal probability, the consecutive numbers might be correlated in some way. One test of this correlation is to fill a square lattice of L^2 sites at random. Consider an array $n(x,y)$ that is initially empty, where $1 \leq x_i, y_i \leq L$. A site is selected randomly by choosing its two coordinates x_i and y_i from two consecutive numbers in the sequence. If the site is empty, it is filled and $n(x_i, y_i) = 1$; otherwise it is not changed. This procedure is repeated t times, where t is the number of Monte Carlo steps per site. That is, the time is increased by $1/L^2$ each time a pair of random numbers is generated. Because this process is analogous to the decay of radioactive nuclei, we expect that the fraction of empty lattice sites should decay as e^{-t}. Determine the fraction of unfilled sites using the random number generator that you have been using for $L = 10$, 15, and 20. Are your results consistent with the expected fraction? Repeat the same test using (7.59) with $a = 231$, $c = 0$, and $m = 65{,}549$. The existence of triplet correlations can be determined by a similar test on a simple cubic lattice by choosing the three coordinates x_i, y_i, and z_i from three consecutive random numbers.

(e) *Parking lot test.* Fill sites as in part (d) and draw the sites that have been filled. Do the filled sites look random, or are there stripes of filled sites? Try $a = 65.549$, $c = 0$, and $m = 231$.

(f) *Hidden correlations.* Another way of checking for correlations is to plot x_{i+k} versus x_i. If there are any obvious patterns in the plot, then there is something wrong with the generator. Use the generator (7.59) with $a = 16{,}807, c = 0$, and $m = 2^{31} - 1$. Can you detect any structure in the plotted points for $k = 1$ to $k = 5$? Test the random number generator that you have been using. Do you see any evidence of lattice structure, for example, equidistant parallel lines? Is the logistic map $x_{n+1} = 4x_n(1 - x_n)$ a suitable random number generator?

(g) *Short-term correlations.* Another measure of short term correlations is the autocorrelation function

$$C(k) = \frac{\langle x_{i+k} x_i \rangle - \langle x_i \rangle^2}{\langle x_i x_i \rangle - \langle x_i \rangle \langle x_i \rangle} \tag{7.62}$$

where x_i is the ith term in the sequence. We have used the fact that $\langle x_{i+k} \rangle = \langle x_i \rangle$; that is, the choice of the origin of the sequence is irrelevant. The quantity $\langle x_{i+k} x_i \rangle$ is found for a particular choice of k by forming all the possible products of $x_{i+k} x_i$ and dividing by the number of products. If x_{i+k} and x_i are not correlated, then $\langle x_{i+k} x_i \rangle = \langle x_{i+k} \rangle \langle x_i \rangle$ and $C(k) = 0$. Is $C(k)$ identically zero for any finite sequence? Compute $C(k)$ for $a = 106$, $c = 1283$, and $m = 6075$.

(h) *Random walk.* A test based on the properties of random walks has been proposed by Vattulainen et al. Assume that a walker begins at the origin of the x-y plane and walks for N steps. Average over M walkers and count the number of walks that end in each quadrant q_i. Use the χ^2 test (7.61) with $y_i \to q_i$, $M = 4$, and $E_i = M/4$. If $\chi^2 > 7.815$ (a 5% probability if the random number generator is perfect), we say that the run fails. The random number generator fails if two out of three independent runs fail. The probability of a perfect generator failing two out of three runs is approximately $3 \times 0.95 \times (0.05)^2 \approx 0.007$. Test several random number generators. □

Problem 7.36. Improving random number generators

One way to reduce sequential correlation and to lengthen the period is to mix or *shuffle* the random numbers produced by a random number generator. A standard procedure is to begin with a list of N random numbers (between 0 and 1) using a given generator `rng`. The number N is arbitrary but should be less than the period of `rng`. Also generate one more random number r_{extra}. Then for each desired random number use the following procedure:

(i) Calculate the integer k given by `(int)(N*r`$_{\text{extra}}$`)`. Use the kth random number r_k from your list as the desired random number.

(ii) Set r_{extra} equal to the random number r_k chosen in step (i).

(iii) Generate a new random number r from `rng` and use it to replace the number chosen in step (i); that is, $r_k = r$.

Consider a random number generator with a relatively short period and strong sequential correlation and show that this shuffling scheme improves the quality of the random number sequence. □

At least some of the statistical tests given in Problem 7.35 should be done whenever serious calculations are contemplated. However, even if a random number generator passes all these tests, there can still be problems in rare cases. Typically, these problems arise when a small number of events have a large weight. In these cases a very small bias in the random number generator might lead to systematic errors, and two generators, which appear equally good as determined by various statistical tests, might give statistically different results in a specific application (see Project 15.34). For this reason it is important that the particular random number generator used be reported along with the actual results. Confidence in the results can also be increased by repeating the calculation with another random number generator.

Because all random number generators are based on a deterministic algorithm, it is always possible to construct a test generator for which a particular algorithm will fail. The success of a random number generator in passing various statistical tests is necessary, but it is not a sufficient condition for its use in all applications. In Project 15.34 we discuss an application of Monte Carlo methods to the Ising model for which some popular random number generators give incorrect results.

7.10 Variational Methods

Many problems in physics can be formulated in terms of a variational principle. In the following, we consider examples of variational principles in geometrical optics and classical mechanics. We then discuss how Monte Carlo methods can be applied to these problems. A more sophisticated application of Monte Carlo methods to a variational problem in quantum mechanics is discussed in Chapter 16.

Our everyday experience of light leads naturally to the concept of light rays. This description of light propagation, called *geometrical* or *ray optics*, is applicable when the wavelength of light is small compared to the linear dimensions of any obstacles or openings. The path of a light ray can be formulated in terms of Fermat's principle of least time: A ray of light follows the path between two points (consistent with any constraints) that requires the least amount of time. Fermat's principle can be adopted as the basis of geometrical optics. For example, Fermat's principle implies that light travels from a point A to a point B in a straight line in a homogeneous medium. Because the speed of light is constant along any path within the medium, the path of shortest time is the path of shortest distance; that is, a straight line from A to B. What happens if we impose the constraint that the light must strike a mirror before reaching B?

The speed of light in a medium can be expressed in terms of c, the speed of light in a vacuum, and the index of refraction n of the medium:

$$v = \frac{c}{n}. \tag{7.63}$$

Suppose that a light ray in a medium with index of refraction n_1 passes through a second medium with index of refraction n_2. The two media are separated by a plane surface. We now show how we can use Fermat's principle and a simple Monte Carlo method to find the path of the light. The analytic solution to this problem using Fermat's principle is found in many texts (cf. Feynman et al.).

Our strategy, as implemented in class `Fermat`, is to begin with a straight path and to make changes in the path at random. These changes are accepted only if they reduce the travel time of the light. Some of the features of `Fermat` and `FermatApp` include:

1. Light propagates from left to right through `N` regions. The index of refraction `n[i]` is uniform in each region `[i]`. The index `i` increases from left to right. We have chosen units such that the speed of light in vacuum equals unity.
2. Because the light propagates in a straight line in each medium, the path of the light is given by the coordinates `y[i]` at each boundary.
3. The coordinates of the light source and the detector are at `(0,y[0])` and `(N,y[N])`, respectively, where `y[0]` and `y[N]` are fixed.
4. The path is the connection of the set of points at the boundary of each region.
5. The path of the light is found by choosing the boundary `i` at random and generating a trial value of `y[i]` that differs from its previous value by a random number between `-dy` to `dy`. If the trial value of `y[i]` yields a shorter travel time, this value becomes the new value for `y[i]`.
6. The path is redrawn whenever it is changed.

Listing 7.6: Fermat class.

```java
package org.opensourcephysics.sip.ch07;
public class Fermat {
   double y[];       // y coordinate of light ray, index is x coordinate
   // light speed of ray for medium starting at index value
   double v[];
   int N;            // number of media
   // change in index of refraction from one region to the next
   double dn;
   double dy = 0.1; // maximum change in y position
   int steps;

   public void initialize() {
      y = new double[N+1];
      v = new double[N];
      double indexOfRefraction = 1.0;
      for(int i = 0;i<=N;i++) {
         y[i] = i; // initial path is a straight line
      }
      for(int i = 0;i<N;i++) {
         v[i] = 1.0/indexOfRefraction;
         indexOfRefraction += dn;
      }
      steps = 0;
   }

   public void step() {
      int i = 1+(int) (Math.random()*(N-1));
      double yTrial = y[i]+2.0*dy*(Math.random()-0.5);
      double previousTime = Math.sqrt(Math.pow(y[i-1]-y[i], 2)+1)/v[i-1]; // left medium
      previousTime += Math.sqrt(Math.pow(y[i+1]-y[i], 2)+1)/v[i]; // right medium
      double trialTime = Math.sqrt(Math.pow(y[i-1]-yTrial, 2)+1)/v[i-1]; // left medium
      trialTime += Math.sqrt(Math.pow(y[i+1]-yTrial, 2)+1)/v[i]; // right medium
      if(trialTime<previousTime) {
         y[i] = yTrial;
      }
      steps++;
   }
}
```

Listing 7.7: Target class for Fermat's principle.

```
package org.opensourcephysics.sip.ch07;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.frames.PlotFrame;

public class FermatApp extends AbstractSimulation {
   Fermat medium = new Fermat();
   PlotFrame path = new PlotFrame("x", "y", "Light path");

   public FermatApp() {
      path.setAutoscaleX(true);
      path.setAutoscaleY(true);
      path.setConnected(true); // draw lines between points
   }

   public void initialize() {
      medium.dn = control.getDouble("Change in index of refraction");
      medium.N = control.getInt("Number of media segments");
      medium.initialize();
      path.clearData();
   }

   public void doStep() {
```

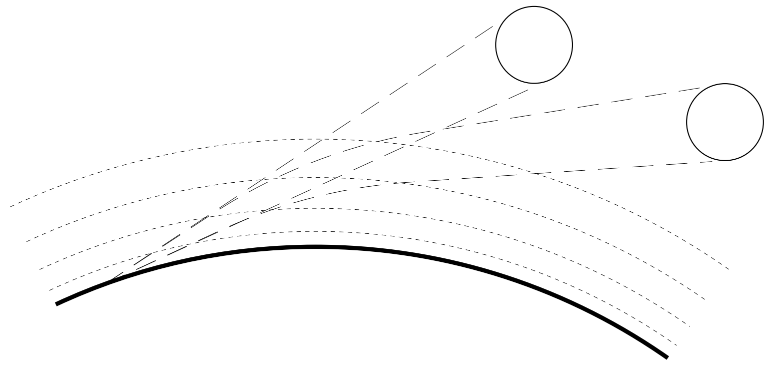


Figure 7.13: Near the horizon, the apparent (exaggerated) position of the sun is higher than the true position of the sun. Note that the light rays from the true sun are curved due to refraction.

```
         medium.step();
         path.clearData();
         for(int i = 0;i<=medium.N;i++) {
            path.append(0, i, medium.y[i]);
         }
         path.setMessage(medium.steps+" steps");
      }

      public void reset() {
         control.setValue("Change in index of refraction", 0.5);
         control.setValue("Number of media segments", 2);
         path.clearData();
         enableStepsPerDisplay(true);
      }

      public static void main(String[] args) {
         SimulationControl.createApp(new FermatApp());
      }
   }
```

Problem 7.37. The law of refraction

(a) Use Fermat and FermatApp to determine the angle of incidence θ_1 and the angle of refraction θ_2 between two media with different indices of refraction. The angles θ_1 and θ_2 are measured from the normal to the boundary. Set $N = 2$ and let the first medium be air ($n_1 \approx 1$) and the second medium be glass ($n_2 \approx 1.5$). Describe the path of the light after a number of trial paths are attempted. Add statements to the program to determine θ_1 and θ_2, the vertical position of the intersection of the light at the boundary between the two media, and the total time for the light to go from (0,y[0]) to (2,y[2]).

(b) Modify the program so that the first medium represents glass ($n_1 \approx 1.5$) and the second medium represents water ($n_2 \approx 1.33$). Verify that your results are consistent with $n_2 \sin\theta_2 = n_1 \sin\theta_1$. □

Problem 7.38. Inhomogeneous media

(a) The earth's atmosphere is thin at the top and dense near the earth's surface. We can model this inhomogeneous medium by dividing the atmosphere into equal width segments each of which is homogeneous. To simulate this atmosphere run your program with `N` = 10 and `dn` = 0.1, and find the path of least time. Use your results to explain why when we see the sun set, the sun is already below the horizon (see Figure 7.13).

(b)* Modify your program to find the appropriate distribution $n(y)$ for a fiber optic cable, which we take to be a flat, long ribbon. In this case the ith region corresponds to a cross-sectional slab through the cable. Although a real cable is three-dimensional, we consider a two-dimensional cable for simplicity. We want the cable to have the property that if a ray of light starts from one side of the cable and ends at the other, the slope dy/dx of the path should be near zero at the edges so that light does not escape from the cable. □

Fermat's principle is an example of an extremum principle. An extremum means that a small change ϵ in an independent variable leads to a change in a function (more precisely, a function of functions) that is proportional to ϵ^2 or a higher power of ϵ. An important extremum principle in classical mechanics is based on the action S:

$$S = \int_{t_0}^{t_{\text{final}}} L\,dt \tag{7.64}$$

where t_0 and t_{final} are the initial and final times, respectively. The Lagrangian L in (7.64) is the kinetic energy minus the potential energy. The extremum principle for the action is known as *the principle of least action* or Hamilton's action principle. The path where (7.64) is stationary (either a minimum or a saddle point) satisfies Newton's second law (for conservative forces). One reason for the importance of the principle of least action is that quantum mechanics can be formulated in terms of an integral over the action (see Section 16.10).

To use (7.64) to find the motion of a single particle in one dimension, we fix the position at the chosen initial and final times, $x(t_0)$ and $x(t_{\text{final}})$, and then choose the velocities and positions for the intermediate times $t_0 < t < t_{\text{final}}$ to minimize the action. One way to implement this procedure numerically is to convert the integral in (7.64) to a sum:

$$S \approx \sum_{i=1}^{N-1} L(t_i)\,\Delta t \tag{7.65}$$

where $t_i = t_0 + i\Delta t$. (The approximation used to obtain (7.65) is known as the rectangular approximation and is discussed in Chapter 11.) For a single particle in one dimension moving in an external potential $u(x)$, we can write

$$L_i \approx \frac{m}{2(\Delta t)^2}(x_{i+1} - x_i)^2 - u(x_i) \tag{7.66}$$

where m is the mass of the particle and $u(x_i)$ is the potential energy of the particle at x_i. The velocity has been approximated as the difference in position divided by the change in time Δt.

Problem 7.39. Principle of least action

(a) Write a program to minimize the action S given in (7.64) for the motion of a single particle in one dimension. Use the approximate form of the Lagrangian given in (7.66). One way to write the program is to modify class `Fermat` so that the vertical coordinate for the light ray becomes the position of the particle, and the horizontal region number `i` becomes the discrete time interval of duration Δt.

(b) Verify your program for the case of free fall for which the potential energy is $u(y) = mgy$. Choose $y(t = 0) = 2\,\text{m}$ and $y(t = 10\,\text{s}) = 8\,\text{m}$ and begin with $N = 20$. Allow the maximum change in the position to be 5 m.

(c) Consider the harmonic potential $u(x) = \frac{1}{2}kx^2$. What shape do you expect the path $x(t)$ to be? Increase N to approximately 50 and estimate the path by minimizing the action. □

It is possible to extend the principle of least action to more dimensions or particles. However, it is necessary to begin with a path close to the optimum one to obtain a good approximation to the optimum path in a reasonable time.

In Problems 7.37–7.39, a simple Monte Carlo algorithm that always accepts paths that reduce the time or action is sufficient. However, for more complicated index of refraction distributions or potentials, it is possible that such a simple algorithm will find only a local minimum, and the global minimum will be missed. The problem of finding the global minimum is very general and is shared by all optimization algorithms if the system has many relative minima. Optimization is a very active area of research in many fields of science and engineering. Ideas from physics, biology, and computer science have led to many improved algorithms. We will discuss some of these algorithms in Chapter 15. In most of these algorithms, paths that are worse than the current path are sometimes accepted in an attempt to climb out of a local minimum. Other algorithms involve ways of sampling over a wider range of possible paths. Another approach is to convert the Monte Carlo algorithm into a deterministic algorithm. We have already mentioned that an analytic variational calculation leads to Newton's second law. Passerone and Parrinello discuss an algorithm for looking for extrema in the action by maintaining the discrete structure in (7.66) and then finding the extremum by taking the derivative with respect to each coordinate, x_i and setting the resulting equations equal to zero. This procedure leads to a set of deterministic equations that need to be solved numerically. The performance can be improved by enforcing energy conservation and using some other tricks.

7.11 Projects

Almost all of the problems in this chapter can be done using more efficient programs, greater number of trials, and larger systems. More applications of random walks and random number sequences are discussed in subsequent chapters. Many more ideas for projects can be gained from the references.

Project 7.40. Competition between diffusion and fragmentation

As we have discussed, random walks are useful for understanding diffusion in contexts more general than the movement of a particle. Consider a particle in solution whose mass can grow either by the absorption of particles or shrink by the loss of small particles, including fragmentation. We can model this process as a random walk by replacing the position of the particle by its mass. One difference between this case and the random walks we have studied so far is that the random variable, the mass, must be positive. The model of Ferkinghoff–Berg et al. can be summarized as follows:

(i) Begin with N objects with some distribution of lengths. Let the integer L_i represent the length of the ith object.

(ii) All the objects change their length by ± 1. This step is analogous to a random walk. If the length of an object becomes equal to 0, it is removed from the system. An easy way to eliminate the ith object is to set its length equal to the length of the last object and reduce N by unity.

(iii) Choose one object at random with a probability that is proportional to the length of the object. Fragment this object into two objects, where the fraction of the mass going to each object is random.

(iv) Repeat steps (ii) and (iii).

(a) Write a program to implement this algorithm in one dimension. One way to implement step (iii) is given in the following code, where `totalMass` is the sum of the lengths of all the objects.

```
int i = 0;    // label of object
// length of first object, all lengths are integers
// choose object to fragment so that choice is proportional to length
int sum = length[0];
int x = (int)(Math.random()*totalMass);
while(sum < x) {
   i++;
   sum += length[i];
}
// if object big enough to fragment, choose random fraction
// for each part
if(length[i] > 1) {
   int partA = 1 + (int)(Math.random()*(length[i]-1));
   int partB = length[i] - partA;
   length[i] = partA;
   length[numberOfObjects] = partB;   // new object
   numberOfObjects++;
}
```

The main quantity of interest is the distribution of lengths $P(L)$. Explore a variety of initial length distributions with a total mass of 5000 for which the distribution is peaked at about 20 mass units. Is the long time behavior of $P(L)$ similar in shape for any initial distribution? Compute the total mass (sum of the lengths) and output this value periodically. Although the total mass will fluctuate, it should remain approximately constant. Why?

(b) Collect data for three different initial distributions with the same number of objects N, and scale $P(L)$ and L so that the three distributions roughly fall on the same curve. For example, you can scale $P(L)$ so that the maximum of the three distributions has the same value. Then multiply each value of L by a factor so that the distributions overlap.

(c) The analytic results suggest that the universal behavior can be obtained by scaling L by the total mass raised to the 1/3 power. Is this prediction consistent with your results? Test this hypothesis by adjusting the initial distributions so that they all have the same total mass. Your results for the long time behavior of $P(L)$ should fall on a universal curve. Why is this universality interesting? How can this result be used to analyze different systems? Would you need to do a new simulation for each value of L?

(d) What happens if step (iii) is done more or less often than each random change of length. Does the scaling change? □

Project 7.41. Application of the pivot algorithm to self-avoiding walks

The algorithms that we have discussed for generating self-avoiding random walks are all based on making *local* deformations of the walk (polymer chain) for a given value of N, the number of bonds. As discussed in Problem 7.31, the time τ between statistically independent configurations is nonzero. The problem is that τ increases with N as some power, for example, $\tau \sim N^3$. This power law dependence of τ on N is called *critical slowing down* and implies that it becomes increasingly more time consuming to generate long walks. We now discuss an example of a *global* algorithm that reduces the dependence of τ on N. Another example of a global algorithm that reduces critical slowing down is discussed in Project 15.32.

(a) Consider the walk shown in Figure 7.14a. Select a site at random and one of the four possible directions. The shorter portion of the walk is rotated (pivoted) to this new direction by treating the walk as a rigid structure. The new walk is accepted only if the new walk is self-avoiding; otherwise, the old walk is retained. (The shorter portion of the walk is chosen to save computer time.) Some typical moves are shown in Figure 7.14. Note that if an end point is chosen, the previous walk is retained. Write a program to implement this algorithm and compute the dependence of the mean square end-to-end distance R^2 on N. Consider values of N in the range $10 \leq N \leq 80$. A discussion of the results and the implementation of the algorithm can be found in MacDonald et al. and Madras and Sokal, respectively.

(b) Compute the correlation time τ for different values of N using the approach discussed in Problem 7.31b. □

Project 7.42. Pattern formation

In Problem 7.34 we saw that simple patterns can develop as a result of random behavior. The phenomenon of pattern formation is of much interest in a variety of contexts ranging from the large scale structure of the universe to the roll patterns seen in convection (for example, smoke rings). In the following, we explore the patterns that can develop in a simple reaction diffusion model based on the reactions, $A+2B \to 3B$ and $B \to C$, where C is inert. Such a reaction is called *autocatalytic*.

In Problem 7.34 we considered chemical reactions in a closed system where the reactions can proceed to equilibrium. In contrast, open systems allow a continuous supply of fresh reactants and a removal of products. These two processes allow steady states to be realized and oscillatory conditions to be maintained indefinitely. In this problem we assume that A is added at a constant rate, and that both A and B are removed by the feed process. Pearson (see references) modeled these processes by two coupled reaction diffusion equations:

$$\frac{\partial A}{\partial t} = D_A \nabla^2 A - AB^2 + f(1-A) \tag{7.67a}$$

$$\frac{\partial B}{\partial t} = D_B \nabla^2 B + AB^2 - (f+k)B. \tag{7.67b}$$

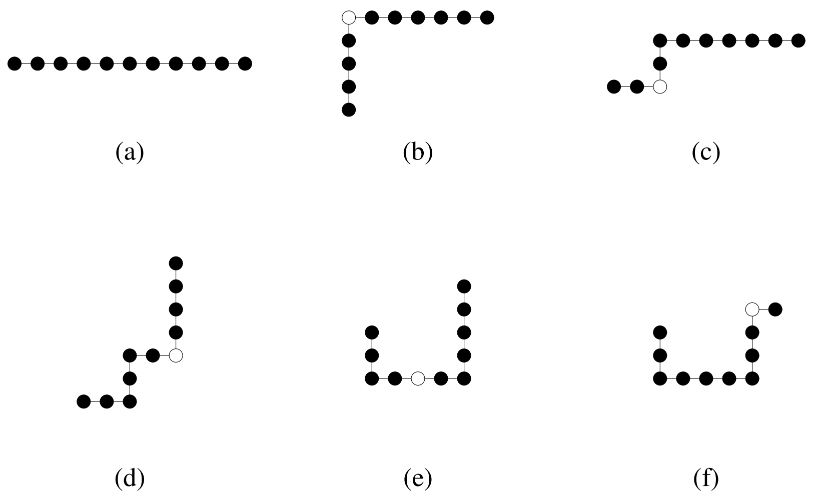


Figure 7.14: Examples of the first several changes generated by the pivot algorithm for a self-avoiding walk of $N = 10$ bonds (11 sites). The open circle denotes the pivot point. This figure is adopted from the article by MacDonald et al.

The AB^2 term represents the reaction $A + 2B \to 3B$. This term is negative in (7.67a) because the reactant A decreases and is positive in (7.67b) because the reactant B increases. The term $+f$ represents the constant addition of A, and the terms $-fA$ and $-fB$ represent the removal process; the term $-kB$ represents the reaction $B \to C$. All the quantities in (7.67) are dimensionless. We assume that the diffusion coefficients are $D_A = 2 \times 10^{-5}$ and $D_B = 10^{-5}$, and the behavior of the system is determined by the values of the rate constant k and the feed rate f.

(a) We first consider the behavior of the reaction kinetics that results when the diffusion terms in (7.67) are neglected. It is clear from (7.67) that there is a trivial steady state solution with $A = 1$, $B = 0$. Are there other solutions, and if so, are they stable? The steady state solutions can be found by solving (7.67) with $\partial A/\partial t = \partial B/\partial t = 0$. To determine the stability, we can add a perturbation and determine whether the perturbation grows or not. However, without the diffusion terms, it is more straightforward to solve (7.67) numerically using a simple Euler algorithm. Choose a time step equal to unity and let $A = 0.1$ and $B = 0.5$ at $t = 0$. Determine the steady state values for $0 < f \leq 0.3$ and $0 < k \leq 0.07$ in increments of $\Delta f = 0.02$ and $\Delta k = 0.005$. Record the steady state values of A and B. Then repeat this exercise for the initial values $A = 0.5$ and $B = 0.1$. You should find that for some values of f and k, only one steady state solution can be obtained for the two initial conditions, and for other initial values of A and B there are two steady state solutions. Try other initial conditions. If you obtain a new solution, change the initial A or B slightly to see if your new solution is stable. On an f versus k plot indicate, where there are two solutions and where there are one. In this way you can determine the approximate phase diagram for this process.

(b) There is a small region in f-k space where one of the steady state solutions becomes unstable and periodic solutions occur (the mechanism is known as a Hopf bifurcation). Try $f = 0.009$, $k = 0.03$, and set $A = 0.1$ and $B = 0.5$ at $t = 0$. Plot the values of A and B versus the time t. Are they periodic? Try other values of f and k and estimate where the periodic solutions occur.

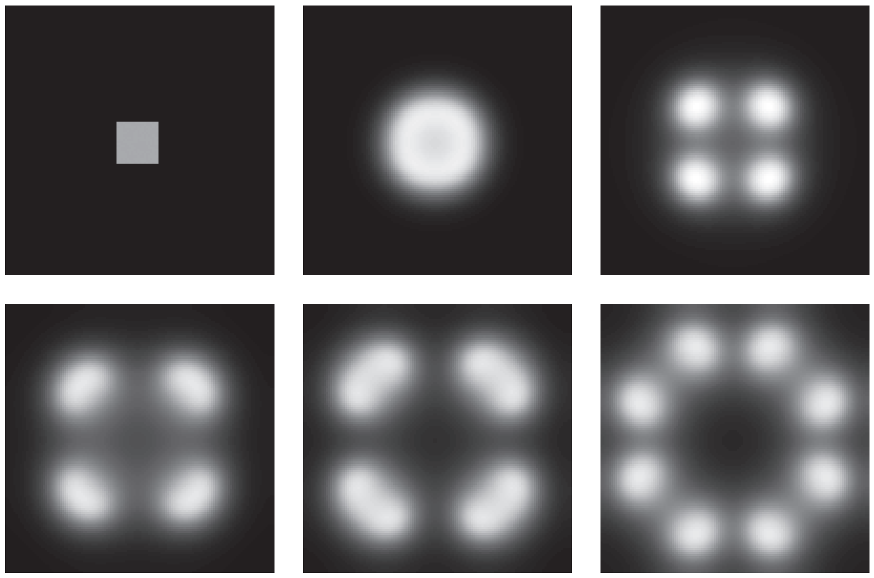


Figure 7.15: Evolution of the pattern starting from the initial conditions suggested in Project 7.42c.

(c) Numerical solutions of the full equation with diffusion (7.67) can be found by making a finite difference approximation to the spatial derivatives as in (3.16) and using a simple Euler algorithm for the time integration. Adopt periodic boundary conditions. Although it is straightforward to write a program to do the numerical integration, an exploration of the dynamics of this system requires much computer resources. However, we can find some preliminary results with a small system and a coarse grid. Consider a 0.5×0.5 system with a spatial mesh of 128×128 grid points on a square lattice. Choose $f = 0.18$, $k = 0.057$, and $\Delta t = 0.1$. Let the entire system be in the initial trivial state ($A = 1$, $B = 0$) except for a 20×20 grid located at the center of the system where the sites are $A = 1/2$, $B = 1/4$ with a $\pm 1\%$ random noise. The effect of the noise is to break the square symmetry. Let the system evolve for approximately 80,000 time steps and look at the patterns that develop. Color code the grid according to the concentration of A, with red representing $A = 1$ and blue representing $A \approx 0.2$ and with several intermediate colors. Very interesting patterns have been found by Pearson (see Figure 7.15). □

Appendix 7A: Random Walks and the Diffusion Equation

To gain some insight into the relation between random walks and the diffusion equation, we first show that the latter implies that $\langle x(t) \rangle$ is zero and $\langle x^2(t) \rangle$ is proportional to t. We rewrite the diffusion equation (7.27) here for convenience:

$$\frac{\partial P(x,t)}{\partial t} = D \frac{\partial^2 P(x,t)}{\partial x^2}. \tag{7.68}$$

To derive the t dependence of $\langle x(t)\rangle$ and $\langle x^2(t)\rangle$ from (7.68), we write the average of any function of x as

$$\langle f(x,t)\rangle = \int_{-\infty}^{\infty} f(x)P(x,t)\,dx. \tag{7.69}$$

The average displacement is given by

$$\langle x(t)\rangle = \int_{-\infty}^{\infty} xP(x,t)\,dx. \tag{7.70}$$

To do the integral on the right-hand side of (7.70), we multiply both sides of (7.68) by x and formally integrate over x:

$$\int_{-\infty}^{\infty} x\frac{\partial P(x,t)}{\partial t}\,dx = D\int_{-\infty}^{\infty} x\frac{\partial^2 P(x,t)}{\partial x^2}\,dx. \tag{7.71}$$

The left-hand side can be expressed as

$$\int_{-\infty}^{\infty} x\frac{\partial P(x,t)}{\partial t}\,dx = \frac{\partial}{\partial t}\int_{-\infty}^{\infty} xP(x,t)\,dx = \frac{d}{dt}\langle x\rangle. \tag{7.72}$$

The right-hand side of (7.71) can be written in the desired form by doing an integration by parts:

$$D\int_{-\infty}^{\infty} x\frac{\partial^2 P(x,t)}{\partial x^2}\,dx = Dx\frac{\partial P(x,t)}{\partial x}\bigg|_{x=-\infty}^{x=\infty} - D\int_{-\infty}^{\infty} \frac{\partial P(x,t)}{\partial x}\,dx. \tag{7.73}$$

The first term on the right hand side of (7.73) is zero because $P(x = \pm\infty, t) = 0$ and all the spatial derivatives of P at $x = \pm\infty$ are zero. The second term is also zero because it integrates to $D[P(x=\infty,t) - P(x=-\infty,t)]$. Hence, we find that

$$\frac{d\langle x\rangle}{dt} = 0 \tag{7.74}$$

or $\langle x\rangle$ is a constant, independent of time. Because $x = 0$ at $t = 0$, we conclude that $\langle x\rangle = 0$ for all t.

To calculate $\langle x^2(t)\rangle$, we can use a similar procedure and perform two integrations by parts. The result is

$$\frac{d}{dt}\langle x^2(t)\rangle = 2D \tag{7.75}$$

or

$$\langle x^2(t)\rangle = 2Dt. \tag{7.76}$$

We see that the random walk and the diffusion equation have the same time dependence. In d-dimensional space, $2D$ is replaced by $2dD$.

The solution of the diffusion equation shows that the time dependence of $\langle x^2(t)\rangle$ is equivalent to the long time behavior of a simple random walk on a lattice. In the following, we show directly that the continuum limit of the one-dimensional random walk model is a diffusion equation.

If there is an equal probability of taking a step to the right or left, the random walk can be written in terms of the simple master equation

$$P(i,N) = \frac{1}{2}[P(i+1,N-1) + P(i-1,N-1)] \tag{7.77}$$

where $P(i,N)$ is the probability that the walker is at site i after N steps. To obtain a differential equation for the probability density $P(x,t)$, we identify $t = N\tau$, $x = ia$, and $P(i,N) = aP(x,t)$, where τ is the time between steps and a is the lattice spacing. This association allows us to rewrite (7.77) in the equivalent form

$$P(x,t) = \frac{1}{2}[P(x+a,t-\tau) + P(x-a,t-\tau)]. \tag{7.78}$$

We rewrite (7.78) by subtracting $P(x,t-\tau)$ from both sides of (7.78) and dividing by τ:

$$\frac{1}{\tau}\Big[P(x,t) - P(x,t-\tau)\Big] = \frac{a^2}{2\tau}\Big[P(x+a,t-\tau) - 2P(x,t-\tau) + P(x-a,t-\tau)\Big]a^{-2}. \tag{7.79}$$

If we expand $P(x,t-\tau)$ and $P(x \pm a,t-\tau)$ in a Taylor series and take the limit $a \to 0$ and $\tau \to 0$ with the ratio $D \equiv a^2/2\tau$ finite, we obtain the diffusion equation

$$\frac{\partial P(x,t)}{\partial t} = D\,\frac{\partial^2 P(x,t)}{\partial x^2}. \tag{7.80a}$$

The generalization of (7.80a) to three dimensions is

$$\frac{\partial P(x,y,z,t)}{\partial t} = D\,\nabla^2 P(x,y,z,t) \tag{7.80b}$$

where $\nabla^2 = \partial^2/\partial x^2 + \partial^2/\partial y^2 + \partial^2/\partial x^2$ is the Laplacian operator. Equation (7.80) is known as the *diffusion* equation and is frequently used to describe the dynamics of fluid molecules.

The direct numerical solution of the prototypical *parabolic* partial differential equation (7.80) is a nontrivial problem in numerical analysis (cf. Press et al. or Koonin and Meredith). An indirect method of solving (7.80) numerically is to use a Monte Carlo method; that is, replace the partial differential equation (7.80) by a corresponding random walk on a lattice with discrete time steps. Because the asymptotic behavior of the partial differential equation and the random walk model are equivalent, this approach uses the Monte Carlo technique as a method of *numerical analysis*. In contrast, if our goal is to understand a random walk lattice model directly, the Monte Carlo technique is a *simulation* method. The difference between simulation and numerical analysis is sometimes in the eyes of the beholder.

Problem 7.43. Biased random walk

Show that the form of the differential equation satisfied by $P(x,t)$ corresponding to a random walk with a drift; that is, a walk for $p \neq q$, is

$$\frac{\partial P(x,t)}{\partial t} = D\,\nabla^2 P(x,y,z,t) - v\,\frac{\partial P(x,t)}{\partial x}. \tag{7.81}$$

How is v related to p and q? □

References and Suggestions for Further Reading

Daniel J. Amit, G. Parisi, and L. Peleti, "Asymptotic behavior of the "true" self-avoiding walk," Phys. Rev. B **27**, 1635–1645 (1983).

Panos Argyrakis, "Simulation of diffusion-controlled chemical reactions," Computers in Physics **6**, 525–579 (1992).

G. T. Barkema, Parthapratim Biswas, and Henk van Beijeren, "Diffusion with random distribution of static traps," Phys. Rev. Lett. **87**, 170601 (2001).

J. M. Bernardo and A. F. M. Smith, *Bayesian Theory* (John Wiley & Sons, 1994). Bayes theorem is stated concisely on page 2.

J. Bernasconi and L. Pietronero, "True self-avoiding walk in one dimension," Phys. Rev. B **29**, 5196–5198 (1984). The authors present results for the exponent ν accurate to 1%.

Philip R. Bevington and D. Keith Robinson, *Data Reduction and Error Analysis for the Physical Sciences*, 3rd ed. (McGraw–Hill, 2003).

I. Carmesin and Kurt Kremer, "The bond fluctuation model: A new effective algorithm for the dynamics of polymers in all spatial dimensions," Macromolecules **21**, 2819–2823 (1988). The bond fluctuation model is an efficient method for simulating the dynamics of polymer chains and would be the basis of an excellent project.

S. Chandrasekhar, "Stochastic problems in physics and astronomy," Rev. Mod. Phys. **15**, 1–89 (1943). This article is reprinted in M. Wax, *Selected Papers on Noise and Stochastic Processes* (Dover, 1954).

William S. Cleveland and Robert McGill, "Graphical perception and graphical methods for analyzing scientific data," Science **229**, 828–833 (1985). There is more to analyzing data than least squares fits.

Mohamed Daoud, "Polymers," Chapter 6 in Armin Bunde and Shlomo Havlin, editors, *Fractals in Science*, Springer-Verlag (1994).

Roan Dawkins and Daniel ben–Avraham, "Computer simulations of diffusion-limited reactions," Comput. Sci. Eng. **3** (1), 72–76 (2001).

R. Everaers, I. S. Graham, and M. J. Zuckermann, "End-to-end distance and asymptotic behavior of self-avoiding walks in two and three dimensions," J. Phys. A **28**, 1271–1293 (1995).

Jesper Ferkinghoff–Borg, Mogens H. Jensen, Joachim Mathiesen, Poul Olesen, and Kim Sneppen, "Competition between diffusion and fragmentation: An important evolutionary process of nature," Phys. Rev. Lett. **91**, 266103 (2003). The results of the model were compared with experimental data on ice crystal sizes and the length distribution of α helices in proteins.

Richard P. Feynman, Robert B. Leighton, and Matthew Sands, *The Feynman Lectures on Physics* (Addison–Wesley, 1963). See Vol. 1, Chapter 26 for a discussion of the principle of least time and Vol. 2, Chapter 19 for a discussion of the principle of least action.

Pierre–Giles de Gennes, *Scaling Concepts in Polymer Physics* (Cornell University Press, 1979). A difficult but important text.

Peter Grassberger, "Pruned-enriched Rosenbluth method: Simulations of θ polymers of chain length up to 1 000 000," Phys. Rev. E **56**, 3682–3693 (1997).

Shlomo Havlin and Daniel ben–Avraham, "Diffusion in disordered media," Adv. Phys. **36**, 695 (1987). Section 7 of this review article discusses trapping and diffusion-limited reactions. Also see Daniel ben–Avraham and Shlomo Havlin, *Diffusion and Reactions in Fractals and Disordered Systems* (Cambridge University Press, 2001).

Shlomo Havlin, George H. Weiss, James E. Kiefer, and Menachem Dishon, "Exact enumeration of random walks with traps," J. Phys. A: Math. Gen. **17**, L347 (1984). The authors discuss a method based on exact enumeration for calculating the survival probability of random walkers on a lattice with randomly distributed traps.

Brian Hayes, "How to avoid yourself," Am. Scientist **86** (4), 314–319 (1998).

Z. Jiang and C. Ebner, "Simulation study of reaction fronts," Phys. Rev. A **42**, 7483–7486 (1990).

Peter R. Keller and Mary M. Keller, *Visual Cues* (IEEE Press, 1993). A well-illustrated book on data visualization techniques.

Donald E. Knuth, *Seminumerical Algorithms*, 2nd ed., Vol. 2 of *The Art of Computer Programming* (Addison-Wesley, 1981). The standard reference on random number generators.

Bruce MacDonald, Naeem Jan, D. L. Hunter, and M. O. Steinitz, "Polymer conformations through 'wiggling'," J. Phys. A **18**, 2627–2631 (1985). A discussion of the pivot algorithm summarized in Project 7.41. Also see Tom Kennedy, "A faster implementation of the pivot algorithm for self-avoiding walks," J. Stat. Phys. **106**, 407–429 (2002).

Vishal Mehra and Peter Grassberger, "Trapping reaction with mobile traps," Phys. Rev. E **65**, 050101-1–4 (R) (2002). This paper discusses the model of a single walker moving on a lattice of traps.

Elliott W. Montroll and Michael F. Shlesinger, "On the wonderful world of random walks," in *Nonequilibrium Phenomena II: From Stochastics to Hydrodynamics*, J. L. Lebowitz and E. W. Montroll, editors (North-Holland Press, 1984). The first part of this delightful review article chronicles the history of the random walk.

M. E. J. Newman and G. T. Barkema, *Monte Carlo Methods in Statistical Physics* (Oxford University, 1999). This book has a good section on random number generators.

Daniele Passerone and Michele Parrinello, "Action-derived molecular dynamics in the study of rare events," Phys. Rev. Lett. **87**, 108302 (2001). This paper describes a deterministic algorithm for finding extrema of the action. Also see D. Passerone, M. Ceccarelli, and M. Parrinello, J. Chem. Phys. **118**, 2025–2032 (2003).

John E. Pearson, "Complex patterns in a simple fluid," Science **261**, 189–192 (1993) or patt-sol/9304003. See also P. Gray and S. K. Scott, "Sustained oscillations and other exotic patterns of behavior in isothermal reactions," J. Phys. Chem. **89**, 22–32 (1985).

Thomas Prellberg, "Scaling of self-avoiding walks and self-avoiding trails in three dimensions," J. Phys. A **34**, L599–602 (2001). The author estimates that $\nu \approx 0.5874(2)$ for the self-avoiding walk in three dimensions.

Thomas Prellberg and Jaroslaw Krawczyk, "Flat histogram version of the pruned and enriched Rosenbluth method," Phys. Rev. Lett. **92**, 120602 (2004). The authors discuss an improved algorithm for simulating self-avoiding walks.

William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery, *Numerical Recipes*, 2nd ed. (Cambridge University Press, 1992). This classic book is available online at `<www.nr.com/>`. See Chapter 15 for a general discussion of the modeling of data, including general linear least squares and nonlinear fits and Chapter 19 for a discussion of the Crank–Nicholson method for solving diffusion-type partial equations.

Sidney Redner, *A Guide to First-Passage Processes* (Cambridge University Press, 2001).

Sidney Redner and Francois Leyvraz, "Kinetics and spatial organization of competitive reactions," Chapter 7 in Armin Bunde and Shlomo Havlin, editors, *Fractals in Science* (Springer–Verlag, 1994).

F. Reif, *Fundamentals of Statistical and Thermal Physics* (McGraw–Hill, 1965). This popular text on statistical physics has a good discussion on random walks (Chapter 1) and diffusion (Chapter 12).

F. Reif, *Statistical and Thermal Physics*, Berkeley Physics, Vol. 5 (McGraw–Hill, 1965). Chapter 2 introduces random walks.

Marshall N. Rosenbluth and Arianna W. Rosenbluth, "Monte Carlo calculation of the average extension of molecular chains," J. Chem. Phys. **23**, 356–359 (1955). One of the first Monte Carlo calculations of the self-avoiding walk.

Joseph Rudnick and George Gaspari, *Elements of the Random Walk* (Cambridge University Press, 2004). A graduate level text, but parts are accessible to undergraduates.

David Ruelle, *Chance and Chaos* (Princeton Publishing Company, 1993). A nontechnical introduction to chaos theory that discusses the relation of chaos to randomness.

Charles Ruhla, *The Physics of Chance* (Oxford University Press, 1992). A delightful book on probability in many contexts.

Andreas Ruttor, Georg Reents, and Wolfgang Kinzel, "Synchronization of random walks with reflecting boundaries," J. Phys. A: Math. Gen. **37**, 8609–8618 (2004).

G. L. Squires, *Practical Physics*, 4th ed. (Cambridge University Press, 2001). An excellent text on the design of experiments and the analysis of data.

John R. Taylor, *An Introduction to Error Analysis*, 2nd ed. (University Science Books, Oxford University Press (1997).

Edward R. Tuffe, *The Visual Display of Quantitative Information*, Graphics Press (1983) and *Envisioning Information* (Graphics Press, 1990). Also see Tuffe's Web site at <www.edwardtufte.com/>.

I. Vattulainen, T. Ala–Nissila, and K. Kankaala, "Physical tests for random numbers in simulations," Phys. Rev. Lett. **73**, 2513 (1994) and "Physical models as tests of randomness," Phys. Rev. E **52**, 3205–3214 (1995). Also see Vattulainen's Web site which has some useful programs: <www.physics.helsinki.fi/ vattulai/rngs.html>.

Peter H. Verdier and W. H. Stockmayer, "Monte Carlo calculations on the dynamics of polymers in dilute solution," J. Chem. Phys. **36**, 227–235 (1962).

Frederick T. Wall and Frederic Mandel, "Macromolecular dimensions obtained by an efficient Monte Carlo method without sample attrition," J. Chem. Phys. **63**, 4592–4595 (1975). An exposition of the reptation method.

George H. Weiss, "A primer of random walkology," Chapter 5 in Armin Bunde and Shlomo Havlin, editors, *Fractals in Science* (Springer–Verlag, 1994).

George H. Weiss and Shlomo Havlin, "Trapping of random walks on the line," J. Stat. Phys. **37**, 17–25 (1984). The authors discuss an analytic approach to the asymptotic behavior of one-dimensional random walkers with randomly placed traps.

George H. Weiss and Robert J. Rubin, “Random walks: Theory and selected applications,” Adv. Chem. Phys. **52**, 363–503 (1983). In spite of its research orientation, much of this review article can be understood by well-motivated students.

Charles A. Whitney, *Random Processes in Physical Systems* (John Wiley and Sons, 1990). An excellent introduction to random processes with many applications to astronomy.

Robert S. Wolff and Larry Yaeger, *Visualization of Natural Phenomena* (Springer–Verlag, 1993).

Chapter 8

The Dynamics of Many-Particle Systems

We simulate the dynamical behavior of many particle systems, such as dense gases, liquids, and solids, and observe their qualitative features. Some of the basic ideas of equilibrium statistical mechanics and kinetic theory are introduced.

8.1 Introduction

Given our knowledge of the laws of physics at the microscopic level, how can we understand the behavior of gases, liquids, and solids and more complex systems such as polymers and proteins? For example, consider two cups of water prepared under similar conditions. Each cup contains approximately 10^{25} molecules which mutually interact and, to a good approximation, move according to the laws of classical physics. Although the intermolecular forces produce a complicated trajectory for each molecule, the observable properties of the water in each cup are indistinguishable and are easy to describe. For example, the temperature of the water in each cup is independent of time even though the positions and velocities of the individual molecules are changing continually.

One way to understand the behavior of a classical many particle system is to simulate the trajectory of each particle. This approach, known as *molecular dynamics*, has been applied to systems of up to 10^9 particles and has given us much insight into a variety of systems in which the particles obey the laws of classical dynamics.

A calculation of the trajectories of many particles would not be very useful unless we know the right questions to ask. Saving these trajectories would quickly fill up any storage medium, and we do not usually care about the trajectory of any particular particle. What are the useful quantities needed to describe these many particle systems? What are the essential characteristics and regularities they exhibit? Questions such as these are addressed by statistical mechanics, and some of the ideas of statistical mechanics are discussed in this chapter. However, the only background needed for this chapter is a knowledge of Newton's laws of motion.

8.2 The Intermolecular Potential

The first step is to specify the model system we wish to simulate. We assume that the dynamics can be treated classically, the molecules are spherical and chemically inert and their internal structure can be ignored, and the interaction between any pair of particles depends only on the distance between them. In this case the total potential energy U is a sum of two-particle interactions:

$$U = u(r_{12}) + u(r_{13}) + \cdots + u(r_{23}) + \cdots = \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} u(r_{ij}) \tag{8.1}$$

where $u(r_{ij})$ depends only on the magnitude of the distance $\mathbf{r}_{ij}$ between particles i and j. The pairwise interaction form (8.1) is appropriate for simple liquids such as liquid argon.

The form of $u(r)$ for electrically neutral molecules can be constructed by a first principles quantum mechanical calculation. Such a calculation is very difficult, and it is usually sufficient to choose a simple phenomenological form for $u(r)$. The most important features of $u(r)$ are a strong repulsion for small r and a weak attraction at large r. The repulsion for small r is a consequence of the Pauli exclusion principle. That is, the electron wave functions of two molecules must distort to avoid overlap, causing some of the electrons to be in different quantum states. The net effect is an increase in kinetic energy and an effective repulsive interaction between the electrons, known as *core repulsion*. The dominant weak attraction at larger r is due to the mutual polarization of each molecule; the resultant attractive potential is called the *van der Waals* potential.

One of the most common phenomenological forms of $u(r)$ is the Lennard–Jones potential:

$$u(r) = 4\epsilon\Big[\Big(\frac{\sigma}{r}\Big)^{12} - \Big(\frac{\sigma}{r}\Big)^{6}\Big]. \tag{8.2}$$

A plot of the Lennard–Jones potential is shown in Figure 8.1. The r^{-12} form of the repulsive part of the interaction was chosen for convenience only and has no fundamental significance. The attractive $1/r^6$ behavior at large r corresponds to the van der Waals interaction.

The Lennard–Jones potential is parameterized by a length σ and an energy ϵ. Note that $u(r) = 0$ at $r = \sigma$, and that $u(r)$ is close to zero for $r > 2.5\sigma$. The parameter ϵ is the depth of the potential at the minimum of $u(r)$; the minimum occurs at a separation $r = 2^{1/6}\,\sigma$.

Problem 8.1. Qualitative properties of the Lennard–Jones interaction

Write a short program to plot the Lennard–Jones potential (8.1) and the magnitude of the corresponding force:

$$\mathbf{f}(r) = -\nabla u(r) = \frac{24\epsilon}{r}\Big[2\Big(\frac{\sigma}{r}\Big)^{12} - \Big(\frac{\sigma}{r}\Big)^{6}\Big]\hat{\mathbf{r}}. \tag{8.3}$$

At what value of r is the force equal to zero? For what values of r is the force repulsive? What is the value of $u(r)$ for $r = 0.8\sigma$? How much does u increase if r is decreased to $r = 0.72\sigma$, a 10% change in r? What is the value of u at $r = 2.5\sigma$? □

8.3 Units

As usual, it is convenient to choose units so that the computed quantities are neither too small nor too large. Because the values of the distance and the energy associated with typical liquids

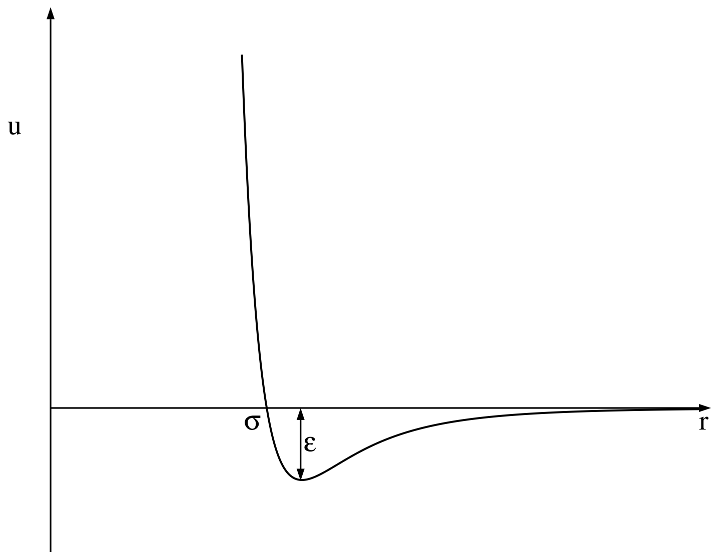


Figure 8.1: Plot of the Lennard–Jones potential $u(r)$. Note that the potential is characterized by a length σ and an energy ϵ.

are very small in SI units, we choose the Lennard–Jones parameters σ and ϵ as the units of distance and energy, respectively. We also choose the unit of mass to be the mass of one atom, m. We can express all other quantities in terms of σ, ϵ, and m. For example, we measure velocities in units of $(\epsilon/m)^{1/2}$, and the time in units of $\sigma(m/\epsilon)^{1/2}$. The values of σ, ϵ, and m for argon are given in Table 8.1. If we use these values, we find that the unit of time is 2.17×10^{-12} s. The units of some of the other physical quantities of interest are also shown in Table 8.1.

All program variables are in reduced units; for example, the time in our molecular dynamics program is expressed in units of $\sigma(m/\epsilon)^{1/2}$. Suppose that we run our molecular dynamics program for 2000 time steps with a time step $\Delta t = 0.01$. The total time of our run is $2000\times0.01 = 20$ in reduced units or 4.34×10^{-11} s for argon (see Table 8.1). The duration of a typical molecular dynamics simulation is in the range of 10–10^4 in reduced units, corresponding to a duration of approximately 10^{-11}–10^{-8} s. The longest practical runs are the order of 10^{-6} s.

8.4 The Numerical Algorithm

Now that we have specified the interaction between the particles, we need to introduce a numerical method for computing the trajectory of each particle. As we have learned, the criteria for a good numerical integration method include that it conserve the phase-space volume and is consistent with the known conservation laws, is time reversible, and is accurate for relatively large time steps to reduce the CPU time needed for the total time of the simulation. These requirements mean that we should use a symplectic algorithm for the relatively long times of interest in molecular dynamics simulations. We adopt the commonly used second-order algorithm:

$$x_{n+1} = x_n + v_n\Delta t + \frac{1}{2}a_n(\Delta t)^2 \tag{8.4a}$$

$$v_{n+1} = v_n + \frac{1}{2}(a_{n+1} + a_n)\Delta t. \tag{8.4b}$$

Quantity	**Unit**	**Value for Argon**
length	σ	3.4×10^{-10} m
energy	ϵ	1.65×10^{-21} J
mass	m	6.69×10^{-26} kg
time	$\sigma(m/\epsilon)^{1/2}$	2.17×10^{-12} s
velocity	$(\epsilon/m)^{1/2}$	1.57×10^{2} m/s
force	ϵ/σ	4.85×10^{-12} N
pressure	ϵ/σ^2	$1.43 \times 10^{-2}\,\mathrm{N \cdot m^{-1}}$
temperature	ϵ/k	120 K

Table 8.1: The system of units used in the molecular dynamics simulations of particles interacting via the Lennard–Jones potential. The numerical values of σ, ϵ, and m are for argon. The quantity k is Boltzmann's constant and has the value $k = 1.38 \times 10^{-23}$ J/K. The unit of pressure is for a two-dimensional system.

To simplify the notation, we have written the algorithm for only one component of the particle's motion. The new position is used to find the new acceleration a_{n+1}, which is used together with a_n to obtain the new velocity v_{n+1}. The algorithm represented by (8.4) is known as the Verlet (or sometimes the velocity Verlet) algorithm (see Appendix 3A). We will use the `Verlet` implementation of the `ODESolver` interface to implement the algorithm. Thus, the x, v_x, y, and v_y values for the ith particle will be stored in the `state` array at `state[4*i]`, `state[4*i+1]`, `state[4*i+2]`, and `state[4*i+3]`, respectively.

8.5 Periodic Boundary Conditions

A useful simulation must incorporate as many of the relevant features of the physical system of interest as possible. Usually we want to simulate a gas, liquid, or a solid in the bulk, that is, systems of at least $N \sim 10^{23}$ particles. In such systems the fraction of particles near the walls of the container is negligibly small. The number of particles that can be studied in a molecular dynamics simulation is typically 10^3–10^5, although we can simulate the order of 10^9 particles using clusters of computers. For these relatively small systems, the fraction of particles near the walls of the container is significant, and hence, the behavior of such a system would be dominated by surface effects.

The most common way of minimizing surface effects and to simulate more closely the properties of a bulk system is to use what are known as *periodic boundary conditions*, although the *minimum image approximation* would be a more accurate name. This boundary condition is familiar to anyone who has played the Pacman computer game. Consider N particles that are constrained to move on a line of length L. The application of periodic boundary conditions is equivalent to considering the line to be a circle, and hence, the maximum separation between any two particles is $L/2$ (see Figure 8.2). The generalization of periodic boundary conditions to two dimensions is equivalent to imagining a box with opposite edges joined so that the box becomes the surface of a torus (the shape of a doughnut or a bagel). The three-dimensional version of periodic boundary conditions cannot be visualized easily, but the same methods can be used.

The implementation of periodic boundary conditions is straightforward. If a particle leaves the box by crossing a boundary in a particular direction, we add or subtract the length L of the box in that direction to the position. One simple way is to use an `if else` statement as shown:

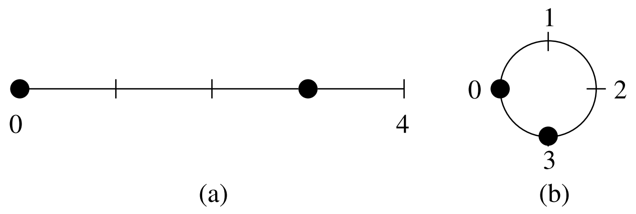


Figure 8.2: (a) Two particles at $x = 0$ and $x = 3$ on a line of length $L = 4$; the distance between the particles is 3. (b) The application of periodic boundary conditions for short range interactions is equivalent to thinking of the line as forming a circle of circumference L. In this case the minimum distance between the two particles is 1.

Listing 8.1: Calculation of the position of particle in the central cell.

```
private double pbcPosition(double s, double L) {
   if(s > L) {
      s -= L;
   } else if(s < 0) {
      s += L;
   }
   return s;
}
```

To compute the minimum distance `ds` in a particular direction between two particles, we can use the method `pbcSeparation` (see Figure 8.2):

Listing 8.2: Calculation of the minimum separation.

```
private double pbcSeparation(double ds, double L) {
   if(ds > 0.5*L) {
      ds -= L;
   } else if(ds < -0.5*L) {
      ds += L;
   }
   return ds;
}
```

The equivalent static methods, `PBC.position` and `PBC.separation` in the Open Source Physics `numerics` package can also be used.

Exercise 8.2. Use of the % operator

(a) Another way to compute the position of a particle in the central cell is to use the `%` (modulus) operator. For example, `17 % 5` equals 2 because 17 divided by 5 leaves a remainder of 2. The `%` operator can also be used with floating point numbers. For example, `10.2 % 5 = 0.2`. Write a little test program to see how the % function works and determine the result of `10.2 % 3.3`, `-10.2 % 3.3`, `10.2 % -3.3`, and `-10.2 % -3.3`. In what way does `%` act like a remainder operator?

(b) From the results of part (a) we might consider writing `x = x % L` as an alternative to Listing 8.1. What about negative values of x? In this case `-17 % 5 = -2`. Because we want the resultant position to be positive, we write

```
return x<0 ? x%L+L: x%L;
```

Explain this syntax and write a program to test if this statement works as claimed.

(c) Write a simple program to determine if the `%` operator is faster than the if-else construction in Listing 8.1. Write another program that compares the speed of calling the `PCB.position` method to that of *inlining* the PBC code. In other words, replace the method call by the above statement. □

We now discuss the nature of periodic boundary conditions. Imagine a set of N particles in a two-dimensional box or cell. The use of periodic boundary conditions implies that the central cell is duplicated an infinite number of times to fill the space. Figure 8.3 shows the first several image cells for $N = 2$. The shape of the central cell must be such that the cell fills space under successive translations. Each image cell contains the original particles in the same relative positions as the central cell. That is, periodic boundary conditions yield an infinite system, although the positions of the particles in the image cells are identical to the positions of the particles in the central cell. These boundary conditions also imply that every point in the cell is equivalent and that there is no surface.

As a particle moves in the original cell, its periodic images move in the image cells. Hence, only the motion of the particles in the central cell needs to be followed. When a particle enters or leaves the central cell, the move is accompanied by an image of that particle leaving or entering a neighboring cell through the opposite face.

The total force on a given particle i is due to the force from every other particle j within the central cell and from the periodic images of particle j. That is, if particle i interacts with particle j in the central cell, then particle i interacts with *all* the periodic replicas of particle j. Hence, in general, there are an infinite number of contributions to the force on any given particle. For long-range interactions such as the Coulomb potential, these contributions have to be included using special methods. For short-range interactions, we can reduce the number of contributions by adopting the minimum image approximation, which assumes that particle i in the central cell interacts only with the nearest image of particle j; the interaction is set equal to zero if the distance of the image from particle i is greater than $L/2$. An example of the minimum image approximation is shown in Figure 8.3.

8.6 A Molecular Dynamics Program

In this section we develop a molecular dynamics program to simulate a two-dimensional system of particles interacting via the Lennard–Jones potential. We choose two rather than three dimensions because it is easier to visualize the results and the calculations are not as time consuming.

In principle, we could define a class for a particle and instantiate an object for each particle. However, this use would be very inefficient and would take up more memory and CPU time than using one class to represent all N particles. Instead we will store the x- and y-components of the positions and velocities in the `state` array and store the accelerations of the particles in a separate array. As usual, we will develop two classes, `LJParticles` and `LJParticlesApp`.

Because the system is deterministic, the nature of the motion is determined by the initial conditions. An appropriate choice of the initial conditions is more difficult than might first appear. For example, how can we choose the initial configuration (a set of positions and velocities) to correspond to a liquid at a desired temperature? According to the equipartition theorem, the

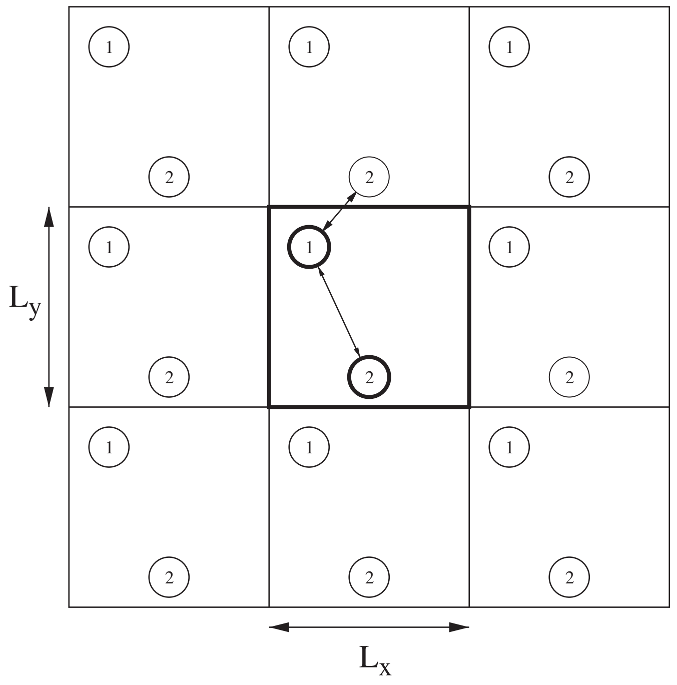


Figure 8.3: Example of the minimum image approximation in two dimensions. The minimum image distance convention implies that the separation between particles 1 and 2 is given by the lesser of the two distances shown.

mean kinetic energy of a particle per degree of freedom is $kT/2$, where k is Boltzmann's constant and T is the temperature. We can generalize this relation to define the temperature at time t by

$$kT(t) = \frac{2}{d}\frac{K(t)}{N} = \frac{1}{Nd}\sum_{i=1}^{N} m_i \mathbf{v}_i(t) \cdot \mathbf{v}_i(t) \tag{8.5}$$

where K is the total kinetic energy of the system, $\mathbf{v}_i$ is the velocity of particle i with mass m_i, and d is the spatial dimension of the system.

We can use (8.5) to choose an initial set of velocities. The following method gives the particles a random set of velocities, sets the total velocity (momentum) to zero, and then rescales the velocities so that the desired initial kinetic energy is achieved.

Listing 8.3: Method for choosing the initial velocities.

```
public void setVelocities() {
   double vxSum = 0.0;
   double vySum = 0.0;
   for(int i = 0;i<N;++i) { // assign random initial velocities
      state[4*i+1] = Math.random() - 0.5;    // vx
      state[4*i+3] = Math.random() - 0.5;    // vy
      vxSum += state[4*i+1];
      vySum += state[4*i+3];
   }
   // zero center of mass momentum
```

```
      double vxcm = vxSum/N;    // center of mass momentum (velocity)
      double vycm = vySum/N;
      for(int i = 0;i<N;++i) {
         state[4*i+1] -= vxcm;
         state[4*i+3] -= vycm;
      }
      // rescale velocities to get desired initial kinetic energy
      double v2sum = 0;
      for(int i = 0;i<N;++i) {
         v2sum += state[4*i+1]*state[4*i+1] + state[4*i+3]*state[4*i+3];
      }
      double kineticEnergyPerParticle = 0.5*v2sum/N;
      double rescale = Math.sqrt(initialKineticEnergy/kineticEnergyPerParticle);
      for(int i = 0;i<N;++i) {
         state[4*i+1] *= rescale;
         state[4*i+3] *= rescale;
      }
   }
```

We will find that setting the initial velocities so that the initial temperature is the desired value does not guarantee that the system will maintain this temperature when it reaches equilibrium. Determining an initial configuration that satisfies the desired conditions is an iterative process.

If the system is a dilute gas, we can choose the initial positions of the particles by placing them at random, making sure that no two particles are too close to one another. If two particles were too close, they would exert a very large repulsive force F on one another, and any simple finite difference integration method would break down because the condition $(F/m)(\Delta t)^2 \ll \sigma$ would not be satisfied. (In dimensionless units, this condition is $F(\Delta t)^2 \ll 1$.) If we assume that the separation between two particles is greater than $2^{1/6}\sigma$, this condition is satisfied. The following method places particles at random such that no two particles are closer than $2^{1/6}\sigma$. Note that we use a `do/while` statement to insure that the body of the loop is executed at least once.

Listing 8.4: Method for choosing the initial positions at random.

```
   public void setRandomPositions() {
      double rMinimumSquared = Math.pow(2.0, 1.0/3.0);
      boolean overlap;
      for(int i = 0;i<N;++i) {
         do {
            overlap = false;
            state[4*i] = Lx*Math.random();      // x
            state[4*i+2] = Ly*Math.random();    // y
            int j = 0;
            while(j<i&&!overlap) {
               double dx = state[4*i]-state[4*j];
               double dy = state[4*i+2]-state[4*j+2];
               if(dx*dx+dy*dy<rMinimumSquared) {
                  overlap = true;
               }
               j++;
            }
         } while(overlap);
      }
   }
```

What is the maximum density that you can reasonably obtain in this way?

Finding a random configuration of particles in which no two particles are closer than $2^{1/6}\sigma$ becomes much too inefficient if the system is dense. It is possible to choose the initial positions randomly without regard to their separations if we include a fictitious drag force proportional to the square of the velocity. The effect of such a force is to dampen the velocity of those particles whose velocities become too large due to the large repulsive forces exerted on them. We then would have to run for a while until all the velocities satisfy the condition $v\Delta t \ll 1$. As the velocities become smaller, we may gradually reduce the friction coefficient.

In general, the easiest way of obtaining an initial configuration with the desired density is to place the particles on a regular lattice. If the temperature is high or if the system is dilute, the system will "melt" and become a liquid or a gas; otherwise, it will remain a solid. If our goal is to equilibrate the system at fluid densities, it is not necessary to choose the correct equilibrium symmetry of the lattice. The method `setRectangularLattice` in Listing 8.5 places the particles on a rectangular lattice. To make the method simple, the user must specify the number of particles per row `nx` and the number per column `ny`. The linear dimensions `Lx` and `Ly` are adjustable parameters and can be varied after initialization to be as close as possible to their desired values. In this way we can vary the density by varying the volume (area) without setting up a new initial configuration.

Listing 8.5: Placement of particles on a rectangular lattice.

```
// place particles on a rectangular lattice
public void setRectangularLattice() {
   double dx = Lx/nx; // distance between columns
   double dy = Ly/ny; // distance between rows
   for(int ix = 0; ix<nx; ++ix) { // loop through particles in a row
      for(int iy = 0; iy<ny; ++iy) { // loop through rows
         int i = ix + iy*ny;
         state[4*i] = dx*(ix+0.5);
         state[4*i+2] = dy*(iy+0.5);
      }
   }
 }
```

The most time consuming part of a molecular dynamics simulation is the computation of the accelerations of the particles. The method `computeAcceleration` determines the total force on each particle due to the other $N-1$ particles and uses Newton's third law to reduce the number of calculations by a factor of two. Hence, for a system of N particles, there are a total of $N(N-1)/2$ possible interactions. Because of the short range nature of the Lennard–Jones potential, we could truncate the force at $r = r_c$ and ignore the forces from particles whose separation is greater than r_c. However, for $N \lesssim 400$, it is easier to include all possible interactions, no matter how small. The quantity `virial` accumulated in `computeAcceleration` is discussed in Section 8.7, where we will see that it is related to the pressure. It is convenient to also calculate the potential energy in `computeAcceleration`. Note that in reduced units, the mass of a particle is unity, and hence, the acceleration and force are equivalent.

Listing 8.6: Calculation of the acceleration.

```
public void computeAcceleration() {
   for(int i = 0;i<N;i++) {
      ax[i] = 0;
      ay[i] = 0;
   }
```

```
      for(int i = 0;i<N-1;i++) {
         for(int j = i+1;j<N;j++) {
            double dx = pbcSeparation(state[4*i]-state[4*j], Lx);
            double dy = pbcSeparation(state[4*i+2]-state[4*j+2], Ly);
            double r2 = dx*dx+dy*dy;
            double oneOverR2 = 1.0/r2;
            double oneOverR6 = oneOverR2*oneOverR2*oneOverR2;
            double fOverR = 48.0*oneOverR6*(oneOverR6-0.5)*oneOverR2;
            double fx = fOverR*dx;
            double fy = fOverR*dy;
            ax[i] += fx;
            ay[i] += fy;
            ax[j] -= fx;
            ay[j] -= fy;
            totalPotentialEnergyAccumulator += 4.0*(oneOverR6*oneOverR6-oneOverR6);
            virialAccumulator += dx*fx+dy*fy;
         }
      }
   }
```

The methods needed for the ODE interface are given in Listing 8.7. Note that the `getRate` method is invoked twice for every call to the `step` method because we are using the Verlet algorithm. The first rate call uses the current positions of the particles, and the second rate call uses the new positions. Because a particle's new position becomes its current position for the next step, we would compute the same accelerations twice. To avoid this inefficiency, we query the ODE solver using the `getRateCounter` method to determine if the position or the velocity is being computed. We store the accelerations in an array during the second computation so that we use these values the next time `getRate` is invoked. This trick is not general and should only be used if you understand exactly how the particular ODE solver behaves. Study the implementation of the `step` method in the `Verlet` class.

Listing 8.7: Methods needed for the `ODE` interface.

```
public void getRate(double[] state, double[] rate) {
   // getRate is called twice for each call to step
   // accelerations computed for every other call to getRate because
   // new velocity is computed from previous and current acceleration
   // Previous acceleration is saved in step method of Verlet
   if(odeSolver.getRateCounter()==1) {
      computeAcceleration();
   }
   for(int i = 0; i<N; i++) {
      rate[4*i] = state[4*i+1];   // rates for positions are velocities
      rate[4*i+2] = state[4*i+3]; // vy
      rate[4*i+1] = ax[i];        // rate for velocity is acceleration
      rate[4*i+3] = ay[i];
   }
   rate[4*N] = 1; // dt/dt = 1
}

public double[] getState() {
   return state;
}
```

```
public void step(HistogramFrame xVelocityHistogram) {
   odeSolver.step();
   double totalKineticEnergy = 0;
   for(int i = 0; i<N; i++) {
      totalKineticEnergy += (state[4*i+1]*state[4*i+1]+state[4*i+3]*state[4*i+3]);
      xVelocityHistogram.append(state[4*i+1]);
      state[4*i] = pbcPosition(state[4*i], Lx);
      state[4*i+2] = pbcPosition(state[4*i+2], Ly);
   }
   totalKineticEnergy *= 0.5;
   steps++;
   totalKineticEnergyAccumulator += totalKineticEnergy;
   totalKineticEnergySquaredAccumulator += totalKineticEnergy*totalKineticEnergy;
   t += dt;
}
```

Note that we accumulate data for the histogram of the x-component of the velocity in the `step` method.

Alternatively, we can implement the Verlet algorithm without the ODE interface. In the following we show the code that would replace the call to the `step` method of the ODE solver. We have used different array names for clarity. This code uses about the same amount of CPU time as the code using the ODE solver.

```
for (int i = 0;i<N;i++) { // use old acceleration
   x[i] += vx[i]*dt + ax[i]*halfdt2;     // halfdt2 = 0.5*dt*dt
   y[i] += vy[i]*dt + ay[i]*halfdt2;
   vx[i] += ax[i]*halfdt;   // add old acceleration, halfdt = 0.5*dt
   vy[i] += ay[i]*halfdt;
   }
   // computes velocity in two steps using old and new acceleration
   computeAcceleration();
   for (int i = 0;i<N;i++) { // add new acceleration
      vx[i] += ax[i]*halfdt;
      vy[i] += ay[i]*halfdt;
   }
```

In Listing 8.8 we give some of the methods for computing the temperature, pressure (see (8.8)), and the heat capacity (see (8.12)). The mean total energy should remain constant, but we compute it to test how well the algorithm conserves the total energy.

Listing 8.8: Methods used to compute averages.

```
public double getMeanTemperature() {
   return totalKineticEnergyAccumulator/(N*steps);
}

public double getMeanEnergy() {
   return totalKineticEnergyAccumulator/steps+totalPotentialEnergyAccumulator/steps;
}

public double getMeanPressure() {
   double meanVirial;
   meanVirial = virialAccumulator/steps;
   // quantity PA/NkT
```

```
      return 1.0+0.5*meanVirial/(N*getMeanTemperature());
   }

   public double getHeatCapacity() {
      double meanTemperature = getMeanTemperature();
      double meanTemperatureSquared = totalKineticEnergySquaredAccumulator/steps;
      // heat capacity related to fluctuations of kinetic energy
      double sigma2 = meanTemperatureSquared-meanTemperature*meanTemperature;
      double denom = sigma2/(N*meanTemperature*meanTemperature) - 1.0;
      return N/denom;
   }

   public void resetAverages() {
      steps = 0;
      virialAccumulator = 0;
      totalPotentialEnergyAccumulator = 0;
      totalKineticEnergyAccumulator = 0;
      totalKineticEnergySquaredAccumulator = 0;
   }
```

The `resetAverages` method is used to set the accumulated averages to zero so that the initial transient behavior can be removed from the computed averages.

We use the `Drawable` interface to display the trajectories of the individual particles.

Listing 8.9: The `draw` method.

```
   public void draw(DrawingPanel panel, Graphics g) {
      if(state==null) {
         return;
      }
      int pxRadius = Math.abs(panel.xToPix(radius)-panel.xToPix(0));
      int pyRadius = Math.abs(panel.yToPix(radius)-panel.yToPix(0));
      g.setColor(Color.red);
      for(int i = 0;i<N;i++) {
         int xpix = panel.xToPix(state[4*i])-pxRadius;
         int ypix = panel.yToPix(state[4*i+2])-pyRadius;
         g.fillOval(xpix, ypix, 2*pxRadius, 2*pyRadius);
      }  // draw central cell boundary
      g.setColor(Color.black);
      int xpix = panel.xToPix(0);
      int ypix = panel.yToPix(Ly);
      int lx = panel.xToPix(Lx)-panel.xToPix(0);
      int ly = panel.yToPix(0)-panel.yToPix(Ly);
      g.drawRect(xpix, ypix, lx, ly);
   }
```

The beginning of the `LJParticles` class includes the usual `import` statements, instance variables, and the `initialize` method. If you include all of the code we have discussed in a single file, you will have a working `LJparticles` class. Alternatively, you can download the source code from the ch08 directory.

Listing 8.10: Beginning of class `LJParticles`.

```
   package org.opensourcephysics.sip.ch08.md;
   import java.awt.*;
```

```
import org.opensourcephysics.display.*;
import org.opensourcephysics.frames.HistogramFrame;
import org.opensourcephysics.numerics.*;

public class LJParticles implements Drawable, ODE {
   public double state[];
   public double ax[], ay[];
   // number of particles, number per row, number per column
   public int N, nx, ny;
   public double Lx, Ly;
   public double rho = N/(Lx*Ly);
   public double initialKineticEnergy;
   public int steps = 0;
   public double dt = 0.01;
   public double t;
   public double totalPotentialEnergyAccumulator;
   public double totalKineticEnergyAccumulator
   public double totalKineticEnergySquaredAccumulator;
   public double virialAccumulator;
   public String initialConfiguration;
   public double radius = 0.5;   // radius of particles on screen
   ODESolver ode_solver = new Verlet(this);

   public void initialize() {
      N = nx*ny;
      t = 0;
      rho = N/(Lx*Ly);
      resetAverages();
      state = new double[1+4*N];
      ax = new double[N];
      ay = new double[N];
      if(initialConfiguration.equals("triangular")) {
         setTriangularLattice();
      } else if(initialConfiguration.equals("rectangular")) {
         setRectangularLattice();
      } else {
         setRandomPositions();
      }
      setVelocities();
      computeAcceleration();
      ode_solver.setStepSize(dt);
   }
```

The target class is given in Listing 8.11. When the user presses the Stop button, various thermodynamic averages are displayed in the message area of the control window. As you will find in Problem 8.8, a time consuming part of a molecular dynamics simulation is equilibrating the system, especially at high densities. The quickest way to do so is to start with a configuration that is typical of the configurations at the desired energy and density. Hence, we will want to be able to read the positions and velocities of the particles from a previously saved file. The class `LJParticlesLoader` allows us to save a configuration. This class is used in the `getLoader` method in class `LJParticlesApp`. To save a given configuration, open the File menu in the control window and choose Save As... . A dialog box will open so that you can choose a name for the file, which will have the extension `xml`. To read a previously saved configuration, choose

Read in the File menu. Notice that in `LJParticlesLoader`, the `loadObject` makes a call to the `computeAcceleration` and `resetAverages` methods of `LJParticles`, so that the simulation can be run starting from the newly loaded configuration by next clicking the Start button. The syntax for saving a model's configuration is described in more detail in Appendix 8A.

Listing 8.11: The `LJParticlesApp` target class.

```
package org.opensourcephysics.sip.ch08.md;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.frames.*;
import org.opensourcephysics.display.GUIUtils;

public class LJParticlesApp extends AbstractSimulation {
   LJParticles md = new LJParticles();
   PlotFrame pressureData = new PlotFrame("time", "PA/NkT",
                                 "Mean pressure");
   PlotFrame temperatureData = new PlotFrame("time", "temperature",
                                    "Mean temperature");
   HistogramFrame xVelocityHistogram = new HistogramFrame("vx", "H(vx)",
                                             "Velocity histogram");
   DisplayFrame display = new DisplayFrame("x", "y",
                              "Lennard-Jones system");

   public void initialize() {
      md.nx = control.getInt("nx"); // number of particles per row
      md.ny = control.getInt("ny"); // number of particles per column
      md.initialKineticEnergy = control.getDouble(
         "initial kinetic energy per particle");
      md.Lx = control.getDouble("Lx");
      md.Ly = control.getDouble("Ly");
      md.initialConfiguration = control.getString("initial configuration");
      md.dt = control.getDouble("dt");
      md.initialize();
      display.addDrawable(md);
      // assumes vmax = 2*initalTemp and bin width = Vmax/N
      display.setPreferredMinMax(0, md.Lx, 0, md.Ly);
      xVelocityHistogram.setBinWidth(2*md.initialKineticEnergy/md.N);
   }

   public void doStep() {
      md.step(xVelocityHistogram);
      pressureData.append(0, md.t, md.getMeanPressure());
      temperatureData.append(0, md.t, md.getMeanTemperature());
   }

   public void stop() {
      control.println("Density = "+decimalFormat.format(md.rho));
      control.println("Number of time steps = "+md.steps);
      control.println("Time step dt = "+decimalFormat.format(md.dt));
      control.println("<T>= "
                      +decimalFormat.format(md.getMeanTemperature()));
      control.println("<E> = "+decimalFormat.format(md.getMeanEnergy()));
      control.println("Heat capacity = "
                      +decimalFormat.format(md.getHeatCapacity()));
      control.println("<PA/NkT> = "
                     +decimalFormat.format(md.getMeanPressure()));
   }

   public void startRunning() {
      md.dt = control.getDouble("dt");
      double Lx = control.getDouble("Lx");
      double Ly = control.getDouble("Ly");
      if((Lx!=md.Lx)||(Ly!=md.Ly)) {
         md.Lx = Lx;
         md.Ly = Ly;
         md.computeAcceleration();
         display.setPreferredMinMax(0, Lx, 0, Ly);
         resetData();
      }
   }

   public void reset() {
      control.setValue("nx", 8);
      control.setValue("ny", 8);
      control.setAdjustableValue("Lx", 20.0);
      control.setAdjustableValue("Ly", 15.0);
      control.setValue("initial kinetic energy per particle", 1.0);
      control.setAdjustableValue("dt", 0.01);
      control.setValue("initial configuration", "rectangular");
      enableStepsPerDisplay(true);
      // draw configurations every 10 steps
      super.setStepsPerDisplay(10);
      // so particles will appear as circular disks
      display.setSquareAspect(true);
   }

   public void resetData() {
      md.resetAverages();
      // clears old data from the plot frames
      GUIUtils.clearDrawingFrameData(false);
   }

   public static XML.ObjectLoader getLoader() {
      return new LJParticlesLoader();
   }

   public static void main(String[] args) {
      SimulationControl control = SimulationControl.createApp(
                                    new LJParticlesApp());
      control.addButton("resetData", "Reset Data");
   }
}
```

Listing 8.12: The LJParticlesLoader class for saving configurations.

```
package org.opensourcephysics.sip.ch08.md;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.display.GUIUtils;
```

```
public class LJParticlesLoader implements XML.ObjectLoader {
   public Object createObject(XMLControl element) {
      return new LJParticlesApp();
   }

   public void saveObject(XMLControl control, Object obj) {
      LJParticlesApp model = (LJParticlesApp) obj;
      control.setValue("initial_configuration",
                       model.md.initialConfiguration);
      control.setValue("state", model.md.state);
   }

   public Object loadObject(XMLControl control, Object obj) {
      // GUI has been loaded with the saved values; now restore the LJ state
      LJParticlesApp model = (LJParticlesApp) obj;
      // reads values from the GUI into the LJ model
      model.initialize();
      model.md.initialConfiguration = control.getString(
         "initial_configuration");
      model.md.state = (double[]) control.getObject("state");
      int N = (model.md.state.length-1)/4;
      model.md.ax = new double[N];
      model.md.ay = new double[N];
      model.md.computeAcceleration();
      model.md.resetAverages();
      // clears old data from the plot frames
      GUIUtils.clearDrawingFrameData(false);
      return obj;
   }
}
```

Problem 8.3. Approach to equilibrium

(a) Consider $N = 64$ particles interacting via the Lennard–Jones potential in a square central cell of linear dimension $L = 10$. Start the system on a square lattice with an initial temperature corresponding to $T = 1.0$. Let $\Delta t = 0.01$ and run the application to make sure that it is working properly. The total energy should be approximately conserved and the trajectories of all 64 particles should be seen on the screen.

(b) The kinetic temperature of the system is given by (8.5). View the evolution of the temperature of the system starting from the initial temperature. Does the temperature reach an equilibrium value? That is, does it eventually fluctuate about some mean value? What is the mean value of the temperature for the given total energy of the system?

(c) Modify method `setRectangularLattice` so that all the particles are initially on the left side of a box of linear dimensions $L_x = 20$ and $L_y = 10$. Does the system become more or less random as time increases?

(d) Modify the program so it computes $n(t)$, the number of particles in the left half of the cell, and plot $n(t)$ as a function of t. What is the qualitative behavior of $n(t)$? What is the mean number of particles on the left half after the system has reached equilibrium? Compare your qualitative results with the results you found in Problem 7.2. □

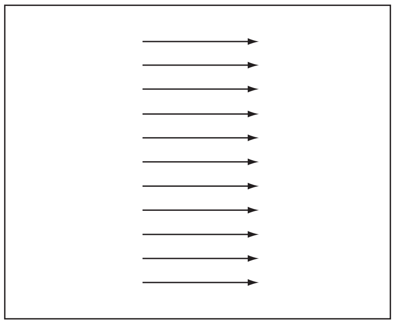


Figure 8.4: Example of a special initial condition; the arrows represent the magnitude and the direction of each particle's velocity.

Problem 8.4. Sensitivity to initial conditions

(a) Modify your program to consider the following initial condition corresponding to $N = 11$ particles moving in the same direction with the same velocity (see Figure 8.4). Choose $L_x = L_y = 10$ and $\Delta t = 0.01$.

```
for (int i = 0; i < N; i++) {
   x[i] = Lx/2;
   y[i] = (i - 0.5)*Ly/N;
   vx[i] = 1;
   vy[i] = 0;
}
```

Does the system eventually reach equilibrium? Why or why not?

(b) Change the velocity of particle 6 so that $v_x(6) = 0.99999$ and $v_y(6) = 0.00001$. Is the behavior of the system qualitatively different than in part (a)? Does the system eventually reach equilibrium? Are the trajectories of the particles sensitive to the initial conditions? Explain why this behavior implies that almost all initial states lead to the same qualitative behavior (for a given total energy).

(c) Modify LJParticlesApp so that the application runs for a predetermined time interval, such as 100 time steps, and then continues with the time reversed process, that is, the motion that would occur if the direction of time was reversed. This reversal is equivalent to letting $\mathbf{v} \to -\mathbf{v}$ for all particles or letting $\Delta t \to -\Delta t$. Do the particles return to their original positions? What happens if you reverse the velocities at a later time? What happens if you choose a smaller value of Δt?

(d) Explain why you can conclude that the system is chaotic. Are the computed trajectories the same as the "true" trajectories? □

From Problems 8.3 and 8.4, we see that from the *microscopic* point of view, the trajectories appear rather complex. In contrast, from the *macroscopic* point of view, the system can be described more simply. For example, in Problem 8.3 we described the approach of the system to equilibrium by specifying $n(t)$, the number of particles in the left half of the cell at time t. Your observations of the macroscopic variable $n(t)$ should be consistent with the following two general properties of systems of many particles:

1. After the removal of an internal constraint, an isolated system changes in time from a "less random" to a "more random" state.
2. A system whose macroscopic state is independent of time is said to be in *equilibrium*. The equilibrium macroscopic state is characterized by relatively small fluctuations about a mean that is independent of time. The relative fluctuations become smaller as the number of particles becomes larger.

In Problems 8.3b and 8.3c we found that the particles filled the box and did not return to their initial configuration. Hence, we were able to define a direction of time. This direction becomes better defined if we consider more particles. Note that Newton's laws of motion are time reversible, and there is no a priori reason that gives the time a preferred direction.

Before we consider other macroscopic quantities, we need to monitor the total energy and verify our claim that the Verlet algorithm maintains conservation of energy with a reasonable choice of Δt. We also introduce a check for momentum conservation.

Problem 8.5. Tests of the Verlet algorithm

(a) One essential check of a molecular dynamics program is that the total energy be conserved to the desired accuracy. Determine the value of Δt necessary for the total energy to be conserved to a given accuracy over a time interval of $t = 2$. One way is to compute $\Delta E_{\max}(t)$, the maximum value of the difference, $|E(t) - E(0)|$, over the time interval t, where $E(0)$ is the initial total energy, and $E(t)$ is the total energy at time t. Verify that $\Delta E_{\max}(t)$ decreases when Δt is made smaller for fixed t. If your application is working properly, $\Delta E_{\max}(t)$ should decrease as approximately $(\Delta t)^2$ because the Verlet algorithm is a second-order algorithm.

(b) A simple way of monitoring how well the program is conserving the total energy is to use a least squares fit of the times series of $E(t)$ to a straight line. The slope of the line can be interpreted as the drift, and the root mean square deviation from the straight line can be interpreted as the noise (σ_y in the notation of Section 7.6). How do the drift and the noise depend on Δt for a fixed time interval t? Most research applications conserve the energy to 1 part in 10^4 or better over the duration of the run.

(c) Because of the use of periodic boundary conditions, all points in the central cell are equivalent and the system is translationally invariant. As you might have learned, translational invariance implies that the total linear momentum is conserved. However, floating point error and the truncation error associated with a finite difference algorithm can cause the total linear momentum to drift. Programming errors might also be detected by checking for conservation of momentum. Hence, it is a good idea to monitor the total linear momentum at regular intervals and reset the total momentum equal to zero if necessary. The method `setVelocities` in Listing 8.3 chooses the velocities so that the total momentum is initially zero. Add a method that resets the total momentum to zero and call it at regular intervals, for example, every 1000–10,000 time steps. How well does class `LJParticles` conserve the total linear momentum for $\Delta t = 0.01$? □

8.7 Thermodynamic Quantities

In the following, we discuss how some of the macroscopic quantities of interest, such as the temperature and the pressure, can be related to time averages over the phase space trajectories of the particles.

We have already introduced the definition of the kinetic temperature in (8.5). The temperature that we measure in a laboratory experiment is the *mean* temperature, which corresponds to the time average of $T(t)$ over many configurations of the particles. For two dimensions ($d = 2$), we write the mean temperature T as

$$kT = \frac{1}{2N}\sum_{i=1}^{N} m_i \overline{\mathbf{v}_i(t)\cdot\mathbf{v}_i(t)} \qquad \text{(two dimensions)} \tag{8.6}$$

where $\overline{X}$ denotes the time average of $X(t)$. The relation (8.6) is an example of the relation of a macroscopic quantity (the mean temperature) to a time average over the trajectories of the particles. (This definition of temperature is not adequate for particles moving relativistically, or if quantum mechanics is important.)

The relation (8.5) holds only if the momentum of the center of mass of the system is zero—we do not want the motion of the center of mass to change the temperature. In a laboratory system, the walls of the container ensure that the center of mass motion is zero (if the mean momentum of the walls is zero). In our simulation, we impose the constraint that the center of mass momentum (in each of the d directions) be zero. Consequently, the system has $dN - d$ independent velocity components rather than dN components, and we should replace (8.6) by

$$kT = \frac{1}{(N-1)d}\sum_{i=1}^{N} m_i \overline{\mathbf{v}_i(t)\cdot\mathbf{v}_i(t)} \qquad \text{(correction for fixed center of mass).} \tag{8.7}$$

The presence of the factor $(N-1)d$ rather than Nd in (8.7) is an example of a *finite size* correction that becomes unimportant for large N. We shall ignore this correction in the following.

Another macroscopic quantity of interest is the mean pressure. The pressure is related to the force per unit area normal to an imaginary surface in the system. By Newton's second law, this force is related to the momentum that crosses the surface per unit time. We could use this relation to determine the pressure, but this relation uses information only from the fraction of particles that are crossing an arbitrary surface at a given time. Instead, we will use the relation of the pressure to the *virial*, which involves all the particles in the system.

In general, the momentum flux across a surface has two contributions. The contribution, NkT/V, where V is the volume (area) of the system, is due to the motion of the particles and is derived in many texts using simple kinetic theory arguments.

The other contribution to the momentum flux arises from the momentum transferred across the surface due to the forces between particles on different sides of the surface. It can be shown that the instantaneous pressure at time t, including both contributions to the momentum flux, is given by

$$P(t)V = NkT(t) + \frac{1}{d}\sum_{i<j}\mathbf{r}_{ij}(t)\cdot\mathbf{F}_{ij}(t) \tag{8.8}$$

where $\mathbf{r}_{ij} = \mathbf{r}_i - \mathbf{r}_j$ and $\mathbf{F}_{ij}$ is the force on particle i due to particle j. The second term in (8.8) is related to the virial and represents the correction to the ideal gas equation of state due to the interactions between the particles. (In two and one dimensions, we replace V by the area and length, respectively.)

The mean pressure, $P = \overline{P(t)}$, is found by computing the time average of the right-hand side of (8.8). The computed quantity is not P, but the ratio

$$\frac{PV}{NkT} - 1 = \frac{1}{dNkT}\sum_{i<j}\overline{\mathbf{r}_{ij}\cdot\mathbf{F}_{ij}}. \tag{8.9}$$

In class `LJParticles`, the sum on the right-hand side of (8.9) is computed in `computeAcceleration` and stored in the variable `virialAccumulator`.

The relation of information at the microscopic level to macroscopic quantities such as the temperature and pressure is one of the fundamental results of statistical mechanics. In brief, molecular dynamics allows us to compute various time averages of the trajectory in phase space over finite time intervals. One practical question is whether our time intervals are sufficiently long enough to allow the system to explore phase space and yield meaningful averages. Calculations in statistical mechanics are done by replacing time averages by *ensemble* averages over all possible configurations. The *quasi-ergodic* hypothesis asserts that these two types of averages give equivalent results if the same quantities are held fixed. In statistical mechanics, the ensemble of systems at fixed E, V and N is called the microcanonical ensemble. Averages in this ensemble correspond to the time averages that we use in molecular dynamics which are at fixed E, V and N. (Molecular dynamics also imposes an additional, but unimportant, constraint on the center of mass motion.) Ensemble averages are explored using Monte Carlo methods in Chapter 15. A test for determining if a molecular dynamics simulation is exploring a reasonable amount of phase space is discussed in Project 8.23.

The goal of the following problems is to explore some of the qualitative features of gases, liquids, and solids. Because we will consider relatively small systems and relatively short runs, our results will only be qualitatively consistent with averages calculated in the thermodynamic limit where $N \to \infty$.

Problem 8.6. Distribution of speeds and velocities

(a) In Section 7.2 we discussed how to use the `HistogramFrame` class from the Open Source Physics library. `LJParticlesApp` uses this class to compute the probability $P(v_x)\Delta v_x$ that a particle has a velocity in the x-direction between v_x and $v_x + \Delta v_x$. Add code to determine $P(v_y)$, the probability density for the y component of the velocity. What are the most probable values for the x and y velocity components? What are their average values? Plot the probability densities $P(v_x)$ versus v_x and $P(v_y)$ versus v_y. Better results can be found by plotting the average $\frac{1}{2}[P(v_x = u) + P(v_y = u)]$ versus u. What is the qualitative form of $P(\mathbf{v})$?

(b) Write a method to compute the equilibrium probability $P(v)\Delta v$ that a particle has a speed between v and $v + \Delta v$. What is the qualitative form of the probability density $P(v)$? Does it have the same qualitative form as $P(\mathbf{v})$, the probability density for the velocity? What is the most probable value of v? What is the approximate width of $P(v)$? Compare your measured result to the theoretical form (in two dimensions):

$$P(v)\,dv = Ae^{-mv^2/2kT}\,v\,dv \tag{8.10}$$

where A is a normalization constant. The form (8.10) of the distribution of speeds is known as the Maxwell–Boltzmann probability distribution.

(c) Repeat part (b) for different densities and temperatures. Does the form of $P(v)$ depend on the density or temperature? □

Problem 8.7. Qualitative properties of a liquid and a gas

(a) Generate an initial configuration using `setRectangularLattice` with $N = 64$ and $L_x = L_y = 12$ and an initial temperature of 2.0. What is the density? Modify your program so that the values of the temperature and pressure are not stored until the system has reached equilibrium. One criterion for equilibrium is to compute the average values of T and P over finite time intervals and check that these averages do not drift with time.

(b) Choose a value of the time step Δt so that the total energy is conserved to the desired accuracy and run the simulation for a sufficient time to estimate the equilibrium pressure and temperature. Compare your estimate for the ratio PV/NkT with its value for an ideal gas. (We have written V for the area of the system, so that the ideal gas equation of state has a familiar form.) Save the final configuration of your simulation in a file (see Appendix 8A).

(c) One way of starting a simulation is to use the positions saved from an earlier run. The simplest way of obtaining an initial condition corresponding to a different density, but the same value of N, is to rescale the positions of the particles and the linear dimensions of the cell. The following code shows one way to do so.

```
for (int i = 0; i < N; i++) {
   x[i] *= rescale;  // add rescale as a class variable
   y[i] *= rescale;
}
Lx = rescale*Lx
Ly = rescale*Ly
```

Incorporate this code into your program in a separate method and add a button that lets the user call this method without initialization. This method must be used with care when increasing the density. If the density is increased too quickly, it is likely that two particles will become very close to each other, so that the force will become too large and the numerical algorithm will break down. Allow the system to equilibrate after making a small density change and then repeat until you reach the desired density. How do you expect P and T to change when the system is compressed? Gradually increase the density and determine how PV/NkT changes with increasing density. Can you distinguish the different phases? (The determination of the phase boundary between a gas, liquid, and a solid is nontrivial and is discussed in Problem 15.26.) □

Another useful thermal quantity is the *heat capacity* at constant volume, which is defined by the relation $C_V = (\partial E/\partial T)_V$. (The subscript V denotes that the partial derivative is taken with the volume held fixed.) C_V is an example of a linear response function, that is, the response of the temperature to a change in the energy of the system. One way to obtain C_V is to determine $T(E)$, the temperature as a function of E. (Remember that a molecular dynamics simulation yields T as a function of E.) The heat capacity is approximately given by $\Delta E/\Delta T$ for two runs that have slightly different temperatures. This method is straightforward, but requires that simulations at different energies be done. An alternative way of determining C_V from the fluctuations of the kinetic energy is discussed in Problem 8.8c.

Problem 8.8. Energy dependence of the temperature and pressure

(a) We found in Problem 8.7 that the total energy is determined by the initial conditions, and the temperature is a derived quantity found only after the system has reached thermal equilibrium. For this reason it is difficult to study the system at a particular temperature. The temperature can be changed to the desired value by rescaling the velocities of the system, but we have to be careful not to increase the velocities too quickly. Run your program to create an equilibrium configuration for $L_x = L_y = 12$ and $N = 64$ and determine $T(E)$, the energy dependence of mean temperature, in the range $T = 1.0$ to $T = 1.2$. Rescale the velocities by the desired amount over some time interval. For example, multiply all the velocities by a factor λ after each time step for a certain number of time steps. In general, the desired temperature is reached by a series of velocity rescalings over a sufficiently long time such that the system remains close to equilibrium during the rescaling.

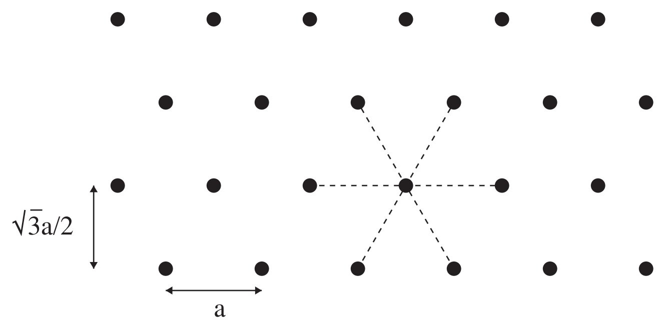


Figure 8.5: Each particle has six nearest neighbors in a triangular lattice.

(b) Use your data for $T(E)$ found in part (a) to plot the total energy E as a function of T. Is T a monotonically increasing function of E? What percentage of the contribution to the heat capacity is due to the potential energy? Why is an accurate determination of C_V difficult to achieve?

(c)* In Chapter 15 we will find that C_V is related to the fluctuations of the total energy in the canonical ensemble in which T, V, N are held fixed. In molecular dynamics simulations, the total energy is fixed, but the kinetic and potential energies can fluctuate. Another way of determining C_V is to relate it to the fluctuations of the kinetic energy. It can be shown that (cf. Ray and Graben)

$$\overline{T^2} - \overline{T}^2 = \frac{d}{2N}(k\overline{T})^2\Big[1 - \frac{dNk}{2C_V}\Big] \tag{8.11}$$

or

$$C_V = \frac{dNk}{2}\Big[1 - \frac{2N}{d}\frac{(\overline{T^2} - \overline{T}^2)}{(k\overline{T})^2}\Big]^{-1}. \tag{8.12}$$

The relation (8.12) reduces to the ideal gas result if $\overline{T^2} = \overline{T}^2$. Method `getHeatCapacity` determines C_V from (8.12). Compare your results obtained using (8.12) with the determination of C_V in part (b). What are the advantages and disadvantages of determining C_V from the fluctuations of the temperature compared to the method used in part (b)? □

Problem 8.9. Ground state energy of two-dimensional lattices

To simulate a solid, we need to choose the shape of the central cell to be consistent with the symmetry of the solid phase of the system. This choice is necessary even though we have used periodic boundary conditions to minimize surface effects. If the cell does not correspond to the correct crystal structure, the particles cannot form a perfect crystal, and some of the particles will wander around in an endless search for their "correct" positions. Consequently, a simulation of a small system at a high density and low temperature would lead to spurious results. In the following, we compute the energy of a Lennard–Jones solid in two dimensions for the square and triangular lattices and determine which symmetry has lower energy.

(a) The symmetry of the triangular lattice can be seen from Figure 8.5. Each particle has six nearest neighbors. Although it is possible to choose the central cell of the triangular lattice to be a rhombus, it is more convenient to choose the cell to be rectangular as in Figure 8.5. For a perfect crystal, the linear dimensions of the cell are L_x and $L_y = \sqrt{3}L_x/2$, respectively. Use method setTriangularLattice in Listing 8.13 to generate the positions of the particles in a triangular lattice. Then compute the potential energy per particle of a system of $N = 64$ particles interacting via the Lennard–Jones potential. Determine the potential energy for $L_x = 8$ and $L_x = 9$.

(b) Determine the potential energy for a square lattice with $L = \sqrt{L_x L_y}$, so that the triangular and square lattices have the same density. Which lattice symmetry has a lower potential energy for a given density? Explain your results in terms of the ability of the triangular lattice to pack the particles closer together. □

Listing 8.13: Method for generating a triangular lattice.

```java
public void setTriangularLattice() {
   double dx = Lx/nx;  // distance between particles on same row
   double dy = Ly/ny;  // distance between rows
   for(int ix = 0; ix<nx; ++ix) {
      for(int iy = 0; iy<ny; ++iy) {
         int i = ix + iy*ny;
         state[4*i+2] = dy*(iy+0.5);
         if(iy%2==0) {
            state[4*i] = dx*(ix+0.25);
         } else {
            state[4*i] = dx*(ix+0.75);
         }
      }
   }
}
```

Problem 8.10. Metastability

If we rapidly lower the temperature of a liquid below its freezing temperature, it is likely that the resulting state will not be an equilibrium crystal, but rather a supercooled liquid. If the properties of the supercooled state do not change with time for a time interval that is sufficiently long to obtain meaningful averages, we say that the system is in a *metastable* state. In general, we must carefully prepare our system so as to minimize the probability that the system becomes trapped in a metastable state. However, there is much interest in metastable states and how they eventually evolve to a more stable state (see Problem 15.20).

(a) What happens if the initial positions of the particles are on the nodes of a square lattice. As we found in Problem 8.9, this symmetry is not consistent with the lowest energy state corresponding to a triangular lattice. If the initial velocities are set to zero, what happens when you run the program? Choose $N = 64$ and $L_x = L_y = 9$.

(b) We can show that the system in part (a) is in a metastable state by giving the particles a small random initial velocity in the interval $[-0.5, +0.5]$. Does the symmetry of the lattice immediately change or is there a delay? When do you begin to see local structure that resembles a triangular lattice?

(c) Repeat part (b) with random velocities in the interval $[-0.1, +0.1]$. □

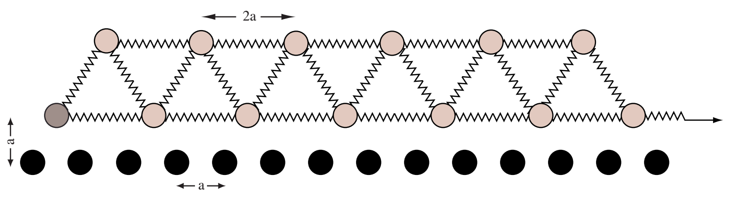


Figure 8.6: Initial configuration for the model of friction discussed in Problem 8.12. The atoms in the sliding object are placed in two rows of a triangular lattice with seven atoms in the bottom row and six atoms in the top row. There is a damping force on the left-most atom in the bottom row (the atom is shaded differently than the other atoms), and there is an external horizontal spring attached to the right-most atom in the bottom row.

Problem 8.11. The solid state and melting

(a) Choose $N = 64$, $L_x = 8$, and $L_y = \sqrt{3}L_x/2$ and place the particles on the nodes of a triangular lattice. Give each particle zero initial velocity. What is the total energy of the system? Do a simulation and measure the temperature and pressure as a function of time. Does the system remain a solid?

(b) Give each particle a random velocity in the interval $[-0.5,+0.5]$. What is the total energy? Equilibrate the system and determine the mean temperature and pressure. Describe the trajectories of the particles. Are the particles localized? Is the system a solid? Save an equilibrium configuration for use in part (c).

(c) Choose the initial configuration to be an equilibrium configuration from part (b) and gradually increase the kinetic energy by a factor of two. What is the new total energy? Describe the qualitative behavior of the motion of the particles. What is the equilibrium temperature and pressure of the system? After equilibrium is reached, increase the temperature again by rescaling the velocities in the same way. Repeat this rescaling and measure $P(T)$ and $E(T)$ for several different temperatures.

(d) Use your results from part (c) to plot $E(T)-E(0)$ and $P(T)$ as a function of T. Is the difference $E(T)-E(0)$ proportional to T? What is the mean potential energy for a harmonic solid? What is its heat capacity?

(e) Choose an equilibrium configuration from part (b) and decrease the density by rescaling L_x, L_y and the particle positions by a factor of 1.1. What is the nature of the trajectories? Decrease the density of the system until the system melts. What is your qualitative criterion for melting? □

Problem 8.12. Microscopic model of friction

In introductory physics texts sliding friction is usually described by the empirical law

$$f = \mu F_N \tag{8.13}$$

where f is the magnitude of the friction force, F_N is the normal force acting on the sliding object, and μ is the coefficient of friction. If the object is not moving, then (8.13), with μ equal to the static coefficient of friction, represents the frictional force needed to start the motion. If the object is moving, then (8.13), with μ equal to the kinetic coefficient of friction, represents the kinetic frictional force, which is assumed to be independent of the speed of the sliding object.

In this problem we explore a simple model discussed by Ringlein and Robbins to investigate the microscopic origin of friction. The stationary surface is modeled by a line of fixed atoms spaced a distance of $a = 2^{1/6}$ apart as shown in Figure 8.6. The sliding object is modeled by two rows of atoms in a triangular lattice configuration initially spaced a distance $2a$ from each other. The bottom row of atoms in the sliding object is a vertical distance a from the line of fixed atoms. The interaction between all the atoms in the two objects occurs via the Lennard–Jones potential. To keep the sliding object together, stiff springs with a spring constant of 500 (in reduced units) connect each atom to its nearest neighbors on the triangular lattice. The left-most atom on the bottom row has a damping force equal to $-10(v_x\hat{x} + v_y\hat{y}$ to help stabilize the motion. In addition, there is an external horizontal spring with spring constant equal to unity attached to the right-most atom on the bottom row. This spring is pulled at a constant rate causing this force on the atom to increase linearly. When this spring force is sufficiently large, the atoms start to move and the spring force suddenly drops. The point at which this decrease occurs defines the magnitude of the static frictional force.

(a) Modify your molecular dynamics program to simulate this model. Choose the sliding object to consist of 13 atoms, 7 on the bottom row and 6 on the top row. Place this system of 13 atoms on the middle of a stationary surface of fixed atoms (100 such atoms should be more than enough). Your program should show the pulling force due to the spring on the right-most atom as a function of time, and a visual display of the atoms in the system. A reasonable rate for pulling the spring is 0.1; that is, the external horizontal spring force is $0.1t - u$, where u is the horizontal displacement of the right-most atom from its initial position.

(b) As the system evolves, you should see the spring force suddenly drop when it reaches a value of about 14. Try different pulling rates and determine if the rate affects your results or the static friction force.

(c) Add a load that is equivalent to increasing the normal force. To add a load W to the system, add a vertical force of $-W/N$ to each of the $N = 13$ atoms in the sliding object. Find the static friction force, f_s as a function of W for W between -20 and $+40$. To what does a negative load correspond? Determine the coefficient of static friction from the slope of f_s versus W.

(d) Reduce the surface area by eliminating 4 atoms, 2 from each row. Rerun your simulations and discuss the results. Repeat for an increased size of 17 atoms and fit your results to the form

$$f_s = \mu_s W + cA \tag{8.14}$$

where A is the number of atoms in the bottom row, and c is a constant. The area dependence in (8.14) is different from what is usually assumed for sliding friction in introductory physics textbooks. The surfaces of macroscopic objects are typically rough at the microscopic level, and thus the effective area of contact is much smaller than the surface area. The effective area can be proportional to the load, and thus both terms in (8.14) can be proportional to the load, which is consistent with the usual assumption made in introductory physics texts. □

8.8 Radial Distribution Function

We can gain more insight into the structure of a many-body system by looking at how the positions of the particles are correlated with one another due to their interactions. The *radial distribution function* $g(r)$ is a measure of this correlation and has the following properties. Suppose that N particles are in a region of volume V with number density $\rho = N/V$. Choose one of the particles to be the origin. Then the mean number of other particles between $\mathbf{r}$ and $\mathbf{r} + d\mathbf{r}$ is defined to be $\rho g(\mathbf{r})\,d\mathbf{r}$. If the interparticle interaction is spherically symmetric and the system is a gas or a liquid, then $g(\mathbf{r})$ depends only on the separation $r = |\mathbf{r}|$. The normalization condition for $g(r)$ is

$$\rho \int g(r)\,d\mathbf{r} = N - 1 \approx N \tag{8.15}$$

where the volume element $d\mathbf{r} = 4\pi r^2\,dr\,(d = 3)$, $2\pi r\,dr\,(d = 2)$, and $2dr\,(d = 1)$. Equation (8.15) implies that if we choose one particle as the origin and count all the other particles in the system, we obtain $N - 1$ particles.

For an ideal gas, there are no correlations between the particles, and the normalization condition (8.15) implies that $g(r) = 1$ for all r. For the Lennard–Jones interaction, we expect that $g(r) \to 0$ as $r \to 0$, because the repulsive force between particles increases rapidly as $r \to \infty$. We also expect that $g(r) \to 1$ as $r \to \infty$, because the correlation of a given particle with the other particles decreases as their separation increases.

Several thermodynamic properties can be obtained from $g(r)$. Because $\rho g(r)$ can be interpreted as the local density of particles about a given particle, the potential energy of interaction between this particle and all other particles between r and $r + dr$ is $u(r)\rho g(r)\,d\mathbf{r}$, if we assume that only two-body interactions are present. The total potential energy is found by integrating over all r and multiplying by $N/2$. The factor of N is included because any of the N particles could be chosen as the particle at the origin, and the factor of $1/2$ is included so that each pair interaction is counted only once. The result is that the mean potential energy per particle can be expressed as

$$\frac{U}{N} = \frac{\rho}{2} \int g(r)u(r)\,d\mathbf{r}. \tag{8.16}$$

It can also be shown that the relation (8.9) for the mean pressure can be rewritten in terms of $g(r)$ so that the equation of state can be expressed as

$$\frac{PV}{NkT} = 1 - \frac{\rho}{2kTd} \int g(r)\,r\frac{du(r)}{dr}\,d\mathbf{r}. \tag{8.17}$$

To determine $g(r)$ for a particular configuration of particles, we first compute $n(r, \Delta r)$, the number of particles in a spherical (circular) shell of radius r and a small, but nonzero width Δr, with the center of the shell centered about each particle. A method for computing $n(r)$ is given in Listing 8.14.

Listing 8.14: Method to compute $n(r)$.

```
public void computeRDF() {
   // accumulate data for n(r)
   for (int i = 0; i < N-1; i++) {
      for (int j = i+1; i < N; j++) {
         double dx = PBC.separation(x[i] - x[j],Lx);
         double dy = PBC.separation(y[i] - y[j],Ly);
         double dy = (dy + Ly) % 0.5*Ly;
```

```
            double r2 = dx*dx + dy*dy;
            double r = Math.sqrt(r2);
            int bin = (int)(r/dr); // dr = shell width
            RDFAccumulator[bin]++;
         }
      }
      numberRDFMeasurements++;
   }
```

The use of periodic boundary conditions in computeRDF implies that the maximum separation between any two particles in the x and y directions is $L_x/2$ and $L_y/2$, respectively. Hence, we can determine $g(r)$ only for $r \leq \frac{1}{2}\min(L_x, L_y)$.

To obtain $g(r)$ from $n(r)$, we note that for a given particle i, we consider only those particles whose index j is greater than the index i (see computeRDF). Hence, there are a total of $\frac{1}{2}N(N-1)$ separations that are considered. In two dimensions we compute $n(r, \Delta r)$ for a circular shell whose area is $2\pi r\Delta r$. These considerations imply that $g(r)$ is related to $n(r)$ by

$$\rho g(r) = \frac{\overline{n(r, \Delta r)}}{\frac{1}{2}N\, 2\pi r\Delta r} \qquad \text{(two dimensions).} \tag{8.18}$$

Note the factor of $N/2$ in the denominator of (8.18). Method normalizeRDF normalizes the array RDFAccumulator and yields $g(r)$.

Listing 8.15: Method for obtaining $g(r)$ from $n(r)$.

```
public void normalizeRDF(PlotFrame dataRDF) {
   double density = N/(Lx*Ly);
   double L = Math.min(Lx,Ly);
   // maximum index is one less than binMax
   int binMax = (int)(L/(2*dr));
   double normalization = density*numberRDFMeasurements*N/2;
   for (int bin = 0; bin < binMax; bin++) {
      double r = bin*dr;
      double shellArea = Math.PI*(Math.pow(r+dr,2) - Math.pow(r,2));
      double RDF = RDFAccumulator[bin]/(normalization*shellArea);
      dataRDF.append(0,dr*(bin+0.5),RDF); // adds results to be plotted
   }
}
```

The shell thickness Δr needs to be sufficiently small so that the important features of $g(r)$ are found, but large enough so that each bin has a reasonable number of contributions. The value of Δr should be a class variable. A reasonable choice for its magnitude is $\Delta r = 0.025$.

Problem 8.13. The structure of $g(r)$ for a dense liquid and a solid

(a) Write a test program that incorporates computeRDF and normalizeRDF and compute $g(r)$ for a system of $N = 64$ particles that are fixed on a triangular lattice with $L_x = 8$ and $L_y = \sqrt{3}L_x/2$. What is the density of the system? What is the nearest neighbor distance between sites? At what value of r does the first maximum of $g(r)$ occur? What is the next nearest distance between sites? Does your calculated $g(r)$ have any other peaks? If so, relate these peaks to the structure of the triangular lattice.

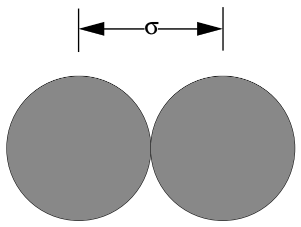


Figure 8.7: The closest distance between two hard disks is σ. The disks exert no force on one another unless they touch.

(b) Modify your molecular dynamics program and compute $g(r)$ for a dense fluid ($\rho > 0.6$, $T \approx 1.0$) with $N \geq 64$. How many peaks in $g(r)$ can you observe? In what ways do they change as the density is increased? How does the behavior of $g(r)$ for a dense liquid compare to that of a dilute gas and a solid? □

8.9 Hard Disks

How can we understand the temperature and density dependence of the equation of state and the structure of a dense liquid? One way to gain more insight into this dependence is to modify the interaction and see how the properties of the system change. In particular, we would like to understand the relative role of the repulsive and attractive parts of the interaction. For this reason, we consider an idealized system of hard disks for which the interaction $u(r)$ is purely repulsive:

$$u(r) = \begin{cases} +\infty, & r < \sigma \\ 0, & r \geq \sigma\ . \end{cases} \tag{8.19}$$

The length σ is the diameter of the hard disks (see Figure 8.7). In three dimensions the interaction (8.19) describes the interaction of hard spheres (billiard balls); in one dimension (8.19) describes the interaction of hard rods.

Because the interaction $u(r)$ between hard disks is a discontinuous function of r, the dynamics of hard disks is qualitatively different than it is for a continuous interaction such as the Lennard–Jones potential. For hard disks the particles move in straight lines at constant speed between collisions and change their velocities instantaneously when a collision occurs. Hence the problem becomes finding the next collision and computing the change in the velocities of the colliding pair. The dynamics is *event driven* and can be computed exactly in principle; in practice, it is limited only by roundoff errors.

The dynamics of a system of hard disks can be treated as a sequence of two-body elastic collisions. The idea is to consider all pairs of particles i and j and to find the collision time t_{ij} for their next collision, ignoring the presence of all other particles. In many cases the particles will be going away from each other and the collision time is infinite. From the collection of collision times for all pairs of particles, we find the minimum collision time. We then move all the particles forward in time until the collision occurs and calculate the postcollision velocities of the colliding pair. The main problem is dealing with the large number of possible collision events.

We first determine the particle velocities of the colliding pair. Consider a collision between particles i and j. Let $\mathbf{v}_i$ and $\mathbf{v}_j$ be their velocities before the collision and $\mathbf{v}'_i$ and $\mathbf{v}'_j$ be their velocities after the collision. Because the particles have equal mass, it follows from conservation of energy and linear momentum that

$$v_i'^2 + v_j'^2 = v_i{}^2 + v_j{}^2 \tag{8.20}$$

$$\mathbf{v}_i' + \mathbf{v}_j' = \mathbf{v}_i + \mathbf{v}_j. \tag{8.21}$$

From (8.21) we have

$$\Delta\mathbf{v}_i = \mathbf{v}_i' - \mathbf{v}_i = -(\mathbf{v}_j' - \mathbf{v}_j) = -\Delta\mathbf{v}_j. \tag{8.22}$$

When two hard disks collide, the force is exerted along the line connecting their centers, $\mathbf{r}_{ij} = \mathbf{r}_i - \mathbf{r}_j$. Hence, the components of the velocities parallel to $\mathbf{r}_{ij}$ are exchanged, and the perpendicular components of the velocities are unchanged. It is convenient to write the velocity of particles i and j as a vector sum of their components parallel and perpendicular to the unit vector $\hat{\mathbf{r}}_{ij} = \mathbf{r}_{ij}/|\mathbf{r}_{ij}|$. We write the velocity of particle i as

$$\mathbf{v}_i = \mathbf{v}_{i,\|} + \mathbf{v}_{i,\perp} \tag{8.23}$$

where $\mathbf{v}_{i,\|} = (\mathbf{v}_i \cdot \hat{\mathbf{r}}_{ij})\hat{\mathbf{r}}_{ij}$, and

$$\mathbf{v}_{i,\|}' = \mathbf{v}_{j,\|} \qquad \mathbf{v}_{j,\|}' = \mathbf{v}_{i,\|} \tag{8.24a}$$

$$\mathbf{v}_{i,\perp}' = \mathbf{v}_{i,\perp} \qquad \mathbf{v}_{j,\perp}' = \mathbf{v}_{j,\perp}. \tag{8.24b}$$

Hence, we can write $\mathbf{v}_i'$ as

$$\begin{aligned}\mathbf{v}_i' &= \mathbf{v}_{i,\|}' + \mathbf{v}_{i,\perp}' = \mathbf{v}_{j,\|} + \mathbf{v}_{i,\perp} \\ &= \mathbf{v}_{j,\|} - \mathbf{v}_{i,\|} + \mathbf{v}_{i,\|} + \mathbf{v}_{i,\perp} \\ &= \Big[(\mathbf{v}_j - \mathbf{v}_i)\cdot\hat{\mathbf{r}}_{ij}\Big]\hat{\mathbf{r}}_{ij} + \mathbf{v}_i. \end{aligned} \tag{8.25}$$

The change in the velocity of particle i at a collision is given by

$$\Delta\mathbf{v}_i = \mathbf{v}_i' - \mathbf{v}_i = -\Big[(\mathbf{v}_i - \mathbf{v}_j)\cdot\hat{\mathbf{r}}_{ij}\Big]\hat{\mathbf{r}}_{ij} \tag{8.26}$$

or

$$\Delta\mathbf{v}_i = -\Delta\mathbf{v}_j = \left(\frac{\mathbf{r}_{ij}\, b_{ij}}{\sigma^2}\right)_{\text{contact}} \tag{8.27}$$

where $b_{ij} = \mathbf{v}_{ij}\cdot\mathbf{r}_{ij}$, $\mathbf{v}_{ij} = \mathbf{v}_i - \mathbf{v}_j$, and we have used the fact that $|\mathbf{r}_{ij}| = \sigma$ at contact.

Exercise 8.14. Velocity distribution of hard rods

Use (8.20) and (8.21) to show that $v_i' = v_j$ and $v_j' = v_i$ in one dimension; that is, two colliding hard rods of equal mass exchange velocities. If you start a system of hard rods with velocities chosen from a uniform random distribution, will the velocity distribution approach the equilibrium Maxwell–Boltzmann distribution? □

We now consider the criteria for a collision to occur. Consider disks i and j at positions $\mathbf{r}_i$ and $\mathbf{r}_j$ at $t = 0$. If they collide at a time t_{ij} later, their centers will be separated by a distance σ:

$$|\mathbf{r}_i(t_{ij}) - \mathbf{r}_j(t_{ij})| = \sigma. \tag{8.28}$$

During the time t_{ij}, the disks move with constant velocities. Hence, we have

$$\mathbf{r}_i(t_{ij}) = \mathbf{r}_i(0) + \mathbf{v}_i(0)\,t_{ij} \tag{8.29}$$

and

$$\mathbf{r}_j(t_{ij}) = \mathbf{r}_2(0) + \mathbf{v}_2(0)\,t_{ij}. \tag{8.30}$$

If we substitute (8.29) and (8.30) into (8.28), we find

$$[\mathbf{r}_{ij} + \mathbf{v}_{ij}t_{ij}]^2 = \sigma^2 \tag{8.31}$$

where $\mathbf{r}_{ij} = \mathbf{r}_i(0) - \mathbf{r}_j(0)$, $\mathbf{v}_{ij} = \mathbf{v}_i(0) - \mathbf{v}_j(0)$, and

$$t_{ij} = \frac{-\mathbf{v}_{ij}\cdot\mathbf{r}_{ij} \pm \sqrt{(\mathbf{v}_{ij}\cdot\mathbf{r}_{ij})^2 - v_{ij}{}^2(r_{ij}{}^2 - \sigma^2)}}{v_{ij}{}^2}. \tag{8.32}$$

Because $t_{ij} > 0$ for a collision to occur, we see from (8.32) that the condition

$$\mathbf{v}_{ij}\cdot\mathbf{r}_{ij} < 0 \tag{8.33}$$

must be satisfied. That is, if $\mathbf{v}_{ij}\cdot\mathbf{r}_{ij} > 0$, the particles are moving away from each other and there is no possibility of a collision.

If the condition (8.33) is satisfied, then the discriminant in (8.32) must satisfy the condition

$$(\mathbf{v}_{ij}\cdot\mathbf{r}_{ij})^2 - v_{ij}{}^2(r_{ij}{}^2 - \sigma^2) \geq 0. \tag{8.34}$$

If the condition (8.34) is satisfied, then the quadratic in (8.32) has two roots. The smaller root corresponds to the physically significant collision because the disks are impenetrable. Hence, the physically significant solution for the time of a collision t_{ij} for particles i and j is given by

$$t_{ij} = \frac{-b_{ij} - \left[b_{ij}{}^2 - v_{ij}{}^2\,(r_{ij}{}^2 - \sigma^2)\right]^{1/2}}{v_{ij}{}^2}. \tag{8.35}$$

Exercise 8.15. Calculation of collision times

Write a short program that determines the collision times (if any) of the following pairs of particles. It would be a good idea to draw the trajectories to confirm your results. Consider the cases: $\mathbf{r}_1 = (2,1)$, $\mathbf{v}_1 = (-1,-2)$, $\mathbf{r}_2 = (1,3)$, $\mathbf{v}_2 = (1,1)$; $\mathbf{r}_1 = (4,3)$, $\mathbf{v}_1 = (2,-3)$, $\mathbf{r}_2 = (3,1)$, $\mathbf{v}_2 = (-1,-1)$; and $\mathbf{r}_1 = (4,2)$, $\mathbf{v}_1 = (-2,\frac{1}{2})$, $\mathbf{r}_2 = (3,1)$, $\mathbf{v}_2 = (-1,1)$. As usual, choose units so that $\sigma = 1$. □

Our hard disk program implements the following steps. We first find the collision times and the collision partners for all pairs of particles i and j. We then do the following.

1. locate the minimum collision time t_{min};
2. advance all particles using a straight line trajectory until the collision occurs; that is, displace particle i by $\mathbf{v}_i\,t_{\text{min}}$ and update its next collision time;
3. compute the postcollision velocities of the colliding pair `nextCollider` and `nextPartner`;
4. calculate the physical quantities of interest and accumulate data;

5. update the collision partners of the colliding pair, `nextCollider` and `nextPartner`, and all other particles that were to collide with either `nextCollider` or `nextPartner` if `nextCollider` and `nextPartner` had not collided first;
6. repeat steps 1–5 indefinitely.

Methods for carrying out these steps are listed in the following:

Listing 8.16: Methods for each step of the hard disk system.

```
public void step() {
   // finds minimum collision time from list of collision times
   minimumCollisionTime();
   // moves particles for time equal to minimum collision time
   move();
   t += timeToCollision;
   // changes velocities of two colliding particles
   contact();
   // sets collision times to bigTime for those particles set to collide with
   // two colliding particles.
   setDefaultCollisionTimes();
   // finds new collision times between all particles and two colliding particles
   newCollisionTimes();
   numberOfCollisions++;
}

public void minimumCollisionTime() {
   // sets collision time very large so that can find minimum collision time
   timeToCollision = bigTime;
   for(int k = 0; k<N; k++) {
      if(collisionTime[k]<timeToCollision) {
         timeToCollision = collisionTime[k];
         nextCollider = k;
      }
   }
   nextPartner = partner[nextCollider];
}

public void move() {
   for(int k = 0; k<N; k++) {
      collisionTime[k] -= timeToCollision;
      x[k] = PBC.position(x[k]+vx[k]*timeToCollision, Lx);
      y[k] = PBC.position(y[k]+vy[k]*timeToCollision, Ly);
   }
}

public void contact() {
   // computes collision dynamics between nextCollider and nextPartner at contact
   double dx = PBC.separation(x[nextCollider]-x[nextPartner], Lx);
   double dy = PBC.separation(y[nextCollider]-y[nextPartner], Ly);
   double dvx = vx[nextCollider]-vx[nextPartner];
   double dvy = vy[nextCollider]-vy[nextPartner];
   double factor = dx*dvx+dy*dvy;
   double delvx = -factor*dx;
   double delvy = -factor*dy;
```

```
      vx[nextCollider] += delvx;
      vy[nextCollider] += delvy;
      vx[nextPartner] -= delvx;
      vy[nextPartner] -= delvy;
      virialSum += delvx*dx+delvy*dy;
   }

   public void setDefaultCollisionTimes() {
      collisionTime[nextCollider] = bigTime;
      collisionTime[nextPartner] = bigTime;
     // sets collision times to bigTime for all particles set to collide
     // with the two colliding particles
      for(int k = 0; k<N; k++) {
         if(partner[k]==nextCollider) {
            collisionTime[k] = bigTime;
         } else if(partner[k]==nextPartner) {
            collisionTime[k] = bigTime;
         }
      }
   }

   public void newCollisionTimes() {
     // finds new collision times for all particles which were set to collide
     // with two colliding particles; also finds new collision
     // times for two colliding particles.
      for(int k = 0; k<N; k++) {
         if((k!=nextCollider)&&(k!=nextPartner)) {
            checkCollision(k, nextPartner);
            checkCollision(k, nextCollider);
         }
      }
   }
```

The colliding pair and the next collision time are found in method `minimumCollisionTime`, and all the particles are moved forward in `move` until contact occurs. The collision dynamics of the colliding pair is computed in method `contact`, where the contribution to the virial is also found. In `setDefaultCollisionTimes` we set all the collisions times to an arbitrarily large value, `bigTime`, for all pairs of particles that need to be updated. Then in `newCollisionTimes` we update the collision times for those particles in step 5.

In method `initialize` we initialize various variables and most importantly compute the minimum collision time for each particle using method `checkCollision`. The *i*th element in the array, `collisionTime`, stores the minimum collision time for particle *i* with all the other particles. The array element `partner[i]` stores the particle label of the collision partner corresponding to this time. The collision time for each particle is initially set to an arbitrarily large value, `bigTime`, to take into account that at any given time, some particles have no collision partners. The methods for setting the initial positions and velocities are the same as those used for simulating Lennard–Jones particles.

Listing 8.17: Method for generating the initial configuration of hard disks.

```
   public void initialize(String configuration) {
      resetAverages();
      x = new double[N];
      y = new double[N];
      vx = new double[N];
      vy = new double[N];
      collisionTime = new double[N];
      partner = new int[N];
      if(configuration.equals("regular")) {
         setRegularPositions();
      } else {
         setRandomPositions();
      }
      setVelocities();
      for(int i = 0; i<N; ++i) {
         // sets unknown collision times to a big number
         collisionTime[i] = bigTime;
      }
      // find initial collision times for all particles
      for(int i = 0; i<N-1; i++) {
         for(int j = i+1; j<N; j++) {
            checkCollision(i, j);
         }
      }
   }

   public void resetAverages() {
      t = 0;
      virialSum = 0;
   }
```

Method `checkCollision` uses the relations (8.33) and (8.35) to determine whether particles `i` and `j` will collide and if so, the time `tij` until their collision. We check for collisions with particle `j` in the central cell as well as with particle `j` in the eight image cells surrounding the central cell as shown in Figure 8.8. For very dilute systems we might need to check further periodic images. For the densities we will consider, such a check should not be necessary.

Listing 8.18: Method for checking the collision time and collision partners of each particle.

```
   public void checkCollision(int i, int j) {
      // consider collisions between i and j and periodic images of j
      double dvx = vx[i]-vx[j];
      double dvy = vy[i]-vy[j];
      double v2 = dvx*dvx+dvy*dvy;
      for(int xCell = -1; xCell<=1; xCell++) {
         for(int yCell = -1; yCell<=1; yCell++) {
            double dx = x[i]-x[j]+xCell*Lx;
            double dy = y[i]-y[j]+yCell*Ly;
            double bij = dx*dvx+dy*dvy;
            if(bij<0) {
               double r2 = dx*dx+dy*dy;
               double discriminant = bij*bij-v2*(r2-1);
               if(discriminant>0) {
                  double tij = (-bij-Math.sqrt(discriminant))/v2;
                  if(tij<collisionTime[i]) {
                     collisionTime[i] = tij;
                     partner[i] = j;
```

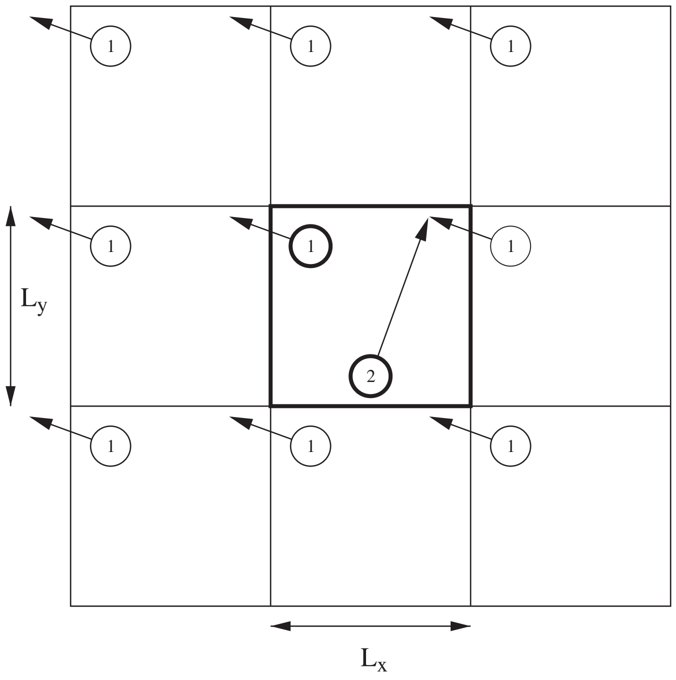


Figure 8.8: The positions and velocities of disks 1 and 2 in the figure are such that disk 1 collides with an image of disk 2 that is not the image closest to disk 1. The periodic images of disk 2 are not shown.

```
                }
                if(tij<collisionTime[j]) {
                    collisionTime[j] = tij;
                    partner[j] = i;
                }
            }
        }
    }
}
}
```

The main thermodynamic quantity of interest for hard disks is the mean pressure P. Because the forces act only when two disks are in contact, we have to modify the form of (8.9). We write $\mathbf{F}_{ij}(t) = \mathbf{I}_{ij}\,\delta(t - t_c)$, where t_c is the time at which the collision occurs. This form of $\mathbf{F}_{ij}$ implies that the force is nonzero only when there is a collision between i and j. The delta function $\delta(t)$ is infinite for $t = 0$ and is zero otherwise; $\delta(t)$ is defined by its use in an integral as shown in (8.36). This form of the force yields

$$\int_0^t \mathbf{I}_{ij}\,\delta(t' - t_c)\,dt' = \mathbf{I}_{ij} = m\Delta\mathbf{v}_{ij} \tag{8.36}$$

where we have used Newton's second law and assumed that a single collision has occurred during the time interval t. The quantity $\Delta\mathbf{v}_{ij}$ is given by $\Delta\mathbf{v}_{ij} = \mathbf{v}'_i - \mathbf{v}_i - (\mathbf{v}'_j - \mathbf{v}_j)$. If we explicitly include the time average to account for all collisions during the time interval t, we can write

(8.9) as

$$\begin{aligned}\frac{PV}{NkT}-1 &= \frac{1}{dNkT}\frac{1}{t}\sum_{ij}\int_0^t \mathbf{r}_{ij}\cdot\mathbf{I}_{ij}\,\delta(t'-t_c)\,dt' \\ &= \frac{1}{dNkT}\frac{1}{t}\sum_{c_{ij}} m\Delta\mathbf{v}_{ij}\cdot\mathbf{r}_{ij}. \qquad (8.37)\end{aligned}$$

The sum in (8.37) is over all collisions c_{ij} between disks i and j in the time interval t; $\mathbf{r}_{ij}$ is the vector between the centers of the disks at the time of a collision; the magnitude of $\mathbf{r}_{ij}$ in (8.37) is σ.

Listing 8.19: Method for calculating the pressure.

```
public double pressure() {
   double area = Lx*Ly;
   return 1 + virialSum/(2*t*N*temperature);
}
```

As discussed in Problem 8.16, an important check on the calculated trajectories of a hard disk system is that no two disks overlap. The following method tests for this condition.

```
public void checkOverlap() {
   for (int i = 0; i < N-1; ++i) {
      for(int j = i+1; j < N; ++j) {
         double dx = PBC.separation(x[i] - x[j],Lx);
         double dy = PBC.separation(y[i] - y[j],Ly);
         if (dx*dx+dy*dy < 1.0) {
            System.out.println("Particles " + i + " and" + j + " overlap");
         }
      }
   }
}
```

To complete class HardDisks, we need to add the class declarations, which we show in Listing 8.20, and the draw method, which is the same as in class LJParticles. You can use a slightly modified version of class LJParticlesApp as the target class for this application, but note that you will need to modify the LJParticlesLoader class to store different arrays. The number of collisions, the time, and a plot of the pressure versus time should be displayed. We will leave the task of writing the target class as an exercise.

Listing 8.20: Class declarations for HardDisks.

```
package org.opensourcephysics.sip.ch08.hd;
import java.awt.*;
import org.opensourcephysics.display.*;
import org.opensourcephysics.numerics.*;

public class HardDisks implements Drawable {
   public double x[], y[], vx[], vy[];
   public double collisionTime[];
   public int partner[];
   public int N;
   public double Lx;
   public double Ly;
   public double keSum = 0, virialSum = 0;
   public int nextCollider, nextPartner;
   public double timeToCollision;
   public double t = 0;
   public double bigTime = 1.0E10;
   public double temperature;
   public int numberOfCollisions = 0;
```

Problem 8.16. Initial test of class `HardDisks`

(a) Because even a small error in computing the trajectories of the disks will eventually lead to their overlap and hence to a fatal error, it is necessary to test class `HardDisks` carefully. For simplicity, start from a lattice configuration. The most important test of the program is to monitor the computed positions of the hard disks for overlaps. If the distance between the centers of any two hard disks is less then unity (distances are measured in units of σ), there must be a serious error in the program. To check for the overlap of hard disks, include method `checkOverlap` in method `step` while you are testing the program.

(b) The temperature for a system of hard disks is constant and can be defined as in (8.6). Why does the temperature not fluctuate as it does for a system of particles interacting with a continuous potential? The constancy of the temperature can be used as another check on your program. What is the effect of increasing all the velocities by a factor of two? What is the natural unit of time? Explain why the state of the system is determined by the density only and not by the temperature.

(c) Generate equilibrium configurations of a system of $N = 64$ disks in a square cell of linear dimension $L = 12$. Suppose that at $t = 0$, the constraint that $0 \leq x \leq 12$ is removed, and the disks are allowed to move in a rectangular cell with $L_x = 24$ and $L_y = 12$. Does the system become more or less random? What is the qualitative nature of the time dependence of $n(t)$, the number of disks on the left half of the cell?

(d) Modify your program so that averages are not computed until the system is in equilibrium. Compute the virial (8.37) and make a rough estimate of the error in your determination of the mean pressure due to statistical fluctuations.

(e) Modify your program so that you can compute the velocity and speed distributions and verify that the computed distributions have the expected forms. □

Problem 8.17. Static properties of hard disks

(a) As we have seen in Section 8.7, a very time consuming part of the simulation is equilibrating a system from an arbitrary initial configuration. One way to obtain a set of initial positions is to add the hard disks sequentially with random positions and reject an additional hard disk if it overlaps any disks already present. Although this method is very inefficient at high densities, try it so that you will have a better idea of how difficult it is to obtain a high density configuration in this way. A much better method is to place the disks on the sites of a lattice.

(b) The largest number of hard disks that can be placed into a fixed volume defines the maximum density. What is the maximum density if the disks are placed on a square lattice? What is the maximum density if the disks are placed on a triangular lattice? Suppose that the initial condition is chosen to be a square lattice with $N = 100$ and $L = 11$ so that each particle has four nearest neighbors. What is the qualitative nature of the system after several hundred collisions have occurred? Do most particles still have four nearest neighbors, or are there regions where most particles have six neighbors?

(c) The dependence of the mean pressure P on the density ρ is of interest, as it is for a system with a continuous potential. Is P a monotonically increasing function of ρ? Is a system of hard disks always a fluid, or is there a fluid to solid transition at higher densities? You will not be able to find definitive answers to these questions for $N = 64$. Most simulations in the 1960s and 70s were done for systems of $N = 108$ hard disks. The largest simulations were for several hundred particles and new insight of the properties of liquids was found. Find the dependence of the pressure on the density, beginning at low densities and slowly increasing the density starting from a configuration from a lower density. At any given time, the maximum density increase is given by the minimum distance between any two disks. To increase the density, rescale all the positions and the cell size so that the minimum distance is reduced by a factor of two. Repeat this procedure until you reach the desired density. You will need to equilibrate the system between rescalings.

(d) Compute the radial distribution function $g(r)$ for the same densities you considered for the Lennard–Jones interaction. Computing $g(r)$ is more subtle for the hard disk system than it is for a system with a continuous potential. For the latter system, we can accumulate the sums needed to compute $g(r)$ at regular intervals and simply take the average of the computed quantities. However, in event driven dynamics, the time does not evolve uniformly. The simplest procedure is to keep track of the number of collisions and to compute the necessary sums after a certain number of collisions has occurred. If the number of collisions is sufficiently large, the time elapsed will be approximately the same. The relation of the pressure to $g(r)$ for hard disks is discussed on page 620.

(e) Compare $g(r)$ for the hard disk and Lennard–Jones interactions at the same density. On the basis of your results, which part of the Lennard–Jones interaction plays the dominant role in determining the structure of a dense Lennard–Jones liquid? □

Simulations of systems of hard disks and hard spheres have shown that the structure of these systems does not differ significantly from the structure of systems with more complicated interactions. Given this insight, our present understanding of liquids is based on the use of the hard sphere (disk) system as a reference system; the differences between the hard sphere interaction and the more complicated interaction of interest are treated as a perturbation about this reference system. Thus, even though the particles interact strongly in a dense gas and a liquid, we now have a perturbation theory of liquids, thanks to the insight gained from simulations. Another important insight that was obtained from simulations is that the solid phase does not require an attractive part of the intermolecular potential. That is, hard spheres and disks have a freezing or melting transition, although the nature of the latter is still a subject of current interest.

In Problem 8.18 we consider two physical quantities associated with the dynamics of a system of hard disks, namely the mean free time and the mean free path, quantities of interest in kinetic theory.

Problem 8.18. Mean free path and collision time

(a) Class `HardDisks` provides the information needed to determine the mean free time t_c, that is, the average time a particle travels between collisions. For example, suppose we know that

40 collisions occurred in a time $t = 2.5$ for a system of $N = 16$ disks. Because two particles are involved in each collision, there was an average of 80/16 collisions per particle. Hence, $t_c = 2.5/(80/16) = 0.5$. Write a method to compute t_c and determine t_c as a function of ρ.

(b) The mean free path ℓ is the mean distance a particle travels between collisions. In introductory textbooks, the relation of ℓ to t_c is given by the simple relation $\ell = \overline{v}t_c$, where $\overline{v}$ is the root-mean square velocity, $\overline{v} = \sqrt{\overline{v^2}}$. Write a method to compute the mean free path of the particles. Note that the displacement of particle i during the time t is $v_i t$, where v_i is the speed of particle i. What relation do you find between ℓ and t_c?

(c) Write a method to determine the distribution of times between collisions. What is the qualitative form of the distribution? How does the width of this distribution depend on ρ? □

8.10 Dynamical Properties

The mean free time and the mean free path are well defined for hard disks for which the meaning of a collision is clear. From kinetic theory we know that both quantities are related to the transport properties of a dilute gas. However, the concept of a collision is not well defined for systems with a continuous interaction, such as the Lennard–Jones potential. In the following, we take a more general approach to the dynamics of a many-body system and discuss how the transport of particles in a system near equilibrium is related to the *equilibrium* properties of the system.

Consider the trajectory of a particular particle, for example, particle 1 in a system that is in equilibrium. At some arbitrarily chosen time $t = 0$, its position is $\mathbf{r}_1(0)$. At a later time t, its displacement is $\mathbf{r}_1(t) - \mathbf{r}_1(0)$. If there was no net force on the particle during this time interval, then $\mathbf{r}_1(t) - \mathbf{r}_1(0)$ would increase linearly with t. However, a particle in a fluid undergoes many collisions, and on the average its net displacement would be zero. A more interesting quantity is the mean square displacement defined as

$$\overline{R(t)^2} = \overline{[\mathbf{r}_1(t) - \mathbf{r}_1(0)]^2}. \tag{8.38}$$

The average in (8.38) is over all possible choices of the time origin. Because the system is in equilibrium, the choice of $t = 0$ is arbitrary, and $\overline{R_1(t)^2}$ depends only on the time difference t.

If the collisions of particle 1 with the other particles are random, then we would suspect that particle 1 undergoes a random walk, and the t-dependence of $\overline{R(t)^2}$ would be given by (see (7.76))

$$\overline{R(t)^2} = 2dDt \qquad (t \to \infty) \tag{8.39}$$

where d is the spatial dimension. The coefficient D in (8.39) is known as the *self-diffusion* coefficient and is an example of a transport coefficient. Because the average behavior of all the particles should be the same, we would find better results if we average over all particles. The relation (8.39) relates the macroscopic transport coefficient D to a microscopic quantity $\overline{R(t)^2}$ and gives us a way of computing D.

The easiest way of computing $\overline{R(t)^2}$ is to save the position of a particle in a file at regular time intervals. We later can use a separate program to read the data file and compute $\overline{R(t)^2}$. To understand the procedure for computing $\overline{R(t)^2}$, we consider a simple example. Suppose that the position of a particle in a one-dimensional system is given by $x(t=0)=1.65$, $x(t=1)=1.62$, $x(t=2)=1.84$, and $x(t=3)=2.22$. If we average over all possible time origins, we obtain

$$\begin{aligned}\overline{R(t=1)^2} &= \frac{1}{3}\Big[\big(x(1)-x(0)\big)^2+\big(x(2)-x(1)\big)^2+\big(x(3)-x(2)\big)^2\Big]\\ &= \frac{1}{3}\Big[0.0009+0.0484+0.1444\Big]v = 0.0646\\ \overline{R(t=2)^2} &= \frac{1}{2}\Big[\big(x(2)-x(0)\big)^2+(x(3)-x(1)\big)^2\Big] = \frac{1}{2}\Big[0.0361+0.36\Big] = 0.1981\\ \overline{R(t=3)^2} &= \Big[x(3)-x(0)\Big]^2 = 0.3249.\end{aligned}$$

Note that there are fewer combinations of the positions as the time difference increases.

In Listing 8.21 we show a method that computes $\overline{R(t)^2}$ assuming that the positions of all N particles have been collected for n times in the arrays xSave[i][k] and ySave[i][k]; the time is indexed by k. Because of periodic boundary conditions, we cannot find the distance moved by a particle by just keeping track of its position. Imagine that a particle moved in only one direction and returned to its original position. If we just subtracted the coordinates of the position, we would find that the particle's displacement was zero when in fact it really moved an amount equal to the length of the simulation cell. To keep track of the movement of each particle, we use two arrays, xWrap and yWrap. Every time particle i moves past the right boundary in the time interval k*dk to (k+1)*dk, xWrap[i][k] is incremented by Lx. Similarly, each time the particle moves past the left boundary xWrap[i][k] is decremented by Lx. A similar procedure is used for yWrap.

Listing 8.21: Listing of method computeR2 for finding the mean square displacement.

```
public void computeR2(PlotFrame data) {
   for(int dk = 1; dk < n-1; ++dk) { // loops over time intervals
      int norm = 0;
      double r2 = 0;
      for(int i = 0; i < N; i++) {  // loops over particles
         // time origin labeled by k0
         for(int k0 = 0; k0 < n-dk-1; ++k0)  { // loops over time origins
          double dx = (xSave[i][k0+dk]+xWrap[i][k0+dk])
                     -(xSave[i][k0]+xWrap[i][k0+dk]);
          double dy = (ySave[i][k0+dk]+yWrap[i][k0+dk])
                     -(ySave[i][k0]+yWrap[i][k0+dk]);
          r2 += dx*dx + dy*dy;
          norm++;
         }
      }
      data.append(0,dk*timeInterval,r2/norm);
   }
}
```

We show our results for $\overline{R(t)^2}$ for a system of Lennard–Jones particles in Figure 8.9. Note that $\overline{R(t)^2}$ increases approximately linearly with t with a slope of roughly 0.61. From (8.39) the corresponding self-diffusion coefficient is $D = 0.61/4 \approx 0.15$. In Problem 8.19 we use method computeR2 to compute the self-diffusion coefficient. An easier but less direct way of computing D is discussed in Project 8.23.

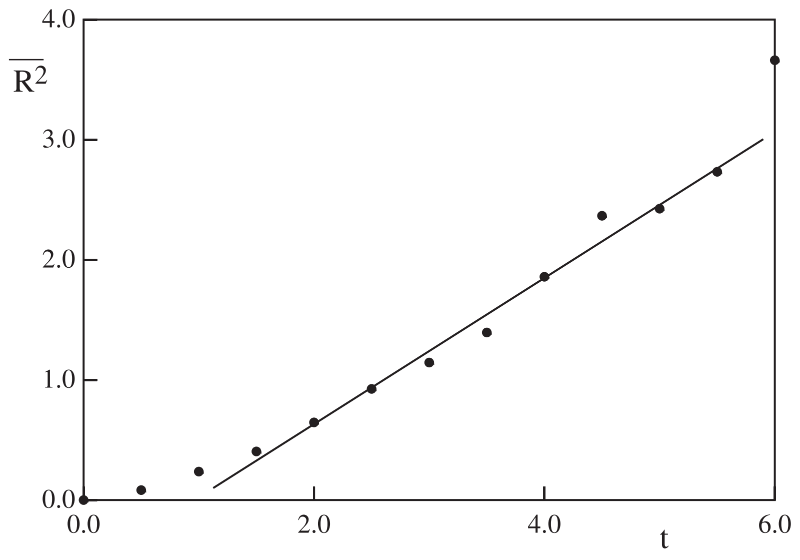


Figure 8.9: The time dependence of the mean square displacement $\overline{R(t)^2}$ for one particle in a two-dimensional Lennard–Jones system with $N = 16$, $L = 5$, and $E = 5.8115$. The position of a particle was saved at intervals of 0.5. Much better results can be obtained by averaging over all particles and over a longer run. The least squares fit was made between $t = 1.5$ and $t = 5.5$. As expected, this fit does not pass through the origin. The slope of the fit is 0.61.

Problem 8.19. The self-diffusion coefficient

(a) Use either your hard disk or molecular dynamics program and visually follow the motion of a particular particle by "tagging" it, for example, by drawing its path with a different color. Describe its motion qualitatively.

(b) Modify your program so that the coordinates of the particles are saved at regular intervals (see Appendix 8A). The optimum time interval needs to be determined empirically. If you save the coordinates too often, the data file will become very large, and you will waste time saving the coordinates. If you do not save the positions often enough, you will lose information. Because the time step Δt must be small compared to any interesting time scale, we know that the time interval for saving the positions must be at least an order of magnitude greater than Δt. A good first guess is to choose the time interval for saving the coordinates to be the order of $10\Delta t$. The easiest procedure for hard disks is to save the coordinates at intervals measured in terms of the number of collisions. If you average over a sufficient number of collisions, you can find the relation between the elapsed time and the number of collisions.

(c) For a finite system, the time difference t cannot be chosen to be too large because the displacement of a particle is bounded. What is the maximum value of $R^2(t)$?

(d) Compute $\overline{R(t)^2}$ for conditions that correspond to a dense fluid. Does $\overline{R(t)^2}$ increase as t^2 as for a free particle or more slowly? Does $\overline{R(t)^2}$ increase linearly with t for longer times?

(e) Use the relation (8.39) to estimate the magnitude of D from the slope of $\overline{R(t)^2}$ for the time interval for which $\overline{R(t)^2}$ is approximately linear. Obtain D for several different temperatures and densities. (A careful study of $\overline{R(t)^2}$ for much larger systems and much longer times would show that $\overline{R(t)^2}$ is not proportional to t in two dimensions. Instead, $\overline{R(t)^2}$ has a term proportional to $t \log t$, which dominates the linear t term if t is sufficiently large. We will not be able to observe the effects of this logarithmic term, and we can interpret our results for $\overline{R(t)^2}$ in terms of an "effective" diffusion coefficient. No such problem exists for three dimensions. See Problem 8.20d.)

(f) Estimate the accuracy of your determination of D. How sensitive is it to the value of Δt? How does this accuracy compare to your estimates of other physical quantities such as the mean pressure?

(g) Compute $\overline{R(t)^2}$ for an equilibrium configuration corresponding to a harmonic solid. What is the qualitative behavior of $\overline{R(t)^2}$?

(h) Compute $\overline{R(t)^2}$ for an equilibrium configuration corresponding to a dilute gas. Is $\overline{R(t)^2}$ proportional to t for small times? Do the particles diffuse over short time intervals? □

Another physically important property is the *velocity autocorrelation function* $C(t)$. Suppose that particle i has velocity $\mathbf{v}_i$ at time $t = 0$. If there was no net force on particle i, its velocity would remain constant. However, its interactions with other particles in the fluid will change the particle's velocity, and we expect that after several collisions, its velocity will not be strongly correlated with its velocity at an earlier time. We define $C(t)$ as

$$C(t) = \frac{1}{v_0^2}\overline{\mathbf{v}_i(t) \cdot \mathbf{v}_i(0)} \tag{8.40}$$

where $v_0^2 = \overline{\mathbf{v}_i(0) \cdot \mathbf{v}_i(0)} = kTd/m$. We have defined $C(t)$ such that $C(t = 0) = 1$. As in our discussion of the mean square displacement, the average in (8.40) is over all possible time origins. Better results would be obtained by averaging over all particles. For large time differences t, we expect $\mathbf{v}_i(t)$ to be independent of $\mathbf{v}_i(0)$, and hence, $C(t) \to 0$ for $t \to \infty$. (We have implicitly assumed that $\overline{\mathbf{v}}_i(t) = 0$.)

It can be shown that the self-diffusion coefficient defined by (8.39) can be related to the integral of $C(t)$:

$$D = v_0^2 \int_0^\infty C(t)\, dt. \tag{8.41}$$

Other transport coefficients such as the shear viscosity and the thermal conductivity can also be expressed as an integral over a corresponding autocorrelation function. The qualitative properties of the velocity autocorrelation function are explored in Problem 8.20.

Problem 8.20. The velocity autocorrelation function

(a) Modify your hard disk or molecular dynamics program so that the velocity of a particular particle is saved at regular time intervals. Then modify method computeR2 so that you can compute $C(t)$. The following code might be useful.

```
for(int timeDiff = 1; timeDiff < maxTimeDiff; timeDiff++) {
   for(int time0 = 0; time0 < maxTime0 - timeDiff; time0++) {
      correl[timeDiff] += vxSave[time0 + timeDiff]*vx[time0];
      correl[timeDiff] += vySave[time0 + timeDiff]*vy[time0];
      normalization[timeDiff]++;
   }
}
```

First compute $C(t)$ for a relatively low density system. Plot $C(t)$ versus t and describe its qualitative behavior. Does it more or less decay exponentially?

(b) Increase the density and compute $C(t)$ again. How does the qualitative behavior of $C(t)$ change? Why does $C(t)$ become negative after a relatively short time?

(c)* To obtain quantitative results, modify your program so that $C(t)$ is averaged over all particles. Compute $C(t)$ for time differences in the range 10–40 mean collision times and densities that are about a factor of two less than maximum close packing. Also choose $N \geq 256$. If you are careful, you will be able to observe that $C(t)$ decays as t^{-1} for very long time differences. This long-time tail is due to hydrodynamic effects; that is, part of the velocity of a particle is stored in a microscopic vortex that dies off very slowly. The existence of this tail was first found by simulations and implies that the self-diffusion constant is not defined in two dimensions because the integral (8.41) does not exist (the integral diverges for large t). In three dimensions, $C(t) \sim t^{-3/2}$ and the self-diffusion coefficient is well defined.

(d) Compute $C(t)$ for an equilibrium solid. Plot $C(t)$ versus t and describe its qualitative behavior. Explain your results in terms of the oscillatory motion of the particles about their lattice sites.

(e) Contrast the behavior of the mean square displacement, the velocity autocorrelation function, and the radial distribution function in the solid and fluid phases and explain how these quantities can be used to indicate the nature of the phase. □

8.11 Extensions

The primary goals of this chapter have been to introduce the method of molecular dynamics, some of the concepts of statistical mechanics and kinetic theory, and the qualitative behavior of systems of many particles. Although we found that simulations of systems as small as 64 particles show some of the qualitative properties of macroscopic systems, we would need to simulate larger systems to obtain quantitative results. Most simulations of systems with simple interactions require only several hundred to several thousand particles to obtain reliable results for equilibrium quantities such as the equation of state. How do we know if the size of our system is sufficient to yield quantitative results? The simple answer is to repeat the simulation for a diferent value of N. In the same spirit, you can determine if your runs are long enough to give statistically meaningful averages.

In general, the most time consuming parts of a molecular dynamics simulation are generating an appropriate initial configuration and doing the bookkeeping necessary for the force and energy calculations. If the force is short range, there are several ways to reduce the equilibration time. For example, suppose we want to simulate a system of 864 particles in three dimensions. We can first simulate a system of 108 particles and allow the small system to come to equilibrium at the desired temperature. After equilibrium has been established, the small system can be replicated twice in each direction to generate the desired system of 864 particles. All of the velocities are reassigned at random using the Maxwell–Boltzmann distribution. Equilibration of the new system is usually established quickly.

The computer time required for our simple molecular dynamics program is order N^2 for each time step. The reason for this N^2- dependence is that the energy and force calculations require sums over all $\frac{1}{2}N(N-1)$ pairs of particles. If the interactions are short range, the time required for these sums can be reduced to approximately order N. The idea is to take advantage of the fact that many pairs of particles are separated by a distance much greater than the effective range r_c of the interparticle interaction. For example, if the distance between two particles interacting via the Lennard–Jones potential is sufficiently large, the magnitude of the potential is so small that it can be neglected. Popular choices for the cutoff r_c are 2.3σ and 2.5σ. The use of a cutoff is equivalent to assuming that $u(r)$ in (8.2) is given by the usual Lennard–Jones form for $r < r_c$ and is zero for $r > r_c$. However, this use of a cutoff implies that $u(r)$ has a discontinuity at $r = r_c$, which means that whenever a particle pair "crosses" the cutoff distance, the energy jumps, thus affecting the apparent energy conservation. To avoid this problem, it is a good idea to modify the potential so as to eliminate the discontinuity in both $u(r)$ and the force $-du/dr$. Hence, we write

$$\tilde{u}(r) = u(r) - u(r_c) - \left.\frac{du(r)}{dr}\right|_{r=r_c} (r - r_c) \tag{8.42}$$

where $u(r)$ is the usual Lennard–Jones potential.

The use of the interparticle potential (8.42) to calculate the force and the energy requires considering only those pairs of particles whose separation is less than r_c. Because testing whether each pair satisfies this criterion is an order N^2 calculation, we have to limit the number of pairs tested. One way is to divide the box into small cells and to only compute the distance between particles that are in the same cell or in nearby cells. Another method is to maintain a list for each particle of its neighbors whose separation is less than a distance r_n, where r_n is chosen to be slightly greater than r_c. The idea is to use the same list of neighbors for several time steps (usually 10–20) so that the time consuming job of updating the list of neighbors does not have to be done too often. The cell method and the neighbor list method do not become efficient until N is approximately a few hundred particles.

Usually, the neighbor list leads to the consideration of fewer particle pairs in the force calculation than the cell list. We provide a method to compute the neighbor list below. A more efficient approach is to use cells to construct the neighbor list.

```
public void computeNeighborList() {
   for(int i = 0; i < N-1; i++) {
      numberInList[i] = 0;
      for(int j = i+1; j < N; j++) {
         double dx = separation(x[i] - x[j],Lx);
         double dy = separation(y[i] - y[j],Ly);
         double r2 = dx*dx + dy*dy;
         if(r2 < r2ListCutoff) {
            list[i][numberInList[i]] = j;
            numberInList[i]++;
         }
      }
   }
}
```

To use this list in method `computeAcceleration`, we replace the for loops by

```
for (int i = 0; i < N-1; i++) {
   for (int k = 0; k < numberInList[i]; k++) {
      int j = list[i][k];
   }
}
```

The method `computeNeighborList` should be called before a particle may have moved a distance equal to the difference $r_n - r_c$. This time depends on the density and the temperature. For dense systems a reasonable value for r_n is 2.7σ. Simulations of small systems can be used to determine the time between calls of `computeNeighborList`.

Note that in method `computeNeighborList`, only particles $j > i$ are included in `list[i]`. In Section 15.10 we will consider Monte Carlo simulations where a particle is chosen at random, and its potential energy of interaction must be computed. In this case we cannot take advantage of Newton's third law, and a neighbor list must be created for all particles that are within a distance r_n of particle i.

***Problem 8.21.** Neighbor lists

(a) Simulate a system of $N = 64$ Lennard–Jones particles in a square cell with $L = 10$ at a temperature $T = 2.0$. After the system has reached equilibrium, determine the shortest time for any particle to move a distance equal to 0.2. Use half this time in the rest of the program as the time between updates of the neighbor list.

(b) Run your simulation with and without the neighbor list starting from identical initial configurations. Choose $r_c = 2.3$ and use the modified potential given in (8.42). Calculate $g(r)$, the pressure, the heat capacity (see Problem 8.8), and the temperature. Make sure your results are identical. Compare the amount of CPU time with and without the use of the neighbor list.

(c) Repeat part (b) with $N = 256$ but with the same density and total energy. You can adjust the total energy by scaling the initial velocities. Increase N until the CPU time for the neighbor list version is faster.

(d) Continue increasing the number of particles by a factor of four, but only use the program with the neighbor list. Determine the CPU time required for one time step as a function of N. □

So far we have discussed molecular dynamics simulations at fixed energy, volume, and number of particles. Molecular dynamics simulations at fixed temperature are discussed in Project 8.24. It is also possible to modify the dynamics so as to do molecular dynamics simulations at constant pressure and to do simulations in which the shape of the cell is determined by the dynamics, rather than imposed by the program. Such a simulation is essential for the study of solid-to-solid transitions where the major change is the shape of the crystal.

In addition to these algorithmic advances, there is much more to learn about the properties of the system. For example, how are transport properties such as the viscosity and the thermal conductivity related to the trajectories? We have also not discussed one of the most fundamental properties of a many-body system, namely, its entropy. In brief, not all macroscopic properties of a many-body system, including the entropy, can be defined as a time average over some function of the phase space coordinates of the particles (but see Ma). However, changes in the entropy can be computed by using thermodynamic integration.

The fundamental limitation of molecular dynamics is the existence of *multiple time scales*. We must choose the time step Δt to be smaller than any physical time scale in the system. For a solid, the smallest time scale is the period of the oscillatory motion of individual particles about their equilibrium positions. If we want to know how the solid responds to the addition of an interstitial particle or a vacancy, we would have to run for millions of small time steps for the vacancy to move several interparticle distances. Although this particular problem can be overcome by using a faster computer, there are many problems for which no imaginable supercomputer would be sufficient. One of the biggest current challenges is the *protein folding problem*. The biological function of a protein is determined by its three-dimensional structure which is encoded by the sequence of amino acids in the protein. At present, we know little about how the protein forms its three-dimensional structure. Such formidable computational challenges remind us that we cannot simply put a problem on a computer and let the computer tell us the answer. In particular, for many problems, molecular dynamics methods need to be complemented by other simulation methods, especially Monte Carlo methods (see Chapter 15).

The emphasis in current applications of molecular dynamics is shifting from studies of simple equilibrium fluids to studies of more complex fluids and nonequilibrium systems. For example, how does a solid form when the temperature of a liquid is lowered quickly? How does a crack propagate in a brittle solid? What is the nature of the glass transition? Molecular dynamics and related methods will play an important role in aiding our understanding of these and many other problems.

8.12 Projects

Many of the pioneering applications of molecular dynamics were done on relatively small systems. It is interesting to peruse the research literature of the past three decades to see how much physical insight was obtained from these simulations. Many research-level problems can be generated by first reproducing previously published work and then extending the work to larger systems or longer run times to obtain better statistics. Many related projects are discussed in Chapter 15.

Project 8.22. The classical Heisenberg model of magnetism

Magnetism is intrinsically a quantum phenomenon. One common model of magnetism is the Heisenberg model which is defined by the Hamiltonian or energy function:

$$H = -J \sum_{<ij>} \mathbf{S}_i \cdot \mathbf{S}_j \tag{8.43}$$

where $\mathbf{S}_i$ is the spin operator at the ith lattice site. The sum is over nearest neighbor sites of the lattice, and the (positive) coupling constant J is a measure of the strength of the interaction between spins. The negative sign indicates that the lowest energy state is ferromagnetic. The magnetic moment of a particle on a site is proportional to the particle's spin, and the proportionality constant is absorbed into the constant J.

For many models of magnetism, such as the Ising model (see Section 15.5), there is no obvious dynamics. However, for the Heisenberg model we can motivate a dynamics using the standard rule for the time evolution of an operator given in quantum mechanics texts. For simplicity, we will consider a one-dimensional lattice. The equation for the time development becomes (see Slanič et al.)

$$\frac{d\mathbf{S}_i}{dt} = J\mathbf{S}_i \times (\mathbf{S}_{i-1} + \mathbf{S}_{i+1}). \tag{8.44}$$

In general, $\mathbf{S}$ in (8.44) is an operator. However, if the magnitude of the spin is sufficiently large, the system can be treated classically, and $\mathbf{S}$ can be interpreted as a three-dimensional unit vector. The dynamics in (8.44) conserves the total energy given in (8.43) and the total magnetization, $\mathbf{M} = \sum_i \mathbf{S}_i$.

We can simulate the classical Heisenberg magnet using an ODE solver to solve the first-order differential equation (8.44).

(a) Explain why there is no obvious way to determine the mean temperature of this system.

(b) Write a program to simulate the Heisenberg model on a one-dimensional lattice using periodic boundary conditions. Choose $J = 1$ and $N \geq 100$. Use the RK4 ODE solver, and plot the energy and magnetization as a function of time. These two quantities should be constant within the accuracy of the ODE solver. Also, plot each component of the spin versus position or draw a three-dimensional representation of the spin at each site so that you can visualize the state of the system.

(c) Begin with all spins in the positive z direction, except for one spin pointing in the negative z direction. Use $N \geq 1000$. Define the energy of spin i as $\epsilon_i = -\mathbf{S}_i \cdot (\mathbf{S}_{i-1} + \mathbf{S}_{i+1})/2$. Plot the local energy as a function of i. Describe how the local energy diffuses. What patterns do you observe? Do the locations of the peaks in the local energy move with a constant speed?

(d) One of the interesting dynamical phenomena we can explore is that of *spin waves*. Begin with all $S_{z,i} = 1$ except for a group of 20 spins, where $S_{x,i} = A\cos ki$, $S_{y,i} = A\sin ki$, and $S_{z,i} = \sqrt{1 - S_{x,i}^2 + S_{y,i}^2}$. Choose $A = 0.2$ and $k = 1$. Describe the motion of the spins. Compute the mean position of this spin wave defined by $x = \sum_i i(1 - S_{z,i})$. Show that x changes linearly with time indicating a constant spin wave velocity. Vary k and A to determine what effect their values have on the speed of the spin wave.

(e) Read about sympletic algorithms in the article by Tsai, Lee, and Landau and write your own ODE solver for one of them. Compare your results to the results you found for the RK4 algorithm. Is the total energy better conserved for the same value of the time step? □

Project 8.23. Single particle metrics and ergodic behavior

As mentioned in Section 8.7, the quasi-ergodic hypothesis assumes that time averages and ensemble averages are identical for a system in thermodynamic equilibrium. The assumption is that if we run a molecular dynamics simulation for a sufficiently long time, then the dynamical trajectory will fill the accessible phase space.

One way to confirm the quasi-ergodic hypothesis is to compute an ensemble average by simulating many independent copies of the system of interest using different initial configurations. Another way is to simulate a very large system and compare the behavior of different parts. A more direct measure of ergodicity (see Thirumalai and Mountain) is based on a comparison of the time averaged quantity $\overline{f_i(t)}$ of f_i for particle i to its average for all other particles. If the system is ergodic, then all particles see the same average environment, and the time average $\overline{f_i(t)}$ for each particle will be the same if t is sufficiently long. Note that $\overline{f_i(t)}$ is the average of the quantity f_i over the time interval t and not the value of f_i at time t. The time average of f_i is defined as

$$\overline{f_i(t)} = \frac{1}{t}\int_0^t f(t')\,dt' \tag{8.45}$$

and the average of $\overline{f_i(t)}$ over all particles is written as

$$\langle f(t)\rangle = \frac{1}{N}\sum_{i=1}^{N} \overline{f_i(t)}. \tag{8.46}$$

One of the physical quantities of interest is the energy of a particle e_i defined as

$$e_i = \frac{p_i^2}{2m_i} + \frac{1}{2}\sum_{i \neq j} u(r_{ij}). \tag{8.47}$$

The factor of 1/2 is included in the potential energy term in (8.47) because the interaction energy is shared between pairs of particles. The above considerations lead us to define the energy *metric*, $\Omega_e(t)$, as

$$\Omega_e(t) = \frac{1}{N}\sum_{i=1}^{N}\Big[\overline{e_i(t)} - \langle e(t)\rangle\Big]^2. \tag{8.48}$$

(a) Compute $\Omega_e(t)$ for a system of Lennard–Jones particles at a relatively high temperature. Determine $e_i(t)$ at time intervals of 0.5 or less and average Ω_e over as many time origins as possible. If the system is ergodic over the time interval t, then it can be shown that $\Omega_e(t)$ decreases as $1/t$. Plot $1/\Omega_e(t)$ versus t. Do you find that $1/\Omega_e(t)$ eventually behaves linearly with t? Nonergodic behavior might be found by rapidly reducing the kinetic energy (a temperature quench) and obtaining an amorphous solid or glass rather than a crystalline solid. However, it would be necessary to consider three-dimensional rather than two-dimensional systems because the latter system forms a crystalline solid very quickly.

(b) Another quantity of interest is the velocity metric Ω_v:

$$\Omega_v(t) = \frac{1}{dN}\sum_{i=1}^{N}\Big[\overline{\mathbf{v}_i(t)} - \langle \mathbf{v}(t)\rangle\Big]^2. \tag{8.49}$$

The factor of $1/d$ in (8.49) is included because the velocity is a vector with d components. If we choose the total momentum of the system to be zero, then $\langle \mathbf{v}(t)\rangle = 0$, and we can write (8.49) as

$$\Omega_v(t) = \frac{1}{dN}\sum_{i=1}^{N}\overline{\mathbf{v}_i(t)}\cdot\overline{\mathbf{v}_i(t)}. \tag{8.50}$$

As we will see, the time dependence of $\Omega_v(t)$ is not a good indicator of ergodicity, but can be used to determine the diffusion coefficient D. We write

$$\overline{\mathbf{v}_i(t)} = \frac{1}{t}\int_0^t \mathbf{v}_i(t')\,dt' = \frac{1}{t}\Big[\mathbf{r}_i(t) - \mathbf{r}_i(0)\Big]. \tag{8.51}$$

If we substitute (8.51) into (8.50), we can express the velocity metric in terms of the mean square displacement:

$$\Omega_v(t) = \frac{1}{dNt^2}\sum_{i=1}^{N}\Big[\mathbf{r}_i(t) - \mathbf{r}_i(0)\Big]^2 = \frac{\overline{R^2(t)}}{dt^2}. \tag{8.52}$$

The average in (8.52) is over all particles. If the particles are diffusing during the time interval t, then $\overline{R^2(t)} = 2dDt$ and

$$\Omega_v(t) = 2D/t. \tag{8.53}$$

From (8.53) we see that $\Omega_v(t)$ goes to zero as $1/t$ as claimed in part (a). However, if the particles are localized (as in a crystalline solid and a glass), then $\overline{R^2}$ is bounded for all t and $\Omega_v(t) \sim 1/t^2$. Because a crystalline solid is ergodic and a glass is not, the velocity metric is not a good measure of the lack of ergodicity. Use the t-dependence of $\Omega_v(t)$ in (8.53) to determine D for the same configurations as in Problem 8.19. □

Project 8.24. Constant temperature molecular dynamics

In the molecular dynamics simulations we have discussed so far, the energy is constant up to truncation, discretization, and floating point errors, and the temperature fluctuates. However, sometimes it is more convenient to do simulations at constant temperature. In Chapter 15 we will see how to simulate systems at constant T, V, and N (the canonical ensemble) by using Monte Carlo methods. However, we can also do constant temperature simulations by modifying the dynamics.

A crude way of maintaining a constant temperature is to rescale the velocities after every time step to keep the mean kinetic energy per particle constant. This approach is equivalent to a constant temperature simulation when $N \to \infty$. However, the fluctuations of the kinetic energy can be non-negligible in small systems. For such systems keeping the total kinetic energy constant in this way is not equivalent to a constant temperature simulation.

A better way of maintaining a constant temperature is based on imagining that every particle in the system is connected to a much larger system called a *heat bath*. The heat bath is sufficiently large so that it has a constant temperature even if it loses or gains energy. The particles in the system of interest occasionally collide with particles in this heat bath. The effect of these collisions is to give the particles random velocities with the desired probability distribution (see Problem 8.6). We first list the algorithm and give its rationale later. Add the following statements to method `step` after all the particles have been moved.

Listing 8.22: Andersen thermostat.

```
for (int i = 0; i < N; i++) {
   if (Math.random() < collisionProbability) {
      double r1 = Math.random();
      double r2 = Math.random()*2.0*Math.PI;
        // vx
      state[4*i+1] = Math.sqrt(-2.0*temperature*Math.log(r1))*Math.cos(r2);
        // vy
      state[4*i+3] = Math.sqrt(-2.0*temperature*Math.log(r1))*Math.sin(r2);
   }
}
```

The parameter `collisionProbability` is much less than unity and determines how often there is a collision with the heat bath. This way of maintaining constant temperature is known as the *Andersen thermostat*.

(a) Do a constant energy simulation as before, using an initial configuration for which the desired temperature is equal to 1.0. Make sure the total momentum is zero. Choose $N = 64$ and place the particles initially on a triangular lattice with $L_x = 10$ and $L_y = \sqrt{3}L_x/2$. Plot the instantaneous temperature defined as in (8.5) and compute the average temperature. Estimate the magnitude of the temperature fluctuations. Repeat your simulation for some other initial configurations.

(b) Modify your program to use the Andersen thermostat at a constant temperature set equal to 1.0. Set `collisionProbability` = 0.0001. Repeat the calculations of part (a) and compare them. Discuss the differences. Do the results change significantly?

(c) Modify your program to do a simple constant kinetic energy ensemble where the velocities are rescaled after every time step so that the total kinetic energy does not change. What is the final temperature now? How do your results compare with parts (a) and (b)? Are the differences in the computed thermodynamic averages statistically significant?

(d) Compute the velocity probability distribution for each case. How do they compare? Consider `collisionProbability` = 0.001 and 0.00001.

(e) A deterministic algorithm for constant temperature molecular dynamics is the Nosé–Hoover thermostat. The idea is to introduce an additional degree of freedom s that plays the role of the heat bath. The derivation of the appropriate equations of motion is an excellent example of the Lagrangian formulation of mechanics. The equations of motion of Nosé–Hoover dynamics are

$$\frac{d\mathbf{p}_i}{dt} = \mathbf{F}_i(t) - s\mathbf{p}_i \tag{8.54}$$

$$\frac{ds}{dt} = \frac{1}{M}\Big[\sum_i \frac{p_i^2}{m_i} - dNkT\Big] \tag{8.55}$$

where T is the desired temperature, and M is a parameter that can be interpreted as the mass associated with the extra degree of freedom. Equation (8.54) is similar to Newton's equations of motion with an additional friction term. However, the coefficient s can be positive or negative. Equation (8.55) defines the way s is changed to control the temperature. Apply the Nosé–Hoover algorithm to simulate a simple harmonic oscillator at constant temperature. Plot the phase space trajectory. If the energy was constant, the trajectory would be an ellipse. How does the shape of the trajectory depend on M? Choose M so that the period of any oscillations due to the finite value of M is much longer than the period of the system. □

Project 8.25. Simulations on the surface of a sphere

Because of the long-range nature of the Coulomb potential, we have to sum all the periodic images of the particles to compute the force on a given particle. Although there are special methods to do these sums so that they converge quickly (Ewald sums), the simulation of systems of charged particles is more difficult than systems with short-range potentials. An alternative approach that avoids periodic boundary conditions is to not have any boundaries at all. For example, if we wish to simulate a two-dimensional system, we can consider the motion of the particles on the surface of a sphere. If the radius of the sphere is sufficiently large, the curvature of the surface can be neglected. Of course, there is a price—the coordinate system is no longer Cartesian.

Although this approach can also be applied to systems with short-range interactions, it is more interesting to apply it to charged particles. The simplest system of interest is a model of charged particles moving in a uniform background of opposite charge to ensure overall charge neutrality, the one-component plasma (OCP). In two dimensions this system is a simplified model of electrons on the surface of liquid Helium. The properties of the OCP are determined by the dimensionless parameter Γ given by the ratio of the potential energy between nearest neighbor particles to the mean kinetic energy of a particle, $\Gamma = (e^2/a)/kT$, where $\rho\pi a^2 = 1$ and ρ is the number density. Systems with $\Gamma >> 1$ are called strongly coupled. For $\Gamma \sim 100$ in two dimensions, the system forms a solid. Strongly coupled one-component plasmas in three dimensions are models of dense astrophysical matter.

Assume that the origin of the coordinate system is at the center of the sphere and that $\mathbf{u}_i$ is a unit vector from the origin to the position of particle i on the sphere. Then $R\theta_{ij}$ is the length of the chord joining particle i and j, where $\cos\theta_{ij} = \mathbf{u}_i \cdot \mathbf{u}_j$. Newton's equation of motion for the ith electron has the form

$$m\ddot{\mathbf{u}}_i = -\frac{e^2}{R^2}\sum_{j\neq i}\frac{1}{\theta_{ij}^2\sin\theta_{ij}}[\mathbf{u}_j - (\cos\theta_{ij}\mathbf{u}_i]. \tag{8.56}$$

Note that the unit vector $\mathbf{w}_{ij} = [\mathbf{u}_j - (\cos\psi_{ij}\mathbf{u}_i]/\sin\theta_{ij}$ is orthogonal to $\mathbf{u}_i$. In addition, we must take into account that the particles must stay on the surface of the sphere, so there is an additional force on particle i toward the center of magnitude $m|\dot{\mathbf{u}}_i|^2/R$.

(a) What are the appropriate units for length, time, and the self-diffusion constant?

(b) Write a program to compute the velocity correlation function given by

$$C(t) = \frac{1}{v_0^2}\overline{\dot{\mathbf{u}}(t) \cdot \dot{\mathbf{u}}(0)} \tag{8.57}$$

where $v_0^2 = \overline{\mathbf{u}(0) \cdot \mathbf{u}(0)}$. To compute the self-diffusion constant D, we let $\cos\theta(t) = \mathbf{u}(t) \cdot \mathbf{u}(0)$, so that $R\theta$ is the circular arc from the initial position of a particle to its position on the sphere at time t. We then define

$$D(t) = \frac{1}{a^2}\frac{\overline{\theta^2(t)}}{4t} \tag{8.58}$$

where D and t are dimensionless variables. The self-diffusion constant D corresponds to the limit $t \to \infty$. Choose $N = 104$ and a radius R corresponding to $\Gamma \approx 36$ as in the original simulations by Hansen et al. and then consider bigger systems. Can you conclude that the self-diffusion exists for the two-dimensional OCP?

(c)* Use a similar procedure to compute the velocity autocorrelation function and the self-diffusion constant D for a two-dimensional system of Lennard–Jones particles. Can you conclude that the self-diffusion exists for this two-dimensional system? □

Project 8.26. Granular matter

Recently, physicists have become very interested in granular matter such as sand. The key difference between molecular systems and granular systems is that the interparticle interactions in the latter are inelastic. The lost energy goes into the internal degrees of freedom of a grain and ultimately is dissipated. From the point of view of the motion of the granular particles, the energy is lost. Experimentalists have studied models of granular material composed of small steel balls or glass beads using sophisticated imaging techniques that can track the motion of individual particles. There have also been many complementary computer simulation studies.

What are some of the interesting properties of granular matter? Because the interactions are inelastic, granular particles will ultimately come to rest unless there is an external source of energy, usually a vibrating wall or gravity (for example, the fall of particles through a funnel). When granular particles come to rest, they can form a granular solid that is different than molecular solids. One difference is that there frequently exists a complex network of force lines within the solid. In addition, unlike ordinary liquids, the pressure does not increase with depth because the walls of the container help support the grains. As a consequence, sand flowing out of an aperture flows at a constant rate independent of the height of the sand above the aperture. For this reason sand is used in hour glasses. Another interesting property is that under some conditions, the large grains in a mixture of large and small grains can move to the top while the container is being vibrated—the "Brazil nut" effect. Under other conditions, the large grains might move to the bottom. What happens depends on the size and density of the large grains compared to the small grains (see Sanders et al.).

It is also known that there is a critical angle for the slope of a sand pile, above which the sand pile is unstable. This slope is called the angle of repose. These and many other effects have been studied using theoretical, computational, and experimental techniques.

The first step in simulating granular matter is to determine the effective force law between particles. For granular gases the details of the force do not influence the qualitative results, as long as the force is purely repulsive and short range, and there is some mechanism for dissipating energy. Common examples of force laws are spring-like forces with stiff spring constants and hard disks with inelastic collisions. For simplicity, we will consider the Lennard–Jones potential with a cut off at $r_c = 2^{1/6}$ so that the force is always repulsive. To remove energy during a collision, we will introduce a viscous damping force given by

$$f_{ij} = -\gamma(\mathbf{v}_{ij} \cdot \mathbf{r}_{ij})\frac{\mathbf{r}_{ij}}{r_{ij}^2} \tag{8.59}$$

where the viscous damping coefficient γ equals 100 in reduced units. A more realistic force model necessary for granular flow problems is given in Hirchfeld et al.

(a) Modify class `LJParticles` so that the cutoff is at $2^{1/6}$. Is the total energy conserved. Include a viscous damping force as in (8.59) and plot the kinetic energy per particle versus time. We will define the kinetic temperature to be the mean kinetic energy per particle. Why does this definition of temperature not have the same significance as the temperature in molecular systems in which the energy is conserved? Choose $N = 64$, $L = 20$, and $\Delta t = 0.001$. Begin with a random configuration and initial kinetic temperature equal to 10. How long does it take for the kinetic temperature to decrease to 10% of its initial value? Describe the spatial distribution of the particles at this time.

(b) Compute the mean kinetic temperature versus time averaged over three runs. What functional form describes your results for the mean kinetic temperature at long times?

(c) To prevent "granular collapse" where the particles ultimately come to rest, we need to add energy to the system. The simplest way of doing so is to give random kicks to randomly selected particles. You can use the same algorithm we used to set the initial velocities in `LJParticles`:

```
int i = (int)(N*math.random())  // selects random particle
// use to generate Gaussian distribution
double r = Math.random();
double a = -Math.log(r);
double theta = 2.0*Math.PI*Math.random();
// assign velocities according to Maxwell-Boltzmann distribution
// using Box-Muller method
state[4*i+1] = Math.sqrt(2.0*desiredKE*a)*Math.cos(theta);    // vx
state[4*i+3] = Math.sqrt(2.0*desiredKE*a)*Math.sin(theta);    // vy
```

(The Box–Muller method is described in Section 11.5.) Assume that at each time step one particle is chosen at random and receives a random kick. Adjust `desiredKE` so that the mean kinetic energy per particle remains roughly constant at about 5.0. Compute the velocity distribution function for each component of the velocity. Compare this distribution on the same plot to the Gaussian distribution:

$$p(v_x) = \frac{1}{\sqrt{2\pi\sigma^2}}e^{-(v_x-\langle v_x\rangle)^2/2\sigma^2} \tag{8.60}$$

where $\sigma^2 = \langle v_x^2\rangle - \langle v_x\rangle^2$. Is the velocity distribution function of the system a Gaussian? If not, give a physical explanation for the difference. □

Appendix 8A: Reading and Saving Configurations

For most of the problems in this chapter, qualitative results can be obtained fairly quickly. However, in research applications, the time for running a simulation is likely to be much longer than a few minutes and runs that require days or even months are not uncommon. In such cases it is important to be able to save the intermediate configurations to prevent the potential loss of data in the case of a computer crash or power failure.

Also, in many cases it is easier to save the configurations periodically and then use a separate program to analyze the configurations and compute the quantities of interest. In addition, if we wish to compute averages as a function of a parameter, such as the temperature, it is convenient to make small changes in the temperature and use the last configuration from the previous run as the initial configuration for the simulation at the new temperature.

The standard Java API has methods for reading and writing files. The usual way of saving a configuration is to use these methods to simply write all the positions and velocities as numbers into a file. Additional simulation parameters and information about the configuration would be saved using a custom format. Although this approach is the traditional one for data storage, the use of a custom format means that you might not remember the format later, and sharing data between programs and other users becomes more difficult.

An alternative is to use a more structured and widely shared format for storing data. The Open Source Physics library has support for the Extensive Markup Language (XML). The XML format offers a number of advantages for computational physics: clear markup of input data and results, standardized data formats, and easier exchange and archival stability of data. In simple terms the main advantage of XML is that it is a human readable format; just by looking at an XML file you can get an idea of the nature of the data.

The XML classes in the Open Source Physics library can be understood by reading the `XML-ExampleApp` example. The XML API is very similar to the control API. For example, we use `setValue` to add data to an XML control, and we use `getInt`, `getDouble`, and `getString` to read data. We start by importing the necessary definitions from the controls package and defining the `main` method for the `ExampleXMLApp` class. Note that `XMLControl` defines an interface and `XMLControlElement` defines an implementation of this interface.

```
import org.opensourcephysics.controls.XMLControl;
import org.opensourcephysics.controls.XMLControlElement;

public class ExampleXMLApp {
   public static void main(String[] args) {
   ...
}
```

The following Java statements are placed in the body of the `main` method. An empty XML document is created using an `XMLControl` object by calling the `XMLControlElement` constructor without any parameters.

```
XMLControl xmlOut = new XMLControlElement();
```

Invoking the control's `setValue` method creates an XML element consisting of a tag and a data value. The tag is the first parameter and the data to be stored is the second. Data that can be stored includes numbers, number arrays, and strings. Because the tag is unique, the data can later be retrieved from the control using the appropriate get method.

```
xmlOut.setValue("comment", "An XML description of an array.");
xmlOut.setValue("x positions", new double[]{1,3,4});
xmlOut.setValue("x velocities", new double[]{0,-1,1});
```

Once the data has been stored in an `XMLControl` object, it can be exported to a file by calling the `write` method. In this example, the name of the file is `MDconfiguration.xml`.

```
xmlOut.write("MDconfiguration.xml");
```

An `XMLControl` can also be used to read XML data from a file. In the next example, we will read from the file that we just saved. We start by instantiating a new `XMLControl` named `xmlIn`.

```
XMLControl xmlIn = new XMLControlElement("particle_configuration.xml");
```

The new `XMLControl` object `xmlIn` contains the same data as the object we saved, `xmlOut`. Its data can be accessed using a tag name. Note that the `getObject` method returns a generic `Object` and must be cast to the appropriate data type.

```
System.out.println(xmlIn.getString("comment"));
double[] xPos = (double[])xmlIn.getObject("x positions");
double[] xVel = (double[])xmlIn.getObject("x velocities");
for(int i = 0; i < xPos.length; i++) {
   System.out.println("x[i] = " + xPos[i] +" vx[i] = " + xVel[i]);
}
```

Exercise 8.27. Saving XML data

(a) Combine the above statements to create a working `XMLControlApp` class. Examine the saved data using a text editor. Describe how the parameters are stored.

(b) Run `HardDisksApp` and save the control's configuration using the Save As item under the file menu in the toolbar. Examine the saved file using a text editor and describe how this file is different from the file you generated in part (a).

(c) What is the minimum amount of information that must be stored in a configuration file to specify the current `HardDisks` state?

(d) Add custom buttons to `HardDisksApp` to store and load the current `HardDisks` state. Test your code by showing that quantities, such as the temperature, remain the same if a configuration is stored and reloaded. □

Open Source Physics user interfaces, such as a `SimulationControl`, store a program's configuration in two steps. During the first step, parameters from the graphical user interface are stored. During the second step, the model is given the opportunity to store runtime data using an `ObjectLoader`. Study the `LJParticlesLoader` class and note how storing and loading are done in the `saveObject` and `loadObject` methods, respectively. You will adapt this `ObjectLoader` to store `HardDisks` data in Problem 8.28. Additional information about how Open Source Physics applications store XML-based configurations is provided in the *Open Source Physics Users Guide*.

Problem 8.28. Hard disk configuration

(a) Create a `HardDisksLoader` class that stores the `HardDisks` runtime data.

(b) Add the `getLoader` method to `HardDisksApp` and test the loader.

```
public static XML.ObjectLoader getLoader() {
   return new HardDisksLoader();
}
```

Method `XML.ObjectLoader getLoader` allows the `SimulationControl` to obtain the `HardDisksLoader`, which will be used to store the runtime data. Data written by the loader's `saveObject` method will be included in the output file when the user saves a program configuration. Describe how the initialization parameters and the runtime data are separated in the XML file. □

Because XML allows for the creation of custom tags, various companies and professional organizations have defined other XML grammars, such as *MathML*. See <`http://xml.comp-phys.org/`> for another example of the use of XML in computational physics.

References and Suggestions for Further Reading

One of the best ways of testing your programs is by comparing your results with known results. The website, <`www.cstl.nist.gov/lj`>, maintained by the National Institute of Standards and Technology (U.S.) provides some useful benchmark results for the Lennard–Jones fluid.

Farid F. Abraham, "Computational statistical mechanics: Methodology, applications and supercomputing," Adv. Phys. **35**, 1–111 (1986). The author discusses both molecular dynamics and Monte Carlo techniques.

B. J. Alder and T. E. Wainwright, "Phase transition for a hard sphere system," J. Chem. Phys. **27**, 1208–1209 (1957).

M. P. Allen and D. J. Tildesley, *Computer Simulation of Liquids* (Clarendon Press, 1987). A classic text on molecular dynamics and Monte Carlo methods.

Jean–Louis Barrat and Jean–Pierre Hansen, *Basic Concepts for Simple and Complex Liquids* (Cambridge University Press, 2003). Also see Jean–Pierre Hansen and Ian R. McDonald, *Theory of Simple Liquids*, 2nd ed. (Academic Press, 1986). Excellent graduate level texts that derive most of the theoretical results mentioned in this chapter.

Kurt Binder, Jürgen Horbach, Walter Kob, Wolfgang Paul, and Fathollah Varnik, "Molecular dynamics simulations," J. Phys.: Condens. Matter **16**, S429–S453 (2004).

R. P. Bonomo and F. Riggi, "The evolution of the speed distribution for a two-dimensional ideal gas: A computer simulation," Am. J. Phys. **52**, 54–55 (1984). The authors consider a system of hard disks and show that the system evolves to the Maxwell–Boltzmann distribution.

J. P. Boon and S. Yip, *Molecular Hydrodynamics* (Dover Publications,1991). Their discussion of transport properties is an excellent supplement to our brief discussion.

Giovanni Ciccotti and William G. Hoover, editors, *Molecular-Dynamics Simulation of Statistical-Mechanics Systems* (North–Holland, 1986).

Giovanni Ciccotti, Daan Frenkel, and Ian R. McDonald, editors, *Simulation of Liquids and Solids* (North–Holland, 1987). A collection of reprints on the simulation of many body systems. Of particular interest are B. J. Alder and T. E. Wainwright, "Phase transition in elastic disks," Phys. Rev. **127**, 359–361 (1962) and earlier papers by the same authors; A. Rahman, "Correlations in the motion of atoms in liquid argon," Phys. Rev. **136**, A405–A411 (1964), the first application of molecular dynamics to systems with continuous potentials; and Loup Verlet, "Computer 'experiments' on classical fluids. I. Thermodynamical properties of Lennard–Jones molecules," Phys. Rev. **159**, 98–103 (1967).

Daan Frenkel and Berend Smit, *Understanding Molecular Simulation: From Algorithms to Applications*, 2nd ed. (Academic Press, 2002). This monograph is one of the best on molecular dynamics and Monte Carlo simulations. It is particularly strong on simulations in various ensembles and on methods for computing free energies.

J. M. Haile, *Molecular Dynamics Simulation* (John Wiley & Sons, 1992). A derivation of the mean pressure using periodic boundary conditions is given in Appendix B.

J. P. Hansen, D. Levesque, and J. J. Weis, "Self-diffusion in the two-dimensional, classical electron gas," Phys. Rev. Lett. **43**, 979–982 (1979).

D. Hirchfeld, Y. Radzyner, and D. C. Rapaport, "Molecular dynamics studies of granular flow through an aperture," Phys. Rev. E **56**, 4404–4415 (1997).

W. G. Hoover, *Molecular Dynamics* (Springer–Verlag, 1986) and W. G. Hoover, *Computational Statistical Mechanics* (Elsevier, 1991).

K. Kadau, T. C. Germann and P. S. Lomdahl, "Large-scale molecular-dynamics simulation of 19 billion particles," Int. J. Mod. Phys. C **15**, 193–201 (2004).

J. Krim, "Friction at macroscopic and microscopic length scales," Am. J. Phys. **70**, 890–897 (2002).

J. Kushick and B. J. Berne, "Molecular dynamics methods: Continuous potentials" in *Statistical Mechanics Part B: Time-Dependent Processes*, Bruce J. Berne, editor (Plenum Press, 1977). Also see the article by Jerome J. Erpenbeck and William Wood on "Molecular dynamics techniques for hard-core systems" in the same volume.

Shang–keng Ma, "Calculation of entropy from data of motion," J. Stat. Phys. **26**, 221 (1981). Also see Chapter 25 of Ma's graduate level text, *Statistical Mechanics* (World Scientific, 1985). Ma discusses a novel approach for computing the entropy directly from the trajectories. Note that the coincidence rate in Ma's approach is related to the recurrence time for a finite system to return to an arbitrarily small neighborhood of almost any given initial state. The approach is intriguing, but is practical only for small systems.

A. McDonough, S. P. Russo, and I. K. Snook, "Long-time behavior of the velocity autocorrelation function for moderately dense, soft-repulsive, and Lennard–Jones fluids," Phys. Rev. E **63**, 026109-1–9 (2001).

S. Ranganathan, G. S. Dubey, and K. N. Pathak, "Molecular-dynamics study of two-dimensional Lennard–Jones fluids," Phys. Rev. A **45**, 5793–5797 (1992).

Dennis Rapaport, *The Art of Molecular Dynamics Simulation*, 2nd ed. (Cambridge University Press, 2004). The most complete text on molecular dynamics written by one of its leading practitioners.

John R. Ray and H. W. Graben, "Direct calculation of fluctuation formulae in the microcanonical ensemble," Mol. Phys. **43**, 1293 (1981).

F. Reif, *Fundamentals of Statistical and Thermal Physics* (McGraw–Hill, 1965). An intermediate level text on statistical physics with a more thorough discussion of kinetic theory than found in most undergraduate texts. *Statistical Physics*, Vol. 5 of the Berkeley Physics Course (McGraw–Hill, 1965), by Reif was one of the first texts to use computer simulations to illustrate the approach of macroscopic systems to equilibrium.

Marco Ronchetti and Gianni Jacucci, editors, *Simulation Approach to Solids* (Kluwer Academic Publishers, 1990). Another excellent collection of classic reprints.

James Ringlein and Mark O. Robbins, "Understanding and illustrating the atomic origins of friction," Am. J. Phys. **72** (7), 884–891 (2004). A very readable paper on the microscopic origins of sliding friction.

Duncan A. Sanders, Michael R. Swift, R. M. Bowley, and P. J. King, "Are Brazil nuts attractive?," Phys. Rev. Lett. **93**, 208002 (2004). An example of a simulation of granular matter.

Tamar Schlick, *Molecular Modeling and Simulation* (Springer–Verlag, 2002). Although the book is at the graduate level, it is an accessible introduction to computational molecular biology.

Leonardo E. Silbert, Deniz Ertas, Gary S. Grest, Thomas C. Halsey, Dov Levine, and Steven J. Plimpton, "Granular flow down an inclined plane: Bagnold scaling and rheology," Phys. Rev. E **64**, 051302-1–14 (2001). This paper discusses the contact force model, which captures the major features of granular interactions.

R. M. Sperandeo Mineo and R. Madonia, "The equation of state of a hard-particle system: A model experiment on a microcomputer," Eur. J. Phys. **7**, 124–129 (1986).

D. Thirumalai and Raymond D. Mountain, "Ergodic convergence properties of supercooled liquids and glasses," Phys. Rev. A **42**, 4574–4587 (1990).

Shan-Ho Tsai, H. K. Lee, and D. P. Landau, "Molecular and spin dynamics simulations using modern integration methods," Am. J. Phys. **73**, 615–624 (2005).

James H. Williams and Glenn Joyce, "Equilibrium properties of a one-dimensional kinetic system," J. Chem. Phys. **59**, 741–750 (1973). Simulations in one dimension are even easier than in two.

Zoran Slanič, Harvey Gould, and Jan Tobochnik, "Dynamics of the classical Heisenberg chain," Computers in Physics **5** (6), 630–635 (1991).

Chapter 9

Normal Modes and Waves

We discuss the physics of wave phenomena and the motivation and use of Fourier transforms.

9.1 Coupled Oscillators and Normal Modes

Terms such as period, amplitude, and frequency are used to describe both waves and oscillatory motion. To understand the relation between waves and oscillatory motion, consider a flexible rope that is under tension with one end fixed. If we flip the free end, a pulse propagates along the rope with a speed that depends on the tension and on the inertial properties of the rope. At the *macroscopic* level, we observe a transverse wave that moves along the length of the rope. In contrast, at the *microscopic* level we see discrete particles undergoing oscillatory motion in a direction perpendicular to the motion of the wave. One goal of this chapter is to use simulations to understand the relation between the microscopic dynamics of a simple mechanical model and the macroscopic wave motion that the model can support.

For simplicity, we first consider a one-dimensional chain of N particles each of mass m. The particles are coupled by massless springs with force constant k. The equilibrium separation between the particles is a. We denote the displacement of particle j from its equilibrium position at time t by $u_j(t)$ (see Figure 9.1). For many purposes the most realistic boundary conditions are to attach particles $j = 1$ and $j = N$ to springs which are attached to fixed walls. We denote the walls by $j = 0$ and $j = N + 1$ and require that $u_0(t) = u_{N+1}(t) = 0$.

The force on an individual particle is determined by the compression or extension of its adjacent springs. The equation of motion of particle j is given by

$$\begin{aligned} m\frac{d^2u_j(t)}{dt^2} &= -k\Big[u_j(t) - u_{j+1}(t)\Big] - k\Big[u_j(t) - u_{j-1}(t)\Big] \\ &= -k\Big[2u_j(t) - u_{j+1}(t) - u_{j-1}(t)\Big]. \end{aligned} \tag{9.1}$$

Equation (9.1) couples the motion of particle j to its two nearest neighbors and describes *longitudinal* oscillations; that is, motion along the length of the system. It is straightforward to show that identical equations hold for the *transverse* oscillations of N identical mass points equally spaced on a stretched massless string (cf. French).

Because the equations of motion (9.1) are linear, that is, only terms proportional to the displacements appear, it is straightforward to obtain analytic solutions of (9.1). We first discuss

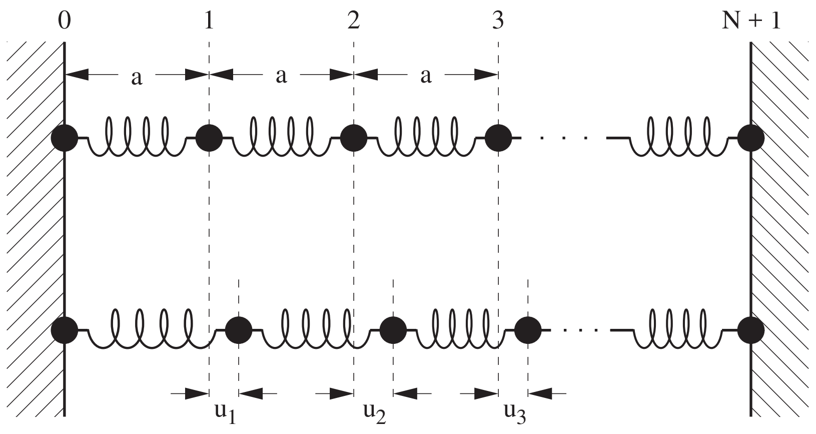


Figure 9.1: A one-dimensional chain of N particles of mass m coupled by massless springs with force constant k. The first and last particles (0 and $N+1$) are attached to fixed walls. The top chain shows the oscillators in equilibrium. The bottom chain shows the oscillators displaced from equilibrium.

these solutions because they will help us interpret the nature of the numerical solutions. To find the *normal modes*, we look for oscillatory solutions for which the displacement of each particle is proportional to $\sin\omega t$ or $\cos\omega t$. We write

$$u_j(t) = u_j \cos\omega t \tag{9.2}$$

where u_j is the amplitude of the displacement of the jth particle. If we substitute the form (9.2) into (9.1), we obtain

$$-\omega^2 u_j = -\frac{k}{m}[2u_j - u_{j+1} - u_{j-1}]. \tag{9.3}$$

We next assume that the amplitude u_j depends sinusoidally on the distance ja:

$$u_j = C\sin qja \tag{9.4}$$

where the constants q and C will be determined. If we substitute (9.4) into (9.3), we find the following condition for ω:

$$-\omega^2 \sin qja = -\frac{k}{m}\big[2\sin qja - \sin q(j-1)a - \sin q(j+1)a\big]. \tag{9.5}$$

We write $\sin q(j\pm1)a = \sin qja\cos qa \pm \cos qja\sin qa$ and find that (9.4) is a solution if

$$\omega^2 = 2\frac{k}{m}\big(1-\cos qa\big). \tag{9.6}$$

We need to find the values of the wavenumber q that satisfy the boundary conditions $u_0 = 0$ and $u_{N+1} = 0$. The former condition is automatically satisfied by assuming a sine instead of a cosine solution in (9.4). The latter boundary condition implies that

$$q = q_n = \frac{\pi n}{a(N+1)} \qquad \text{(fixed boundary conditions)} \tag{9.7}$$

where $n = 1, \dots, N$. The corresponding possible values of the wavelength λ are related to q by $q = 2\pi/\lambda$, and the corresponding values of the angular frequencies are given by

$$\omega_n{}^2 = 2\frac{k}{m}[1 - \cos q_n a] = 4\frac{k}{m}\sin^2\frac{q_n a}{2}, \tag{9.8}$$

or

$$\omega_n = 2\sqrt{\frac{k}{m}}\sin\frac{q_n a}{2}. \tag{9.9}$$

The relation (9.9) between ω_n and q_n is known as a *dispersion relation*.

A particular value of the integer n corresponds to the nth normal mode. We write the (time-independent) normal mode solutions as

$$u_{j,n} = C \sin q_n ja. \tag{9.10}$$

The linear nature of the equation of motion (9.1) implies that the time dependence of the displacement of the jth particle can be written as a superposition of normal modes:

$$u_j(t) = C\sum_{n=1}^{N}\Big(A_n \cos\omega_n t + B_n \sin\omega_n t\Big)\sin q_n ja. \tag{9.11}$$

The coefficients A_n and B_n are determined by the initial conditions:

$$u_j(t=0) = C\sum_{n=1}^{N} A_n \sin q_n ja \tag{9.12a}$$

$$v_j(t=0) = C\sum_{n=1}^{N} \omega_n B_n \sin q_n ja. \tag{9.12b}$$

To solve (9.12) for A_n and B_n, we note that the normal mode solutions $u_{j,n}$ are orthogonal; that is, they satisfy the condition

$$\sum_{j=1}^{N} u_{j,n}\, u_{j,m} \propto \delta_{n,m}. \tag{9.13}$$

The Kronecker δ symbol $\delta_{n,m} = 1$ if $n = m$ and is zero otherwise. It is convenient to normalize the $u_{j,n}$ so that they are orthonormal; that is,

$$\sum_{j=1}^{N} u_{j,n}\, u_{j,m} = \delta_{n,m}. \tag{9.14}$$

It is easy to show that the choice, $C = 1/\sqrt{(N+1)/2}$, in (9.4) and (9.10) insures that (9.14) is satisfied.

We now use the orthonormality condition (9.14) to determine the A_n and B_n coefficients. If we multiply both sides of (9.12) by $C \sin q_m ja$, sum over j, and use the orthogonality condition (9.14), we obtain

$$A_n = C\sum_{j=1}^{N} u_j(0)\sin q_n ja \tag{9.15a}$$

$$B_n = C\sum_{j=1}^{N} (v_j(0)/\omega_n)\sin q_n ja. \tag{9.15b}$$

For example, if the initial displacement of every particle is zero, and the initial velocity of every particle is zero except for $v_1(0) = 1$, we find $A_n = 0$ for all n, and

$$B_n = \frac{C}{\omega_n} \sin q_n a. \tag{9.16}$$

The corresponding solution for $u_j(t)$ is

$$u_j(t) = \frac{2}{N+1} \sum_{n=1}^{N} \frac{1}{\omega_n} \cos \omega_n t \sin q_n a \sin q_n j a. \tag{9.17}$$

What is the solution if the particles start in a normal mode; that is, $u_j(t=0) \propto \sin q_2 j a$?

The `Oscillators` class in Listing 9.1 displays the analytic solution (9.11) of the oscillator displacements. The `draw` method uses a single circle that is repeatedly set equal to the appropriate world coordinates. The initial positions are calculated and stored in the `y` array in the `Oscillators` constructor. When an oscillator is drawn, the position array is multiplied by the given mode's sinusoidal (phase) factor to produce a time-dependent displacement.

Listing 9.1: The `Oscillators` class models the time evolution of a normal mode of a chain of coupled oscillators.

```java
package org.opensourcephysics.sip.ch09;
import java.awt.Graphics;
import org.opensourcephysics.display.*;

public class Oscillators implements Drawable {
   OscillatorsMode normalMode;
   Circle circle = new Circle();
   double[] x; // drawing positions
   double[] u; // displacement
   double time = 0;

   public Oscillators(int mode, int N) {
      u = new double[N+2]; // includes the two ends of the chain
      x = new double[N+2]; // includes the two ends of the chain
      normalMode = new OscillatorsMode(mode, N);
      double xi = 0;
      for(int i = 0;i<N+2;i++) {
         x[i] = xi;
         u[i] = normalMode.evaluate(xi); // initial displacement
         // increment x[i] by lattice spacing of one
         xi++;
      }
   }

   public void step(double dt) {
      time += dt;
   }

   public void draw(DrawingPanel drawingPanel, Graphics g) {
      normalMode.draw(drawingPanel, g); // draw initial condition
      double phase = Math.cos(time*normalMode.omega);
      for(int i = 0, n = x.length;i<n;i++) {
         circle.setXY(x[i], u[i]*phase);
         circle.draw(drawingPanel, g);
      }
   }
}
```

The OscillatorsMode class in Listing 9.2 instantiates a normal mode. This class stores the mode frequency and implements the Function interface to evaluate the analytic solution. It draws a light gray outline of the initial analytic solution using a FunctionDrawer.

Listing 9.2: The OscillatorsMode class models a normal mode of a chain of coupled oscillators.

```
package org.opensourcephysics.sip.ch09;
import java.awt.*;
import org.opensourcephysics.display.*;
import org.opensourcephysics.numerics.*;

public class OscillatorsMode implements Drawable, Function {
   static final double OMEGA_SQUARED = 1; // equals k/m
   FunctionDrawer functionDrawer;         // draws the initial condition
   double omega;                          // oscillation frequency of mode
   double wavenumber;                     // wavenumber = 2*pi/wavelength
   double amplitude;

   OscillatorsMode(int mode, int N) {
      amplitude = Math.sqrt(2.0/(N+1));
      omega = 2*Math.sqrt(OMEGA_SQUARED)
            *Math.abs(Math.sin(mode*Math.PI/(2*(N+1))));
      wavenumber = Math.PI*mode/(N+1);
      functionDrawer = new FunctionDrawer(this);
      // draws the initial displacement
      functionDrawer.initialize(0, N+1, 300, false);
      functionDrawer.color = Color.LIGHT_GRAY;
   }

   public double evaluate(double x) {
      return amplitude*Math.sin(x*wavenumber);
   }

   public void draw(DrawingPanel panel, Graphics g) {
      functionDrawer.draw(panel, g);
   }
}
```

The OscillatorsApp target class extends AbstractSimulation, creates an Oscillators object, and displays the particle displacements as transverse oscillations. The complete listing is available in the ch09 code package, but it is not given here because it is similar to other animations.

Problem 9.1. Normal modes

(a) How many modes are there for a chain of 16 oscillators? Predict the initial positions of the oscillators for modes 1 and 14 and compare your prediction to the program's output.

(b) Because a normal mode is a standing wave, it can be written as the sum of right and left traveling sinusoidal waves, $f_+ = A\sin(kx-\omega t)$ and $f_- = A\sin(kx+\omega t)$, respectively. Does the phase velocity $v = \omega/k$ depend on the mode (wavelength)? In other words, how does the speed depend on the wavelength [see (9.9)]?

(c) Determine the wavelength and frequency for modes with the largest and smallest wavelengths. Note that you can click-drag within the window to measure position. The time is displayed in the yellow message box. Compare your measured values with (9.9).

(d) Are negative mode numbers acceptable? Give two different listings of mode numbers that contain a complete set of modes for $N = 16$.

(e) Write a short stand-alone program to verify that the normal mode solutions (9.10) are orthonormal.

(f) Modify your program to show the evolution of the superposition of two or more normal modes. Compare this evolution to that of a single mode. □

As seen in Problem 9.1, the number of unique nontrivial solutions is equal to the number of oscillators. Modes with mode numbers $1, 2, 3, \ldots, N$ form a complete set and all other modes are indistinguishable from them. (The number of modes is related to the Nyquist frequency and is discussed further in Section 9.3.)

The analytic solution (9.11), together with the initial conditions, represent the complete solution of the displacement of the particles. We can use a computer to calculate the sum in (9.11) and plot the time dependence of the displacements $u_j(t)$. There are many interesting extensions that are amenable to an analytic solution. What is the effect of changing the boundary conditions? What happens if the spring constants are not all equal but are chosen from a probability distribution? What happens if we vary the masses of the particles? For these cases we can follow a similar approach and look for the eigenvalues ω_n and eigenvectors $u_{j,n}$ of the matrix equation

$$\mathbf{T}\,\mathbf{u} = \omega^2\,\mathbf{u}. \tag{9.18}$$

The matrix elements $T_{i,j}$ are zero except for

$$T_{i,i} = \frac{1}{m_i}[k_{i,i+1} + k_{i,i-1}] \tag{9.19a}$$

$$T_{i,i+1} = -\frac{k_{i,i+1}}{m_i} \tag{9.19b}$$

$$T_{i,i-1} = -\frac{k_{i,i-1}}{m_i} \tag{9.19c}$$

where $k_{i,j}$ is the spring constant between particles i and j. The solution of matrix equations such as (9.18) is a well- studied problem in linear programming, and an open source library such as LINPAC available from NetLIB (<`www.netlib.org`>) or a stand-alone program such as Octave (<`www.octave.org`>) can be used to obtain the solutions.

9.2 Numerical Solutions

Because we are also interested in the effects of nonlinear forces between the particles, for which the matrix approach is inapplicable, we study the numerical solution of the equations of motion (9.1) directly.

To use the ODE interface, we need to remember that the ordering of the variables in the coupled oscillator state array is important because the implementations of some ODE solvers, such as Verlet and Euler–Richardson, make explicit assumptions about the ordering. Our standard ordering is to follow a variable by its derivative. For example, the state vector of an N oscillator chain is ordered as $\{u_0, v_0, u_1, v_1, \ldots, u_N, v_N, u_{N+1}, v_{N+1}, t\}$. Note that the state array includes variables for the chain's end points although the velocity rate corresponding to the end points is always zero. We include the time as the last variable because we will sometimes model time-dependent external forces. With this ordering, the `getRate` method is implemented as follows:

```java
static final double OMEGA_SQUARED = 1;        // equals k/m
public void getRate(double[] state, double[] rate) {
   for(int i = 1, N = x.length-1; i<N; i++) {  // skip ends
      rate[2*i] = state[2*i+1]; // displacement rate
      rate[2*i+1] = -OMEGA_SQUARED*(2*state[2*i]-state[2*i-2]-state[2*i+2]);
   }
   rate[state.length-1] = 1;
 }
```

Problem 9.2. Numerical solution

(a) Modify the `Oscillators` class to solve the dynamical equations of motion by implementing the `ODE` interface. Compare the numerical and the analytic solution for $N = 10$ using an algorithm that is well suited to oscillatory problems.

(b) What is the maximum deviation between the analytic and numeric solution of $u_j(t)$? How well is the total energy conserved in the numerical solution? How does the maximum deviation and the conservation of the total energy change when the time step Δt is reduced? Justify your choice of numerical algorithm. □

Problem 9.3. Dynamics of coupled oscillators

(a) Use your program for Problem 9.2 for $N = 2$ using units such that the ratio $k/m = 1$. Choose the initial values of u_1 and u_2 so that the system is in one of its two normal modes, for example, $u_1 = u_2 = 0.5$ and set the initial velocities equal to zero. Describe the displacement of the particles. Is the motion of each particle periodic in time? To answer this question, add code that plots the displacement of each particle versus the time. Then consider the other normal mode, for example, $u_1 = 0.5$, $u_2 = -0.5$. What is the period in this case? Does the system remain in a normal mode indefinitely? Finally, choose the initial particle displacements equal to random values between -0.5 and $+0.5$. Is the motion of each particle periodic in this case?

(b) Consider the same questions as in part (a) but with $N = 4$ and $N = 10$. Consider the $n = 2$ mode for $N = 4$ and the $n = 3$ and $n = 8$ modes for $N = 10$. (See (9.10) for the form of the normal mode solutions.) Also consider random initial displacements. □

Problem 9.4. Different boundary conditions

(a) Modify your program from Problem 9.3 so that periodic boundary conditions are used; that is, $u_0 = u_N$ and $u_1 = u_{N+1}$. Choose $N = 10$ and the initial condition corresponding to the normal mode (9.10) with $n = 2$. Does this initial condition yield a normal mode solution for periodic boundary conditions? (It might be easier to answer this question by plotting

u_i versus time for two or more particles.) For fixed boundary conditions, there are $N+1$ springs, but for periodic boundary conditions, there are N springs. Why? Choose the initial condition corresponding to the $n = 2$ normal mode, but replace $N+1$ by N in (9.7). Does this initial condition correspond to a normal mode? Now try $n = 3$ and other values of n. Which values of n give normal modes? Only sine functions can be normal modes for fixed boundary conditions [see (9.4)]. Can there be normal modes with cosine functions if we use periodic boundary conditions?

(b) Modify your program so that free boundary conditions are used, which means that the masses at the end points are connected to only one nearest neighbor. A simple way to implement this boundary condition is to set $u_0 = u_1$ and $u_N = u_{N+1}$. Choose $N = 10$, and use the initial condition corresponding to the $n = 3$ normal mode found using fixed boundary conditions. Does this condition correspond to a normal mode for free boundary conditions? Is $n = 2$ a normal mode for free boundary conditions? Are the normal modes purely sinusoidal?

(c) Choose free boundary conditions and $N \geq 10$. Let the initial condition be a pulse of the form $u_1 = 0.2, u_2 = 0.6, u_3 = 1.0, u_4 = 0.6, u_5 = 0.2$, and all other $u_j = 0$. After the pulse reaches the right end, what is the phase of the reflected pulse; that is, are the displacements in the reflected pulse in the same direction as the incoming pulse (a phase shift of zero degrees) or in the opposite direction (a phase shift of 180 degrees)? What happens for fixed boundary conditions? Choose N to be as large as possible so that it is easy to distinguish the incident and reflected waves.

(d) Choose $N \geq 20$ and let the spring constants on the right half of the system be four times greater than the spring constants on the left half. Use fixed boundary conditions. Set up a pulse on the left side. Is there a reflected pulse at the boundary between the two types of springs? If so, what is its relative phase? Compare the amplitude of the reflected and transmitted pulses. Consider the same questions with a pulse that is initially on the right side. □

Problem 9.5. Motion of coupled oscillators with external forces

(a) Modify your program from Problem 9.4 so that an external force F_{ext} is exerted on the first particle

$$F_{\text{ext}}/m = 0.5 \cos \omega t \tag{9.20}$$

where ω is the angular frequency of the external force. Let the initial displacements and velocities of all N particles be zero. Choose $N = 3$ and consider the response of the system to an external force for $\omega = 0.5$ to 4.0 in steps of 0.5. Record $A(\omega)$, the maximum amplitude of any particle, for each value of ω. Repeat the simulation for $N = 10$.

(b) Choose ω to be one of the normal mode frequencies. Does the maximum amplitude remain constant or does it increase with time? How can you use the response of the system to an external force to determine the normal mode frequencies? Discuss your results in terms of the power input $F_{\text{ext}} v_1$?

(c) In addition to the external force exerted on the first particle, add a damping force equal to $-\gamma v_i$ to all the oscillators. Choose the damping constant $\gamma = 0.05$. How do you expect the system to behave? How does the maximum amplitude depend on ω? Are the normal mode frequencies changed when $\gamma \neq 0$? □

In Problem 9.5 we saw that a boundary produces reflections that lead to resonances. Although it is not possible to eliminate all reflections, it can be shown analytically that it is possible to absorb waves at a single frequency ω by using a free boundary and a judicious choice of the mass and damping coefficient for the last oscillator (see the text by Main). These conditions are

$$m_{N-1} = \frac{m}{2} \tag{9.21a}$$

$$\gamma_{N-1} = \frac{k}{\omega}\sin qa \tag{9.21b}$$

where a is the separation between oscillators, q is the wavenumber, and ω is the angular frequency. Problem 9.6 shows that this choice leads to a transparent boundary (at the selected frequency) and is akin to impedance matching in electrical transmission lines enabling us to study traveling waves. In Section 9.7 we will study the classical wave equation in detail.

Problem 9.6. Traveling waves

(a) Modify the oscillator program by imposing a sinusoidal amplitude on the first oscillator:

$$u_1(t) = 0.5\sin\omega t. \tag{9.22}$$

Run the program with $\omega = 1.0$ and observe the right traveling wave, but note that reflections soon produce a left traveling wave.

(b) Implement transparent boundary conditions on the right-hand side. Is the first wave crest totally transmitted at the end, or is there some residual reflection? Explain. Add a small overall damping to the chain to remove transient effects.

(c) The dispersion relation (9.9) predicts a cutoff frequency:

$$\omega_c^2 = 4\frac{k}{m}. \tag{9.23}$$

What happens if we apply an external driving force above this frequency? □

Problem 9.7. Evanescent waves

Increase ω past the cutoff frequency in the traveling wave simulation that was found in Problem 9.6. Do you still observe waves? Is energy being transported along the chain? □

Waves above the cutoff frequency are known as *evanescent waves*. In Problem 9.8 we show how these waves lead to a classical counterpart of quantum mechanical tunneling.

Problem 9.8. Tunneling

(a) Model a traveling wave on a $N = 64$ particle chain with mass $m = 1$ and $k = 1$, but assign $m = 4$ to eight oscillators near the center. Drive the first particle in the chain with a frequency of 0.113. (This value is slightly higher than the frequency above which the wave amplitude falls off exponentially.) Describe the steady-state motion in the left region, the central region of heavier masses, and the right region.

(b) Lower the frequency in part (a) until you observe maximum transmission through the barrier. Describe the steady-state motion in the left region, the central barrier, and the right region. Explain how this system can be used as a frequency filter. □

9.3 Fourier Series

In Section 9.1, we showed that the displacement of a single particle can be written as a linear combination of normal modes, that is, a linear superposition of sinusoidal terms. In general, an arbitrary periodic function $f(t)$ of period T can be expressed as a sum of sines and cosines:

$$f(t) = \frac{1}{2}a_0 + \sum_{k=1}^{\infty}\Big(a_k \cos\omega_k t + b_k \sin\omega_k t\Big) \tag{9.24}$$

where

$$\omega_k = k\omega_0 \text{ and } \omega_0 = \frac{2\pi}{T}. \tag{9.25}$$

The quantity ω_0 is the fundamental frequency. Such a sum is called a Fourier series. The sine and cosine terms in (9.24) for $k = 2, 3, \ldots$ represent the second, third, ..., and higher order harmonics. The Fourier coefficients a_k and b_k are given by

$$a_k = \frac{2}{T}\int_0^T f(t)\cos\omega_k t\, dt \tag{9.26a}$$

$$b_k = \frac{2}{T}\int_0^T f(t)\sin\omega_k t\, dt. \tag{9.26b}$$

The constant term $\frac{1}{2}a_0$ in (9.24) is the average value of $f(t)$. The expressions in (9.26) for the coefficients follow from the orthogonality conditions:

$$\frac{2}{T}\int_0^T \sin\omega_k t \sin\omega_{k'} t\, dt = \delta_{k,k'} \tag{9.27a}$$

$$\frac{2}{T}\int_0^T \cos\omega_k t \cos\omega_{k'} t\, dt = \delta_{k,k'} \tag{9.27b}$$

$$\frac{2}{T}\int_0^T \sin\omega_k t \cos\omega_{k'} t\, dt = 0. \tag{9.27c}$$

In general, an infinite number of terms is needed to represent an arbitrary periodic function exactly. In practice, a good approximation usually can be obtained by including a relatively small number of terms. Unlike a power series, which can approximate a function only near a particular point, a Fourier series can approximate a function at almost every point. The `Synthesize` class in Listing 9.3 evaluates such a series given the Fourier coefficients a and b.

Listing 9.3: A class that synthesizes a function using a Fourier series.

```
package org.opensourcephysics.sip.ch09;
import org.opensourcephysics.numerics.Function;

public class Synthesize implements Function {
   double[] cosCoefficients, sinCoefficients;
   // cosine and sine coefficients
   double a0; // the constant term
   double omega0;

   public Synthesize(double period, double a0, double[] cosCoef,
                     double[] sinCoef) {
```

```
      omega0 = Math.PI*2/period;
      cosCoefficients = cosCoef;
      sinCoefficients = sinCoef;
      this.a0 = a0;
   }

   public double evaluate(double x) {
      double f = a0/2;
      // sum the cosine terms
      for(int i = 0, n = cosCoefficients.length;i<n;i++) {
         f += cosCoefficients[i]*Math.cos(omega0*x*(i+1));
      }
      // sum the sine terms
      for(int i = 0, n = sinCoefficients.length;i<n;i++) {
         f += sinCoefficients[i]*Math.sin(omega0*x*(i+1));
      }
      return f;
   }
}
```

The SynthesizeApp class creates a Synthesize object by defining the values of the nonzero Fourier coefficients and draws the result of the Fourier series. Because the Synthesize class implements the Function interface, we can plot the Fourier series and see how the sum can represent an arbitrary periodic function. An easy way to do so is to create a FunctionDrawer and add it to a drawing frame as shown in Listing 9.4. The FunctionDrawer handles the routine task of generating a curve from the given function to produce a drawing.

Listing 9.4: A program that displays a Fourier series.

```
package org.opensourcephysics.sip.ch09;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.display.FunctionDrawer;
import org.opensourcephysics.frames.DisplayFrame;

public class SynthesizeApp extends AbstractCalculation {
   DisplayFrame frame = new DisplayFrame("x", "f(x)", "Fourier Synthesis");

   public void calculate() {
      double xmin = control.getDouble("xmin");
      double xmax = control.getDouble("xmax");
      int N = control.getInt("N");
      double period = control.getDouble("period");
      double[] sinCoefficients = (double[]) control.getObject(
                                 "sin coefficients");
      double[] cosCoefficients = (double[]) control.getObject(
                                 "cos coefficients");
      FunctionDrawer functionDrawer = new FunctionDrawer(new Synthesize
          (period, 0, cosCoefficients, sinCoefficients));
      functionDrawer.initialize(xmin, xmax, N, false);
      frame.clearDrawables();             // remove old function drawer
      frame.addDrawable(functionDrawer); // add new function drawer
   }

   public void reset() {
      control.setValue("xmin", -1);
      control.setValue("xmax", 1);
      control.setValue("N", 300);
      control.setValue("period", 1);
      control.setValue("sin coefficients", new double[] {
         1.0, 0, 1.0/3.0, 0, 1.0/5.0, 0, 0
      });
      control.setValue("cos coefficients", new double[] {
         0, 0, 0, 0, 0, 0, 0
      });
      calculate();
   }

   public static void main(String[] args) {
      CalculationControl.createApp(new SynthesizeApp());
   }
}
```

Problem 9.9. Fourier synthesis

(a) The process of approximating a function by adding together a fundamental frequency and harmonics of various amplitudes is *Fourier synthesis*. The SynthesizeApp class shows how a sum of harmonic functions can represent an arbitrary periodic function. Consider the series

$$f(t) = \frac{2}{\pi}(\sin t + \frac{1}{3}\sin 3t + \frac{1}{5}\sin 5t + \cdots). \tag{9.28}$$

Describe the nature of the plot of $f(t)$ when only the first three terms in (9.28) are retained. Increase the number of terms until you are satisfied that (9.28) represents the desired function with sufficient accuracy. What function is represented by the infinite series?

(b) Modify SynthesizeApp so that you can create an initial state with an arbitrary number of terms. (Do not use the control to input each coefficient. Input only the total number of terms.) Consider the series (9.28) with at least 32 terms. For what values of t does the finite sum most faithfully represent the exact function? For what values of t does it not? Why is it necessary to include a large number of terms to represent $f(t)$ where it has sharp edges? The small oscillations that increase in amplitude as a sharp edge is approached are known as the *Gibbs phenomenon*.

(c) Modify SynthesizeApp and determine the function that is represented by the Fourier series with coefficients $a_k = 0$ and $b_k = (2/k\pi)(-1)^{k-1}$ for $k = 1, 2, 3, \ldots$. Approximately how many terms in the series are required? □

So far we have considered how a sum of sines and cosines can approximate a known periodic function. More typically, we generate a time series consisting of N data points, $f(t_i)$ where $t_i = 0, \Delta, 2\Delta, \ldots, (N-1)\Delta$. The Fourier series approximation to this data assumes that the data repeats itself with a period T given by $T = N\Delta$. Our goal is to determine the Fourier coefficients a_k and b_k. We will see that these coefficients contain important physical information.

Because we know only a finite number of data points $f_i \equiv f(t_i)$, it is possible to find only a finite set of Fourier coefficients. For a given value of Δ, what is the largest frequency component we can extract? In the following, we give a plausibility argument that suggests that the maximum frequency we can analyze is given by

$$\omega_Q \equiv \frac{\pi}{\Delta} \quad \text{or} \quad f_Q \equiv \frac{1}{2\Delta} \qquad \text{(Nyquist frequency)} \tag{9.29}$$

where f_Q is the Nyquist frequency.

One way to understand (9.29) is to analyze a sine wave, $\sin \omega t$. If we choose $\Delta = \pi/\omega$; that is, sample at the Nyquist frequency, we would sample two points per cycle. If we sample at or below this frequency, we sample during the sine's positive and negative displacement. If we sample above this frequency, we would sample during the positive displacement of one crest and the positive displacement of another crest, thereby completely missing the negative displacement. In Problem 9.10, we explore sampling at frequencies higher than the Nyquist frequency.

Problem 9.10. Nyquist demonstration

Run `OscillatorsApp` with 16 particles in normal mode 27. What is the nature of the motion that you observe? Because the particles sample the analytic function $f(x)$ which sets the initial position at intervals Δ that are larger than $1/(2\lambda)$, the particle positions cannot accurately represent the high spatial frequencies. What mode number corresponds to sampling at the Nyquist frequency? □

The Nyquist frequency is very important in signal processing because the sampling theorem due to Nyquist and Shannon states that a continuous function is completely determined by N sampling points if the signal has passed though a filter with a cutoff frequency less than f_Q. This theorem is easy to understand. The filter will take out all the high frequency modes and allow only N modes of the signal to pass through. With N sampling points we can solve for the amplitudes of each mode using (9.35) (also see Section 9.6).

One consequence of (9.29) is that there are $\omega_Q/\omega_0 + 1$ independent coefficients for a_k (including a_0) and ω_Q/ω_0 independent coefficients for b_k, a total of $N+1$ independent coefficients. (Recall that $\omega_Q/\omega_0 = N/2$, where $\omega_0 = 2\pi/T$ and $T = N\Delta$.) Because $\sin \omega_Q t = 0$ for all values of t that are multiples of Δ, we have $b_{N/2} = 0$ from (9.26b). Consequently, there are $N/2 - 1$ values for b_k, and hence, a total of N Fourier coefficients that can be computed. This conclusion is reasonable because the number of meaningful Fourier coefficients should be the same as the number of data points.

The `Analyze` class computes the Fourier coefficients a_k and b_k of a function $f(t)$ defined between $t = 0$ and $t = T$ at intervals of Δ. To compute the coefficients, we do the integrals in (9.26) numerically using the simple rectangular approximation (see Section 11.1):

$$a_k \approx \frac{2\Delta}{T} \sum_{i=0}^{N-1} f(t_i) \cos \omega_k t_i \tag{9.30a}$$

$$b_k \approx \frac{2\Delta}{T} \sum_{i=0}^{N-1} f(t_i) \sin \omega_k t_i \tag{9.30b}$$

where the ratio $2\Delta/T = 2/N$.

Listing 9.5: The `Analyze` class calculates the Fourier coefficients of a function.

```
package org.opensourcephysics.sip.ch09;
import org.opensourcephysics.numerics.Function;
```

```
public class Analyze {
   Function f;
   double period, delta;
   double omega0;
   int N;

   Analyze(Function f, int N, double delta) {
      this.f = f;
      this.delta = delta;
      this.N = N;
      period = N*delta;
      omega0 = 2*Math.PI/period;
   }

   double getSineCoefficient(int n) {
      double sum = 0;
      double t = 0;
      for(int i = 0;i<N;i++) {
         sum += f.evaluate(t)*Math.sin(n*omega0*t);
         t += delta;
      }
      return 2*delta*sum/period;
   }

   double getCosineCoefficient(int n) {
      double sum = 0;
      double t = 0;
      for(int i = 0;i<N;i++) {
         sum += f.evaluate(t)*Math.cos(n*omega0*t);
         t += delta;
      }
      return 2*delta*sum/period;
   }
}
```

The AnalyzeApp program reads a function and creates an Analyze object to plot the Fourier coefficients a_k and b_k versus k. Note how we use a *parser* to convert a string of characters into a function using a ParsedFunction object. Because typing mistakes are common when entering mathematical expressions, we return from the calculate method if a syntax error is detected.

Listing 9.6: A program that analyzes the Fourier components of a function.

```
package org.opensourcephysics.sip.ch09;
import java.awt.*;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.display.*;
import org.opensourcephysics.frames.*;
import org.opensourcephysics.numerics.*;

public class AnalyzeApp extends AbstractCalculation {
   PlotFrame frame = new PlotFrame("frequency", "coefficients",
                        "Fourier analysis");

   public AnalyzeApp() {
      frame.setMarkerShape(0, Dataset.POST);
      frame.setMarkerColor(0, new Color(255, 0, 0, 128));
      // semitransparent red
      frame.setMarkerShape(1, Dataset.POST);
      frame.setMarkerColor(1, new Color(0, 0, 255, 128));\
      // semitransparent blue
      frame.setXYColumnNames(0, "frequency", "cos");
      frame.setXYColumnNames(1, "frequency", "sin");
   }

   public void calculate() {
      double delta = control.getDouble("delta");
      int N = control.getInt("N");
      int numberOfCoefficients = control.getInt("number of coefficients");
      String fStr = control.getString("f(t)");
      Function f = null;
      try {
         f = new ParsedFunction(fStr, "t");
      } catch(ParserException ex) {
         control.println("Error parsing function string: "+fStr);
         return;
      }
      Analyze analyze = new Analyze(f, N, delta);
      double f0 = 1.0/(N*delta);
      for(int i = 0;i<=numberOfCoefficients;i++) {
         frame.append(0, i*f0, analyze.getCosineCoefficient(i));
         frame.append(1, i*f0, analyze.getSineCoefficient(i));
      }
      // Data tables can be displayed  by the user using
      // the tools menu but this statemment does so explicitly.
      frame.showDataTable(true);
   }

   public void reset() {
      control.setValue("f(t)", "sin(pi*t/10)");
      control.setValue("delta", 0.1);
      control.setValue("N", 200);
      control.setValue("number of coefficients", 10);
      calculate();
   }

   public static void main(String[] args) {
      CalculationControl.createApp(new AnalyzeApp());
   }
}
```

In Problem 9.11 we compute the Fourier coefficients for several functions. We will see that if $f(t)$ is a sum of sinusoidal functions with different periods, it is essential that the period $T = N\Delta$ in the Fourier analysis program be an integer multiple of the periods of all the functions in the sum. If T does not satisfy this condition, then the results for some of the Fourier coefficients will be spurious. In practice, the solution to this problem is to vary the sampling interval Δ and the total time over which the signal $f(t)$ is sampled. Fortunately, the results for the power spectrum (see Section 9.6) are less ambiguous than the values for the Fourier coefficients themselves.

Problem 9.11. Fourier analysis

(a) Use the `AnalyzeApp` class with $f(t) = \sin \pi t/10$. Determine the first three nonzero Fourier coefficients by doing the integrals in (9.26) analytically before running the program. Choose the number of data points to be $N = 200$ and the sampling time $\Delta = 0.1$. Which Fourier components are nonzero? Repeat your analysis for $N = 400, \Delta = 0.1$; $N = 200, \Delta = 0.05$; $N = 205, \Delta = 0.1$; and $N = 500, \Delta = 0.1$, and other combinations of N and Δ. Explain your results by comparing the period of $f(t)$ with $N\Delta$, the assumed period. If the combination of N and Δ are not chosen properly, do you find any spurious results for the coefficients?

(b) Consider the functions $f_1(t) = \sin \pi t/10 + \sin \pi t/5$, $f_2(t) = \sin \pi t/10 + \cos \pi t/5$, and $f_3(t) = \sin \pi t/10 + \frac{1}{2}\cos \pi t/5$, and answer the same questions as in part (a) for each function. What combinations of N and Δ give reasonable results for each function?

(c) Consider a function that is not periodic, but goes to zero for $|t|$ large. For example, try $f(t) = t^4 e^{-t^2}$ and $f(t) = t^3 e^{-t^2}$. Interpret the difference between the Fourier coefficients of these two functions. □

As shown in Appendix 9A, sine and cosine functions in a Fourier series can be combined into exponential functions with complex coefficients and complex exponents. We express $f(t)$ as

$$f(t) = \sum_{k=-\infty}^{\infty} c_k e^{i\omega_k t} \tag{9.31}$$

where

$$\omega_k = k\omega_0 \text{ and } \omega_0 = \frac{2\pi}{T} \tag{9.32}$$

and use (9.24) to express the complex coefficients c_k in terms of a_k and b_k:

$$c_k = \frac{1}{2}(a_k - ib_k) \tag{9.33a}$$

$$c_0 = \frac{1}{2}a_0 \tag{9.33b}$$

$$c_{-k} = \frac{1}{2}(a_k + ib_k). \tag{9.33c}$$

The coefficients c_k can be expressed in terms of $f(t)$ by using (9.33) and (9.26) and the fact that $e^{\pm i\omega_k t} = \cos\omega_k t \pm i \sin\omega_k t$. The result is

$$c_k = \frac{1}{T}\int_{-T/2}^{T/2} f(t)e^{-i\omega_k t}\,dt. \tag{9.34}$$

As in (9.30), we can approximate the integral in (9.34) using the rectangular approximation. We write

$$g(\omega_k) \equiv c_k \frac{T}{\Delta} \approx \sum_{j=-N/2}^{N/2} f(j\Delta)e^{-i\omega_k j\Delta} = \sum_{j=-N/2}^{N/2} f(j\Delta)e^{-i2\pi kj/N}. \tag{9.35}$$

If we multiply (9.35) by $e^{i2\pi kj'/N}$, sum over k, and use the orthogonality condition

$$\sum_{k=-N/2}^{N/2} e^{i2\pi kj/N}\, e^{-i2\pi kj'/N} = N\delta_{j,j'} \tag{9.36}$$

we obtain the inverse Fourier transform

$$f(j\Delta) = \frac{1}{N}\sum_{k=-N/2}^{N/2} g(\omega_k)\, e^{i2\pi kj/N} = \frac{1}{N}\sum_{k=-N/2}^{N/2} g(\omega_k)\, e^{i\omega_k t_j}. \tag{9.37}$$

The frequencies ω_k for $k > N/2$ are greater than the Nyquist frequency ω_Q. We can interpret the frequencies for $k > N/2$ as negative frequencies equal to $(k-N)\omega_0$ (see Problem 9.13). The occurrence of negative frequency components is a consequence of the use of the exponential functions rather than sines and cosines. Note that $f(t)$ is real if $g(-\omega_k) = g(\omega_k)$ because the $\sin\omega_k$ terms in (9.37) cancel due to symmetry.

The calculation of a single Fourier coefficient using (9.30) requires approximately $\mathcal{O}(N)$ multiplications. Because the complete Fourier transform contains N complex coefficients, the calculation requires $\mathcal{O}(N^2)$ multiplications and may require hours to complete if the sample contains just a few megabytes of data. Because many of the calculations are redundant, it is possible to organize the calculation so that the computational time is order $N \log N$. Such an algorithm is called a *fast Fourier transform* (FFT) and is discussed in Appendix 9B. The improvement in speed is dramatic. A dataset containing 10^6 points requires $\approx 6 \times 10^6$ multiplications rather than $\approx 10^{12}$. Because we will use this algorithm to study diffraction and other phenomena, and because coding this algorithm is nontrivial, we have provided an implementation of the FFT in the Open Source Physics numerics package. We can use this FFT class to transform between time and frequency or position and wavenumber. The FFTApp program shows how the FFT class is used.

Listing 9.7: The FFTApp program computes the fast Fourier transform of a function and displays the coefficients.

```
package org.opensourcephysics.sip.ch09;
import java.text.DecimalFormat;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.numerics.FFT;

public class FFTApp extends AbstractCalculation {
   public void calculate() {
      // output format
      DecimalFormat decimal = new DecimalFormat("0.0000");
      // number of Fourier coefficients
      int N = 8;
      // array that will be transformed
      double[] z = new double[2*N];
      // FFT implementation for N points
      FFT fft = new FFT(N);
      // mode or harmonic of e^(i*x)
      int mode = control.getInt("mode");
      double x = 0, delta = 2*Math.PI/N;
      for(int i = 0;i<N;i++) {
         z[2*i] = Math.cos(mode*x);    // real component of e^(i*mode*x)
         z[2*i+1] = Math.sin(mode*x); // imaginary component of e^(i*mode*x)
         x += delta;                   // increase x
      }
      // transform data; data will be in wrap-around order
      fft.transform(z);
      for(int i = 0;i<N;i++) {
         System.out.println("index = "+i+"\t real = "
                            +decimal.format(z[2*i])+"\t imag = "
                            +decimal.format(z[2*i+1]));
      }
   }

   public void reset() {
      control.setValue("mode", -1);
   }

   public static void main(String[] args) {
      CalculationControl.createApp(new FFTApp());
   }
}
```

The FFT class replaces the data in the input array with transformed values. (Make an array copy if you need to retain the original data.) Because the transformation assumes N complex data points and Java does not support a primitive complex data type, the input array has length $2N$. The real part of the nth data point is located in array element $2n$ and the imaginary part is in element $2n+1$. The transformed data maintains this same ordering of real and imaginary parts. We test the FFT class in Problem 9.12.

Problem 9.12. The FFT class

(a) The FFTApp initializes the array that will be transformed by evaluating the following complex function:

$$f_n = f(n\Delta) = e^{in\Delta} = \cos n\Delta + i \sin n\Delta \tag{9.38}$$

where n is an integer and $\Delta = 2\pi/N$ is the interval between data points. We refer to n as the mode variable in the program. What happens to the Fourier component if the phase of the complex exponential is shifted by α? In other words, what happens if the data is initialized using $f_n = f(n\Delta) = e^{in\Delta+\alpha}$?

(b) Modify and run FFTApp to show that the fast Fourier transform produces a single component only if the grid contains an integer number of wavelengths.

(c) Change the number of grid points N and show that the value of the nonzero Fourier coefficient is equal to N if the input function is (9.38). Note that some other FFT implementations normalize the result by dividing by N.

(d) Show that the original function can be recovered by invoking the FFT.inverse method. □

Problem 9.13. Negative frequencies

(a) Use (9.38) as the input function and show that FFTApp produces the same Fourier coefficients if $\omega = 2\pi/(N\Delta)$ or $\omega = -2\pi/(N\Delta)$. Repeat for $\omega = 4\pi/(N\Delta)$ and $\omega = -4\pi/(N\Delta)$.

(b) Compute the Fourier coefficients using $f_n = f(n\Delta) = \cos n\Delta$, where $n = 0, 1, 2, \ldots, N-1$. Repeat using a sine function. Interpret your results using

$$\cos\theta = \frac{e^{i\theta} + e^{-i\theta}}{2} \tag{9.39a}$$

$$\sin\theta = \frac{e^{i\theta} - e^{-i\theta}}{2}. \tag{9.39b}$$

□

As shown in Problem 9.13, the Fourier coefficient indices for $n \geq N/2$ should be interpreted as negative frequencies. The transformed data still contains N frequencies, and these frequencies are still separated by $2\pi/(N\Delta)$.

Because the frequencies switch signs at the array's midpoint index $N/2$, we refer to the transformed data as being in *wrap-around order*. The toNaturalOrder method can be used to sort and normalize the Fourier components in order of increasing frequency starting at $\omega_Q = -\pi/\Delta$ and continuing to $\omega_Q = \pi/\Delta((N-2)/N)$ if N is even and continuing to $\omega_Q = \pi/\Delta$ if N is odd.

Exercise 9.14. Natural order

Invoke the toNaturalOrder method after performing the FFT in the FFTApp program. Modify the print statement so that the natural frequency is shown and repeat Problem 9.13. If N is even, the Fourier components have a frequency separation $\Delta\omega$ given by

$$\Delta\omega = \frac{2\pi}{N\Delta}. \tag{9.40}$$

What is the frequency separation if N is odd? □

As we have seen, computing Fourier transformations is straightforward but requires a fair amount of bookkeeping. To simplify the process, we have defined the FFTFrame class in the frames package to perform a FFT and display the coefficients. This utility class accepts either data arrays or functions as input parameters in the doFFT method. The code shown in Listing 9.8 transforms an input array. We use the FFTFrame in Problem 9.12.

Listing 9.8: The FFTCalculationApp displays the coefficients of the function $e^{2\pi nx}$.

```java
package org.opensourcephysics.sip.ch09;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.frames.FFTFrame;

public class FFTCalculationApp extends AbstractCalculation {
   FFTFrame frame = new FFTFrame("frequency", "amplitude",
                       "FFT Frame Test");

   public void calculate() {
      double xmin = control.getDouble("xmin");
      double xmax = control.getDouble("xmax");
      int n = control.getInt("N");
      double
         xi = xmin, delta = (xmax-xmin)/n;
      double[] data = new double[2*n];
      int mode = control.getInt("mode");
      for(int i = 0;i<n;i++) {
         data[2*i] = Math.cos(mode*xi);
         data[2*i+1] = Math.sin(mode*xi);
         xi += delta;
      }
      frame.doFFT(data, xmin, xmax);
      frame.showDataTable(true);
   }
```

```
    public void reset() {
        control.setValue("mode", 1);
        control.setValue("xmin", 0);
        control.setValue("xmax", "2*pi");
        control.setValue("N", 32);
        calculate();
    }

    public static void main(String[] args) {
        CalculationControl.createApp(new FFTCalculationApp());
    }
}
```

Problem 9.15. Spatial Fourier transforms and phase

So far we have considered only nonnegative values of t for functions $f(t)$. Spatial Fourier transforms are of interest in many contexts, and these transforms usually involve both positive and negative values of x.

(a) Write a program using a `CalculationControl` that computes the real and imaginary parts of the Fourier transform $\phi(q)$ of a complex function $\psi(x) = f(x) + ig(x)$, where $f(x)$ and $g(x)$ are real and x has both positive and negative values. Note that the wavenumber $q = 2\pi/L$ is analogous to the angular frequency $\omega = 2\pi/T$.

(b) Compute the Fourier transform of the Gaussian function $\psi(x) = e^{-bx^2}$ in the interval $[-5,5]$. Examine $\psi(x)$ and $\phi(q)$ for at least three values of b such that the Gaussian is contained within the interval. Does $\phi(q)$ appear to be a Gaussian? Choose a reasonable criterion for the half-width of $\psi(x)$ and measure its value. Use the same criterion to measure the half-width of $\phi(q)$. How do these widths depend on b? How does the width of $\phi(q)$ change as the width of $\psi(x)$ increases?

(c) Repeat part (b) with the function $\psi(x) = Ae^{-b(x-x_0)^2}$ for various values of x_0. What effect does shifting the peak have on $\phi(q)$?

(d) Repeat part (b) with the function $\psi(x) = Ae^{-bx^2}e^{iq_0x}$ for various values of q_0. What effect does the phase oscillation have on $\phi(q)$? □

9.4 Two-Dimensional Fourier Series

The extension of the ideas of Fourier analysis to two dimensions is simple and direct. We will use two-dimensional FFTs when we study diffraction in Section 9.9.

If we assume a function of two variables $f(x,y)$, then a two-dimensional series is constructed using harmonics of both variables. The basis functions are the products of one dimensional basis functions $e^{ixq_x}e^{iyq_y}$, and the Fourier series is written as a sum of these harmonics:

$$f(x,y) = \sum_{n=-N/2}^{N/2} \sum_{m=-M/2}^{M/2} c_{n,m}\, e^{iq_n x} e^{iq_m y} \tag{9.41}$$

where

$$q_n = \frac{2\pi n}{X} \quad \text{and} \quad q_m = \frac{2\pi m}{Y}. \tag{9.42}$$

The function $f(x, y)$ is assumed to be periodic in both x and y with periods X and Y, respectively. The Fourier coefficients are again calculated by integrating the product of the function with a basis function:

$$c_{n,m} = \int_{-X/2}^{X/2} \int_{-Y/2}^{Y/2} f(x, y) e^{i(q_n x + q_m y)} \, dx \, dy. \tag{9.43}$$

Because of the large number of coefficients $c_{n,m}$, the discrete two-dimensional Fourier transform is best implemented using the FFT algorithm. The `FFT2DCalculationApp` program shows how to compute a two-dimensional FFT using the `FFT2DFrame` utility class.

Listing 9.9: The `FFT2DCalculationApp` program computes the two-dimensional fast Fourier transform of a function and shows the resulting coefficients using a grid plot.

```
package org.opensourcephysics.sip.ch09;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.frames.FFT2DFrame;

public class FFT2DCalculationApp extends AbstractCalculation {
   FFT2DFrame frame = new FFT2DFrame("k_x", "k_y", "2D FFT");

   public void calculate() {
      int xMode = control.getInt("x mode"),
          yMode = control.getInt("y mode");
      double xmin = control.getDouble("xmin");
      double xmax = control.getDouble("xmax");
      int nx = control.getInt("Nx");
      double ymin = control.getDouble("ymin");
      double ymax = control.getDouble("ymax");
      int ny = control.getInt("Ny");
      // data stored in row-major format
      double[] zdata = new double[2*nx*ny];
      double y = 0, yDelta = 2*Math.PI/ny;
      for(int iy = 0;iy<ny;iy++) { // loop over rows in array
      // offset to beginning of a row;  each row is nx long
         int offset = 2*iy*nx;
         double x = 0, xDelta = 2*Math.PI/nx;
         for(int ix = 0;ix<nx;ix++) {
            // z function is e^(i*xmode*x)e^(i*ymode*y)
            zdata[offset+2*ix] = //real part
               Math.cos(xMode*x)*Math.cos(yMode*y)
               -Math.sin(xMode*x)*Math.sin(yMode*y);
            zdata[offset+2*ix+1] = // imaginary part
               Math.sin(xMode*x)*Math.cos(yMode*y)
               +Math.cos(xMode*x)*Math.sin(yMode*y);
            x += xDelta;
         }
         y += yDelta;
      }
      frame.doFFT(zdata, nx, xmin, xmax, ymin, ymax);
   }
   public void reset() {
      control.setValue("x mode", 0);
      control.setValue("y mode", 1);
      control.setValue("xmin", 0);
      control.setValue("xmax", "2*pi");
      control.setValue("ymin", 0);
      control.setValue("ymax", "2*pi");
      control.setValue("Nx", 16);
      control.setValue("Ny", 16);
   }

   public static void main(String[] args) {
      CalculationControl.createApp(new FFT2DCalculationApp());
   }
}
```

The `FFT2DFrame` is based on FFT routines contributed to the GNU Scientific Library (GSL) by Brian Gough and adapted to Java by Bruce Miller at NIST. We initialize our data array to conform to the GSL API using a one-dimensional array such that rows follow sequentially. This ordering is known as row-major format. Because the input function is assumed to be complex, the array has dimension $2N_xN_y$, where N_x and N_y are the number of grid points in the x and y direction, respectively. The `FFT2DFrame` object transforms and displays the data when the `doFFT` method is invoked.

Exercise 9.16. Two-dimensional FFT

Write a program to transform a two-dimensional Gaussian using the `FFT2DFrame` class. Note that color is used to represent the complex phase. What happens if the Gaussian is not centered on the grid? □

9.5 Fourier Integrals

Fourier analysis can be extended to approximate waveforms that do not repeat themselves by converting the Fourier sum over discrete frequency components to an integral. The *Fourier integral* transforms a continuous function of time $f(t)$ into a continuous function of frequency $g(\omega)$ as follows:

$$g(\omega) = \int_{-\infty}^{\infty} f(t) e^{i\omega t}\, dt. \tag{9.44}$$

The inverse transformation reverses this process:

$$f(t) = \frac{1}{2\pi} \int_{-\infty}^{\infty} g(\omega) e^{i\omega t}\, d\omega. \tag{9.45}$$

Equations (9.44) and (9.45) are known as a Fourier transform pair.

Because we need to store functions such as $f(t)$ and $g(\omega)$ at a discrete number of points, Fourier integrals are usually approximated using Fourier series with a large number of terms and a large sampling time. In other words, the time over which the signal is measured is large compared to the period of interest. Exercise 9.17 illustrates how a Fourier series for a pulse train approaches a continuous frequency spectrum as the period approaches infinity.

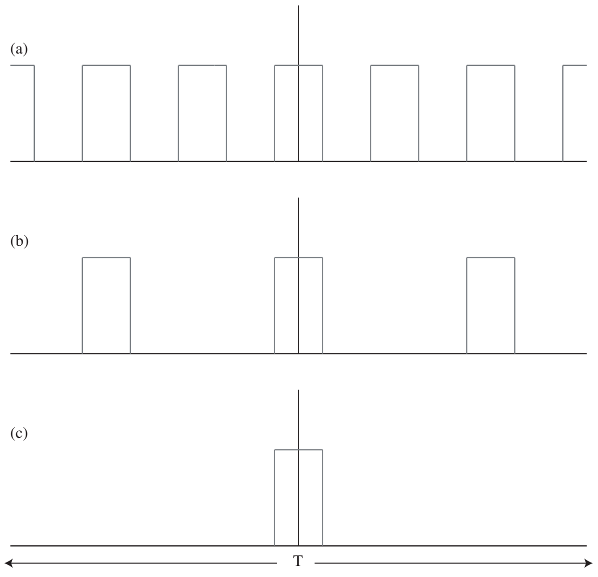


Figure 9.2: A series of pulses with increasing periods.

Exercise 9.17. Fourier integral

Write a program to plot the frequency spectrum of the waveforms shown in Figure 9.2. Use the FFT algorithm. How does the frequency spectrum change as the waveform becomes less periodic? How does the finite time interval affect the result? □

9.6 Power Spectrum

The power output of a periodic electrical signal $f(t)$ is proportional to the integral of the signal squared:

$$P = \frac{1}{T}\int_0^T |f(t)|^2\,dt. \tag{9.46}$$

Another way to look at the power is to calculate the power P associated with the Fourier components ω_k of a signal that has been sampled at regular intervals. If we substitute (9.31) into (9.46), rearrange terms, and apply the orthogonality condition (9.14), we obtain

$$P = \sum_{j=-N/2}^{N/2}\sum_{k=-N/2}^{N/2} c_j^* c_k \frac{1}{T}\int_0^T e^{i(\omega_k-\omega_j)t}\,dt = \sum_{k=-N/2}^{N/2} |c_k|^2. \tag{9.47}$$

From (9.47) we conclude that the average power of $|f(t)|^2$ is the sum of the power in each frequency component $P(\omega_k) = |c_k|^2$. This result is one form of Parseval's theorem.

In most measurements, the function $f(t)$ corresponds to an amplitude, and the power or intensity is proportional to the square of this amplitude, or for complex functions, the modulus squared. The *power spectrum* $P(\omega_k)$ is proportional to the power associated with a particular frequency component embedded in the quantity of interest. Well-defined peaks in $P(\omega_k)$ often correspond to normal mode frequencies.

What happens to the power associated with frequencies greater than the Nyquist frequency? To answer this question, consider two choices of the Nyquist frequency ω_{Q_a} and $\omega_{Q_b} > \omega_{Q_a}$ and the corresponding sampling times $\Delta_b < \Delta_a$. The calculation with $\Delta = \Delta_b$ is more accurate because the sampling time is smaller. Suppose that this calculation of the spectrum yields the result that $P(\omega > \omega_a) > 0$. What happens if we compute the power spectrum using $\Delta = \Delta_a$? The power associated with $\omega > \omega_a$ must be "folded" back into the $\omega < \omega_a$ frequency components. For example, the frequency component at $\omega + \omega_a$ is added to the true value at $\omega - \omega_a$ to produce an incorrect value at $\omega - \omega_a$ in the computed power spectrum. This phenomenon is called *aliasing* and leads to spurious results. Aliasing occurs in calculations of $P(\omega)$ if the latter does not vanish above the Nyquist frequency. To avoid aliasing, it is necessary to sample more frequently or to remove the high frequency components from the signal before sampling the data.

Although the power spectrum can be computed by a simple modification of `AnalyzeApp`, it is a good idea to use the FFT for many of the following problems.

Problem 9.18. Aliasing

Sample the sinusoidal function, $\sin 2\pi t$, and display the resulting power spectrum using sampling frequencies above and below the Nyquist frequency. Start with a sampling time of $\Delta = 0.1$ and increase the time until $\Delta = 10.0$.

(a) Is the power spectrum sharp? That is, is all the power located in a single frequency? Does your answer depend on the ratio of the period to the sampling time?

(b) Explain the appearance of the power spectrum for $\Delta = 1.25$, $\Delta = 1.75$, and $\Delta = 2.5$.

(c) What is the power spectrum if you sample at the Nyquist frequency or twice the Nyquist frequency? □

Problem 9.19. Examples of power spectra

(a) Create a data set with N points corresponding to $f(t) = 0.3\cos(2\pi t/T) + r$, where r is a uniform random number between 0 and 1 and $T = 4$. Plot $f(t)$ versus t in time intervals of $\Delta = 4T/N$ for $N = 128$. Can you visually detect the periodicity? Compute the power spectrum using the same sampling interval $\Delta = 4T/N$. Does the frequency dependence of the power spectrum indicate that there are any special frequencies? Repeat with $T = 16$. Are high or low frequency signals easier to pick out from the random background?

(b) Simulate a one-dimensional random walk and compute the time series $x^2(t)$, where $x(t)$ is the distance from the origin of the walk after t steps. Average $x^2(t)$ over several trials. Compute the power spectrum for a walk of $t \leq 256$. In this case $\Delta = 1$, the time between steps. Do you observe any special frequencies?

(c) Let f_n be the nth member of a random number sequence. As in part (b), $\Delta = 1$. Compute the power spectrum of the random number generator. Do you detect any periodicities? If so, is the random number generator acceptable? □

Problem 9.20. Power spectrum of coupled oscillators

(a) Modify your program developed in Problem 9.2 so that the power spectrum of the position of one of the N particles is computed at the end of the simulation. Set $\Delta = 0.1$ so that the Nyquist frequency is $\omega_Q = \pi/\Delta \approx 31.4$. Choose the time of the simulation equal to $T = 25.6$ and let $k/m = 1$. Plot the power spectrum $P(\omega)$ at frequency intervals equal to $\Delta\omega = \omega_0 = 2\pi/T$. First choose $N = 2$ and choose the initial conditions so that the system is in a normal mode. What do you expect the power spectrum to look like? What do you find? Then choose $N = 10$ and choose initial conditions corresponding to various normal modes. Is the power spectrum the same for all particles?

(b) Repeat part (a) for $N = 2$ and $N = 10$ with random initial particle displacements between -0.5 and $+0.5$ and zero initial velocities. Can you detect all the normal modes in the power spectrum? Repeat for a different set of random initial displacements.

(c) Repeat part (a) for initial displacements corresponding to the equal sum of two normal modes. Do the power spectrum show two peaks? Are these peaks of equal height?

(d) Recompute the power spectrum for $N = 10$ with $T = 6.4$. Is this time long enough? How can you tell? □

Problem 9.21. Quasiperiodic power spectra

(a) Write a program to compute the power spectrum of the circle map (6.62). Begin by exploring the power spectrum for $K = 0$. Plot $\ln P(\omega)$ versus ω, where $P(\omega)$ is proportional to the modulus squared of the Fourier transform of x_n. Begin with 256 iterations. How do the power spectra differ for rational and irrational values of the parameter Ω? How are the locations of the peaks in the power spectra related to the value of Ω?

(b) Set $K = 1/2$ and compute the power spectra for $0 < \Omega < 1$. Does the power spectra differ from the spectra found in part (a)?

(c) Set $K = 1$ and compute the power spectra for $0 < \Omega < 1$. How does the power spectra compare to those found in parts (a) and (b)? □

In Problem 9.20 we found that the peaks in the power spectrum yield information about the normal mode frequencies. In Problems 9.22 and 9.23 we compute the power spectra for a system of coupled oscillators with disorder. Disorder can be generated by having random masses or random spring constants (or both). We will see that one effect of disorder is that the normal modes are no longer simple sinusoidal functions. Instead, some of the modes are localized, meaning that only some of the particles move significantly while the others remain essentially at rest. This effect is known as *Anderson localization*. Typically, we find that modes above a certain frequency are *localized*, and those below this threshold frequency are *extended*. The threshold frequency is well defined for large systems. All states are localized in the limit of an infinite chain with any amount of disorder. The dependence of localization on disorder in systems of coupled oscillators in higher dimensions is more complicated.

Problem 9.22. Localization with a single defect

(a) Modify your program developed in Problem 9.2 so that the mass of one oscillator is equal to one fourth that of the others. Set $N = 20$ and use fixed boundary conditions. Compute the power spectrum over a time $T = 51.2$ using random initial displacements between -0.5 and $+0.5$ and zero initial velocities. Sample the data at intervals of $\Delta = 0.1$. The normal mode frequencies correspond to the well-defined peaks in $P(\omega)$. Consider at least three different sets of random initial displacements to insure that you find all the normal mode frequencies.

(b) Apply an external force $F_e = 0.3 \sin \omega t$ to each particle. (The steady-state behavior occurs sooner if we apply an external force to each particle instead of just one particle.) Because the external force pumps energy into the system, it is necessary to add a damping force to prevent the oscillator displacements from becoming too large. Add a damping force equal to $-\gamma v_i$ to all the oscillators with $\gamma = 0.1$. Choose random initial displacements and zero initial velocities and use the frequencies found in part (a) as the driving frequencies ω. Describe the motion of the particles. Is the system driven to a normal mode? Take a "snapshot" of the particle displacements after the system has run for a sufficiently long time, so that the patterns repeat themselves. Are the particle displacements simple sinusoidal functions? Sketch the approximate normal mode patterns for each normal mode frequency. Which of the modes appear localized and which modes appear to be extended? What is the approximate cutoff frequency that separates the localized from the extended modes? □

Problem 9.23. Localization in a disordered chain of oscillators

(a) Modify your program so that the spring constants can be varied by the user. Set $N = 10$ and use fixed boundary conditions. Consider the following set of 11 spring constants: 0.704, 0.388, 0.707, 0.525, 0.754, 0.721, 0.006, 0.479, 0.470, 0.574, 0.904. To help you determine all the normal modes, we provide two of the normal mode frequencies: $\omega \approx 0.28$ and 1.15. Find the power spectrum using the procedure outlined in Problem 9.22a.

(b) Apply an external force $F_e = 0.3 \sin \omega t$ to each particle and find the normal modes as outlined in Problem 9.22b.

(c) Repeat parts (a) and (b) for another set of random spring constants for $N = 40$. Discuss the nature of the localized modes in terms of the specific values of the spring constants. For example, is the edge of a localized mode at a spring that has a relatively large or small spring constant?

(d) Repeat parts (a) and (b) for uniform spring constants but random masses between 0.5 and 1.5. Is there a qualitative difference between the two types of disorder? □

In 1955 Fermi, Pasta, and Ulam used the Maniac I computer at Los Alamos to study a chain of oscillators. Their surprising discovery might have been the first time a qualitatively new result, instead of a more precise number, was found from a simulation. To understand their results, we need to discuss an idea from statistical mechanics that was discussed in Project 8.23.

A fundamental assumption of statistical mechanics is that an isolated system of particles is quasi-ergodic; that is, the system will evolve through all configurations consistent with the conservation of energy. A system of linearly coupled oscillators is not quasi-ergodic, because if the system is initially in a normal mode, it stays in that normal mode forever. Before 1955 it was believed that if the interaction between the particles is weakly nonlinear (and the number of particles is sufficiently large), the system would be quasi-ergodic and evolve through the different normal modes of the linear system. In Problem 9.24 we will find, as did Fermi, Pasta, and Ulam, that the behavior of the system is much more complicated. The question of ergodicity in this system is known as the FPU problem.

Problem 9.24. Nonlinear oscillators

(a) Modify your program so that cubic forces between the particles are added to the linear spring forces. That is, let the force on particle i due to particle j be

$$F_{ij} = -(u_i - u_j) - \alpha(u_i - u_j)^3 \tag{9.48}$$

where α is the amplitude of the nonlinear term. Choose the masses of the particles to be unity. Consider $N = 10$ and choose initial displacements corresponding to a normal mode of the linear ($\alpha = 0$) system. Compute the power spectrum over a time $T = 51.2$ with $\Delta = 0.1$ for $\alpha = 0$, 0.1, 0.2, and 0.3. For what value of α does the system become ergodic; that is, for what value of α are the heights of all the normal mode peaks approximately the same?

(b) Repeat part (a) for the case where the displacements of the particles are initially random. Use the same set of random displacements for each value of α.

(c)* We now know that the number of oscillators is not as important as the magnitude of the nonlinear interaction. Repeat parts (a) and (b) for $N = 20$ and 40 and discuss the effect of increasing the number of particles. □

9.7 Wave Motion

Our simulations of coupled oscillators have shown that the microscopic motion of the individual oscillators leads to macroscopic wave phenomena. To understand the transition between microscopic and macroscopic phenomena, we reconsider the oscillations of a linear chain of N particles with equal spring constants k and equal masses m. As we found in Section 9.1, the equations of motion of the particles can be written as [see (9.1)]

$$\frac{d^2u_j(t)}{dt^2} = -\frac{k}{m}\Big[2u_j(t) - u_{j+1}(t) - u_{j-1}(t)\Big] \qquad (j = 1,\ldots,N). \tag{9.49}$$

We consider the limits $N \to \infty$ and $a \to 0$ with the length of the chain Na fixed. We will find that the discrete equations of motion (9.49) can be replaced by the continuous *wave equation*

$$\frac{\partial^2 u(x,t)}{\partial t^2} = c^2\,\frac{\partial^2 u(x,t)}{\partial x^2} \tag{9.50}$$

where c has the dimension of velocity.

We obtain the wave equation (9.50) as follows. First we replace $u_j(t)$, where j is a discrete variable, by the function $u(x,t)$, where x is a continuous variable, and rewrite (9.49) in the form

$$\frac{\partial^2 u(x,t)}{\partial t^2} = \frac{ka^2}{m}\,\frac{1}{a^2}\Big[u(x+a,t) - 2u(x,t) + u(x-a,t)\Big]. \tag{9.51}$$

We have written the time derivative as a partial derivative because the function u depends on two variables. If we use the Taylor series expansion

$$u(x \pm a) = u(x) \pm a\frac{du}{dx} + \frac{a^2}{2}\frac{d^2u}{dx^2} + \ldots \tag{9.52}$$

it is easy to show that as $a \to 0$, the quantity

$$\frac{1}{a^2}\Big[u(x+a,t) - 2u(x,t) + u(x-a,t)\Big] \to \frac{\partial^2 u(x,t)}{\partial x^2}. \tag{9.53}$$

(We have written a spatial derivative as a partial derivative for the same reason as before.) The wave equation (9.50) is obtained by substituting (9.53) into (9.51) with $c^2 = ka^2/m$. If we introduce the linear mass density $\mu = M/a$ and the tension $T = ka$, we can express c in terms of μ and T and obtain the familiar result $c^2 = T/\mu$.

It is straightforward to show that any function of the form $f(x \pm ct)$ is a solution to (9.50). Among these many solutions to the wave equation are the familiar forms:

$$u(x,t) = A\cos\frac{2\pi}{\lambda}(x \pm ct) \tag{9.54a}$$

$$u(x,t) = A\sin\frac{2\pi}{\lambda}(x \pm ct). \tag{9.54b}$$

Because the wave equation is linear and hence, satisfies a superposition principle, we can understand the behavior of a wave of arbitrary shape by representing its shape as a sum of sinusoidal waves.

One way to solve the wave equation (9.50) numerically is to retrace our steps back to the discrete equations (9.49) to find a discrete form of the wave equation that is convenient for numerical calculations. The conversion of a continuum equation to a physically motivated discrete form frequently leads to useful numerical algorithms. From (9.53) we see how to approximate the second derivative by a finite difference. If we replace a by Δx and take Δt to be the time step, we can rewrite (9.49) by

$$\frac{1}{(\Delta t)^2}[u(x,t+\Delta t) - 2u(x,t) + u(x,t-\Delta t)] = \\ \frac{c^2}{(\Delta x)^2}[u(x+\Delta x,t) - 2u(x,t) + u(x-\Delta x,t)]. \tag{9.55}$$

The quantity Δx is the spatial interval. The result of solving (9.55) for $u(x,t+\Delta t)$ is

$$\begin{aligned} u(x,t+\Delta t) &= 2(1-b)u(x,t) \\ &\quad + b[u(x+\Delta x,t) + u(x-\Delta x,t)] - u(x,t-\Delta t) \end{aligned} \tag{9.56}$$

where $b = (c\Delta t/\Delta x)^2$. Equation (9.56) expresses the displacements at time $t+\Delta t$ in terms of the displacements at the current time t and at the previous time $t-\Delta t$.

Problem 9.25. Solution of the discrete wave equation

(a) Write a program to compute the numerical solutions of the discrete wave equation (9.56). Three spatial arrays corresponding to $u(x)$ at times $t+\Delta t$, t, and $t-\Delta t$ are needed. Denote the displacement $u(j\Delta x)$ by the array element `u[j]` where $j = 0,\ldots,N+1$. Use periodic boundary conditions so that $u_0 = u_N$ and $u_1 = u_{N+1}$. Draw lines between the displacements at neighboring values of x. Note that the initial conditions require the specification of u at $t = 0$ and at $t = -\Delta t$. Let the waveform at $t = 0$ and $t = -\Delta t$ be $u(x,t=0) = \exp(-(x-10)^2)$ and $u(x,t=-\Delta t) = \exp(-(x-10+c\Delta t)^2)$, respectively. What is the direction of motion implied by these initial conditions?

(b) Our first task is to determine the optimum value of the parameter b. Let $\Delta x = 1$ and $N \geq 100$ and try the following combinations of c and Δt: $c = 1, \Delta t = 0.1$; $c = 1, \Delta t = 0.5$; $c = 1, \Delta t = 1$; $c = 1, \Delta t = 1.5$; $c = 2, \Delta t = 0.5$; and $c = 2, \Delta t = 1$. Verify that the value $b = (c\Delta t)^2 = 1$ leads to the best results; that is, for this value of b, the initial form of the wave is preserved.

(c) It is possible to show that the discrete form of the wave equation with $b = 1$ is exact up to numerical roundoff error (cf. DeVries). Hence, we can replace (9.56) by the simpler algorithm

$$u(x, t + \Delta t) = u(x + \Delta x, t) + u(x - \Delta x, t) - u(x, t - \Delta t). \tag{9.57}$$

That is, the solutions of (9.57) are equivalent to the solutions of the original partial differential equation (9.50). Try several different initial waveforms and show that if the displacements have the form $f(x \pm ct)$, the waveform maintains its shape with time. For the remaining problems, we will use (9.57) corresponding to $b = 1$. Unless otherwise specified, choose $c = 1$, $\Delta x = \Delta t = 1$, and $N \geq 100$ in the following problems. □

Problem 9.26. Velocity of waves

(a) Use the waveform given in Problem 9.25a and verify that the speed of the wave is unity by determining the distance traveled in a given amount of time. Because we have set $\Delta x = \Delta t = 1$ and $b = 1$, the speed $c = 1$. (A way of incorporating different values of c is discussed in Problem 9.27c.)

(b) Replace the waveform considered in part (a) by a sinusoidal wave that fits exactly; that is, choose $u(x, t) = \sin(qx - \omega t)$ such that $\sin q(N + 1) = 0$. Measure the period T of the wave by measuring the time it takes for successive maxima to pass a given point. What is the wavelength λ of the wave? Does it depends on the value of q? The frequency of the wave is given by $f = 1/T$. Verify that $\lambda f = c$. □

Problem 9.27. Reflection of waves

(a) Consider a wave of the form $u(x, t) = e^{-(x-10-ct)^2}$. Use fixed boundary conditions so that $u_0 = u_{N+1} = 0$. What happens to the reflected wave?

(b) Modify your program so that free boundary conditions are incorporated: $u_0 = u_1$ and $u_N = u_{N+1}$. Compare the phase of the reflected wave to your result from part (a).

(c) What happens to a pulse at the boundary between two media? Set $c = 1$ and $\Delta t = 1$ on the left side of your grid and $c = 2$ and $\Delta t = 0.5$ on the right side. These choices of c and Δt imply that $b = 1$ on both sides, but that the right side is updated twice as often as the left side. What happens to a pulse that begins on the left side and moves to the right? Is there both a reflected and transmitted wave at the boundary between the two media? What is their relative phase? Find a relation between the amplitude of the incident pulse and the amplitudes of the reflected and transmitted pulses. Repeat for a pulse starting from the right side. □

Problem 9.28. Superposition of waves

(a) Consider the propagation of the wave determined by $u(x, t = 0) = \sin(4\pi x/N)$. What must $u(x, -\Delta t)$ be so that the wave moves in the positive x direction? Test your answer by doing a simulation. Use periodic boundary conditions. Repeat for a wave moving in the negative x direction.

(b) Simulate two waves moving in opposite directions each with the same spatial dependence given by $u(x,0) = \sin(4\pi x/N)$. Describe the resultant wave pattern. Repeat the simulation for $u(x,0) = \sin(8\pi x/N)$.

(c) Assume that $u(x,0) = \sin q_1 x + \sin q_2 x$ with $q_1 = 10\pi/N$ and $q_2 = 12\pi/N$. Describe the qualitative form of $u(x,t)$ for fixed t. What is the distance between modulations of the amplitude? Estimate the wavelength associated with the fine ripples of the amplitude. Estimate the wavelength of the envelope of the wave. Find a simple relation for these two wavelengths in terms of the wavelengths of the two sinusoidal terms. This phenomena is known as *beats*.

(d) Consider the motion of the two Gaussian pulses moving in opposite directions, $u_1(x,0) = e^{-(x-10)^2}$ and $u_2(x,0) = e^{-(x-90)^2}$. Choose the array at $t = -\Delta t$ as in Problem 9.25. What happens to the two pulses when they overlap or partially overlap? Do they maintain their shape? While they are going through each other, is the displacement $u(x,t)$ given by the sum of the displacements of the individual pulses? □

Problem 9.29. Standing waves

(a) In Problem 9.28c we considered a *standing wave*, the continuum analog of a normal mode of a system of coupled oscillators. As is the case for normal modes, each point of the wave has the same time dependence. For fixed boundary conditions, the displacement is given by $u(x,t) = \sin qx \cos \omega t$, where $\omega = cq$, and the wavenumber q is chosen so that $\sin qN = 0$. Choose an initial condition corresponding to a standing wave for $N = 100$. Describe the motion of the particles and compare it with your observations of standing waves on a rope.

(b) Establish a standing wave by displacing one end of a system periodically. The other end is fixed. Let $u(x,0) = u(x,-\Delta t) = 0$, and $u(x = 0,t) = A \sin \omega t$ with $A = 0.1$. How long must the simulation run before you observe standing waves? How large is the standing wave amplitude? □

We have seen that the wave equation can support pulses that propagate indefinitely without distortion. In addition, because the wave equation is linear, the sum of any two solutions is also a solution, and the principle of superposition is satisfied. As a consequence, we know that two pulses can pass through each other unchanged. We have also seen that similar phenomena exist in the discrete system of linearly coupled oscillators. What happens if we create a pulse in a system of nonlinear oscillators? As an introduction to nonlinear wave phenomena, we consider a system of N coupled oscillators with the potential energy of interaction given by

$$V = \frac{1}{2}\sum_{j=1}^{N}\left[e^{-(u_j-u_{j-1})} - 1\right]^2. \tag{9.58}$$

This form of the interaction is known as the Morse potential. All parameters in the potential (such as the overall strength of the potential) have been set to unity. The force on the jth particle is

$$F_j = -\frac{\partial V}{\partial u_j} = Q_j(1 - Q_j) - Q_{j+1}(1 - Q_{j+1}) \tag{9.59a}$$

where

$$Q_j = e^{-(u_j-u_{j-1})}. \tag{9.59b}$$

In linear systems it is possible to set up a pulse of any shape and maintain the shape of the pulse indefinitely. In a nonlinear system, there also exist solutions that maintain their shape, but we will find in Problem 9.30 that not all pulse shapes do so. The pulses that maintain their shape are called *solitons*.

Problem 9.30. Solitons

(a) Modify the program developed in Problem 9.2 so that the force on particle j is given by (9.59). Use periodic boundary conditions. Choose $N \geq 60$ and an initial pulse of the form $u(x,t) = 0.5\, e^{-(x-10)^2}$. You should find that the initial pulse splits into two pulses plus some noise. Describe the motion of the pulses (solitons). Do they maintain their shape, or is this shape modified as they move? Describe the motion of the particles far from the pulse. Are they stationary?

(b) Save the displacements of the particles when the peak of one of the solitons is located near the center of your display. Is it possible to fit the shape of the soliton to a Gaussian? Continue the simulation and after one of the solitons is relatively isolated, set `u(j) = 0` for all `j` far from this soliton. Does the soliton maintain its shape?

(c) Repeat part (b) with a pulse given by $u(x,0) = 0$ everywhere except for $u(20,0) = u(21,0) = 1$. Do the resulting solitons have the same shape as in part (b)?

(d) Begin with the same Gaussian pulse as in part (a) and run until the two solitons are well separated. Then change at random the values of `u(j)` for particles in the larger soliton by about 5% and continue the simulation. Is the soliton destroyed? Increase the perturbation until the soliton is no longer discernible.

(e) Begin with a single Gaussian pulse as in part (a). The two resultant solitons will eventually "collide." Do the solitons maintain their shape after the collision? The principle of superposition implies that the displacement of the particles is given by the sum of the displacements due to each pulse. Does the principle of superposition hold for solitons?

(f) Compute the speeds, amplitudes, and width of the solitons produced from a single Gaussian pulse. Take the amplitude of a soliton to be the largest value of its displacement and the half-width to correspond to the value of x at which the displacement is half its maximum value. Repeat these calculations for solitons of different amplitudes by choosing the initial amplitude of the Gaussian pulse to be 0.1, 0.3, 0.5, 0.7, and 0.9. Plot the soliton speed and width versus the corresponding soliton amplitude.

(g) Change the boundary conditions to free boundary conditions and describe the behavior of the soliton as it reaches a boundary. Compare this behavior with that of a pulse in a system of linear oscillators.

(h) Begin with an initial sinusoidal disturbance that would be a normal mode for a linear system. Does the sinusoidal mode maintain its shape? Compare the behavior of the nonlinear and linear systems. □

9.8 Interference

Interference is one of the most fundamental characteristics of all wave phenomena. The term *interference* is used when there are a small number of sources, and the term *diffraction* when the

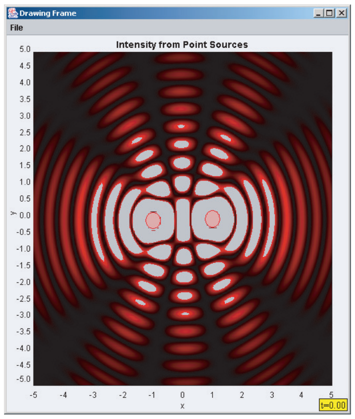


Figure 9.3: The computed energy density in the vicinity of two point sources.

number of sources is large and can be treated as a continuum. Because it is relatively easy to observe interference and diffraction phenomena with light, we discuss these phenomena in this context.

Consider the field from one or more point sources lying in a plane. The electric field at position $\mathbf{r}$ associated with the light emitted from a monochromatic point source at $\mathbf{r}_1$ is a spherical wave radiating from that point. This wave can be thought of as the real part of a complex exponential:

$$E(\mathbf{r},t) = \frac{A}{|\mathbf{r}-\mathbf{r}_1|} e^{i(q|\mathbf{r}-\mathbf{r}_1|-\omega t)} \tag{9.60}$$

where $|\mathbf{r}-\mathbf{r}_1|$ is the distance between the source and the point of observation and q is the wavenumber $2\pi/\lambda$. The superposition principle implies that the total electric field at $\mathbf{r}$ from N point sources at $\mathbf{r}_i$ is

$$E(\mathbf{r},t) = e^{-i\omega t} \sum_{n=1}^{N} \frac{A_n}{|\mathbf{r}-\mathbf{r}_n|} e^{i(q|\mathbf{r}-\mathbf{r}_n|)} = e^{-i\omega t}\mathcal{E}(\mathbf{r}). \tag{9.61}$$

The time evolution can be expressed as an oscillatory complex exponential $e^{-i\omega t}$ that multiplies a complex space part $\mathcal{E}(\mathbf{r})$. The spatial part $\mathcal{E}(\mathbf{r})$ is a *phasor* which contains both the maximum value of the electric field and the time within a cycle when the physical field reaches its maximum value. As the system evolves, the complex electric field $E(\mathbf{r},t)$ oscillates between purely real and purely imaginary values. Both the energy density (the energy per unit volume) and the light intensity (the energy passing through a unit area) are proportional to the square of the magnitude of the phasor. Because light fields oscillate at $\approx 6\times10^{14}$ Hz, typical optics experiments observe the time average (rms value) of $\mathcal{E}$ and do not observe the phase angle.

Huygens's principle states that each point on a wavefront (a surface of constant phase) can be treated as the source of a new spherical wave or *Huygens's wavelet*. The wavefront at some later time is constructed by summing these wavelets. The HuygensApp program implements Huygens's principle by assuming superposition from an arbitrary number of point sources and displaying a two-dimensional animation of (9.61) as shown in Figure 9.3 and described in Appendix 9C.

Sources are represented by circles and are added to the frame when a custom button invokes the `createSource` method.

```
public void createSource() {
   InteractiveShape ishape = InteractiveShape.createCircle(0, 0, 0.5);
   frame.addDrawable(ishape);
   initPhasors();
   frame.repaint();
}
```

Users can create as many sources as they wish. The program later retrieves a list of sources from the frame using the latter's `getDrawables` method.

The program uses $n \times n$ arrays to store the real and imaginary values. The code fragment from the `initPhasors` method shown in the following starts the process by obtaining a list of point sources in the frame. We then use an `Iterator` to access each source as we sum the vector components at each grid point.

```
ArrayList list=frame.getDrawables();    // gets list of point sources
// creates an iterator for the list
Iterator it=list.iterator();
// these two statements are combined in the final code
```

`List` and `Iterator` are interfaces implemented by the objects returned by `frame.getDrawables` and `list.iterator`, respectively. As the name implies, an iterator is a convenient way to access a list without explicitly counting its elements. The iterator's `getNext` method retrieves elements from the list, and the `hasNext` method returns true if the end of the list has not been reached.

The `initPhasors` method in HuygensApp computes the phasors at every point by summing the phasors at each grid point. Note how the distance from the source to the observation point is computed by converting the grid's index values to world coordinates.

```
Iterator it = frame.getDrawables().iterator();  // source iterator
while(it.hasNext()) {
   InteractiveShape source = (InteractiveShape) it.next();
   // world coordinates for source
   double xs = source.getX(), ys = source.getY();
   for(int ix = 0;ix<n;ix++) {
      double x = frame.indexToX(ix);
      double dx = (xs-x);  // source -> gridpoint x separation
      for(int iy = 0;iy<n;iy++) {
         double y = frame.indexToY(iy);
         double dy = (ys-y);  // charge -> gridpoint y separation
         double r = Math.sqrt(dx*dx+dy*dy);
         realArray[ix][iy] += (r==0) ? 0 : Math.cos(PI2*r)/r;
         imagArray[ix][iy] += (r==0) ? 0 : Math.sin(PI2*r)/r;
      }
   }
}
```

To calculate the real and imaginary components of the phasor, the distance from the source to the grid point is determined in terms of the wavelength λ, and the time is is determined in terms of the period T. For example, for green light one unit of distance is $\approx 5\times10^{-7}$ m and one unit of time is $\approx 1.6\times10^{-15}$ s.

The simulation is performed by multiplying the phasors by $e^{-i\omega t}$ in the `doStep` method. Multiplying each phasor by $e^{-i\omega t}$ mixes the phasor's real and imaginary components. We then obtain the physical field from (9.61) by taking the real part:

$$E(\mathbf{r},t) = \mathrm{Re}[e^{-i\omega t}\mathcal{E}(\mathbf{r})] = \mathrm{Re}[\mathcal{E}]\cos\omega t + \mathrm{Im}[\mathcal{E}]\sin\omega t. \tag{9.62}$$

Listing 9.10 shows the entire HuygensApp class. A custom button is used to create sources at the origin. Because the source is an InteractiveShape, it can be repositioned using the mouse. The program also implements the InteractiveMouseHandler interface to recalculate the phasors when the source is moved. (See Section 5.7 for a discussion of interactive handlers.)

Listing 9.10: The HuygensApp class simulates the energy density from one or more point sources.

```
package org.opensourcephysics.sip.ch09;
import java.util.*;
import java.awt.event.*;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.display.*;
import org.opensourcephysics.display2d.*;
import org.opensourcephysics.frames.*;

public class HuygensApp extends AbstractSimulation implements
               InteractiveMouseHandler {
   static final double PI2 = Math.PI*2;
   Scalar2DFrame frame = new Scalar2DFrame("x", "y",
                              "Intensity from point sources");
   double time = 0;
   double[][] realPhasor, imagPhasor, amplitude;
   int n;    // grid points on a side
   double a; // side length

   public HuygensApp() {
      // interpolated plot looks best
      frame.convertToInterpolatedPlot();
      frame.setPaletteType(ColorMapper.RED);
      frame.setInteractiveMouseHandler(this);
   }

   public void initialize() {
      n = control.getInt("grid size");
      a = control.getDouble("length");
      frame.setPreferredMinMax(-a/2, a/2, -a/2, a/2);
      realPhasor = new double[n][n];
      imagPhasor = new double[n][n];
      amplitude = new double[n][n];
      frame.setAll(amplitude);
      initPhasors();
   }

   void initPhasors() {
      for(int ix = 0;ix<n;ix++) {
         for(int iy = 0;iy<n;iy++) {
            imagPhasor[ix][iy] = realPhasor[ix][iy] = 0; // zero the phasor
         }
      }
      // an iterator for the sources in the frame
      Iterator it = frame.getDrawables().iterator(); // source iterator
      // counts the number of sources
      int counter = 0;
      while(it.hasNext()) {
         InteractiveShape source = (InteractiveShape) it.next();
         counter++;
         double
            xs = source.getX(), ys = source.getY();
         for(int ix = 0;ix<n;ix++) {
            double x = frame.indexToX(ix);
            double dx = (xs-x);
            for(int iy = 0;iy<n;iy++) {
               double y = frame.indexToY(iy);
               double dy = (ys-y);
               double r = Math.sqrt(dx*dx+dy*dy);
               realPhasor[ix][iy] += (r==0) ? 0 : Math.cos(PI2*r)/r;
               imagPhasor[ix][iy] += (r==0) ? 0 : Math.sin(PI2*r)/r;
            }
         }
      }
      double cos = Math.cos(-PI2*time);
      double sin = Math.sin(-PI2*time);
      for(int ix = 0;ix<n;ix++) {
         for(int iy = 0;iy<n;iy++) {
            // only the real part of the complex field is physical
            double re = cos*realPhasor[ix][iy]-sin*imagPhasor[ix][iy];
            amplitude[ix][iy] = re*re;
         }
      }
      frame.setZRange(false, 0, 0.2*counter); // scale the intensity
      frame.setAll(amplitude);
   }

   public void reset() {
      time = 0;
      control.setValue("grid size", 128);
      control.setValue("length", 10);
      frame.clearDrawables();
      frame.setMessage("t = "+decimalFormat.format(time));
      control.println("Source button creates a new source.");
      control.println("Drag sources after they are created.");
      initialize();
   }

   public void createSource() {
      InteractiveShape ishape = InteractiveShape.createCircle(0, 0, 0.5);
      frame.addDrawable(ishape);
      initPhasors();
      frame.repaint();
   }

   public void handleMouseAction(InteractivePanel panel, MouseEvent evt) {
      panel.handleMouseAction(panel, evt); // panel moves the source
      if(panel.getMouseAction()==InteractivePanel.MOUSE_DRAGGED) {
         initPhasors();
      }
```

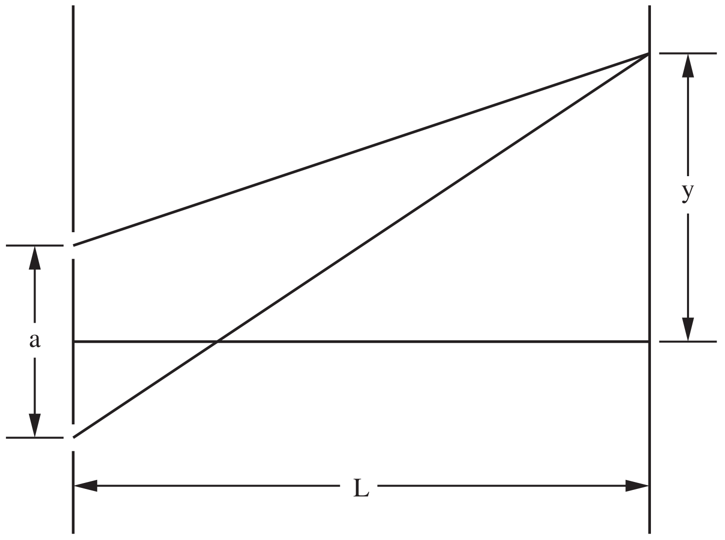


Figure 9.4: Young's double slit experiment. The figure defines the quantities a, L, and y used in Problem 9.31.

```
   }

   protected void doStep() {
      time += 0.1;
      double cos = Math.cos(-PI2*time);
      double sin = Math.sin(-PI2*time);
      for(int ix = 0;ix<n;ix++) {
         for(int iy = 0;iy<n;iy++) {
            double re = cos*realPhasor[ix][iy]-sin*imagPhasor[ix][iy];
            amplitude[ix][iy] = re*re;
         }
      }
      frame.setAll(amplitude);
      frame.setMessage("t="+decimalFormat.format(time));
   }

   public static void main(String[] args) {
      OSPControl control = SimulationControl.createApp(new HuygensApp());
      control.addButton("createSource", "Source");
   }
}
```

The classic example of interference is Young's double slit experiment (see Figure 9.4). Imagine two narrow parallel slits separated by a distance a and illuminated by a light source that emits light of only one frequency (monochromatic light). If the light source is placed on the line bisecting the two slits and the slit opening is very narrow, the two slits become coherent light sources with equal phases. We first assume that the slits act as point sources, for example, pinholes. A screen that displays the intensity of the light from the two sources is placed a distance L away. What do we see on the screen?

In the following problems we discuss writing programs to determine the intensity of light that is observed on a screen due to a variety of geometries. The wavelength of the light sources, the positions of the sources $\mathbf{r}_i$, and the observation points on the screen need to be specified. Your program should instantiate the necessary point sources and compute a plot showing the intensity on the obervation screen located to the right of the sources by summing the phasors. Although we suggest that you use Listing 9.10 as a guide, it is unrealistic to compute a two-dimensional grid that covers the entire region from the source to the screen. It is more efficient to plot the intensity on the screen, thereby reducing the computation to a single loop over the screen coordinate y.

Problem 9.31. Point source interference

(a) Derive an analytic expression for E from two and three point sources if the screen is far from the sources.

(b) Compute and plot the intensity of light on a screen due to two small sources (a source that emits spherical waves). Compute the phasors using (9.61) and find the intensity by taking the magnitude of $\mathcal{E}$. Let a be the distance between the sources and y be the vertical position along the screen as measured from the central maximum. Set $L = 200\,\text{mm}$, $a = 0.1\,\text{mm}$, the wavelength of light $\lambda = 5000\,\text{Å}$ ($1\,\text{Å} = 10^{-7}\,\text{mm}$), and consider $-5.0\,\text{mm} \leq y \leq 5.0\,\text{mm}$ (see Figure 9.4). Describe the interference pattern you observe. Identify the locations of the intensity maxima and plot the intensity of the maxima as a function of y. Compare your result to the analytic expression for a two slit diffraction pattern.

(c) Repeat part (b) for $L = 0.5\,\text{mm}$ and $1.0\,\text{mm} \leq y \leq 1.0\,\text{mm}$. Note that in this case L is not much greater than a, and hence we cannot ignore the r dependence of $|\mathbf{r} - \mathbf{r}_i|^{-1}$ in (9.61). □

Problem 9.32. Diffraction grating

High resolution optical spectroscopy is done with multiple slits. In its simplest form, a diffraction grating consists of N parallel slits. Compute the intensity of light for $N = 3$, 4, 5, and 10 slits with $\lambda = 5000\,\text{Å}$, slit separation $a = 0.01\,\text{mm}$, screen distance $L = 200\,\text{mm}$, and $-15\,\text{mm} \leq y \leq 15\,\text{mm}$. How do the intensity of the peaks and their separation vary with N? □

In our analysis of the double slit and the diffraction grating, we assumed that each slit was a pinhole that emits spherical waves. In practice, real slits are much wider than the wavelength of visible light. In Problem 9.33 we consider the pattern of light produced when a plane wave is incident on an aperture such as a single slit. To do so, we use Huygens's principle and replace the slit by many coherent sources of spherical waves. This equivalence is not exact, but is applicable when the aperture width is large compared to the wavelength.

Problem 9.33. Single slit diffraction

(a) Compute the time averaged intensity of light diffracted from a single slit of width 0.02 mm by replacing the slit by $N = 20$ point sources spaced 0.001 mm apart. Choose $\lambda = 5000\,\text{Å}$, $L = 200\,\text{mm}$, and consider $-30\,\text{mm} \leq y \leq 30\,\text{mm}$. What is the width of the central peak? How does the width of the central peak compare to the width of the slit? Do your results change if N is increased?

(b) Determine the position of the first minimum of the diffraction pattern as a function of the wavelength, slit width, and distance to the screen.

(c) Compute the intensity pattern for $L = 1$ mm and 50 mm. Is the far field condition satisfied in this case? How do the patterns differ? □

Problem 9.34. A more realistic double slit simulation

Reconsider the intensity distribution for double slit interference using slits of finite width. Modify your program to simulate two "thick" slits by replacing each slit by 20 point sources spaced 0.001 mm apart. The centers of the thick slits are $a = 0.1$ mm apart. How does the intensity pattern change? □

***Problem 9.35.** Diffraction pattern from a rectangular aperture

We can use a similar approach to determine the diffraction pattern due to a two-dimensional thin opaque mask with an aperture of finite width and height near the center. The simplest approach is to divide the aperture into little squares and to consider each square as a source of spherical waves. Similarly, we can divide the viewing screen or photographic plate into small regions or cells and calculate the time averaged intensity at the center of each cell. The calculations are straightforward, but time consuming because of the necessity of evaluating the cosine function many times. The less straightforward part of the problem is deciding how to plot the different values of the calculated intensity on the screen. One way is to plot "points" at random locations in each cell so that the number of points is proportional to the computed intensity at the center of the cell. Suggested parameters are $\lambda = 5000$ Å and $N = 200$ mm for a 1 mm × 3 mm aperture. □

If the number of sources N becomes large, the summation becomes the Huygens–Fresnel integral. Optics texts discuss how different approximations can be used to evaluate (9.61). For example, if the sources are much closer to each other than they are to the screen (the *far field* condition), we obtain the condition for Fraunhofer diffraction. Otherwise, we obtain Fresnel diffraction.

9.9 Fraunhofer Diffraction

The contemporary approach to diffraction is based on Fourier analysis. Consider a plane wave incident on a one-dimensional aperture. Using Huygen's idea that every point within the aperture serves as the source of spherical secondary wavelets, we can approximate the aperture as a linear array of in-phase coherent oscillators. If the spatial extent of the array is small and the distance to the observing screen is large, the wavelet amplitudes arriving at the screen will be essentially equal for wavelets within the aperture and zero for wavelets on the opaque mask. The total field is thus given by the real part of the sum:

$$E(\mathbf{r},t) = E_0 a_1 e^{i(qr_1-\omega t)} + E_0 a_2 e^{i(qr_2-\omega t)} + E_0 a_3 e^{i(qr_2-\omega t)} + \cdots + E_0 a_N e^{i(qr_N-\omega t)} \tag{9.63}$$

where $a_i = 1$ within the aperture and $a_i = 0$ outside the aperture, and the wavenumber q and the angular frequency ω have their usual meaning. Equation (9.63) can be factored into a complex phasor and time-dependent phase shift

$$E(\mathbf{r},t) = e^{-i\omega t}\mathcal{E}(\mathbf{r}) \tag{9.64}$$

where

$$\mathcal{E}(\mathbf{r}) = E_0 e^{iqr_1}\Big[1 + a_1 e^{iq(r_2-r_1)} + a_2 e^{iq(r_3-r_1)} + \cdots + a_N e^{iq(r_N-r_1)}\Big]. \tag{9.65}$$

If the grid spacing d is uniform, it follows that the phase difference between adjacent sources is $\delta = qd\sin\theta$, where θ is the angle between the aperture and a point on the screen. We substitute δ into (9.65) and observe that the field can be expressed as an overall phase times a Fourier sum:

$$\mathcal{E}(\mathbf{r}) = E_0 e^{ikqr_1}\Big[1 + \sum_{n=1}^{N} a_n e^{in\delta}\Big]. \tag{9.66}$$

Equation (9.66) shows that a Fraunhofer (far field) diffraction pattern can be obtained by dividing the aperture using a uniform grid and computing the complex Fourier transform. Because the intensity is proportional to the square of the electric field, the diffraction pattern is the modulus squared of the Fourier transform of the aperture's transmission function. Listing 9.11 uses the one-dimensional fast Fourier transform to compute the Fraunhofer diffraction pattern for a slit.

Listing 9.11: The FraunhoferApp program computes the Fraunhofer diffraction pattern from a slit.

```
package org.opensourcephysics.sip.ch09;
import org.opensourcephysics.frames.*;
import org.opensourcephysics.numerics.FFT;

public class FraunhoferApp {
   static final double PI2 = Math.PI*2;
   // Math.log is natural log
   static final double LOG10 = Math.log(10);
   static final double ALPHA = Math.log(1.0e-3)/LOG10; // cutoff value

   public static void main(String[] args) {
      PlotFrame plot = new PlotFrame("x", "intensity",
                           "Fraunhofer diffraction");
      int N = 512;
      FFT fft = new FFT(N);
      double[] cdata = new double[2*N];
      double a = 10; // aperture screen dimension
      double dx = (2*a)/N;
      double x = -a;
      for(int ix = 0;ix<N;ix++) {
         cdata[2*ix] = (Math.abs(x)<0.4) ? 1 : 0; // slit
         cdata[(2*ix)+1] = 0;
         x += dx;
      }
      fft.transform(cdata);
      fft.toNaturalOrder(cdata);
      double max = 0;
      // find the max intensity value
      for(int i = 0;i<N;i++) {
         double real = cdata[2*i];
         double imag = cdata[(2*i)+1];
         max = Math.max(max, (real*real)+(imag*imag));
         plot.append(0, i, (real*real)+(imag*imag));
      }
      plot.setVisible(true);
      // create N by 30 raster plot to show an image
      int[][] data = new int[N][30];
      // compute pixel intensity
      for(int i = 0;i<N;i++) {
         double real = cdata[2*i];
         double imag = cdata[(2*i)+1]
         // log scale increases visibility of weaker fringes
         double logIntensity = Math.log(((real*real)+(imag*imag))/max)
                               /LOG10;
         int intensity = (logIntensity<=ALPHA)
                        ? 0 : (int) (254*(1-(logIntensity/ALPHA)));
         for(int j = 0;j<30;j++) {
            data[i][j] = intensity;
         }
      }
      RasterFrame frame =
          new RasterFrame("Fraunhofer Diffraction (log scale)");
      frame.setBWPalette();
      frame.setAll(data); // send the fft data to the raster frame
      frame.setVisible(true);
      frame.setDefaultCloseOperation(javax.swing.JFrame.EXIT_ON_CLOSE);
   }
}
```

To emphasize the weaker regions of the diffraction pattern, the program plots the logarithm of the intensity. First the intensity is normalized to its peak value; then the logarithm is taken and all values less than a cutoff are truncated. The resulting range is mapped linearly to $(0,255)$ to set the gray scale.

Problem 9.36. Two-dimensional apertures

Modify FraunhoferApp to show the diffraction pattern from a double slit and compare the computed diffraction pattern to the analytic result. □

The Fraunhofer2DApp program computes the Fraunhofer diffraction pattern for a circular aperture using a two-dimensional FFT. This program is used in Problem 9.37 but is not listed because it is too long and because it is similar to Listing 9.11.

Problem 9.37. Two-dimensional apertures

(a) Compile and run the Fraunhofer2DApp program. Compute the diffraction pattern using aperture radii of 4λ and 0.25λ in a mask with dimension $a\lambda$. How does the radius influence the diffraction pattern? How does the rectangular grid influence the pattern? How can the effect of the rectangular grid be reduced?

(b) Because a typical computer monitor displays only 256 gray scale values, Fraunhofer2DApp uses a logarithmic scale to enhance the visibility of the fringes. Add code to display a linear plot of intensity as a function of radius using a slice through the center of the pattern.

(c) Compute the diffraction pattern for an annular ring with inner radius 1.8λ and outer radius 2.2λ. Why is there a bright spot at the center of the diffraction pattern? What effect does the finite width of the annular ring produce?

(d) Compute the diffraction pattern for a rectangular aperture with width 2λ and height 6λ. Describe the effect of the asymmetry of the slit. □

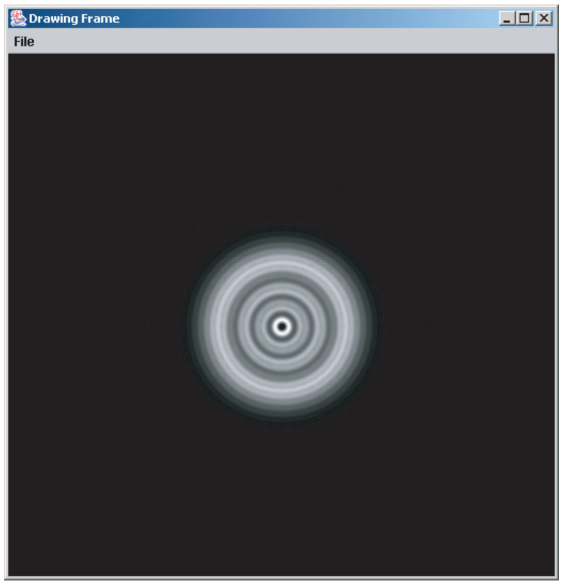


Figure 9.5: Computed Fresnel diffraction on a screen illuminated by a uniform plane wave and located $0.5 \times 10^6 \lambda$ from a circular aperture of radius 2000λ.

Problem 9.38. Diffraction patterns due to multiple apertures

(a) Compute the diffraction pattern due to a 5×5 array of rectangular slits. Each slit has a width of 0.5 and a height of 0.25 and is offset by (1,1) from neighboring slits using units such that $\lambda = 1$. The aperture array is centered within a mask ten units on a side. What is the effect of the asymmetry of the slit? What happens if the number of slits is decreased or increased?

(b) Compute the diffraction pattern due to 25 randomly placed rectangular slits. Each slit should have a width of 0.5 and a height of 0.25 and be placed within a region ten units on a side. Do not be concerned if rectangles overlap.

(c) Compare the results from (a) and (b). What effect does the random placement have on the pattern? □

9.10 Fresnel Diffraction

Fourier analysis can be used to compute the Fresnel diffraction pattern by decomposing a wave incident on an aperture into a sum of plane waves and then propagating each plane wave from the aperture mask to the screen. A plane wave with wavenumber (q_x, q_y, q_z) propagating in a homogenous environment can be written as

$$\mathcal{U} = \mathcal{U}_0 e^{i(q_x x + q_y y + q_z z)} \tag{9.67}$$

where $\mathcal{U}_0$ is the amplitude of the field at the origin, and (q_x, q_y, q_z) is a vector of length $2\pi/\lambda$ in the direction of propagation. If we place a viewing screen perpendicular to the direction of the incoming light at a point z_0 along the z axis, then the field on the screen is

$$\mathcal{U} = \mathcal{U}_0 e^{iq_z z_0} = \mathcal{U}_0 e^{iz_0(q^2 - q_x^2 - q_y^2)^{1/2}} \tag{9.68}$$

where we have used the fact that $q_x^2 + q_y^2 + q_z^2 = q^2$.

We now place an aperture mask at the origin $z = 0$ in the xy-plane and illuminate it from the left by a plane wave. Because the aperture truncates the incident plane wave, we obtain a more complicated field $\mathcal{U}_0(q_x, q_y)$ that contains both q_x and q_y spatial components:

$$\mathcal{U}_0(q_x, q_y) = \iint_{\text{aperture}} e^{i(q_x x + q_y y)}\, dx\, dy. \tag{9.69}$$

The field that propagates from the origin contains the Fourier components of the aperture mask. In other words, because we have truncated the wave, we have a field composed of a mixture of plane waves with wavenumbers (q_x, q_y, q_z). Each field component is multiplied by the $e^{iq_z z_0}$ phase factor in (9.68) as it propagates toward the viewing screen at z_0. The following steps summarize the algorithm:

1. Compute the Fourier transformation of the aperture (9.69) to obtain the field's components in the plane of the aperture.
2. Multiply each component by the propagation phase factor $e^{iz_0(q^2 - q_x^2 - q_y^2)^{1/2}}$.
3. Compute the inverse transformation to obtain the amplitude.
4. Compute the magnitude squared of the amplitude to obtain the intensity.

The Fresnel diffraction pattern algorithm is implemented in Listing 9.12. Note that the field includes evanescent waves if $q^2 - q_x^2 - q_y^2 < 0$.

Listing 9.12: The FresnelApp program computes the Fresnel diffraction pattern from a circular aperture.

```java
package org.opensourcephysics.sip.ch09;
import org.opensourcephysics.frames.RasterFrame;
import org.opensourcephysics.numerics.FFT2D;

public class FresnelApp {
   final static double PI2 = Math.PI*2;
   final static double PI4 = PI2*PI2;

   public static void main(String[] args) {
   //power of 2 for optimum speed
      int N = 512;
      // distance from aperture to screen
      double z = 0.5e+6;
      FFT2D fft2d = new FFT2D(N, N);
      double[] cdata = new double[2*N*N]; // complex data
      double a = 6000;                    // aperture mask dimension
      double
         dx = 2*a/N, dy = 2*a/N;
      double x = -a;
      for(int ix = 0;ix<N;ix++) {
         int offset = 2*ix*N;
         double y = -a;
         for(int iy = 0;iy<N;iy++) {
            double r2 = (x*x+y*y);
            cdata[offset+2*iy] = (r2<4e6) ? 1 : 0; // circular aperture
            cdata[offset+2*iy+1] = 0;
            y += dy;
         }
         x += dx;
      }
      fft2d.transform(cdata);
      // get arrays containing the wavenumbers in wrapped order
      double[] kx = fft2d.getWrappedOmegaX(-a, a);
      double[] ky = fft2d.getWrappedOmegaY(-a, a);
      for(int ix = 0;ix<N;ix++) {
         int offset = 2*ix*N; //offset to beginning of row
         for(int iy = 0;iy<N;iy++) {
            double radical = PI4-kx[ix]*kx[ix]-ky[iy]*ky[iy];
            if(radical>0) {
               double phase = z*Math.sqrt(radical);
               double real = Math.cos(phase);
               double imag = Math.sin(phase);
               double temp = cdata[offset+2*iy];
               cdata[offset+2*iy] = real*cdata[offset+2*iy]
                                    -imag*cdata[offset+2*iy+1];
               cdata[offset+2*iy+1] = real*cdata[offset+2*iy+1]+imag*temp;
            } else { //evanescent waves decay exponentially
               double decay = Math.exp(-z*Math.sqrt(-radical));
               cdata[offset+2*iy] *= decay;
               cdata[offset+2*iy+1] *= decay;
            }
         }
      }
      fft2d.inverse(cdata);
      double max = 0;
      for(int i = 0;i<N*N;i++) {      // find max intensity
         double real = cdata[2*i];
         double imag = cdata[2*i+1];
         max = Math.max(max, real*real+imag*imag);
      }
      // intensity is squared magnitude of the amplitude
      int[] data = new int[N*N];
      for(int i = 0, N2 = N*N;i<N2;i++) {
         double real = cdata[2*i];   // real
         double imag = cdata[2*i+1]; // imaginary
         data[i] = (int) (255*(real*real+imag*imag)/max);
      }
      // raster for least memory and best speed
      RasterFrame frame = new RasterFrame("Fraunhofer Diffraction");
      frame.setBWPalette();
      frame.setAll(data, N, -0.5, 0.5, -0.5, 0.5);
      frame.setVisible(true);
      frame.setDefaultCloseOperation(javax.swing.JFrame.EXIT_ON_CLOSE);
   }
}
```

Because the algorithm in Listing 9.12 depends only on the linearity of the wave equation, it is exact and may be applied to many practical optics problems. Its main limitation occurs when

z_0 is large because of rapid oscillations in the phase factor. In this case, the far field (Fraunhofer) approximation usually becomes applicable and should be used.

Problem 9.39. Fresnel diffraction

The diffraction pattern from a circular aperture is important because most lenses, mirrors, and optical instruments have cylindrical symmetry (see Figure 9.5).

(a) Compute the diffraction pattern due to a circular aperture with a radius of $1000\,\lambda$ at a screen distance of $10^5\,\lambda$. Is the center of the diffraction pattern dark or light? Reposition the screen to $2{\times}10^5\,\lambda$ and repeat the calculation. Does the intensity at the center of the shadow change? Use a 512×512 grid to sample a region of space $5000\,\lambda$ on a side.

(b) Replace the screen by a circular disk. That is, use an aperture mask that is opaque if $r < 1000\lambda$. Is the center of the screen light or dark? Does the center change from bright to dark if the screen is repositioned? □

Problem 9.40. Optical resolution

Consider a mask containing two circular openings of radius 500λ separated by 100λ. Do a simulation to determine how far the screen can be placed from the aperture mask and still observe two distinct shadows. □

Appendix 9A: Complex Fourier Series

A function $f(t)$ with period T can be expressed in terms of a trigonometric Fourier series:

$$f(t) = \frac{1}{2}a_0 + \sum_{k=1}^{\infty}\Big(a_k \cos\omega_k t + b_k \sin\omega_k t\Big) \tag{9.70}$$

where $\omega_k = k\omega_0$ and $\omega_0 = 2\partial/T$. To derive the exponential form of this series, we express the sine and cosine functions as

$$\sin x = \frac{e^{ix} - e^{-ix}}{2i} \tag{9.71a}$$

$$\cos x = \frac{e^{ix} + e^{-ix}}{2} \tag{9.71b}$$

We substitute (9.71) into (9.70) and obtain

$$f(t) = \frac{1}{2}a_0 + \sum_{k=1}^{\infty}\Big[e^{ik\omega_0 t}\frac{a_k - ib_k}{2} + e^{-ik\omega_0 t}\frac{a_k + ib_k}{2}\Big]. \tag{9.72}$$

We use $1/i = -i$ and define new Fourier coefficients as follows:

$$c_0 \equiv \frac{1}{2}a_0 \tag{9.73a}$$

$$c_k \equiv \frac{a_k - ib_k}{2} \tag{9.73b}$$

$$c_{-k} \equiv \frac{a_{-k} - ib_{-k}}{2} = \frac{a_k + ib_k}{2} \tag{9.73c}$$

where the right-hand side of (9.73c) follows from $a_k = a_{-k}$ and $b_k = -b_{-k}$. We substitute these coefficients into (9.72) and find

$$f(t) = c_0 + \sum_{k=1}^{\infty} c_k e^{ik\omega_0 t} + \sum_{k=1}^{\infty} c_{-k} e^{-ik\omega_0 t} \tag{9.74}$$

or

$$f(t) = \sum_{k=0}^{\infty} c_k e^{ik\omega_0 t} + \sum_{k=1}^{\infty} c_{-k} e^{-ik\omega_0 t}. \tag{9.75}$$

Finally, we re-index the second sum from -1 to $-\infty$

$$f(t) = \sum_{k=0}^{\infty} c_k e^{ik\omega_0 t} + \sum_{k=-1}^{-\infty} c_k e^{ik\omega_0 t} \tag{9.76}$$

and combine the summations to obtain the exponential form:

$$f(t) = \sum_{k=-\infty}^{\infty} c_k e^{ik\omega_0 t}. \tag{9.77}$$

Appendix 9B: Fast Fourier Transform

The fast Fourier transform (FFT) was discovered independently in a variety of contexts by many workers. There are several variations of the algorithm, and we describe a version due to Danielson and Lanczos. The goal is to compute the Fourier transform $g(\omega_k)$ given the data set $f(j\Delta) \equiv f_j$ of (9.35). For convenience we rewrite the relation

$$g_k \equiv g(\omega_k) = \sum_{j=0}^{N-1} f(j\Delta)\, e^{-i2\pi kj/N} \tag{9.78}$$

and introduce the complex number W given by

$$W = e^{-i2\pi/N}. \tag{9.79}$$

The following algorithm works with any complex data set if N is a power of two. Real data sets can be transformed by setting the array elements corresponding to the imaginary part equal to 0.

To understand the FFT algorithm, we consider the case $N = 8$ and rewrite (9.78) as

$$g_k = \sum_{j=0,2,4,6} f(j\Delta)\, e^{-i2\pi kj/N} + \sum_{j=1,3,5,7} f(j\Delta)\, e^{-i2\pi kj/N} \tag{9.80a}$$

$$= \sum_{j=0,1,2,3} f(2j\Delta) e^{-i2\pi k2j/N} + \sum_{j=0,1,2,3} f((2j+1)\Delta) e^{-i2\pi k(2j+1)/N} \tag{9.80b}$$

$$= \sum_{j=0,1,2,3} f(2j\Delta) e^{-i2\pi kj/(N/2)} + W^k \sum_{j=0,1,2,3} f((2j+1)\Delta)\, e^{-i2\pi kj/(N/2)} \tag{9.80c}$$

$$= g_k^{\mathrm{e}} + W^k g_k^{\mathrm{o}} \tag{9.80d}$$

where $W^k = e^{-i2\pi k/N}$. The quantity $g^{\rm e}$ is the Fourier transform of length $N/2$ formed from the even components of the original $f(j\Delta)$; $g^{\rm o}$ is the Fourier transform of length $N/2$ formed from the odd components.

We can continue this decomposition if N is a power of two. That is, we can decompose $g^{\rm e}$ into its $N/4$ even and $N/4$ odd components, $g^{\rm ee}$ and $g^{\rm eo}$, and decompose $g^{\rm o}$ into its $N/4$ even and $N/4$ odd components, $g^{\rm oe}$ and $g^{\rm oo}$. We find

$$g_k = g_k^{\rm ee} + W^{2k} g_k^{\rm eo} + W^k g_k^{\rm oe} + W^{3k} g_k^{\rm oo}. \tag{9.81}$$

One more decomposition leads to

$$\begin{aligned} g_k &= g_k^{\rm eee} + W^{4k} g_k^{\rm eeo} + W^{2k} g_k^{\rm eoe} + W^{6k} g_k^{\rm eoo} \\ &\quad + W^k g_k^{\rm oee} + W^{5k} g_k^{\rm oeo} + W^{3k} g_k^{\rm ooe} + W^{7k} g_k^{\rm ooo}. \end{aligned} \tag{9.82}$$

At this stage each of the Fourier transforms in (9.82) uses only one data point. We see from (9.78) with $N = 1$ that the value of each of these Fourier transforms, $g_k^{\rm eee}$, $g_k^{\rm eeo}$,..., is equal to the value of f at the corresponding data point. Note that for $N = 8$, we have performed $3 = \log_2 N$ decompositions. In general, we would perform $\log_2 N$ decompositions.

There are two steps to the FFT. First, we reorder the components so that they appear in the order given in (9.82). This step makes the subsequent calculations easier to organize. To see how to do the reordering, we rewrite (9.82) using the values of f:

$$\begin{aligned} g_k &= f(0) + W^{4k} f(4\Delta) + W^{2k} f(2\Delta) + W^{6k} f(6\Delta) \\ &\quad + W^k f(\Delta) + W^{5k} f(5\Delta) + W^{3k} f(3\Delta) + W^{7k} f(7\Delta). \end{aligned} \tag{9.83}$$

We use a trick to obtain the ordering in (9.83) from the original order $f(0\Delta)$, $f(1\Delta)$, ..., $f(7\Delta)$. Part of the trick is to refer to each g in (9.82) by a string of "e" and "o" characters. We assign 0 to "e" and 1 to "o" so that each string represents the binary representation of a number. If we reverse the order of the representation; that is, set 110 to 011, we obtain the value of f we want. For example, the fifth term in (9.82) contains $g^{\rm oee}$, corresponding to the binary number 100. The reverse of this number is 001, which equals 1 in decimal notation, and hence, the fifth term in (9.83) contains the function $f(1\Delta)$. Convince yourself that this bit reversal procedure works for the other seven terms.

The first step in the FFT algorithm is to use this bit reversal procedure on the original array representing the data. In the next step this array is replaced by its Fourier transform. If you want to save your original data, it is necessary to first copy the data to another array before passing the array to a FFT implementation. The `SimpleFFT` class implements the Danielson–Lanczos algorithm using three loops. The outer loop runs over $\log_2 N$ steps. For each of these steps, N calculations are performed in the two inner loops. As can be seen in Listing 9.13, in each pass through the innermost loop, each element of the array `g` is updated once by the quantity `temp` formed from a power of W multiplied by the current value of an appropriate element of `g`. The power of W used in `temp` is changed after each pass through the innermost loop. The power of the FFT algorithm is that we do not separately multiply each $f(j\Delta)$ by the appropriate power of W. Instead, we first take pairs of $f(j\Delta)$ and multiply them by an appropriate power of W to create new values for the array `g`. Then we repeat this process for pairs of the new array elements (each array element now contains four of the $f(j\Delta)$). We repeat this process until each array element contains a sum of all N values of $f(j\Delta)$ with the correct powers of W multiplying each term to form the Fourier transform.

Listing 9.13: A simple implementation of the FFT algorithm.

```
package org.opensourcephysics.sip.ch09;
public class SimpleFFT {
   public static void transform(double[] real, double[] imag) {
      int N = real.length;
      int pow = 0;
      while(N/2>0) {
         if(N%2==0) { // N should be even
            pow++;
            N /= 2;
         } else {
            throw new IllegalArgumentException("Number of points in
                  this FFT implementation must be even.");
         }
      }
      int N2 = N/2;
      int jj = N2;
      // rearrange input according to bit reversal
      for(int i = 1;i<N-1;i++) {
         if(i<jj) {
            double tempRe = real[jj];
            double tempIm = imag[jj];
            real[jj] = real[i];
            imag[jj] = imag[i];
            real[i] = tempRe;
            imag[i] = tempIm;
         }
         int k = N2;
         while(k<=jj) {
            jj = jj-k;
            k = k/2;
         }
         jj = jj+k;
      }
      jj = 1;
      for(int p = 1;p<=pow;p++) {
         int inc = 2*jj;
         double
            wp1 = 1, wp2 = 0;
         double theta = Math.PI/jj;
         double
            cos = Math.cos(theta), sin = -Math.sin(theta);
         for(int j = 0;j<jj;j++) {
            for(int i = j;i<N;i += inc) {
               // calculate the transform of 2^p
               int ip = i+jj;
               double tempRe = wp1*real[ip]-wp2*imag[ip];
               double tempIm = wp2*real[ip]+wp1*imag[ip];
               real[ip] = real[i]-tempRe;
               imag[ip] = imag[i]-tempIm;
               real[i] = real[i]+tempRe;
               imag[i] = imag[i]+tempIm;
            }
```

```
               double temp = wp1;
               wp1 = wp1*cos-wp2*sin;
               wp2 = temp*sin+wp2*cos;
            }
            jj = inc;
         }
      }
   }
```

Exercise 9.41. Testing the FFT algorithm

1. Test the SimpleFFT class for $N = 8$ by going through the code by hand and showing that the class reproduces (9.83).
2. Display the Fourier coefficients of random real values of $f(j\Delta)$ for $N = 8$ using both SimpleFFT and the direct computation of the Fourier coefficients based on (9.34). Compare the two sets of data to insure that there are no errors in SimpleFFT. Repeat for a random collection of complex data points.
3. Modify the SimpleFFT class to compute the inverse Fourier transform defined by (9.37). The inverse Fourier transform of a Fourier transformed data set should be the original data set.
4. Compute the CPU time as a function of N for $N = 16, 64, 256,$ and 1024 for the SimpleFFT algorithm and the direct computation. You can use the currentTimeMillis method in System class to record the time.

   ```
   int n = 10;
   long startTime = System.currentTimeMillis();
   for(int i=0; i<n; i++) { // average for better results
      fft.transform();
   }
   long endTime = System.currentTimeMillis();
   System.out.println("time/FFT = "+((endTime-startTime)/n));
   ```

   Verify that the dependence on N is what you expect.
5. Compare the CPU time as a function of N for the SimpleFFT class and the FFT class in the numerics package.
6. Compare the CPU time for the FFT class in the numerics package using two slightly different values of N such that one value is a power of two and the other is not. □

Appendix 9C: Plotting Scalar Fields

Imagine a plate that is heated at an interior point and cooled along its edges. In principle, the temperature of this plate can be measured at every point. A scalar quantity, such as temperature, pressure, or light intensity, that is defined throughout a region of space is known as a *scalar field*. The Open Source Physics library contains a number of tools that help us visualize two-dimensional scalar fields. A more complete description of two-dimensional visualization tools is available in *Open Source Physics User's Guide*.

An image in which pixels are color coded can be used to visualize a scalar field. The frames package defines a `RasterFrame` class that makes this process easy and efficient if the scalar field can be represented by integers from 0 to 255. The following program shows how such a `RasterFrame` is used.

Listing 9.14: A scalar field visualization using a raster frame.

```
package org.opensourcephysics.sip.ch09;
import org.opensourcephysics.frames.RasterFrame;

public class RasterFrameApp {
   public static void main(String[] args) {
      RasterFrame frame = new RasterFrame("x", "y", "Raster Frame");
      frame.setPreferredMinMax(-10, 10, -10, 10);
      // generate random data
      int nx = 256, ny = 256;
      int[][] data = new int[nx][ny];
      for(int i = 0;i<nx;i++) {
         for(int j = 0;j<ny;j++) {
            data[i][j] = (int) (255*Math.random());
         }
      }
      frame.setAll(data);
      frame.setVisible(true);
      frame.setDefaultCloseOperation(javax.swing.JFrame.EXIT_ON_CLOSE);
   }
}
```

After the scalar field's values are calculated, the raster's pixels are set using the `setBlock` method. Note that the $[0,0]$ array element maps to the lower left hand pixel. Because the image raster's mapping has been optimized for speed, the image cannot be resized. The on-screen image size in pixels always matches the array size. Listing 9.11 uses a raster frame to display a Fraunhofer diffraction pattern.

Although the `RasterFrame` makes it easy to work with integer-based data, it lacks the flexibility for more advanced visualizations. It is unsuitable if the array size is small or if the dynamic range of the scalar field is too large. The `Scalar2DFrame` class overcomes these limitations. Using a `Scalar2DFrame` allows us to view the data using different representations, such as contour plots and three-dimensional surface plots. Listing 9.15 shows a `RasterFrame` being used to visualize the function $f(x,y) = xy$.

Listing 9.15: A scalar field test program.

```
package org.opensourcephysics.sip.ch09;
import org.opensourcephysics.frames.Scalar2DFrame;

public class Scalar2DFrameApp {
   final static int SIZE = 16; // array size

   public static void main(String[] args) {
      Scalar2DFrame frame = new Scalar2DFrame("x", "y", "Scalar Field");
      double[][] data = new double[16][16];
      frame.setAll(data, -10, 10, -10, 10);
      // generate sample data
      for(int i = 0;i<SIZE;i++) {
         double x = frame.indexToX(i);
         for(int j = 0;j<SIZE;j++) {
            double y = frame.indexToY(j);
            data[i][j] = x*y;
         }
      }
      frame.setAll(data);
      frame.setVisible(true);
      frame.setDefaultCloseOperation(javax.swing.JFrame.EXIT_ON_CLOSE);
   }
}
```

Exercise 9.42. Scalar field visualization

Run the scalar field test program and describe the various types of visualizations available under the Tools menu. Which visualizations give respectable representations if the grid is small? What is the maximum grid size that can be used with each type of visualization and still give acceptable performance on your computer? □

References and Suggestions for Further Reading

David C. Champeney, *Fourier Transforms and Their Physical Applications* (Academic Press, 1973).

James B. Cole, Rudolph A. Krutar, Susan K. Numrich, and Dennis B. Creamer, "Finite-difference time-domain simulations of wave propagation and scattering as a research and educational tool," Computers in Physics **9**, 235–239 (1995).

Frank S. Crawford, *Waves*, Berkeley Physics Course, Vol. 3 (McGraw–Hill, 1968). A delightful book on waves of all types. The home experiments are highly recommended. One observation of wave phenomena equals many computer demonstrations.

Paul DeVries, *A First Course in Computational Physics* (John Wiley & Sons, 1994). Part of our discussion of the wave equation is based on Chapter 7. There are also good sections on the numerical solution of other partial differential equations, Fourier transforms, and the FFT.

N. A. Dodd, "Computer simulation of diffraction patterns," Phys. Educ. **18**, 294–299 (1983).

P. G. Drazin and R. S. Johnson, *Solitons: An Introduction* (Cambridge University Press, 1989). This book focuses on analytic solutions to the Korteweg–de Vries equation which has soliton solutions.

Richard P. Feynman, Robert B. Leighton, and Matthew Sands, *The Feynman Lectures on Physics*, Vol. 1 (Addison–Wesley, 1963). Chapters relevant to wave phenomena include Chapters 28–30 and Chapter 33.

A. P. French, *Vibrations and Waves* (W. W. Norton & Co., 1971). An introductory level text that emphasizes mechanical systems.

Robert Guenther, *Modern Optics*, Vol. 1 (John Wiley & Sons, 1990). Chapter 6 discusses Fourier analysis and Chapters 9–12 apply Fourier analysis to the study of Fraunhofer and Fresnel diffraction and holography.

Eugene Hecht, *Optics*, 4th ed. (Addison–Wesley, 2002). An intermediate level optics text that emphasizes wave concepts.

Akira Hirose and Karl E. Lonngren, *Introduction to Wave Phenomena* (John Wiley & Sons, 1985). An intermediate level text that treats the general properties of waves in various contexts.

Amy Kolan, Barry Cipra, and Bill Titus, "Exploring localization in nonperiodic systems," Computers in Physics **9** (4), 387–395 (1995). An elementary discussion of how to solve the problem of a chain of coupled oscillators with disorder using transfer matrices.

J. F. James, *A Student's Guide to Fourier Transforms*, 2nd ed. (Cambridge University Press, 2002).

William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery, *Numerical Recipes*, 2nd ed., Cambridge University Press (1992). See Chapter 12 for a discussion of the fast Fourier transform.

Iain G. Main, *Vibrations and Waves in Physics* (Cambridge University Press, 1993). See Chapter 12 for a discussion of a chain of coupled oscillators.

Masud Mansuripur, *Classical Optics and its Applications* (Cambridge University Press, 2002). See Chapter 2 for a discussion of Fourier optics.

Timothy J. Rolfe, Stuart A. Rice, and John Dancz, "A numerical study of large amplitude motion on a chain of coupled nonlinear oscillators," J. Chem. Phys. **70**, 26–33 (1979). Problem 9.30 is based on this paper.

Garrison Sposito, *An Introduction to Classical Dynamics* (John Wiley & Sons, 1976). A good discussion of the coupled harmonic oscillator problem is given in Chapter 6.

William J. Thompson, *Computing for Scientists and Engineers* (John Wiley & Sons, 1992). See Chapters 9 and 10 for a discussion of Fourier transform methods.

Michael L. Williams and Humphrey J. Maris, "Numerical study of phonon localization in disordered systems," Phys. Rev. B **31**, 4508–4515 (1985). The authors consider the normal modes of a two-dimensional system of coupled oscillators with random masses. The idea of using mechanical resonance to extract the normal modes is the basis of a new numerical method for finding the eigenmodes of large lattices. See Kousuke Yukubo, Tsuneyoshi Nakayama, and Humphrey J. Maris, "Analysis of a new method for finding eigenmodes of very large lattice systems," J. Phys. Soc. Japan **60**, 3249 (1991).

Chapter 10

Electrodynamics

We compute the electric fields due to static and moving charges, describe methods for computing the electric potential in boundary value problems, and solve Maxwell's equations numerically.

10.1 Static Charges

Suppose we want to know the electric field $\mathbf{E}(\mathbf{r})$ at the point $\mathbf{r}$ due to N point charges $q_1, q_2, \ldots, q_N$ at fixed positions $\mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_N$. We know that $\mathbf{E}(\mathbf{r})$ satisfies a superposition principle and is given by

$$\mathbf{E}(\mathbf{r}) = K \sum_{i}^{N} \frac{q_i}{|\mathbf{r} - \mathbf{r}_i|^3} (\mathbf{r} - \mathbf{r}_i) \tag{10.1}$$

where $\mathbf{r}_i$ is the fixed location of the ith charge, and K is a constant that depends on our choice of units. One of the difficulties associated with electrodynamics is the competing systems of units. In the SI (or rationalized MKS) system of units, the charge is measured in coulombs (C) and the constant K is given by

$$K = \frac{1}{4\pi\epsilon_0} \approx 9.0 \times 10^9\,\mathrm{N \cdot m^2/C^2} \qquad \text{(SI units).} \tag{10.2}$$

The constant ϵ_0 is the electrical permittivity of free space. This choice of units is not convenient for computer programs because $K \gg 1$. Another popular system of units is the Gaussian (cgs) system for which the constant K is absorbed into the unit of charge so that $K = 1$. Charge is in electrostatic units or esu. One feature of Gaussian units is that the electric and magnetic fields have the same units. For example, the (Lorentz) force on a particle of charge q and velocity $\mathbf{v}$ in an electric field $\mathbf{E}$ and a magnetic field $\mathbf{B}$ has the form

$$\mathbf{F} = q(\mathbf{E} + \frac{\mathbf{v}}{c} \times \mathbf{B}) \qquad \text{(Gaussian units).} \tag{10.3}$$

These virtues of the Gaussian system of units lead us to adopt this system for this chapter even though SI units are used in introductory texts.

10.2 Electric Fields

The electric field is an example of a *vector field* because it defines a vector quantity at every point in space. One way to visualize this field is to divide space into a discrete grid and to draw arrows in the direction of **E** at the vertices of this grid. The length of the arrow can be chosen to be proportional to the magnitude of the electric field. Another possibility is to use color or gray scale to represent the magnitude. Because we have found that using an arrow's color rather than its length to represent field strength produces a more effective representation of vector fields over a wider dynamic range, the `Vector2DFrame` class in the Open Source Physics frames package uses the color representation (see Appendix 10A).

The `ElectricFieldApp` program computes the electric field due to an arbitrary number of point charges. A charge is created using the control's x, y, and q parameters when the Calculate button is clicked. Each time the Calculate button is clicked, another charge is created. Whenever a charge is added or moved, the electric field is recomputed in the `calculateField` method. The Reset button removes all charges.

Because the number of charges can change, we need a way to obtain the position and charge for each point charge so that the electric field can be computed. The drawing panel for `frame` contains a list of all drawable objects, but we need a way to obtain only those objects that are of type `Charge`. The following statements show one way of doing this.

```
List chargeList = frame.getDrawables(Charge.class);
Iterator it = chargeList.iterator();
```

The argument `Charge.class` tells the `frame.getDrawables` method to return only a list of objects that can be cast to the `Charge` class.[1] These objects are then placed in an object of type `ArrayList` which implements the Java interface `List`. The `iterator` method returns an object called `it` which implements the `Iterator` interface. We then use the object `it` to loop through all the charges using the `next` and `hasNext` methods of the interface `Iterator`. As the name implies, an iterator is a convenient way to access a list without explicitly counting its elements. You will modify `ElectricFieldApp` to include a moving test charge in Problem 10.1.

Listing 10.1: `ElectricFieldApp` computes and displays the electric field from a list of point charges.

```
package org.opensourcephysics.sip.ch10;
import java.awt.event.MouseEvent;
import java.util.Iterator;
import java.util.List;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.display.*;
import org.opensourcephysics.frames.Vector2DFrame;

public class ElectricFieldApp extends AbstractCalculation implements
      InteractiveMouseHandler {
   int n = 20;                                      // grid points on a side
   double a = 10;                                   // viewing side length
   double[][][] eField = new double[2][n][n]; // stores electric field
   Vector2DFrame frame = new Vector2DFrame("x", "y", "Electric field");

   public ElectricFieldApp() {
```

[1] The syntax `Charge.class` uses a small portion of the Java reflection API to determine the type of object that is being requested. The reflection API is an advanced feature of Java.

```
      frame.setPreferredMinMax(-a/2, a/2, -a/2, a/2);
      frame.setZRange(false, 0, 2);
      frame.setAll(eField); // sets the vector field
      frame.setInteractiveMouseHandler(this);
   }

   public void calculate() {
      double x = control.getDouble("x");
      double y = control.getDouble("y");
      double q = control.getDouble("q");
      Charge charge = new Charge(x, y, q);
      frame.addDrawable(charge);
      calculateField();
   }

   public void reset() {
      control.println(
          "Calculate creates a new charge and updates the field.");
      control.println("You can drag charges.");
      frame.clearDrawables(); // removes all charges
      control.setValue("x", 0);
      control.setValue("y", 0);
      control.setValue("q", 1);
      calculateField();
   }

   void calculateField() {
      for(int ix = 0;ix<n;ix++) {
         for(int iy = 0;iy<n;iy++) {
            eField[0][ix][iy] = eField[1][ix][iy] = 0; // zeros field
         }
      }
      // the charges in the frame
      List chargeList = frame.getDrawables(Charge.class);
      Iterator it = chargeList.iterator();
      while(it.hasNext()) {
         Charge charge = (Charge) it.next();
         double
            xs = charge.getX(), ys = charge.getY();
         for(int ix = 0;ix<n;ix++) {
            double x = frame.indexToX(ix);
            // distance of charge to gridpoint
            double dx = (x-xs);
            for(int iy = 0;iy<n;iy++) {
               double y = frame.indexToY(iy);
               double dy = (y-ys);           // charge to gridpoint
               double r2 = dx*dx+dy*dy;      // distance squared
               double r3 = Math.sqrt(r2)*r2; // distance cubed
               if(r3>0) {
                  eField[0][ix][iy] += charge.q*dx/r3;
                  eField[1][ix][iy] += charge.q*dy/r3;
               }
            }
            }
         }
         frame.setAll(eField);
      }

      public void handleMouseAction(InteractivePanel panel, MouseEvent evt) {
         panel.handleMouseAction(panel, evt); // panel moves the charge
         if(panel.getMouseAction()==InteractivePanel.MOUSE_DRAGGED) {
          // remove this line if user interface is sluggish
            calculateField();
            panel.repaint();
         }
      }

      public static void main(String[] args) {
         CalculationControl.createApp(new ElectricFieldApp());
      }
   }
```

To make the program interactive, the `ElectricFieldApp` class implements the `Interactive-MouseHandler` to process mouse events when a charge is dragged. (See Section 5.7 for a discussion of interactive panels and interactive mouse handlers.) The class registers its interest in handling these events using the `setInteractiveMouseHandler` method. The handler passes the event to the panel to move the charge and then recalculates the field. Note that the `Charge` class in Listing 10.2 inherits from the `InteractiveCircle` class.[2]

Listing 10.2: The `Charge` class extends the `InteracticeCircle` class and adds the charge property.

```
package org.opensourcephysics.sip.ch10;
import java.awt.Color;
import org.opensourcephysics.display.InteractiveCircle;

public class Charge extends InteractiveCircle {
   double q = 0;

   public double getQ() {
      return q;
   }

   public Charge(double x, double y, double q) { //
      super(x, y);
      this.q = q;
      if(q>0) {
         color = Color.red;
      } else {
         color = Color.blue;
      }
   }
}
```

[2]Dragging may become sluggish if too many computations are performed within the mouse action method.

Problem 10.1. Motion of a charged particle in an electric field

(a) Test `ElectricFieldApp` by adding one charge at a time at various locations. Do the electric field patterns look reasonable? For example, does the electric field point away from positive charges and toward negative charges? How well is the magnitude of the electric field represented?

(b) Modify `ElectricFieldApp` so that it uses `AbstractSimulation` to compute the motion of a test particle of mass m and charge q in the presence of the electric field created by a fixed distribution of point charges. That is, create a drawable test charge that implements the `ODE` interface and add it to the vector field frame. Use the same approach that was used for the trajectory problems in Chapter 5. The acceleration of the charge is given by $q\mathbf{E}/m$, where $\mathbf{E}$ is the electric field due to the fixed point charges. Use a higher-order algorithm to advance the position and velocity of the particle. (Ignore the effects of radiation due to accelerating charges.)

(c) Assume that $\mathbf{E}$ is due to a charge `q(1)` = 1.5 fixed at the origin. Simulate the motion of a charged particle of mass $m = 0.1$ and charge $q = 1$ initially at $x = 1, y = 0$. Consider the following initial conditions for its velocity: $v_x = 0, v_y = 0$; $v_x = 1, v_y = 0$; $v_x = 0, v_y = 1$; and $v_x = -1$, $v_y = 0$. Is the trajectory of the particle tangent to the field vectors? Explain.

(d) Assume that the electric field is due to two fixed point charges: `q(1)` = 1 at `x(1)` = 2, `y(1)` = 0 and `q(2)` = −1 at `x(2)` = −2, `y(2)` = 0. Place a charged particle of unit mass and unit positive charge at $x = 0.05, y = 0$. What do you expect the motion of this charge to be? Do the simulation and determine the qualitative nature of the motion.

e)* Consider the motion of a charged particle in the vicinity of the electric dipole defined in part (d). Choose the initial position to be five times the separation of the charges in the dipole. Do you find any bound orbits? Do you find any closed orbits or do all orbits show some precession? □

10.3 Electric Field Lines

Another way of visualizing the electric field is to draw *electric field lines*. The properties of these lines are as follows:

1. An electric field line is a directed line whose tangent at every position is parallel to the electric field at that position.
2. The lines are smooth and continuous except at singularities such as point charges. (It makes no sense to talk about the electric field *at* a point charge.)
3. The density of lines at any point in space is proportional to the magnitude of the field at that point. This property implies that the total number of electric field lines from a point charge is proportional to the magnitude of that charge. The value of the proportionality constant is chosen to provide the clearest pictorial representation of the field. The drawing of field lines is art plus science.

The `FieldLineApp` program draws electric field lines in two dimensions. The program makes extensive use of the `FieldLine` class which implements the following algorithm:

1. Begin at a point (x,y) and compute the components E_x and E_y of the electric field vector $\mathbf{E}$ using (10.1).
2. Draw a small line segment of size $\Delta s = |\Delta \mathbf{s}|$ tangent to $\mathbf{E}$ at that point. The components of the line segment are given by
$$\Delta x = \Delta s \frac{E_x}{|\mathbf{E}|} \text{ and } \Delta y = \Delta s \frac{E_y}{|\mathbf{E}|}. \tag{10.4}$$
3. Iterate the process beginning at the new point $(x+\Delta x, y+\Delta y)$. Continue until the field line approaches a point charge singularity or escapes toward infinity.

This field line algorithm is equivalent to solving the following differential equations:
$$\frac{dx}{ds} = \frac{E_x}{|\mathbf{E}|} \tag{10.5a}$$
$$\frac{dy}{ds} = \frac{E_y}{|\mathbf{E}|}. \tag{10.5b}$$

Because a field line extends in both directions from the algorithm's starting point, the computation must be repeated in the $(-E_x/|\mathbf{E}|, -E_y/|\mathbf{E}|)$ direction to obtain a complete visualization of the field line. Note that this algorithm draws a correct field line but does not draw a collection of field lines with a density proportional to the field intensity.

To draw the field lines, a computation starts when a user double clicks in the panel and end the computation when the field line approaches a point charge or when the magnitude of the field becomes too small. Although we can easily describe these stopping conditions, we do not know how long the computation will take, and we might want to compute multiple field lines simultaneously. An elegant way to do this computation is to use threads.

As we discussed in Section 2.6, Java programs can have multiple threads to separate and organize related tasks. A thread is an *independent* task within a single program that shares the program's data with other threads.[3] In the following example, we create a thread to compute the solution of the differential equation for an electric field line. It is natural to use threads in this context because the drawing of a field line involves starting the field line, drawing each piece of the field line, and then stopping the calculation when some stopping condition is met. The computation begins when the `FieldLine` object is created and ends when the stopping condition is satisfied.

A thread executes statements within an object, such as `FieldLine`, that implements the `Runnable` interface. This interface consists of a single method, the `run` method, and the thread executes the code within this method. The run method is not invoked directly, but is invoked automatically by the thread after the thread is started. When the run method exits, the thread that invoked the run method stops executing and is said to die. After a thread dies, it cannot be restarted. Another thread must be created if we wish to invoke the run method a second time.

We build a `FieldLine` class by subclassing `Thread` and adding the necessary drawing and differential equation capabilities using the `Drawable` and `ODE` interfaces, respectively. This class is shown in Listing 10.3.

Listing 10.3: The `FieldLine` class computes an electric field line using a `Thread`.
```
package org.opensourcephysics.sip.ch10;
```

[3]The *Open Source Physics User's Guide* describes simulation threads in more detail.

```
import java.awt.Graphics;
import java.util.*;
import org.opensourcephysics.display.*;
import org.opensourcephysics.numerics.*;

public class FieldLine implements Drawable, ODE, Runnable {
   DrawingFrame frame;
   double[] state = new double[2]; // Ex and Ey for ODE
   ODESolver odeSolver = new RK45MultiStep(this);
   ArrayList chargeList;          // list of charged particles
   Trail trail;
   double stepSize;
   volatile boolean done = false;

   public FieldLine(DrawingFrame frame, double x0, double y0,
                    double stepSize) {
      this.stepSize = stepSize;
      this.frame = frame;
      odeSolver.setStepSize(stepSize);
      state[0] = x0;
      state[1] = y0;
      chargeList = frame.getDrawables(Charge.class);
      trail = new Trail();
      trail.addPoint(x0, y0);
      Thread thread = new Thread(this);
      thread.start();
   }

   public double[] getState() {
      return state;
   }

   public void getRate(double[] state, double[] rate) {
      double ex = 0;
      double ey = 0;
      for(Iterator it = chargeList.iterator();it.hasNext();) {
         Charge charge = (Charge) it.next();
         double dx = (charge.getX()-state[0]);
         double dy = (charge.getY()-state[1]);
         double r2 = dx*dx+dy*dy;
         double r = Math.sqrt(r2);
         if((r<2*stepSize)||(r>100)) { // done if too close or too far
            done = true;
         }
         ex += (r==0) ? 0 : charge.q*dx/r2/r;
         ey += (r==0) ? 0 : charge.q*dy/r2/r;
      }
      double mag = Math.sqrt(ex*ex+ey*ey);
      rate[0] = (mag==0) ? 0 : ex/mag;
      rate[1] = (mag==0) ? 0 : ey/mag;
   }

   public void run() {
      int counter = 0;
      while(((counter<1000)&&!done)) {
         odeSolver.step();
         trail.addPoint(state[0], state[1]);
         if(counter%50==0) {     // repaint every 50th step
            frame.repaint();
            try {
               Thread.sleep(20); // give the event queue a chance
            } catch(InterruptedException ex) {}
         }
         counter++;
         Thread.yield();
      }
      frame.repaint();
   }

   public void draw(DrawingPanel panel, Graphics g) {
      trail.draw(panel, g);
   }
}
```

The `FieldLine` constructor saves a reference to the list of charges to calculate the electric field using (10.1). The loop in the `run` method solves the differential equation and stores the solution in a drawable trail. The loop is exited when the field line is close to a charge or when the magnitude of the field becomes too small. Because there are situations where the field line will never stop, this loop is executed no more than 1000 times.

The `FieldLineApp` program instantiates a field line when the user double clicks within the panel. Adding a charge or moving a charge removes all field lines from the panel. Study how the `handleMouseAction` allows the user to drag charges and to initiate the drawing of field lines. You are asked to modify this program in Problem 10.2.

Listing 10.4: The `FieldLineApp` program computes an electric field line when the user clicks within the panel.

```
package org.opensourcephysics.sip.ch10;
import java.awt.event.MouseEvent;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.display.*;
import org.opensourcephysics.frames.DisplayFrame;

public class FieldLineApp extends AbstractCalculation implements
      InteractiveMouseHandler {
   DisplayFrame frame = new DisplayFrame("x", "y", "Field lines");

   public FieldLineApp() {
      frame.setInteractiveMouseHandler(this);
      frame.setPreferredMinMax(-10, 10, -10, 10);
   }

   public void calculate() {
      // remove old field lines
      frame.removeObjectsOfClass(FieldLine.class);
      double x = control.getDouble("x");
      double y = control.getDouble("y");
      double q = control.getDouble("q");
      Charge charge = new Charge(x, y, q);
      frame.addDrawable(charge);
   }

   public void reset() {
      control.println(
          "Calculate creates a new charge and clears the field lines.");
      control.println("You can drag charges.");
      control.println("Double click in display to compute a field line.");
      frame.clearDrawables(); // remove charges and field lines
      control.setValue("x", 0);
      control.setValue("y", 0);
      control.setValue("q", 1);
   }

   public void handleMouseAction(InteractivePanel panel, MouseEvent evt) {
      panel.handleMouseAction(panel, evt); // panel handles dragging
      switch(panel.getMouseAction()) {
      case InteractivePanel.MOUSE_DRAGGED :
         if(panel.getInteractive()==null) {
            return;
         }
          // field is invalid
         frame.removeObjectsOfClass(FieldLine.class);
         // repaint to keep the screen up to date
         frame.repaint();
         break;
      case InteractivePanel.MOUSE_CLICKED :
         // check for double click
         if(evt.getClickCount()>1) {
            double
               x = panel.getMouseX(), y = panel.getMouseY();
            FieldLine fieldLine = new FieldLine(frame, x, y, +0.1);
            panel.addDrawable(fieldLine);
            fieldLine = new FieldLine(frame, x, y, -0.1);
            panel.addDrawable(fieldLine);
         }
         break;
      }
   }

   public static void main(String[] args) {
      CalculationControl.createApp(new FieldLineApp());
   }
}
```

Problem 10.2. Verification of field line program

(a) Draw field lines for a few simple sets of one, two, and three charges. Choose sets of charges for which all have the same sign and sets for which they are different. Verify that the field lines never connect charges of the same sign. Why do field lines never cross? Are the units of charge and distance relevant?

(b) Compare FieldLineApp and ElectricFieldApp. Which representation conveys more information? Consider how each program provides (or does not provide) information about the electric field magnitude and direction. Discuss some of the difficulties with making an accurate field line diagram.

(c) FieldLine uses a constant value for Δs. Modify the algorithm so that the calculation continues when a field line moves off the screen but speed up the algorithm by increasing the value of Δs.

(d) Removing a field line from the drawing panel in the reset method does not stop the thread. Improve the performance of the program by modifying ElectricFieldApp so that a field line's done variable is set to false when it is removed from the drawing panel. □

Problem 10.3. Electric field lines from point charges

(a) Modify FieldLineApp so that a charge starts ten field lines per unit of charge whenever a new charge is added to the panel or when a charge is moved. Start these field lines close to each charge in such a way that they propagate away from the charge. Should you start these field lines on both positive and negative charges? Explain your answer.

(b) Draw the field lines for an electric dipole.

(c) Draw the field lines for the electric quadrupole with q(1) = 1, x(1) = 1, y(1) = 1, q(2) = −1, x(2) = −1, y(2) = 1, q(3) = 1, x(3) = −1, y(3) = −1, q(4) = −1, x(4) = 1, and y(4) = −1.

(d) A continuous charge distribution can be approximated by a large number of closely spaced point charges. Draw the electric field lines due to a row of ten equally spaced unit charges located between −2.5 and +2.5 on the x-axis. How does the electric field distribution compare to the distribution due to a single point charge?

(e) Repeat part (c) with two rows of equally spaced positive charges on the lines $y = 0$ and $y = 1$, respectively. Then consider one row of positive charges and one row of negative charges. □

Problem 10.4. Field lines due to infinite line of charge

(a) The FieldLineApp program plots field lines in two dimensions. Sometimes this restriction can lead to spurious results (see Freeman). Consider four identical charges placed at the corners of a square. Use the program to plot the field lines. What, if anything, is wrong with the results? What should happen to the field lines near the center of the square?

(b) The two-dimensional analog of a point charge is an infinite line (thin cylinder) of charge perpendicular to the plane. The electric field due to an infinite line of charge is proportional to the linear charge density and inversely proportional to the distance (instead of the distance squared) from the line of charge to a point in the plane. Modify the FieldLine class to compute the field lines from line charges with $E(r) = 1/r$. Use your modified class to draw the field lines due to four identical line charges located at the corners of a square and compare the field lines with your results in part (a).

(c) Use your modified program from part (b) to draw the field lines for the two-dimensional analogs of the distributions considered in Problem 10.3. Compare the results for two and three dimensions and discuss any qualitative differences.

(d) Can your program be used to demonstrate Gauss's law using point charges? What about line charges? □

10.4 Electric Potential

It often is easier to analyze the behavior of a system using energy rather than force concepts. We define the electric potential $V(\mathbf{r})$ by the relation

$$V(\mathbf{r}_2) - V(\mathbf{r}_1) = -\int_{\mathbf{r}_1}^{\mathbf{r}_2} \mathbf{E} \cdot d\mathbf{r} \tag{10.6}$$

or

$$\mathbf{E}(\mathbf{r}) = -\nabla V(\mathbf{r}). \tag{10.7}$$

Only differences in the potential between two points have physical significance. The gradient operator ∇ is given in Cartesian coordinates by

$$\nabla = \frac{\partial}{\partial x}\hat{\mathbf{x}} + \frac{\partial}{\partial y}\hat{\mathbf{y}} + \frac{\partial}{\partial z}\hat{\mathbf{z}} \tag{10.8}$$

where the vectors $\hat{\mathbf{x}}$, $\hat{\mathbf{y}}$, and $\hat{\mathbf{z}}$ are unit vectors along the x-, y-, and z-axes, respectively. If V depends only on the magnitude of $\mathbf{r}$, then (10.7) becomes $E(r) = -dV(r)/dr$. Recall that $V(r)$ for a point charge q relative to a zero potential at infinity is given by

$$V(r) = \frac{q}{r} \qquad \text{(Gaussian units).} \tag{10.9}$$

The surface on which the electric potential has an equal value everywhere is called an *equipotential surface* (a curve in two dimensions). Because $\mathbf{E}$ is in the direction in which the electric potential decreases most rapidly, the electric field lines are orthogonal to the equipotential surfaces at any point.

The Open Source Physics frames package contains the Scalar2DFrame class to provide graphical representations of scalar fields (see Appendix 9B). Problem 10.5 uses a scalar field plot to show the electric potential. The following code fragment shows how to calculate the electric potential at a grid point.

```java
List chargeList = frame.getDrawables(Charge.class);
Iterator it = chargeList.iterator();
while (it.hasNext()) {
   Charge charge = (Charge) it.next();
   double xs = charge.getX(), ys = charge.getY();
   for (int ix = 0; ix < n; ix++) {
      double x= frame.indexToX(ix);
      double dx = (xs - x); // charge gridpoint separation
      for (int iy = 0; iy < n; iy++) {
         double y= frame.indexToY(iy);
         double dy = (ys -y); //charge gridpoint separation
         double r2 = dx * dx + dy * dy;
         double r = Math.sqrt(r2);
         if (r > 0) {
            eField[ix][iy] += charge.q/r;
         }
      }
   }
}
frame.setAll(eField);
```

Problem 10.5. Equipotential contours

(a) Write a program based on `ElectricFieldApp` that draws equipotential lines using the charge distributions considered in Problem 10.3.

(b) Explain why equipotential surfaces (lines in two dimensions) never cross. □

We can use the orthogonality between the electric field lines and the equipotential lines to modify `FieldLineApp` so that it draws the latter. Because the components of the line segment $\Delta\mathbf{s}$ parallel to the electric field line are given by $\Delta x = \Delta s(E_x/E)$ and $\Delta y = \Delta s(E_y/E)$, the components of the line segment perpendicular to $\mathbf{E}$, and hence, parallel to the equipotential line, are given by $\Delta x = -\Delta s(E_y/E)$ and $\Delta y = \Delta s(E_x/E)$. It is unimportant whether the minus sign is assigned to the x or y component, because the only difference would be the direction that the equipotential lines are drawn.

Problem 10.6. Equipotential lines

(a) Write a program that is based on `FieldLineApp` and `FieldLine` to draw some of the equipotential lines for the charge distributions considered in Problem 10.3. Use a mouse click to determine the initial position of an equipotential line. The equipotential calculation should stop when the line returns close to the starting point or after an unreasonable number of calculations. The program should also kill the thread when the user moves a charge, hits the Reset button, or when the application terminates.

(b) What would a higher density of equipotential lines mean if we drew lines such that each adjacent line differed from a neighboring one by a fixed potential difference?

(c) Explain why equipotential surfaces never cross. □

Problem 10.7. The electric potential due to a finite sheet of charge

Consider a uniformly charged nonconducting plate of total charge Q and linear dimension L centered at $(0,0,0)$ in the x-y plane. In the limit $L \to \infty$ with the charge density $\sigma = Q/L^2$ a constant, we know that the electric field is normal to the sheet and its magnitude is given by $2\pi\sigma$ (Gaussian units). What is the electric field due to a finite sheet of charge? A simple method is to divide the plate into a grid of p square regions on a side such that each region is sufficiently small to be approximated by a point charge of magnitude $q = Q/p^2$. Because the potential is a scalar, it is easier to compute the total potential rather than the total electric field from the $N = p^2$ point charges. Use the relation (10.9) for the potential from a point charge and write a program to compute $V(z)$ and hence, $E_z = -\partial V(z)/\partial z$ for points along the z-axis and perpendicular to the sheet. Take $L = 1$, $Q = 1$, and $p = 10$ for your initial calculations. Increase p until your results for $V(z)$ do not change significantly. Plot $V(z)$ and E_z as a function of z and compare their z-dependence to their infinite sheet counterparts. □

***Problem 10.8.** Electrostatic shielding

We know that the (static) electric field is zero inside a conductor, all excess charges reside on the surface of the conductor, and the surface charge density is greatest at the points of greatest curvature. Although these properties are plausible, it is instructive to do a simulation to see how these properties follow from Coulomb's law. For simplicity, consider the conductor to be two-dimensional so that the potential energy is proportional to $\ln r$ rather than $1/r$ (see Problem 10.4). It is also convenient to choose the surface of the conductor to be an ellipse.

(a) If we are interested only in the final distribution of the charges and not in the dynamics of the system, we can use a Monte Carlo method. Our goal is to find the minimum energy configuration beginning with the N charges randomly placed within a conducting ellipse. One method is to choose a charge i at random and make a trial change in the position of the charge. The trial position should be no more than δ from the old position and still be within the ellipse. Choose $\delta \approx b/10$, where b is the semiminor axis of the ellipse. Compute the change in the total potential energy given by (in arbitrary units)

$$\Delta U = -\sum_j [\ln r_{ij}^{(\text{new})} - \ln r_{ij}^{(\text{old})}]. \tag{10.10}$$

The sum is over all charges in the system not including i. If $\Delta U > 0$, then reject the trial move; otherwise accept it. Repeat this procedure many times until very few trial moves are accepted. Write a program to implement this Monte Carlo algorithm. Run the simulation for $N \geq 20$ charges inside a circle and then repeat the simulation for an ellipse. How are the charges distributed in the (approximately) minimum energy distribution? Which parts of the ellipse have a higher charge density?

(b) Repeat part (a) for a two-dimensional conductor, but assume that the potential energy $U \sim 1/r$. Do the charges move to the surface?

(c) Is it sufficient that the interaction be repulsive for the results of parts (a) and (b) to hold?

(d) Repeat part (a) with the added condition that there is a fixed positive charge of magnitude $N/2$ located outside the ellipse. How does this fixed charge affect the charge distribution? Are the excess free charges still at the surface? Try different positions for the fixed charge.

(e) Repeat parts (a) and (b) for $N = 50$ charges located within an ellipsoid in three dimensions.

□

10.5 Numerical Solutions of Boundary Value Problems

In Section 10.1 we found the electric fields and potentials due to a fixed distribution of charges. Suppose that we do not know the positions of the charges but instead know only the potential on a set of boundaries surrounding a charge-free region. This information is sufficient to determine the potential $V(\mathbf{r})$ at any point within the charge-free region.

The direct method of solving for $V(x,y,z)$ is based on Laplace's equation which can be expressed in Cartesian coordinates as

$$\nabla^2 V(x,y,z) \equiv \frac{\partial^2 V}{\partial x^2} + \frac{\partial^2 V}{\partial y^2} + \frac{\partial^2 V}{\partial z^2} = 0. \tag{10.11}$$

The problem is to find the function $V(x,y,z)$ that satisfies (10.11) and the specified boundary conditions. This type of problem is an example of a *boundary value* problem. Because analytic methods for regions of arbitrary shape do not exist, the only general approach is to use numerical methods. Laplace's equation is not a new law of physics, but can be derived directly from (10.7) and the relation $\boldsymbol{\nabla}\cdot\mathbf{E} = 0$ or indirectly from Coulomb's law in regions of space where there is no charge.

For simplicity, we consider only two-dimensional boundary value problems for $V(x,y)$. We use a finite difference method and divide space into a discrete grid of sites located at the coordinates (x,y). In Problem 10.9b, we show that in the absence of a charge at (x,y), the discrete form of Laplace's equation satisfies the relation

$$\begin{aligned} V(x,y) \approx\ & \frac{1}{4}[V(x+\Delta x,y)+V(x-\Delta x,y) \\ & +V(x,y+\Delta y)+V(x,y-\Delta y)] \qquad \text{(two dimensions)} \end{aligned} \tag{10.12}$$

where $V(x,y)$ is the value of the potential at the site (x,y). Equation (10.12) says that $V(x,y)$ is the average of the potential of its four nearest neighbor sites. This remarkable property of $V(x,y)$ can be derived by approximating the partial derivatives in (10.11) by finite differences (see Problem 10.9b).

In Problem 10.9(a) we verify (10.12) by calculating the potential due to a point charge at a point in space we select and at the four nearest neighbors. As the form of (10.12) implies, the average of the potential at the four neighboring sites should equal the potential at the center site. We assume the form (10.9) for the potential $V(r)$ due to a point charge, a form that satisfies Laplace's equation for $r \neq 0$.

Problem 10.9. Verification of the difference equation for the potential

(a) Modify `PotentialFieldApp` to compare the computed potential at a point to the average of the potential at its four nearest neighbor sites. Choose reasonable values for the spacings Δx and Δy and consider a point that is not too close to the source charge. Do similar measurements for other points. Does the relative agreement with (10.12) depend on the distance of the point to the source charge? Choose smaller values of Δx and Δy and determine if your results are in better agreement with (10.12). Does it matter whether Δx and Δy have the same value?

(b) Derive the finite difference equation (10.12) for $V(x,y)$ using the second-order Taylor expansion:

$$V(x+\Delta x,y) = V(x,y)+\Delta x\frac{\partial V(x,y)}{\partial x}+\frac{1}{2}(\Delta x)^2\frac{\partial^2 V(x,y)}{\partial x^2}+\cdots \tag{10.13}$$

$$V(x,y+\Delta y) = V(x,y)+\Delta y\frac{\partial V(x,y)}{\partial y}+\frac{1}{2}(\Delta y)^2\frac{\partial^2 V(x,y)}{\partial y^2}+\cdots. \tag{10.14}$$

The effect of including higher derivatives is discussed by MacDonald (see references). □

Now that we have found that (10.12), a finite difference form of Laplace's equation, is consistent with Coulomb's law, we adopt (10.12) as the basis for computing the potential for systems for which we cannot calculate the potential directly. In particular, we consider problems where the potential is specified on a closed surface that divides space into interior and exterior regions in which the potential is independently determined. For simplicity, we consider only two-dimensional geometries. The approach, known as the *relaxation method*, is based on the following algorithm:

1. Divide the region of interest into a rectangular grid spanning the region. The region is enclosed by a surface (curve in two dimensions) with specified values of the potential along the curve.

2. Assign to a boundary site the potential of the boundary nearest the site.

3. Assign all interior sites an arbitrary potential (preferably a reasonable guess).
4. Compute new values for the potential V for each interior site. Each new value is obtained by finding the average of the previous values of the potential at the four nearest neighbor sites.
5. Repeat step (4) using the values of V obtained in the previous iteration. This iterative process is continued until the potential at each interior site is computed to the desired accuracy.

The program shown in Listing 10.5 implements this algorithm using a grid of voltages and a boolean grid to signal the presence of a conductor.

Listing 10.5: The LaplaceApp program solves the Laplace equation using the relaxation method.

```
package org.opensourcephysics.sip.ch10;
import java.awt.event.*;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.display.*;
import org.opensourcephysics.display2d.*;
import org.opensourcephysics.frames.*;

public class LaplaceApp extends AbstractSimulation implements
               InteractiveMouseHandler {
   Scalar2DFrame frame = new Scalar2DFrame("x", "y",
            "Electric potential");
   boolean[][] isConductor;
   double[][] potential; // electric potential
   double maximumError;
   int gridSize;          // number of sites on side of grid

   public LaplaceApp() {
      frame.setInteractiveMouseHandler(this);
   }

   public void initialize() {
      maximumError = control.getDouble("maximum error");
      gridSize = control.getInt("size");
      initArrays();
      frame.setVisible(true);
      frame.showDataTable(true); // show the data table
   }

   public void initArrays() {
      isConductor = new boolean[gridSize][gridSize];
      potential = new double[gridSize][gridSize];
      frame.setPaletteType(ColorMapper.DUALSHADE);
      // isConductor array is false by default
      // voltage in potential array is 0 by default
      for(int i = 0;i<gridSize;i++) {        // initialize the sides
         isConductor[0][i] = true;           // left boundary
         isConductor[gridSize-1][i] = true; // right boundary
         isConductor[i][0] = true;           // bottom boundary
         isConductor[i][gridSize-1] = true; // top boundary
      }
      // set potential on inner conductor
      for(int i = 5;i<gridSize-5;i++) {
         potential[gridSize/3][i] = 100;
         isConductor[gridSize/3][i] = true;
         potential[2*gridSize/3][i] = -100;
         isConductor[2*gridSize/3][i] = true;
      }
      frame.setAll(potential);
   }

   public void doStep() {
      double error = 0;
      for(int i = 1;i<gridSize-1;i++) {
         for(int j = 1;j<gridSize-1;j++) {
            // change the voltage for nonconductors
            if(!isConductor[i][j]) {
               double v = (potential[i-1][j]+potential[i+1][j]
                           +potential[i][j-1]+potential[i][j+1])/4;
               double dv = potential[i][j]-v;
               error = Math.max(error, Math.abs(dv));
               potential[i][j] = v;
            }
         }
      }
      frame.setAll(potential);
      if(error<maximumError) {
         // stop the simulation thread
         animationThread = null;
         control.calculationDone("Computation done.");
      }
   }

   public void reset() {
      control.setValue("maximum error", 0.1);
      control.setValue("size", 31);
      initialize();
   }

   public void handleMouseAction(InteractivePanel panel, MouseEvent evt) {
      switch(panel.getMouseAction()) {
      case InteractivePanel.MOUSE_DRAGGED :
      case InteractivePanel.MOUSE_PRESSED :
         double x = panel.getMouseX(); // mouse x in world units
         double y = panel.getMouseY();
         int i = frame.xToIndex(x);    // closest array index
         int j = frame.yToIndex(y);
         frame.setMessage("V="+decimalFormat.format(potential[i][j]));
         break;
      case InteractivePanel.MOUSE_RELEASED :
         panel.setMessage(null);
         break;
      }
   }

   public static void main(String[] args) {
      SimulationControl.createApp(new LaplaceApp());
   }
}
```

As the algorithm loops through the grid sites, it first checks if each grid site is a conductor. If it is, the site is skipped. If not, a new potential is calculated and assigned to the proper element in the `potential` array. A local variable named `maximumError` keeps track of the maximum difference between the potential at a site and the average potential of the four neighbors. This variable is used to determine the end of the simulation.

In Problems 10.10–10.12 you are asked to modify `LaplaceApp` to compute the potential for various geometries.

Problem 10.10. Numerical solution of the potential within a rectangular region

(a) Modify `LaplaceApp` to determine the potential $V(x,y)$ in a square region with linear dimension $L = 10$. The boundary of the square is at a potential $V = 10$. Choose the grid size $\Delta x = \Delta y = 1$. Before you run the program, guess the exact form of $V(x,y)$ and set the initial values of the interior potential 10% lower than the exact answer. How many iterations are necessary to achieve 1% accuracy? Decrease the grid size by a factor of two and determine the number of iterations that are now necessary to achieve 1% accuracy.

(b) Consider the same geometry as in part (a), but set the initial potential at the interior sites equal to zero except for the center site whose potential is set equal to four. Does the potential distribution evolve to the same values as in part (a)? What is the effect of a poor initial guess? Are the final results independent of your initial guess?

(c) Modify `LaplaceApp` so that the value of the potential at the four sides is 5, 10, 5, and 10, respectively (see Figure 10.1). Sketch the equipotential surfaces. What happens if the potential is 10 on three sides and 0 on the fourth? Start with a reasonable guess for the initial values of the potential at the interior sites and iterate until 1% accuracy is obtained.

(d)* Consider the same initial choice of the potential as in part (b) and focus your attention on the potential at the sites near the center of the square. If the central site has an initial potential of four, what is the potential at the nearest neighbor sites after the first iteration? Follow the distribution of the potential as a function of the number of iterations and verify that the nature of the relaxation of the potential to its correct distribution is closely related to *diffusion* (see Chapter 7). It may be helpful to increase the number of sites in the grid and the initial value of the potential at the central site to see the nature of the relaxation more clearly. □

In Problem 10.10, we implemented a simple version of the relaxation method known as the Jacobi method. In particular, the new potential at each site is based on the values of the potentials at the neighboring sites at the previous iteration. After the entire lattice is visited, the potential at each site is updated *simultaneously*. The difficulty with this relaxation method is that it converges very slowly. The use of more general relaxation methods is discussed in many texts (cf. Sadiku or Press et al.). In Problem 10.11 we consider a method known as *Gauss–Seidel* relaxation.

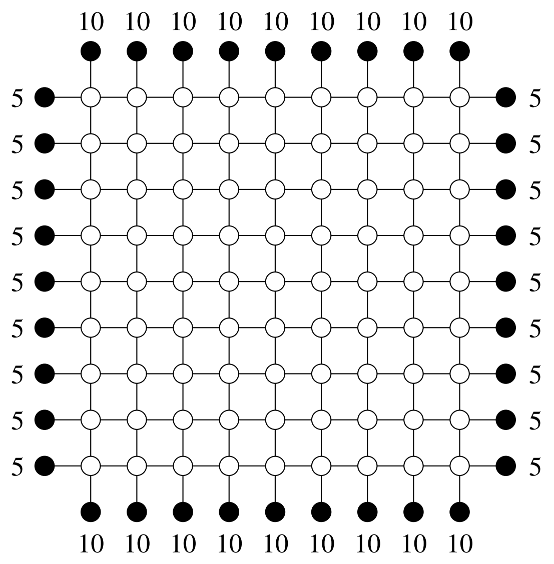


Figure 10.1: Potential distribution considered in Problem 10.10c. The number of interior sites in each direction is nine.

Problem 10.11. Gauss–Seidel relaxation

(a) Modify the program that you used in Problem 10.10 so that the potential at each site is updated sequentially. That is, after the average potential of the nearest neighbor sites of site i is computed, update the potential at i immediately. In this way the new potential of the next site is computed using the most recently computed values of its nearest neighbor potentials. Are your results better, worse, or about the same as for the simple relaxation method?

(b) Imagine coloring the alternate sites of a grid red and black, so that the grid resembles a checkerboard. Modify the program so that all the red sites are updated first, and then all the black sites are updated. This ordering is repeated for each iteration. Do your results converge any more quickly than in part (a)?

c)* The slow convergence of the relaxation methods we have explored is due to the fact that it takes a long time for a change in the potential at one site to effect changes further away. We can improve the Gauss–Seidel method by using an *overrelaxation* method that updates the new potential as follows:

$$V_{\text{new}}(x,y) = wV_{\text{ave}}(x,y) + (1-w)V(x,y) \tag{10.15}$$

where $V_{\text{ave}}(x,y)$ is the average of the potential of the four neighbors of (x,y). The overrelaxation parameter w is in the range $1 < w < 2$. The effect of w is to cause the potential to change by a greater amount than in the simple relaxation procedure. Explore the dependence of the rate of convergence on w. A relaxation method that increases the rate of convergence is explored in Project 10.26. □

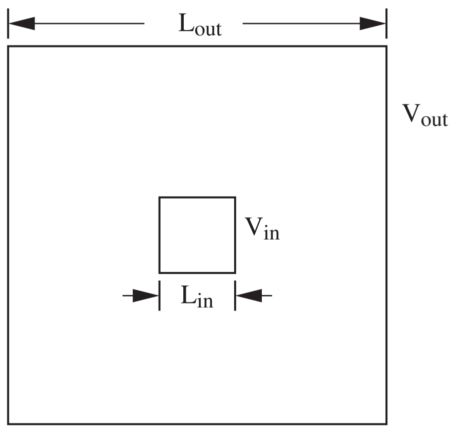


Figure 10.2: The geometry of the two concentric squares considered in Problem 10.12.

Problem 10.12. The capacitance of concentric squares

(a) Use a relaxation method to compute the potential distribution between the two concentric square cylinders shown in Figure 10.2. The potential of the outer square conductor is $V_{\rm out} = 10$, and the potential of the inner square conductor is $V_{\rm in} = 5$. The linear dimensions of the exterior and interior squares are $L_{\rm out} = 25$ and $L_{\rm in} = 5$, respectively. Modify your program so that the potential of the interior square is fixed. Sketch the equipotential surfaces.

(b) A system of two conductors with charge Q and $-Q$ respectively has a capacitance C that is defined as the ratio of Q to the potential difference ΔV between the two conductors. Determine the capacitance per unit length of the concentric cylinders considered in part (a). In this case $\Delta V = 5$. The charge Q can be determined from the fact that near a conducting surface, the surface charge density σ is given by $\sigma = E_n/4\pi$, where E_n is the magnitude of the electric field normal to the surface. E_n can be approximated by the relation $-\delta V/\delta r$, where δV is the potential difference between a boundary site and an adjacent interior site a distance δr away. Use the result of part (a) to compute δV for each site adjacent to the two square surfaces. Use this information to determine E_n for the two surfaces and the charge per unit length on each conductor. Are the charges equal and opposite in sign? Compare your numerical result to the capacitance per unit length, $1/2 \ln r_{\rm out}/r_{\rm in}$, of a system of two concentric circular cylinders of radii $r_{\rm out}$ and $r_{\rm in}$. Assume that the circumference of each cylinder equals the perimeter of the corresponding square, that is, $2\pi r_{\rm out} = 4L_{\rm out}$ and $2\pi r_{\rm in} = 4L_{\rm in}$.

(c) Move the inner square 1 cm off center and repeat the calculations of parts (a) and (b). How do the potential surfaces change? Is there any qualitative difference if we set the inner conductor potential equal to -5? □

Laplace's equation holds only in charge-free regions. If there is a charge density $\rho(x,y,z)$ in the region, we need to use *Poisson's* equation which can be written as

$$\nabla^2 V(\mathbf{r}) = \frac{\partial^2 V}{\partial x^2} + \frac{\partial^2 V}{\partial y^2} + \frac{\partial^2 V}{\partial z^2} = -4\pi\rho(\mathbf{r}) \tag{10.16}$$

where $\rho(\mathbf{r})$ is the charge density. The difference form of Poisson's equation is given in two dimensions by

$$\begin{aligned} V(x,y) \approx\ & \frac{1}{4}\Big[V(x+\Delta x,y)+V(x-\Delta x,y)+V(x,y+\Delta y)+V(x,y-\Delta y)\Big] \\ & + \frac{1}{4}\Delta x\Delta y\, 4\pi\rho(x,y). \end{aligned} \tag{10.17}$$

Note that the product $\rho(x,y)\Delta x\Delta y$ is the total charge in a $\Delta x \times \Delta y$ region centered at (x,y).

Problem 10.13. Surface charge

(a) Poisson's equation can be used to find the surface charge on a conductor after Laplace's equation has been solved. The potential is fixed at the boundary sites. If we assume the boundary is a conductor with some thickness, we can assume that the potential for the next layer of sites outside the boundary has the same potential as the boundary. If we use this assumption, then after we have solved numerically for the potential of the interior sites, we will find that the average value of the neighbors of a boundary site will not equal the imposed potential. From (10.17) the difference will equal $\Delta x\Delta y\pi\rho(x,y)$. Modify `LaplaceApp` to calculate and display the surface charge density, assuming $\Delta x = \Delta y = 1$. Notice that because we are in two dimensions the "surface" charge density, $\Delta x\Delta y\rho(x,y)$, is a linear density of charge per unit length.

(b) Consider the same system as in Problem 10.10(c) and find the surface charge density on the boundary sites. Make a reasonable choice for assigning the potential at the corner sites.

(c) Model a system with the boundary at a potential $V = 0$ and a centered interior rectangle of 6×12 at a potential of $V = 10$. Where is the charge density the highest?

(d) Repeat if the interior rectangle is placed close to an edge. □

Problem 10.14. Numerical solution of Poisson's equation

(a) Consider a square of linear dimension $L = 25$ whose boundary is fixed at a potential equal to $V = 10$. Assume that the interior region has a uniform charge density ρ such that the total charge is $Q = 1$. Modify `LaplaceApp` to compute the potential distribution for this case. Compare the equipotential surfaces obtained for this case to that found in Problem 10.12.

(b) Find the potential distribution if the charge distribution of part (a) is restricted to a 5×5 square at the center.

(c) Find the potential distribution if the charge distribution of part (a) is restricted to a 1×1 square at the center. How does the potential compare to that of a point charge without the boundary? □

***Problem 10.15.** Vector potential and magnetic fields

The magnetic field from arbitrary currents can also be obtained using Poisson's equation. The field is generated from a vector potential $\mathbf{A}$ that satisfies

$$\nabla^2\mathbf{A} = \mu\mathbf{j} \tag{10.18}$$

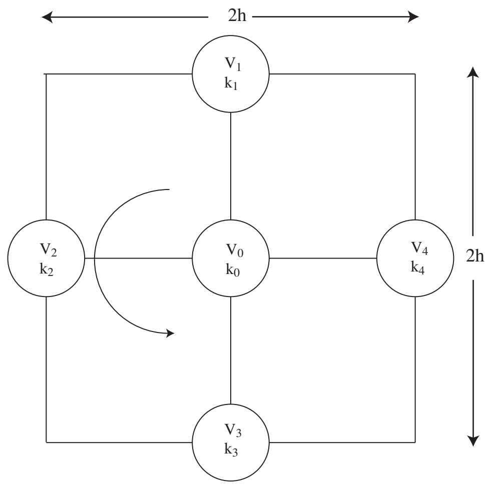


Figure 10.3: The grid used to compute the integral in (10.21) is based on Gauss's law for the electric field.

where **j** is the current density in the wires and μ is the magnetic permeability. If current flows only in the z direction, then $\mathbf{j} = (0, 0, j_z(x, y))$ and $\mathbf{A} = (0, 0, A_z(x, y))$, and we again have a two-dimensional problem that can be solved using the relaxation method.

Do a simulation that models the magnetic field from an arbitrary number of wires. Combine features of the `ElectricFieldApp` and the `LaplaceApp` programs. The program should read the control and create a current carrying wire when a custom button is clicked. The computation is performed using the animation's `doStep` method to perform a Gauss–Seidel relaxation step. Compute the magnetic field after the computation converges by computing the curl of the vector potential:

$$\mathbf{B} = \nabla \times \mathbf{A}. \tag{10.19}$$

See (10.52) for how to compute the curl when only discrete values are available. □

Dielectrics can be added to the solution of Laplace's equation by adding an array to store the dielectric constant k at every grid site and imposing the condition

$$D_{1n} = D_{2n} \tag{10.20}$$

where $\mathbf{D} = k\mathbf{E}$ and k is the dielectric susceptibility. This condition is equivalent to

$$0 = \oint_l k\boldsymbol{\nabla}V \cdot d\mathbf{l} = \oint_l k\frac{\partial V}{\partial n}dl \tag{10.21}$$

where $\partial V/\partial n$ denotes the derivative of V normal to the contour l. The vector $d\mathbf{l}$ is the two-dimensional equivalent of a surface vector. Its magnitude is the length of a line segment and its direction is perpendicular to the tangent of the segment. If we approximate (10.21) along each edge of length $2h$ using a finite difference for the derivative, we obtain

$$0 = k_1\frac{V_1 - V_0}{h}2h + k_2\frac{V_2 - V_0}{h}2h + k_3\frac{V_3 - V_0}{h}2h + k_4\frac{V_4 - V_0}{h}2h. \tag{10.22}$$

We rearrange terms in (10.22) and find a modified form of (10.12) that includes the dielectric

$$V_0 = \frac{1}{4(k_1 + k_2 + k_3 + k_4)}[k_1V_1 + k_2V_2 + k_3V_3 + k_4V_4] \tag{10.23}$$

where k_i is the average dielectric constant at a site where the electric potential is V_i.

Problem 10.16. Capacitor with dielectric

(a) Modify your Laplace program to include a dielectric medium. That is, create an array of dielectric susceptibilities and implement (10.23) using a relaxation algorithm. Be sure to set the dielectric array elements to unity in free space and inside conductors.

(b) Test your algorithm by creating a capacitor consisting of +10 and −10 potential plates near the center of the grid. Initialize the dielectric susceptibility to two in half the capacitor and run the program. Use a `Scalar2DFrame` to display the electric potential, but note that some representations of the scalar field are more appropriate than others. Compare the spacing between the contour lines inside and outside the dielectric. Why does the spacing change?

(c) The bound charge on the surface of a dielectric can be computed by subtracting $V(x,y)$ from the average of the potential at the four nearest neighbor sites. You are, in effect, using (10.17) to solve for the charge. Implement this calculation and describe the bound charge on the surface of the dielectric. □

10.6 Random Walk Solution of Laplace's Equation

In Section 10.5 we found that the solution to Laplace's equation in two dimensions at the point (x,y) is given by

$$V(x,y) = \frac{1}{4}\sum_{i=1}^{4} V(i) \tag{10.24}$$

where $V(i)$ is the value of the potential at the ith neighbor. A generalization of this result is that the potential at any point equals the average of the potential on a circle (or sphere in three dimensions) centered about that point.

The relation (10.24) can be given a probabilistic interpretation in terms of random walks (see Problem 10.10d). Suppose that many random walkers are at the site (x,y) and each walker "jumps" to one of its four neighbors (on a square grid) with equal probability $p = 1/4$. From (10.24) we see that the average potential found by the walkers after jumping one step is the potential at (x,y). This relation generalizes to walkers that visit a site on a closed surface with fixed potential. The random walk algorithm for computing the solution to Laplace's equation can be stated as:

1. Begin at a point (x,y) where the value of the potential is desired and take a step in a random direction.

2. Continue taking steps until the walker reaches the surface. Record $V_b(i)$, the potential at the boundary site i. A typical walk is shown in Figure 10.4.

3. Repeat steps (1) and (2) n times and sum the potential found at the surface each time.

4. The value of the potential at the point (x,y) is estimated by

$$V(x,y) = \frac{1}{n}\sum_{i=1}^{n} V_b(i) \tag{10.25}$$

where n is the total number of random walkers.

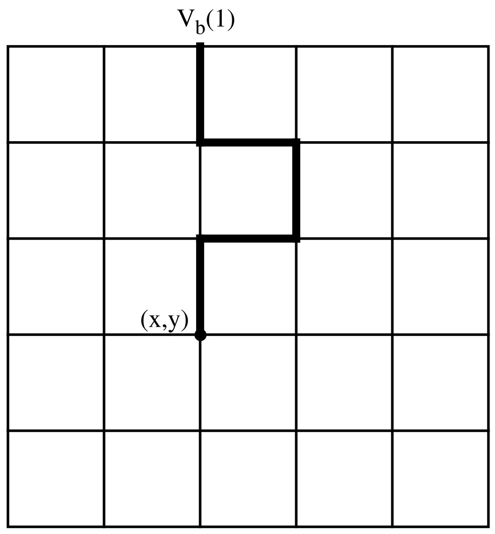


Figure 10.4: A random walk on a 6×6 grid starting at the point $(x, y) = (3, 3)$ and ending at the boundary site $V_b(3, 6)$ where the potential is recorded.

Problem 10.17. Random walk solution of Laplace's equation

(a) Consider the square region shown in Figure 10.1 and compare the results of the random walk method with the results of the relaxation method (see Problem 10.10c). Try $n = 100$ and $n = 1000$ walkers and choose a point near the center of the square.

(b) Repeat part (a) for other points within the square. Do you need more or less walkers when the potential near the surface is desired? How quickly do your answers converge as a function of n? □

The disadvantage of the random walk method is that it requires many walkers to obtain a good estimate of the potential at each site. However, if the potential is needed at only a small number of sites, then the random walk method might be more appropriate than the relaxation method, which requires the potential to be computed at all points within the region. Another case where the random walk method is appropriate is when the geometry of the boundary is fixed, but the potential in the interior for a variety of different boundary potentials is needed. In this case the quantity of interest is $G(x, y, x_b, y_b)$, the number of times that a walker from the point (x, y) lands at the boundary (x_b, y_b). The random walk algorithm is equivalent to the relation

$$V(x,y) = \frac{1}{n} \sum_{x_b, y_b} G(x, y, x_b, y_b) V(x_b, y_b) \tag{10.26}$$

where the sum is over all sites on the boundary. We can use the same function G for different distributions of the potential on a given boundary. G is an example of a Green's function, a function that you will encounter in advanced treatments of electrodynamics and quantum mechanics (cf. Section 16.9). Of course, if we change the geometry of the boundary, we have to recompute the function G.

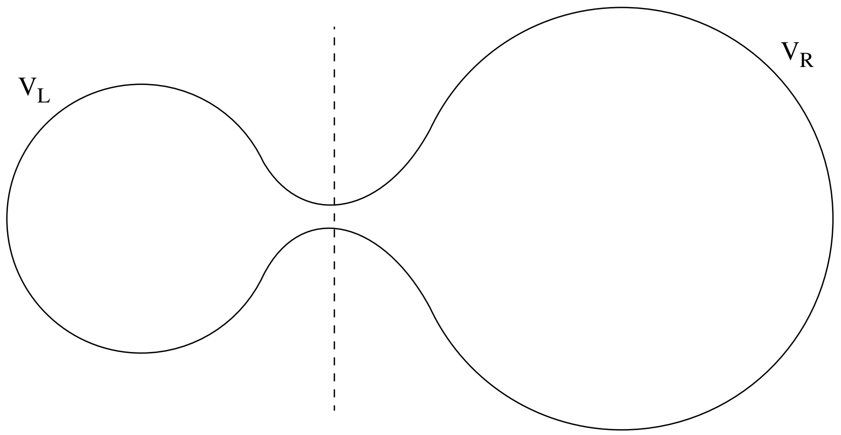


Figure 10.5: Two regions of space connected by a narrow neck. The boundary of the left region has a potential V_L, and the boundary of the right region has a potential V_R.

Problem 10.18. Green's function solution of Laplace's equation

(a) Compute the Green's function $G(x,y,x_b,y_b)$ for the same geometry considered in Problem 10.17. Use at least 200 walkers at each interior site to estimate G. Because of the symmetry of the geometry, you can determine some of the values of G from other values without doing an additional calculation. Store your results for G in a file.

(b) Use your results for G found in part (a) to determine the potential at each interior site when the boundary potential is the same as in part (a), except for five boundary sites which are held at $V = 20$. Find the locations of the five boundary sites that maximize the potential at the interior site located at $(3,5)$. Repeat the calculation to maximize the potential at $(5,3)$. Use trial and error guided by your physical intuition. □

The random walk algorithm can help us gain additional insight into the nature of the solutions of Laplace's equation. Suppose that you have a boundary similar to the one shown in Figure 10.5. The potentials on the left and right boundaries are V_L and V_R, respectively. If the neck between the two sides is narrow, it is clear that a random walker starting on the left side has a low probability of reaching the other side. Hence, we can conclude that the potential in the interior of the left side is approximately V_L, except very near the neck.

Poisson's equation can also be solved using the random walk method. In this case, the potential is given by

$$V(x,y) = \frac{1}{n}\sum_{\alpha} V(\alpha) + \frac{\pi\Delta x\Delta y}{n}\sum_{i,\alpha}\rho(x_{i,\alpha},y_{i,\alpha}) \tag{10.27}$$

where α labels the walker and i labels the site visited by the walker. That is, each time a walker is at site i, we add the charge density at that site to the second sum in (10.27).

10.7 *Fields Due to Moving Charges

The fact that accelerating charges radiate electromagnetic waves is one of the more important results in the history of physics. In this section we discuss a numerical algorithm for computing the electric and magnetic fields due to the motion of charged particles. The algorithm is very general, but requires some care in its application.

To understand the algorithm, we need a few results that can be conveniently found in Feynman's lectures. We begin with the fact that the scalar potential at the observation point **R** due to a stationary particle of charge q is

$$V(\mathbf{R}) = \frac{q}{|\mathbf{R} - \mathbf{r}|} \tag{10.28}$$

where **r** is the position of the charged particle. The electric field is given by

$$\mathbf{E}(\mathbf{R}) = -\frac{\partial V(\mathbf{R})}{\partial \mathbf{R}} \tag{10.29}$$

where $\partial V(\mathbf{R})/\partial \mathbf{R}$ is the gradient with respect to the coordinates of the observation point. (Note that our notation for the observation point differs from that used in other sections of this chapter.) How do the relations (10.28) and (10.29) change when the particle is moving? We might guess that because it takes a finite time for the disturbance due to a charge to reach the point of observation, we should modify (10.28) by writing

$$V(\mathbf{R}) \overset{?}{=} \frac{q}{r_{\text{ret}}} \tag{10.30}$$

where

$$r_{\text{ret}} = |\mathbf{R} - \mathbf{r}(t_{\text{ret}})|. \tag{10.31}$$

The quantity r_{ret} is the separation of the charged particle from the observation point **R** at the retarded time t_{ret}. The latter is the time at which the particle was at $\mathbf{r}(t_{\text{ret}})$ such that a disturbance starting at $\mathbf{r}(t_{\text{ret}})$ and traveling at the speed of light would reach **R** at time t; t_{ret} is given by the implicit equation

$$t_{\text{ret}} = t - \frac{r_{\text{ret}}(t_{\text{ret}})}{c} \tag{10.32}$$

where t is the observation time and c is the speed of light.

Although the above reasoning is plausible, the relation (10.30) is not quite correct (cf. Feynman et al. for a derivation of the correct result). We need to take into account that the potential due to the charge is a maximum if the particle is moving toward the observation point and a minimum if it is moving away. The correct result can be written as

$$V(\mathbf{R}, t) = \frac{q}{r_{\text{ret}}\big(1 - \hat{\mathbf{r}}_{\text{ret}} \cdot \mathbf{v}_{\text{ret}}/c\big)} \tag{10.33}$$

where

$$\mathbf{v}_{\text{ret}} = \frac{d\mathbf{r}(t)}{dt}\Big|_{t=t_{\text{ret}}} \tag{10.34}$$

and $\hat{\mathbf{r}} = \mathbf{r}/r$.

To find the electric field of a moving charge, we recall that the electric field is related to the time rate of change of the magnetic flux. Hence, we expect that the total electric field at the observation point **R** has a contribution due to the magnetic field created by the motion of the charge. We know that the magnetic field due to a moving charge is given by

$$\mathbf{B} = \frac{1}{c}\frac{q\mathbf{v} \times \mathbf{r}}{r^3}. \tag{10.35}$$

If we define the vector potential **A** as

$$\mathbf{A} = \frac{q}{r}\frac{\mathbf{v}}{c} \tag{10.36}$$

we can express **B** in terms of **A** as

$$\mathbf{B} = \nabla \times \mathbf{A}. \tag{10.37}$$

As we did for the scalar potential V, we argue that the correct formula for **A** is

$$\mathbf{A}(\mathbf{R}, t) = q\frac{\mathbf{v}_{\text{ret}}/c}{r_{\text{ret}}\Big(1 - \hat{\mathbf{r}}_{\text{ret}} \cdot \mathbf{v}_{\text{ret}}/c\Big)}. \tag{10.38}$$

Equations (10.33) and (10.38) are known as the Liénard–Wiechert form of the potentials.

The contribution to the electric field **E** from V and **A** is given by

$$\mathbf{E} = -\nabla V - \frac{1}{c}\frac{\partial \mathbf{A}}{\partial t}. \tag{10.39}$$

The derivatives in (10.39) are with respect to the observation coordinates. The difficulty associated with calculating these derivatives is that the potentials depend on t_{ret}, which in turn depends on **R**, **r**, and t. The result can be expressed as

$$\mathbf{E}(\mathbf{R}, t) = \frac{q r_{\text{ret}}}{\Big(\mathbf{r}_{\text{ret}} \cdot \mathbf{u}_{\text{ret}}\Big)^3}\Big[\mathbf{u}_{\text{ret}}(c^2 - v_{\text{ret}}^2) + \mathbf{r}_{\text{ret}} \times \Big(\mathbf{u}_{\text{ret}} \times \mathbf{a}_{\text{ret}}\Big)\Big] \tag{10.40}$$

where

$$\mathbf{u}_{\text{ret}} \equiv c\hat{\mathbf{r}}_{\text{ret}} - \mathbf{v}_{\text{ret}}. \tag{10.41}$$

The acceleration of the particle is given by $\mathbf{a}_{\text{ret}} = d\mathbf{v}(t)/dt|_{t=t_{\text{ret}}}$. We can also show using (10.37) that the magnetic field **B** is given by

$$\mathbf{B} = \hat{\mathbf{r}}_{\text{ret}} \times \mathbf{E}. \tag{10.42}$$

The above discussion is not rigorous but leads to the correct expressions for **E** and **B**. We suggest that you accept (10.40) and (10.42) in the same spirit as you accepted Coulomb's law and the Biot-Savart law. All of classical electrodynamics can be reduced to (10.40) and (10.42) if we assume that the sources of all fields are charges, and all electric currents are due to the motion of charged particles. Note that (10.40) and (10.42) are consistent with the special theory of relativity and reduce to known results in the limit of stationary charges and steady currents.

Although (10.40) and (10.42) are deceptively simple (we do not even have to solve any differential equations), it is difficult to calculate the fields analytically even if the position of a charged particle is an analytic function of time. The difficulty is that we must find the retarded time t_{ret} from (10.32) for each observation position **R** and time t. For example, consider a charged particle whose motion is sinusoidal, that is, $x(t_{\text{ret}}) = A\cos\omega t_{\text{ret}}$. To calculate the fields at the position $\mathbf{R} = (X, Y, Z)$ at time t, we need to solve the following transcendental equation for t_{ret}:

$$t_{\text{ret}} = t - \frac{r_{\text{ret}}}{c} = t - \frac{1}{c}\sqrt{(X - A\cos^2\omega t_{\text{ret}})^2 + Y^2 + Z^2}. \tag{10.43}$$

The solution of (10.43) can be expressed as a root finding problem for which we need to find the zero of the function $f(t_{\text{ret}})$:

$$f(t_{\text{ret}}) = t - t_{\text{ret}} - \frac{r_{\text{ret}}}{c}. \tag{10.44}$$

There are various ways of finding the solution for the retarded time. For example, if the motion of the charges is given by an analytic expression, we can employ *Newton's method* or the *bisection method*. Because we will store the path of the charged particle, we use a simple method that looks for a change in the sign of the function $f(t_{\text{ret}})$ along the path. First find a value t_a such that $f(t_a) > 0$ and another value t_b such that $f(t_b) < 0$. Because $f(t_{\text{ret}})$ is continuous, there is a value of t_{ret} in the interval $t_a < t_{\text{ret}} < t_b$ such that $f(t_{\text{ret}}) = 0$. This technique is used in the `RadiatingCharge` class shown in Listing 10.6. Note that the particle's path is a sinusoidal oscillation specified in method `evaluate`. The name `evaluate` is used because `RadiatingCharge` implements the `Function` interface, which requires an `evaluate` method. What is the maximum velocity for a particle that moves according to this function?

Listing 10.6: The `RadiatingCharge` class computes the radiating electric and magnetic fields using Liénard–Wiechert potentials.

```java
package org.opensourcephysics.sip.ch10;
import java.awt.Graphics;
import org.opensourcephysics.display.*;
import org.opensourcephysics.numerics.*;

public class RadiatingCharge implements Drawable, Function {
   Circle circle = new Circle(0, 0, 5);
   double t = 0;                          // time
   double dt = 0.5;                       // time step
   // current number of points in storage
   int numPts = 0;
   double[][] path = new double[3][1024]; // storage for t,x,y
   double[] r = new double[2];
   double[] v = new double[2];
   double[] u = new double[2];
   double[] a = new double[2];
   // maximum velocity for charge in units where c = 1
   double vmax;

   public RadiatingCharge() {
      resetPath();
   }

   private void resizePath() {
      int length = path[0].length;
      if(length>32768) { // drop half the points
         System.arraycopy(path[0], length/2, path[0], 0, length/2);
         System.arraycopy(path[1], length/2, path[1], 0, length/2);
         System.arraycopy(path[2], length/2, path[2], 0, length/2);
         numPts = length/2;
         return;
      }
      double[][] newPath = new double[3][2*length]; // new path
      System.arraycopy(path[0], 0, newPath[0], 0, length);
      System.arraycopy(path[1], 0, newPath[1], 0, length);
      System.arraycopy(path[2], 0, newPath[2], 0, length);
      path = newPath;
   }

   void step() {
      t += dt;
      if(numPts>=path[0].length) {
         resizePath();
      }
      path[0][numPts] = t;
      path[1][numPts] = evaluate(t); // x position of charge
      path[2][numPts] = 0;
      numPts++;
   }

   void resetPath() {
      numPts = 0;
      t = 0;
      path = new double[3][1024];    // storage for t,x,y
      path[0][numPts] = t;
      path[1][numPts] = evaluate(t); // x position of charge
      path[2][numPts] = 0;
      numPts++; // initial position has been added
   }

   void electrostaticField(double x, double y, double[] field) {
      double dx = x-path[1][0];
      double dy = y-path[2][0];
      double r2 = dx*dx+dy*dy;
      double r3 = r2*Math.sqrt(r2);
      double ex = dx/r3;
      double ey = dy/r3;
      field[0] = ex;
      field[1] = ey;
      field[2] = 0; // magnetic field
   }

   double dsSquared(int i, double t, double x, double y) {
      double dt = t-path[0][i];
      double dx = x-path[1][i];
      double dy = y-path[2][i];
      return dx*dx+dy*dy-dt*dt;
   }

   void calculateRetardedField(double x, double y, double[] field) {
      int first = 0;
      int last = numPts-1;
      double ds_first = dsSquared(first, t, x, y);
      if(ds_first>=0) { // field has not yet propagated to the location
         electrostaticField(x, y, field);
         return;
      }
      while((ds_first<0)&&(last-first)>1) {
         int i = first+(last-first)/2; // bisect the interval
         double ds = dsSquared(i, t, x, y);
         if(ds<=0) {
            ds_first = ds;
            first = i;
         } else {
            last = i;
         }
      }
      double t_ret = path[0][first]; // time where ds changes sign
      r[0] = x-evaluate(t_ret);       // evaluate x at retarded time
      r[1] = y;                       // evaluate y at retarded time
      // derivative of x at retarded time
      v[0] = Derivative.centered(this, t_ret, dt);
      v[1] = 0;                       // derivative of y at retarded time
      // acceleration of x at retarded time
      a[0] = Derivative.second(this, t_ret, dt);
      a[1] = 0; // acceleration of y at retarded time
      double rMag = Vector2DMath.mag2D(r); // magnitdue of r
      u[0] = r[0]/rMag-v[0];
      u[1] = r[1]/rMag-v[1];
      double r_dot_u = Vector2DMath.dot2D(r, u);
      double k = rMag/r_dot_u/r_dot_u/r_dot_u;
      // u cross a is perpendicular to plane of motion
      double u_cross_a = Vector2DMath.cross2D(u, a);
      double[] temp = {r[0], r[1]};
      temp = Vector2DMath.crossZ(temp, u_cross_a); // r cross u
       // (c*c - v*v) where c = 1
      double c2v2 = 1-Vector2DMath.dot2D(v, v);
      double ex = k*(u[0]*c2v2+temp[0]);
      double ey = k*(u[1]*c2v2+temp[1]);
      field[0] = ex;
      field[1] = ey;
      field[2] = k*Vector2DMath.cross2D(temp, r)/rMag;
   }

   public void draw(DrawingPanel panel, Graphics g) {
      circle.setX(evaluate(t));
      // draw the charged particle on the screen
      circle.draw(panel, g);
   }

   public double evaluate(double t) {
      return 5*Math.cos(t*vmax/5.0);
   }
}
```

The `RadiatingCharge` class computes the electric field due to an oscillating charge using the Liénard–Wiechert potentials. We choose units such that the speed of light $c = 1$. As the charge moves, it stores its ith data point in a two-dimensional array `path[3][i]` containing the time, its x-position, and its y-position. To find the retarded time at the position (x, y), we use the `dsSquared` method to compute the square of the space-time interval between the given location and points along the path. The square of the space-time separation is defined as

$$\Delta s^2 = \Delta x^2 + \Delta y^2 - c^2 \Delta t^2 \tag{10.45}$$

where $\Delta x = x - x_{\text{path}}$, $\Delta y = y - y_{\text{path}}$, and $\Delta t = t - t_{\text{path}}$. The last point on the path contains the current position of the charge so Δs^2 must be positive because Δt is zero (unless the charge is at the observation point (x, y) in which case Δs^2 is zero and the field is infinite due to the $1/r^2$ dependence). The calcRetardedField method evaluates Δs^2 at the first point in the trajectory to determine if it is negative. We assume the charge was stationary for $t < 0$ and compute the electrostatic field if Δs^2 is positive at the trajectory's first point where $t = 0$. If Δs^2 is negative at the trajectory's first point, we repeatedly bisect the path into smaller and smaller segments while checking to see if Δs^2 remains negative at the beginning of the segment and positive at the end. In this way we can find the retarded time when we have a path segment bounded by two data points. Note that the RadiatingCharge class uses the Vector2DMath class to perform the necessary vector arithmetic. This helper class is not listed but is available in the ch10 code package.

The RadiatingEFieldApp program is shown in Listing 10.7. It displays the electric field in the x-y plane using a Vector2DFrame. The calculateFields method computes the retarded field at every grid point. The simulation's doStep method invokes this method after it moves the charge.

Listing 10.7: The RadiatingEFieldApp program computes the radiating electric and magnetic fields using Liénard–Wiechert potentials.

```java
package org.opensourcephysics.sip.ch10;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.frames.Vector2DFrame;

public class RadiatingEFieldApp extends AbstractSimulation {
   Vector2DFrame frame = new Vector2DFrame("x", "y", "Electric field");
   RadiatingCharge charge = new RadiatingCharge();
   int gridSize;  // linear dimension of grid used to compute fields
   double[][][] Exy; // x and y components of electric field
   double xmin = -20, xmax = 20, ymin = -20, ymax = 20;

   public RadiatingEFieldApp() {
      frame.setPreferredMinMax(xmin, xmax, ymin, ymax);
      frame.setZRange(false, 0, 0.2);
      frame.addDrawable(charge);
   }

   public void initialize() {
      gridSize = control.getInt("size");
      Exy = new double[2][gridSize][gridSize];
      // maximum speed of charge
      charge.vmax = control.getDouble("vmax");
      charge.dt = control.getDouble("dt");
      frame.setAll(Exy);
      initArrays();
   }

   private void initArrays() {
      charge.resetPath();
      calculateFields();
   }

   private void calculateFields() {
      double[] fields = new double[3]; // Ex, Ey, Bz
      for(int i = 0;i<gridSize;i++) {
         for(int j = 0;j<gridSize;j++) {
            // x location where we calculate the field
            double x = frame.indexToX(i);
            // y location where we calculate the field
            double y = frame.indexToY(j);
            // return the retarded time
            charge.calculateRetardedField(x, y, fields);
            Exy[0][i][j] = fields[0]; // Ex
            Exy[1][i][j] = fields[1]; // Ey
         }
      }
      frame.setAll(Exy);
   }

   public void reset() {
      control.setValue("size", 31);
      control.setValue("dt", 0.5);
      control.setValue("vmax", 0.9);
      initialize(); // initialize the model
   }

   protected void doStep() {
      charge.step();
      calculateFields();
   }

   public static void main(String[] args) {
      SimulationControl.createApp(new RadiatingEFieldApp());
   }
}
```

Problem 10.19. Field lines from an accelerating charge

(a) Read the code for `RadiatingEFieldApp` carefully to understand the correspondence between the program and the analytic results, (10.40) and (10.42), discussed in the text.

(b) Describe qualitatively the nature of the electric and magnetic fields from an oscillating point charge. How does the electric field differ from that of a static charge at the origin? What happens as the speed increases? The physics breaks down if the maximum speed is greater than c. Does the algorithm break down? Explain.

(c) Modify the program to show the magnetic field in the x-y plane using a `Scalar2DFrame` to show the B_z vector component.

(d) Modify the program to observe a charge moving with uniform circular motion about the origin. What happens as the speed of the charge approaches the speed of light? □

Problem 10.20. Spatial dependence of the radiating fields

(a) As waves propagate from an accelerating point source, the total power that passes through a spherical surface of radius R remains constant. Because the surface area is proportional to R^2, the power per unit area or intensity is proportional to $1/R^2$. Also, because the intensity is proportional to E^2, we expect that $E \propto 1/R$ far from the source. Modify the program to verify this result for a charge that is oscillating along the x-axis according to $x(t) = 0.2\cos t$. Plot $|E|$ as a function of the observation time t for a fixed position, such as $\mathbf{R} = (10, 10, 0)$. The field should oscillate in time. Find the amplitude of this oscillation. Next double the distance of the observation point from the origin. How does the amplitude depend on R?

(b) Repeat part (a) for several directions and distances. Generate a polar diagram showing the amplitude as a function of angle in the x-y plane. Is the radiation greatest along the line in which the charge oscillates? □

Problem 10.21. Fields from a charge moving at constant velocity

(a) Use RadiationApp to calculate $\mathbf{E}$ due to a charged particle moving at constant velocity toward the origin, for example, $x(t_{\rm ret}) = 1 - 2t_{\rm ret}$. Take a snapshot at $t = 0.5$ and compare the field lines with those you expect from a stationary charge.

(b) Modify RadiationApp so that $x(t_{\rm ret}) = 1 - 2t_{\rm ret}$ for $t_{\rm ret} < 0.5$ and $x(t_{\rm ret}) = 0$ for $t_{\rm ret} > 0.5$. Describe the field lines for $t > 0.5$. Does the particle accelerate at any time? Is there any radiation? □

Problem 10.22. Frequency dependence of an oscillating charge

(a) The radiated power at any point in space is proportional to E^2. Plot $|E|$ versus time at a fixed observation point (for example, $X = 10, Y = Z = 0$) and calculate the frequency dependence of the amplitude of $|E|$ due to a charge oscillating at the frequency ω. It is shown in standard textbooks that the power associated with radiation from an oscillating dipole is proportional to ω^4. How does the ω-dependence that you measured compare to that for dipole radiation? Repeat for a much bigger value of R and explain any differences.

(b) Repeat part (a) for a charge moving in a circle. Are there any qualitative differences? □

10.8 *Maxwell's Equations

In Section 10.7 we found that accelerating charges produce electric and magnetic fields that depend on position and time. We now investigate the direct relation between changes in $\mathbf{E}$ and $\mathbf{B}$ given by the differential form of Maxwell's equations:

$$\frac{\partial \mathbf{B}}{\partial t} = -\frac{1}{c}\nabla \times \mathbf{E} \tag{10.46}$$

$$\frac{\partial \mathbf{E}}{\partial t} = c\nabla \times \mathbf{B} - 4\pi \mathbf{j} \tag{10.47}$$

where $\mathbf{j}$ is the electric current density. We can regard (10.46) and (10.47) as the basis of electrodynamics. In addition to (10.46) and (10.47), we need the relation between $\mathbf{j}$ and the charge density ρ that expresses the conservation of charge:

$$\frac{\partial \rho}{\partial t} = -\nabla \cdot \mathbf{j}. \tag{10.48}$$

A complete description of electrodynamics requires (10.46), (10.47), and (10.48), and the initial values of all currents and fields.

For completeness, we obtain the Maxwell's equations that involve $\nabla \cdot \mathbf{B}$ and $\nabla \cdot \mathbf{E}$ by taking the divergence of (10.46) and (10.47), substituting (10.48) for $\nabla \cdot \mathbf{j}$, and then integrating over time. If the initial fields are zero, we obtain (using the relation $\nabla \cdot (\nabla \times \mathbf{a}) = 0$ for any vector $\mathbf{a}$)

$$\nabla \cdot \mathbf{E} = 4\pi\rho \tag{10.49}$$

$$\nabla \cdot \mathbf{B} = 0. \tag{10.50}$$

If we introduce the electric and magnetic potentials, it is possible to convert the first-order equations (10.46) and (10.47) to second-order differential equations. However, the familiar first-order equations are better suited for numerical analysis. To solve (10.46) and (10.47) numerically, we need to interpret the curl and divergence of a vector. As its name implies, the curl of a vector measures how much the vector twists around a point. A coordinate free definition of the curl of an arbitrary vector $\mathbf{W}$ is

$$(\nabla \times \mathbf{W}) \cdot \hat{\mathbf{S}} = \lim_{S\to 0} \frac{1}{S} \oint_C \mathbf{W} \cdot d\mathbf{l} \tag{10.51}$$

where $\mathbf{S}$ is the area of any surface bordered by the closed curve C, and $\hat{\mathbf{S}}$ is a unit vector normal to the surface S.

Equation (10.51) gives the component of $\nabla \times \mathbf{W}$ in the direction of $\hat{\mathbf{S}}$ and suggests a way of computing the curl numerically. We divide space into cubes of linear dimension Δl. The rectangular components of $\mathbf{W}$ can be defined either on the edges or on the faces of the cubes. We compute the curl using both definitions. We first consider a vector $\mathbf{B}$ that is defined on the edges of the cubes so that the curl of $\mathbf{B}$ is defined on the faces. (We use the notation $\mathbf{B}$ because we will find that it is convenient to define the magnetic field in this way.) Associated with each cube is one edge vector and one face vector. We label the cube by the coordinates corresponding to its lower left front corner; the three components of $\mathbf{B}$ associated with this cube are shown in Figure 10.6a. The other edges of the cube are associated with B vectors defined at neighboring cubes.

The discrete version of (10.51) for the component of $\nabla \times \mathbf{B}$ defined on the front face of the cube (i, j, k) is

$$(\nabla \times \mathbf{B}) \cdot \hat{\mathbf{S}} = \frac{1}{(\Delta l)^2} \sum_{i=1}^{4} B_i \Delta l_i \tag{10.52}$$

where $S = (\Delta l)^2$, and B_i and l_i are shown in Figures 10.6b and 10.6c, respectively. Note that two of the B_i are associated with neighboring cubes.

The components of a vector can also be defined on the faces of the cubes. We call this vector $\mathbf{E}$ because it will be convenient to define the electric field in this way. In Figure 10.7(a) we show the components of $\mathbf{E}$ associated with the cube (i, j, k). Because $\mathbf{E}$ is normal to a cube face, the components of $\nabla \times \mathbf{E}$ lie on the edges. The components E_i and l_i are shown in Figures 10.7(b) and 10.7(c), respectively. The form of the discrete version of $\nabla \times \mathbf{E}$ is similar to (10.52) with B_i replaced by E_i, where E_i and l_i are shown in Figures 10.7(b) and 10.7(c), respectively. The z-component of $\nabla \times \mathbf{E}$ is along the left edge of the front face.

A coordinate free definition of the divergence of the vector field $\mathbf{W}$ is

$$\nabla \cdot \mathbf{W} = \lim_{V\to 0} \frac{1}{V} \oint_S \mathbf{W} \cdot d\mathbf{S} \tag{10.53}$$

where V is the volume enclosed by the closed surface $\mathbf{S}$. The divergence measures the average flow of the vector through a closed surface. An example of the discrete version of (10.53) is given in (10.54).

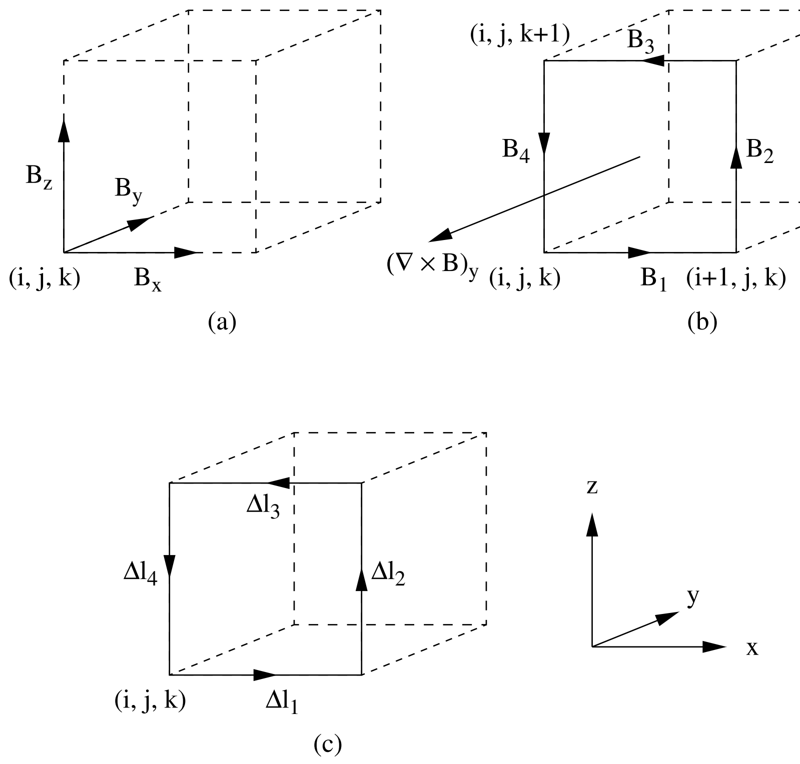


Figure 10.6: Calculation of the curl of **B** defined on the edges of a cube. (a) The edge vector **B** associated with cube (i,j,k). (b) The components B_i along the edges of the front face of the cube. $B_1 = B_x(i,j,k)$, $B_2 = B_z(i+1,j,k)$, $B_3 = -B_x(i,j,k+1)$, and $B_4 = -B_z(i,j,k)$. (c) The vector components $\mathbf{\Delta l}_i$ on the edges of the front face. (The y-component of $\nabla \times \mathbf{B}$ defined on the face points in the negative y direction.)

We now discuss where to define the quantities ρ, **j**, **E**, and **B** on the grid. It is natural to define the charge density ρ at the center of a cube. From the continuity equation (10.48), we see that this definition leads us to define **j** at the faces of the cube. Hence, each face of a cube has a number associated with it corresponding to the current density flowing parallel to the outward normal to that face. Given the definition of **j** on the grid, we see from (10.47) that the electric field **E** and **j** should be defined at the same places, and hence, we define the electric field on the faces of the cubes. Because **E** is defined on the faces, it is natural to define the magnetic field **B** on the edges of the cubes. Our definitions of the vectors **j**, **E**, and **B** on the grid are now complete.

We label the faces of cube c by the symbol f_c. If we use the simplest finite difference method with a discrete time step Δt and discrete spatial interval $\Delta x = \Delta y = \Delta z \equiv \Delta l$, we can write the continuity equation as

$$\left[\rho\Big(c, t + \frac{1}{2}\Delta\Big) - \rho\Big(c, t - \frac{1}{2}\Delta t\Big)\right] = -\frac{\Delta t}{\Delta l}\sum_{f_c=1}^{6} j(f_c, t). \tag{10.54}$$

The factor of $1/\Delta l$ comes from the area of a face $(\Delta l)^2$ used in the surface integral in (10.53) divided by the volume $(\Delta l)^3$ of a cube. In the same spirit, the discretization of (10.47) can be

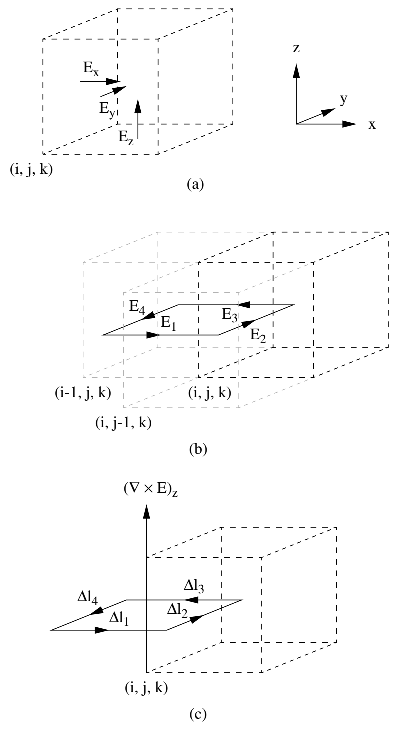


Figure 10.7: Calculation of the curl of the vector **E** defined on the faces of a cube. (a) The face vector **E** associated with the cube (i,j,k). The components associated with the left, front, and bottom faces are $E_x(i,j,k), E_y(i,j,k), E_z(i,j,k)$, respectively. (b) The components E_i on the faces that share the front left edge of the cube (i,j,k). $E_1 = E_x(i,j-1,k), E_2 = E_y(i,j,k), E_3 = -E_x(i,j,k)$, and $E_4 = -E_y(i-1,j,k)$. The cubes associated with E_1 and E_4 also are shown. (c) The vector components Δl_i on the faces that share the left front edge of the cube. (The z-component of the curl of **E** defined on the left edge points in the positive z direction.)

written as

$$E(f,t+\frac{1}{2}\Delta t) - E\Big(f,t-\frac{1}{2}\Delta t\Big) = \Delta t\Big[\boldsymbol{\nabla}\times\mathbf{B} - 4\pi j(f,t)\Big]. \tag{10.55}$$

Note that **E** in (10.55) and ρ in (10.54) are defined at different times than **j**. As usual, we choose units such that $c = 1$.

We next need to define a square around which we can discretize the curl. If **E** is defined on the faces, it is natural to use the square that is the border of the faces. As we have discussed, this choice implies that we should define the magnetic field on the edges of the cubes. We write (10.55) as:

$$E(f,t+\frac{1}{2}\Delta t) - E(f,t-\frac{1}{2}\Delta t) = \Delta t\Big[\frac{1}{\Delta l}\sum_{e_f=1}^{4} B(e_f,t) - 4\pi j(f,t)\Big] \tag{10.56}$$

where the sum is over e_f, the four edges of the face f (see Figure 10.7b). Note that B is defined at the same time as j. In a similar way we can write the discrete form of (10.46) as

$$B(e, t+\Delta t) - B(e,t) = -\frac{\Delta t}{\Delta l}\sum_{f_e=1}^{4} E\Big(f_e, t+\frac{1}{2}\Delta t\Big) \tag{10.57}$$

where the sum is over f_e, the four faces that share the same edge e (see Figure 10.7b).

We now have a well-defined algorithm for computing the spatial dependence of the electric and magnetic field, the charge density, and the current density as a function of time. This algorithm was developed by Yee, an electrical engineer, in 1966, and independently by Visscher, a physicist, in 1988 who also showed that all of the integral relations and other theorems that are satisfied by the continuum fields are also satisfied for the discrete fields.

Usually, the most difficult part of this algorithm is specifying the initial conditions because we cannot simply place a charge somewhere. The reason is that the initial fields appropriate for this charge would not be present. Indeed, our rules for updating the fields and the charge densities reflect the fact that the electric and magnetic fields do not appear instantaneously at all positions in space when a charge appears, but instead evolve from the initial appearance of a charge. Of course, charges do not appear out of nowhere, but appear by disassociating from neutral objects. Conceptually, the simplest initial condition corresponds to two charges of opposite sign moving oppositely to each other. This condition corresponds to an initial current on one face. From this current a charge density and thus an electric field appears using (10.54) and (10.56), respectively, and a magnetic field appears using (10.57).

Because we cannot compute the fields for an infinite lattice, we need to specify the boundary conditions. The easiest method is to use fixed boundary conditions such that the fields vanish at the edges of the lattice. If the lattice is sufficiently large, fixed boundary conditions are a reasonable approximation. However, fixed boundary conditions usually lead to nonphysical reflections off the edges, and a variety of approaches have been used including boundary conditions equivalent to a conducting medium that gradually absorbs the fields. In some cases physically motivated boundary conditions can be employed. For example, in simulations of microwave cavity resonators (see Problem 10.24), the appropriate boundary conditions are that the tangential component of **E** and the normal component of **B** vanish at the boundary.

As we have noted, **E** and ρ are defined at different times than **B** and **j**. This half-step approach leads to well behaved equations that are stable over a range of parameters. An analysis of the stability requirement for the Yee-Visscher algorithm shows that the time step Δt must be smaller than the spatial grid Δl by:

$$c\Delta t \leq \frac{\Delta l}{\sqrt{3}} \qquad \text{(stability requirement)}. \tag{10.58}$$

The `Maxwell` class implements the Visscher-Yee finite difference algorithm for solving Maxwell's equations. The field and current data are stored in multi-dimensional arrays `E`, `B`, and `J`. The first index determines the vector component. The last three indices represent the three spatial coordinates. The `current` method models a positive current flowing for one time unit. This current flow produces both electric and magnetic fields. Because charge is conserved, the current flow produces an electrostatic dipole. Negative charge remains at the source and a positive charge is deposited at the destination. Note that the `doStep` method invokes a `damping` method that reduces the fields at points near the boundaries, thereby absorbing the emitted radiation and reducing the reflected electromagnetic waves. Your understanding of the Yee-Visscher algorithm for finding solutions to Maxwell's equations will be enhanced by carefully reading the `MaxwellApp` program and the `Maxell` class.

Listing 10.8: The `Maxwell` class implements the Yee-Visscher finite difference approximation to Maxwell's equations.

```
package org.opensourcephysics.sip.ch10;

public class Maxwell {
   // static variables determine units and time scale
   static final double pi4 = 4*Math.PI;
   static final double dt = 0.03;
   static final double dl = 0.1;
   static final double escale = dl/(4*Math.PI*dt);
   static final double bscale = escale*dl/dt;
   static final double jscale = 1;
   double dampingCoef = 0.1; // damping coefficient near boundaries
   int size;
   double t;                // time
   double[][][][] E, B, J;

   public Maxwell(int size) {
      this.size = size;
      // 3D arrays for electric field, magnetic field, and current
      // last three indices indicate location, first index indicates
      // x, y, or z component
      E = new double[3][size][size][size];
      B = new double[3][size][size][size];
      J = new double[3][size][size][size];
   }

   public void doStep() {
      current(t); // update the current
      computeE(); // step electric field
      computeB(); // step magnetic field
      damping();  // damp transients
      t += dt;
   }

   void current(double t) {
      final int mid = size/2;
      double delta = 1.0;
      for(int i = -3;i<5;i++) {
         J[0][mid+i][mid][mid] = (t<delta) ? +1 : 0;
      }
   }

   void computeE() {
      for(int x = 1;x<size-1;x++) {
         for(int y = 1;y<size-1;y++) {
            for(int z = 1;z<size-1;z++) {
               double curlBx = (B[1][x][y][z]-B[1][x][y][z+1]
                               +B[2][x][y+1][z]-B[2][x][y][z])/dl;
               E[0][x][y][z] += dt*(curlBx-pi4*J[0][x][y][z]);
               double curlBy = (B[2][x][y][z]-B[2][x+1][y][z]
                               +B[0][x][y][z+1]-B[0][x][y][z])/dl;
               E[1][x][y][z] += dt*(curlBy-pi4*J[1][x][y][z]);
```

```
                double curlBz = (B[0][x][y][z]-B[0][x][y+1][z]
                                 +B[1][x+1][y][z]-B[1][x][y][z])/dl;
                E[2][x][y][z] += dt*(curlBz-pi4*J[2][x][y][z]);
            }
        }
    }
}

void computeB() {
    for(int x = 1;x<size-1;x++) {
        for(int y = 1;y<size-1;y++) {
            for(int z = 1;z<size-1;z++) {
                double curlEx = (E[2][x][y][z]-E[2][x][y-1][z]
                                 +E[1][x][y][z-1]-E[1][x][y][z])/dl;
                B[0][x][y][z] -= dt*curlEx;
                double curlEy = (E[0][x][y][z]-E[0][x][y][z-1]
                                 +E[2][x-1][y][z]-E[2][x][y][z])/dl;
                B[1][x][y][z] -= dt*curlEy;
                double curlEz = (E[1][x][y][z]-E[1][x-1][y][z]
                                 +E[0][x][y-1][z]-E[0][x][y][z])/dl;
                B[2][x][y][z] -= dt*curlEz;
            }
        }
    }
}

void damping() {
    for(int i = 0;i<size;i++) {
        for(int j = 0;j<size;j++) {
            // w used to index cell near boundary subject to damping
            for(int w = 0;w<4;w++) {
                for(int comp = 0;comp<3;comp++) {
                    E[comp][w][i][j] -= dampingCoef*E[comp][w][i][j];
                    E[comp][size-w-1][i][j] -= dampingCoef
                            *E[comp][size-w-1][i][j];
                    E[comp][i][w][j] -= dampingCoef*E[comp][i][w][j];
                    E[comp][i][size-w-1][j] -= dampingCoef
                            *E[comp][i][size-w-1][j];
                    E[comp][i][j][w] -= dampingCoef*E[comp][i][j][w];
                    E[comp][i][j][size-w-1] -= dampingCoef
                            *E[comp][i][j][size-w-1];
                    B[comp][w][i][j] -= dampingCoef*B[comp][w][i][j];
                    B[comp][size-w-1][i][j] -= dampingCoef
                            *B[comp][size-w-1][i][j];
                    B[comp][i][w][j] -= dampingCoef*B[comp][i][w][j];
                    B[comp][i][size-w-1][j] -= dampingCoef
                            *B[comp][i][size-w-1][j];
                    B[comp][i][j][w] -= dampingCoef*B[comp][i][j][w];
                    B[comp][i][j][size-w-1] -= dampingCoef
                            *B[comp][i][j][size-w-1];
                }
            }
        }
```

```
         }
      }
   }
```

Listing 10.9: The `MaxwellApp` program computes and displays the electric field by solving Maxwell's equations.

```
package org.opensourcephysics.sip.ch10;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.frames.Vector2DFrame;

public class MaxwellApp extends AbstractSimulation {
   Vector2DFrame frame = new Vector2DFrame("x", "y",
         "EField in XY Plane");
   int size;
   Maxwell maxwell;
   // x and y components of E for middle plane in z direction
   double[][][] Exy;

   public MaxwellApp() {
      frame.setZRange(false, 0, 1.0);
   }

   public void reset() {
      control.setValue("size", 31);
      control.setValue("dt", 0.5);
   }

   public void initialize() {
      size = control.getInt("size");
      Exy = new double[2][size][size];
      maxwell = new Maxwell(size);
      frame.setAll(Exy);
      frame.setPreferredMinMax(0, Maxwell.dl*size, Maxwell.dl*size, 0);
      plotField();
   }

   protected void doStep() {
      maxwell.doStep();
      plotField();
      frame.setMessage("t="+decimalFormat.format(maxwell.t));
   }

   void plotField() {
      double[][][][] E = maxwell.E; // electric field
      int mid = size/2;
      for(int i = 0;i<size;i++) {
         for(int j = 0;j<size;j++) {
            Exy[0][i][j] = E[0][i][j][mid];  // Ex
            Exy[1][i][j] = E[1][i][j][mid];  // Ey
         }
      }
      frame.setAll(Exy);
   }
   public static void main(String[] args) {
      SimulationControl.createApp(new MaxwellApp());
   }
}
```

The MaxwellApp program shows the electric field in the x-y plane. The x-y components of the electric field are represented by arrows whose length is fixed and whose color indicates the field magnitude at each position where the field is defined.

Problem 10.23. Fields from a current loop

(a) A steady current in the middle of the x-y plane is turned on at $t = 0$ and left on for one time unit. Before running the program, predict what you expect to see. Compare your expectations with the results of the simulation. Use $\Delta t = 0.03$, $\Delta l = 0.1$, and take the number of cubes in each direction to be `n(1) = n(2) = n(3) = 8`.

(b) Add a plot of the magnetic field. Where should the viewing plane be placed to produce the best visualization? How should the plane be oriented? Predict what you expect to see before you run the simulation.

(c) Verify the stability requirement (10.58), by running your program with $\Delta t = 0.1$ and $\Delta l = 0.1$. Then try $\Delta t = 0.05$ and $\Delta l = \Delta t\sqrt{3}$. What happens to the results in part (a) if the stability requirement is not satisfied?

(d) Modify the current density in part (a) so that $\mathbf{j}$ oscillates sinusoidally. What happens to the electric and magnetic field vectors?

(e) How much must you change the factor dampingCoef in the damping method before you can visually see a difference in the simulation? What problems occur when the damping is removed?

f)* The amplitude of the fields far from the current loop should be characteristic of radiation fields for which the amplitude falls off as $1/r$, where r is the distance from the current loop to the observation point. Do a simulation to detect this dependence if you have sufficient computer resources. □

Problem 10.24. Microwave cavity resonators

(a) Cavity resonators are a practical way of storing energy in the form of oscillating electric and magnetic fields without losing as much energy as would be dissipated in a resonant LC circuit. Consider a cubical resonator of linear dimension L whose walls are made of a perfectly conducting material. The tangential components of $\mathbf{E}$ and the normal component of $\mathbf{B}$ vanish at the walls. Standing microwaves can be set up in the box of the form (cf. Reitz et al.)

$$E_x = E_{x0} \cos k_x x \sin k_y y \sin k_z z\, e^{i\omega t} \tag{10.59a}$$

$$E_y = E_{y0} \cos k_y y \sin k_x x \sin k_z z\, e^{i\omega t} \tag{10.59b}$$

$$E_z = E_{z0} \cos k_z z \sin k_x x \sin k_y y\, e^{i\omega t}. \tag{10.59c}$$

The wave vector $\mathbf{k} = (k_x, k_y, k_z) = (m_x\pi/L, m_y\pi/L, m_z\pi/L)$, where m_x, m_y, and m_z are integers. A particular mode is labeled by the integers (m_x, m_y, m_z). The initial electric field is perpendicular to $\mathbf{k}$, and $\omega = ck$. Implement the boundary conditions at $(x = 0, y = 0, z = 0)$ and $(x = L, y = L, z = L)$. Set $\Delta t = 0.05$, $\Delta l = 0.1$, and $L = 1$. At $t = 0$, set $\mathbf{B} = 0$, $\mathbf{j} = 0$ (there are no currents within the cavity), and use (10.59) with $(m_x, m_y, m_z) = (0, 1, 1)$ and $E_{x0} = 1$. Plot the field components at specific positions as a function of t and find the resonant frequency ω. Compare your computed value of ω with the analytic result. Do the magnetic fields change with time? Are they perpendicular to $\mathbf{k}$ and $\mathbf{E}$?

(b) Repeat part (a) for two other modes.

(c) Repeat part (a) with a uniform random noise added to the initial field at all positions. Assume the amplitude of the noise is δ and describe the resulting fields for $\delta = 0.1$. Are they similar to those without noise? What happens for $\delta = 0.5$? More quantitative results can be found by computing the power spectrum $|E(\omega)|^2$ for the electric field at a few positions. What is the order of magnitude of δ for which the maximum of $|E(\omega)|^2$ at the standing wave frequency is swamped by the noise?

(d) Change the shape of the container slightly by removing a 0.1×0.1 cubical box from each of the corners of the original resonator. Do the standing wave frequencies change? Determine the standing wave frequency by adding noise to the initial fields and looking at the power spectrum. How do the standing wave patterns change?

(e) Change the shape of the container slightly by adding a 0.1×0.1 cubical box at the center of one of the faces of the original resonator. Do the standing wave frequencies change? How do the standing wave patterns change?

(f) Cut a 0.2×0.2 square hole in a face in the y-z plane and double the computational region in the x direction. Begin with a $(0, 1, 1)$ standing wave and observe how the fields "leak" out of the hole. □

Problem 10.25. Billiard microwave cavity resonators

(a) Repeat Problem 10.24a for $L_x = L_y = 2$, $L_z = 0.2$, $\Delta l = 0.1$, and $\Delta t = 0.05$. Indicate the magnitude of the electric field in the $L_z = 0.1$ plane by a color code. Choose an initial normal mode field distribution and describe the pattern that you obtain. Then repeat your calculation for a random initial field distribution.

(b) Place an approximately circular conductor in the middle of the cavity of radius $r = 0.4$. Describe the patterns that you see. Such a geometry leads to chaotic trajectories for particles moving within such a cavity (see Project 6.26). Is there any evidence of chaotic behavior in the field pattern?

(c) Repeat part (b) with the circular conductor placed off center. □

10.9 Projects

Part of the difficulty in understanding electromagnetic phenomena is visualizing its three-dimensional nature. Many interesting problems can be posed based on the simple, but nontrivial question of how three-dimensional electromagnetic fields can be best represented visually in various contexts (cf. Belcher and Olbert). However, we have not suggested projects in this area because of their difficulty.

Many of the techniques used in this chapter, for example, the random walk method and the relaxation method for solving Laplace's equation, have applications in other fields, especially problems in fluid flow and transport. Similarly, the multigrid method, discussed in Project 10.26, has far reaching applications.

Project 10.26. Multigrid method

In general, the relaxation method for solving Laplace's equation is very slow even when using overrelaxation. The reason is that the local updates of the relaxation method cannot quickly take into account effects at very large length scales. The *multigrid method* greatly improves performance by using relaxation at many length scales. The important idea is to use a relaxation method to find the values of the potential on coarser and coarser grids, and then to use the coarse grid values to determine the fine grid values. The fine grid relaxation updates take into account effects at short length scales. If we define the initial grid by a lattice spacing $b = 1$, then the coarser grids are characterized by $b = 2^n$, where n determines the coarseness of the grid and is known as the grid level. We need to decide how to use the fine grid values of the potential to assign values to a coarser grid, and then how to use a coarse grid to assign values to a finer grid. The first step is called *restriction* and the second step is called *prolongation*. There is some flexibility on how to do these two operations. We discuss one approach.

We define the centers of the sites of the coarse grid to be located at the centers of every other site of the fine grid. That is, if the set $\{i, j\}$ represents the positions of the sites of the fine grid, then $\{2i, 2j\}$ represents the positions of the coarse grid sites. The fine grid sites that are at the same position as a coarse grid point are assigned the value of the potential of the corresponding coarse grid point. The fine grid sites that have two coarse grid points as nearest neighbors are assigned the average value of these two coarse grid sites. The other fine grid sites have four coarse grid sites as next nearest neighbors and are assigned the average value of these four coarse grid sites. This prescription specifies how values on the fine grid are computed using the values on the coarse grid.

In the full weighting prolongation method, each coarse grid site receives one fourth of the potential of the fine grid site at the same position, one eighth of the potential for the four nearest neighbor sites of the fine grid, and one sixteenth of the potential for the four next nearest neighbor points of the fine grid. The sum of these fractions, $1/4 + 4(1/8) + 4(1/16)$, adds up to unity. An alternative procedure, known as half weighting, ignores the next nearest neighbors and uses one half of the potential of the fine grid site at the same position as the coarse grid site.

(a) Write a program that implements the multigrid method using Gauss–Seidel relaxation on a checkerboard lattice (see Problem 10.11b). In its simplest form the program should allow the user to intervene and decide whether to go to a finer or coarser grid, or to remain at the same level for the next relaxation step. Have the program print the potential at each site of the current level after each relaxation step. Test your program on a 4×4 grid whose boundary sites are all equal to unity, and whose initial internal sites are set to zero. Make sure that the boundary sites of the coarser grids are also set to unity.

(b) The exact solution for part (a) gives a potential of unity at each point. How many relaxation steps does it take to reach unity within 0.1% at every site by simply using the 4×4 grid? How many steps does it take if you use one coarse grid and continue until the coarse grid values are within 0.1% of unity? Is it necessary to carry out any fine grid relaxation steps to reach the desired accuracy on the fine grid? Next start with the coarsest scale, which is just one site. How many relaxation steps does it take now?

(c) Repeat part (b) but change the boundary so that one side of the boundary is held at a potential of 0.5. Experiment with different sequences of prolongation, restriction, and relaxation.

(d) Assume that the boundary points alternate between zero and unity and repeat part (b). Does the multigrid method work? Should one go up and down in levels many times instead of staying at the coarsest level and then going down to the finest level? □

Appendix A: Vector Fields

The frames package contains the `Vector2DFrame` class for displaying two-dimensional vector fields. To use this class we instantiate a multi-dimensional array to store components of the vector. The first array index indicates the component, the second index indicates the column or x-position, and the third index indicates the row or y-position. The vectors in the visualization are set by passing the data array to the frame using the `setAll` method. The program in Listing 10.10 demonstrates how this is done by displaying the electric field of a unit charge located at the origin.

Listing 10.10: A vector field test program.

```
package org.opensourcephysics.sip.ch10;
import javax.swing.JFrame;
import org.opensourcephysics.frames.Vector2DFrame;

public class VectorPlotApp {
   public static void main(String[] args) {
      Vector2DFrame frame =
           new Vector2DFrame("x", "y", "Vector field");
      double a = 2; // half width of frame in world coordinates
      frame.setPreferredMinMax(-a, a, -a, a);
      int nx = 15, ny = 15; // grid sizes in x and y direction
      // generate sample data
      double[][][] vectorField = new double[2][nx][ny];
      frame.setAll(vectorField); // vector field displays zero data
      for(int i = 0;i<nx;i++) {
         double x = frame.indexToX(i);
         for(int j = 0;j<ny;j++) {
            double y = frame.indexToY(j);
            double r2 = x*x+y*y;               // distance squared
            double r3 = Math.sqrt(r2)*r2;      // distance cubed
            vectorField[0][i][j] = (r2==0) ? 0 : x/r3; // x component
            vectorField[1][i][j] = (r2==0) ? 0 : y/r3; // y component
         }
      }
      frame.setAll(vectorField); // vector field displays new data
      frame.setVisible(true);
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
   }
}
```

The arrows in the visualization have a fixed length that is chosen to fill the viewing area. The arrow's color represents the field's magnitude. We have found that using an arrow's color rather than its length to represent field strength produces a more effective representation of vector fields over a wider dynamic range. The frame's Legend menu item under Tools shows this mapping. The appropriate representation of vector fields is an active area of interest.

Problem 10.27. Gradient of a scalar field

The gradient of a scalar field, $A(x,y)$, defines a vector field. In a two-dimensional Cartesian coordinate system, the components of the gradient are equal to the derivative of the scalar field along the x- and y-axes, respectively

$$\nabla A = \frac{\partial A}{\partial x}\hat{\mathbf{x}} + \frac{\partial A}{\partial y}\hat{\mathbf{y}}. \tag{10.60}$$

Write a short program that displays both a scalar field and its gradient. (Hint: Define a function and use numerical derivatives along the rows and columns.) Create separate frames for the scalar and vector field visualizations. The *Open Source Physics: A User's Guide with Examples* manual describes how a vector field visualization can be superimposed on a scalar field visualization. □

References and Suggestions for Further Reading

Forman S. Acton, *Numerical Methods That Work* (Harper & Row, 1970); corrected edition (Mathematical Association of America, 1990). Chapter 18 discusses solutions to Laplace's equation using the relaxation method and alternative approaches.

John W. Belcher and Stanislaw Olbert, "Field line motion in classical electromagnetism," Am. J. Phys. **71**, 220–228 (2003). The authors discuss some of the difficulties with field lines that change in time, and suggest a procedure where the local direction of motion of a field line is in the same direction as the Poynting vector.

Charles K. Birdsall and A. Bruce Langdon, *Plasma Physics via Computer Simulation* (McGraw–Hill, 1985).

D. H. Choi and W. J. R. Hoefer, "The finite-difference-time-domain method and its application to eigenvalue problems," IEEE Trans. Microwave Theory and Techniques **34**, 1464–1469 (1986). The authors use Yee's algorithm to model microwave cavity resonators.

David M. Cook, *The Theory of the Electromagnetic Field* (Prentice Hall, 1975). One of the first books to introduce numerical methods in the context of electromagnetism.

Robert Ehrlich, Jaroslaw Tuszynski, Lyle Roelofs, and Ronald Stoner, *Electricity and Magnetism Simulations: The Consortium for Upper-Level Physics Software* (John Wiley & Sons, 1995).

Richard P. Feynman, Robert B. Leighton, and Matthew Sands, *The Feynman Lectures on Physics*, Vol. 2 (Addison–Wesley, 1963).

T. E. Freeman, "One-, two- or three-dimensional fields?," Am. J. Phys. **63**, 273–274 (1995).

R. L. Gibbs, Charles W. Beason, and James D. Beason, "Solutions to boundary value problems of the potential type by random walk method," Am. J. Phys. **43**, 782–785 (1975).

R. H. Good, "Dipole radiation: Simulation using a microcomputer," Am. J. Phys. **52**, 1150–1151 (1984). The author discusses a graphical simulation of dipole radiation.

David J. Griffiths, *Introduction to Electrodynamics*, 3rd ed. (Prentice Hall, 1999). A classic undergraduate text on electromagnetism. See also David J. Griffiths and Daniel Z. Uvanovic, "The charge distribution on a conductor for non-coulombic potentials," Am. J. Phys. **69**, 435–440 (2001), O. F. de Alcantara Bonfim and David Griffiths, "Comment on 'Charge density on a thin straight wire, revisited,' by J. D. Jackson [Am. J. Phys. **68** (9), 789-Ð799 (2000)]," Am. J. Phys. **69**, 515–516 (2001); and O. F. de Alcantara Bonfim, David J. Griffiths, and Sasha Hinkley, "Chaotic and hyperchaotic motion of a charged particle in a magnetic dipole field," Int. J. Bifurcation and Chaos**10**, 265–271 (2000).

R. W. Hockney and J. W. Eastwood, *Computer Simulation Using Particles* (McGraw–Hill, 1981).

Steven E. Koonin and Dawn C. Meredith, *Computational Physics* (Addison–Wesley, 1990). See Chapter 6 for a discussion of the numerical solution of elliptic partial differential equations of which Laplace's and Poisson's equations are examples.

William M. MacDonald, "Discretization and truncation errors in a numerical solution of Laplace's equation," Am. J. Phys. **62**, 169–173 (1994).

William H. Press and Saul A. Teukolsky, "Multigrid methods for boundary value problems I," Computers in Physics **5** (5), 514 (1991).

Edward M. Purcell, *Electricity and Magnetism*, 2nd ed., Berkeley Physics Course, Vol. 2 (McGraw-Hill, 1985). A well known text that discusses the relaxation method.

John R. Reitz, Frederick J. Milford, and Robert W. Christy, *Foundations of Electromagnetic Theory*, 3rd ed. (Addison–Wesley, 1979). This text discusses microwave cavity resonators.

Matthew N. O. Sadiku, *Numerical Techniques in Electromagnetics*, 2nd ed. (CRC Press, 2001).

A. Taflove and M. E. Brodwin, "Numerical solution of steady state electromagnetic scattering problems using the time dependent Maxwell equations," IEEE Trans. Microwave Theory and Techniques **23**, 623–630 (1975). The authors derive the stability conditions for the Yee algorithm.

P. B. Visscher, *Fields and Electrodynamics* (John Wiley & Sons, 1988). An intermediate level text that incorporates computer simulations and analysis into its development.

P. J. Walker and I. D. Johnston, "Computer model clarifies spontaneous charge distribution in conductors," Computers in Physics **9**, 42 (1995).

Gregg Williams, "An introduction to relaxation methods," Byte **12** (1), 111–124 (1987). The author discusses the application of relaxation methods to the solution of the two-dimensional Poisson's equation.

K. S. Yee, "Numerical solution of initial boundary value problems involving Maxwell's equations in isotropic media," IEEE Trans. Antennas and Propagation **14**, 302–307 (1966). Yee uses the discretized Maxwell's equations to model the scattering of electromagnetic waves off a perfectly conducting rectangular obstacle.

Chapter 11

Numerical and Monte Carlo Methods

Simple classical and Monte Carlo methods including importance sampling are illustrated in the context of the numerical evaluation of definite integrals.

11.1 Numerical Integration Methods in One Dimension

In this chapter we will learn that we can use sequences of random numbers to estimate definite integrals, a problem that seemingly has nothing to do with randomness. To place the Monte Carlo numerical integration methods in perspective, we will first discuss several common classical methods for numerically evaluating definite integrals. We will find that these methods, although usually preferable in low dimensions, are impractical for multidimensional integrals and that Monte Carlo methods are essential for the evaluation of the latter if the number of dimensions is sufficiently high.

Consider the one-dimensional definite integral of the form

$$F = \int_a^b f(x)\,dx. \tag{11.1}$$

For some choices of the integrand $f(x)$, the integration in (11.1) can be done analytically, found in tables of integrals, or evaluated as a series. However, there are relatively few functions that can be evaluated analytically and most functions must be integrated numerically.

Most classical methods of numerical integration are based on the geometrical interpretation of the integral (11.1) as the area under the curve of the function $f(x)$ from $x = a$ to $x = b$ (see Figure 11.1). In these methods the x-axis is divided into n equal intervals of width Δx, where Δx is given by

$$\Delta x = \frac{b-a}{n} \tag{11.2a}$$

and

$$x_n = x_0 + n\,\Delta x. \tag{11.2b}$$

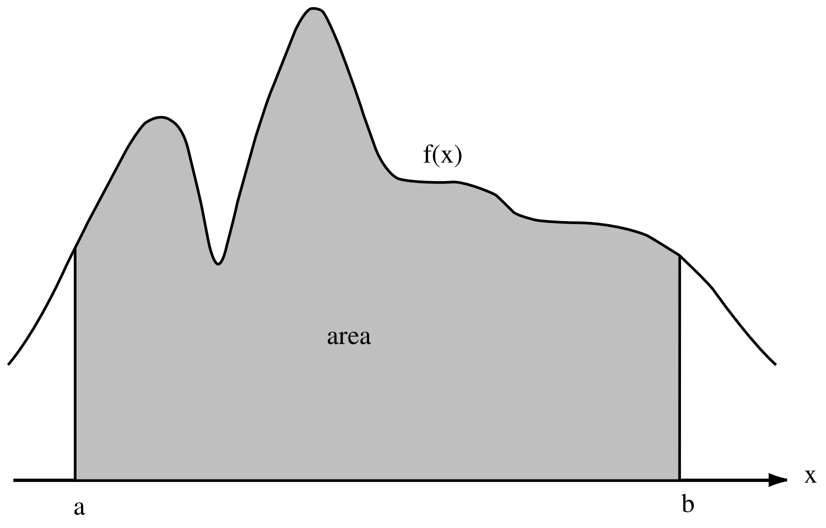


Figure 11.1: The integral F equals the area under the curve $f(x)$.

In the above, $x_0 = a$ and $x_n = b$.

The simplest approximation of the area under the curve $f(x)$ is the sum of the area of the rectangles shown in Figure 11.2. In the *rectangular* approximation, $f(x)$ is evaluated at the *beginning* of the interval, and the approximate of the integral F_n is given by

$$F_n = \sum_{i=0}^{n-1} f(x_i)\Delta x \qquad \text{(rectangular approximation).} \tag{11.3}$$

In the *trapezoidal* approximation, the integral is approximated by a sum of trapezoids. The area is computed by choosing one side equal to $f(x)$ at the beginning of the interval and the other side equal to $f(x)$ at the end of the interval. This approximation is equivalent to replacing the function by a straight line connecting the values of $f(x)$ at the beginning and the end of each interval. Because the area of the trapezoid from x_i to x_{i+1} is given by $\frac{1}{2}[f(x_{i+1}) + f(x_i)]\Delta x$, the total area F_n of the trapezoids is given by

$$F_n = \left[\frac{1}{2}f(x_0) + \sum_{i=1}^{n-1} f(x_i) + \frac{1}{2}f(x_n)\right]\Delta x \quad \text{(trapezoidal approximation).} \tag{11.4}$$

A generally more accurate method is to use a quadratic or parabolic interpolation procedure through adjacent triplets of points. (The general problem of interpolation between data points using polynomials is discussed in Appendix 11D.) For example, the equation of the second-order polynomial that passes through the points (x_0, y_0), (x_1, y_1), and (x_2, y_2) can be written as

$$\begin{aligned} y(x) = {} & \frac{(x - x_1)(x - x_2)}{(x_0 - x_1)(x_0 - x_2)}y_0 + \frac{(x - x_0)(x - x_2)}{(x_1 - x_0)(x_1 - x_2)}y_1 \\ & + \frac{(x - x_0)(x - x_1)}{(x_2 - x_0)(x_2 - x_1)}y_2. \end{aligned} \tag{11.5}$$

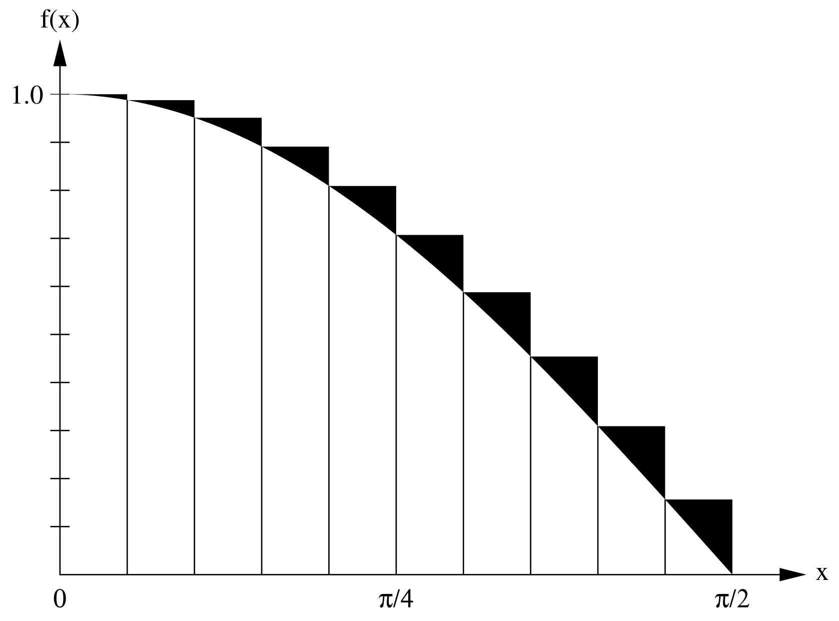


Figure 11.2: The rectangular approximation for $f(x) = \cos x$ for $0 \le x \le \pi/2$. The error is shaded. The error for various values of the number of intervals n is given in Table 11.1.

What is the value of $y(x)$ at $x = x_1$? The area under the parabola $y(x)$ between x_0 and x_2 can be found by integration and is given by

$$F_0 = \frac{1}{3}\Big(y_0 + 4y_1 + y_2\Big)\Delta x \tag{11.6}$$

where $\Delta x = x_1 - x_0 = x_2 - x_1$. The total area under all the parabolic segments yields the parabolic approximation for the total area:

$$\begin{aligned} F_n = {} & \frac{1}{3}[f(x_0) + 4f(x_1) + 2f(x_2) + 4f(x_3) + \cdots \\ & + 2f(x_{n-2}) + 4f(x_{n-1}) + f(x_n)]\Delta x \qquad \text{(Simpson's rule).} \end{aligned} \tag{11.7}$$

This approximation is known as *Simpson's rule*, although a more descriptive name would be the *parabolic approximation*. Note that Simpson's rule requires that n be even.

To write a program that implements the rectangular approximation, we must define the function we wish to integrate. Although we could define a new integration class for each function (or change the function in the class and recompile each time), it is convenient to input the name of the function as a string and then *parse* the string so that the function can be evaluated. The `ParsedFunction` class in the numerics package is designed for this task.

```
String str = "cos(x)"; // default string; this string could be an input
Function f;
try {
   f = new ParsedFunction(str);
   } catch (ParserException ex) {
    // recover if str does not represent a valid function
   }
```

Because the ParsedFunction is often used with keyboard input and it is common for users to mistype the name of a function, the ParsedFunction constructor throws an exception that must be caught.

One way to display a function in a drawing panel is to evaluate the function $f(x)$ at various x values and plot the $(x, f(x))$ data points. Although we could do so using a loop to add a predetermined number of points to a data set, a better way is to use the FunctionDrawer class in the display package. The FunctionDrawer evaluates a given function at every pixel location within a drawing panel thereby producing a plot with optimum resolution.

```
// drawingPanel created previously
drawingPanel.addDrawable(new FunctionDrawer(f));
```

We next define the class RectangularApproximation which computes the area under the curve using the rectangular approximation. This class also displays the rectangles used to compute the area. Note how we have extended the Dataset class to produce the visualization.

Listing 11.1: The class RectangularApproximation illustrates the nature of the rectangular approximation.

```
package org.opensourcephysics.sip.ch11;
import org.opensourcephysics.display.Dataset;
import org.opensourcephysics.numerics.Function;

public class RectangularApproximation extends Dataset {
   double sum = 0; // the inegral

   public RectangularApproximation(Function f, double a, double b,
       int num) {
       // transparent red
      setMarkerColor(new java.awt.Color(255, 0, 0, 128));
      setMarkerShape(Dataset.AREA);
      sum = 0;
      double x = a; // lower limit
      double y = f.evaluate(a);
      double dx = (b-a)/num;
      // ude methods in Dataset superclass
      append(x, 0); // start on the x axis
      append(x, y); // the top left hand corner of the first rectangle
      while(x<b) {         // b is the upper limit
         x += dx;
         append(x, y);    // top right hand corner of current rectangle
         sum += y;
         y = f.evaluate(x); // calculate a new y at the new x
         // top left hand corner of the next rectangle
         append(x, y);
      }
      append(x, 0); // finish on the x axis
      sum *= dx;
   }
}
```

Listing 11.2: The target class is defined in the NumericalIntegrationApp class.

```
package org.opensourcephysics.sip.ch11;
```

```java
import org.opensourcephysics.controls.*;
import org.opensourcephysics.display.*;
import org.opensourcephysics.frames.*;
import org.opensourcephysics.numerics.*;

public class NumericalIntegrationApp extends AbstractCalculation {
   PlotFrame plotFrame = new PlotFrame("x", "f(x)",
                 "Numerical Integration Visualization");

   public void reset() {
      control.setValue("f(x)", "cos(x)");
      control.setValue("lower limit a", 0);
      control.setValue("upper limit b", Math.PI/2);
      control.setValue("n", 4);
   }

   public void calculate() {
      String fstring = control.getString("f(x)");
      double a = control.getDouble("lower limit a");
      double b = control.getDouble("upper limit b");
      int n = control.getInt("n"); // number of intervals
      Function f;
      try {
         f = new ParsedFunction(fstring);
      } catch(ParserException ex) {
         control.println(ex.getMessage());
         plotFrame.clearDrawables();
         return;
      }
      plotFrame.clearDrawables();
      // sets the domain of x to the integration limits
      plotFrame.setPreferredMinMaxX(a, b);
      plotFrame.addDrawable(new FunctionDrawer(f));
      RectangularApproximation approximate =
               new RectangularApproximation(f, a, b, n);
      plotFrame.addDrawable(approximate);
      plotFrame.setMessage("~ area = "
            +decimalFormat.format(approximate.sum));
      control.println("approximate area under curve = "+approximate.sum);
   }

   public static void main(String[] args) {
      CalculationControl.createApp(new NumericalIntegrationApp());
   }
}
```

We consider the accuracy of the rectangular approximation for the integral of $f(x) = \cos x$ from $x = 0$ to $x = \pi/2$ by comparing the numerical results shown in Table 11.1 with the exact answer of unity. Note that the error decreases as n^{-1}. This observed n^{-1}-dependence of the error is consistent with the analytic derivation of the n-dependence of the error obtained in Appendix 11A. We explore the n-dependence of the error associated with other numerical integration methods in Problems 11.1 and 11.2.

n	F_n	Δ_n
2	1.34076	0.34076
4	1.18347	0.18347
8	1.09496	0.09496
16	1.04828	0.04828
32	1.02434	0.02434
64	1.01222	0.01222
128	1.00612	0.00612
256	1.00306	0.00306
512	1.00153	0.00153
1024	1.00077	0.00077

Table 11.1: Rectangular approximations of the integral of $\cos x$ from $x = 0$ to $x = \pi/2$ as a function of n, the number of intervals. The error Δ_n is the difference between the rectangular approximation and the exact result of unity. Note that the error Δ_n decreases approximately as n^{-1}; that is, if n is increased by a factor of 2, Δ_n is decreased by a factor 2.

Problem 11.1. The rectangular and midpoint approximations

(a) Test `NumericalIntegrationApp` by reproducing the results in Table 11.1.

(b) Use the rectangular approximation to determine the approximate definite integrals of $f(x) = 2x + 3x^2 + 4x^3$ and $f(x) = e^{-x}$ for $0 \le x \le 1$ and $f(x) = 1/x$ for $1 \le x \le 2$. What is the approximate n-dependence of the error in each case?

(c) A straightforward modification of the rectangular approximation is to evaluate $f(x)$ at the *midpoint* of each interval. Define a `MidpointApproximation` class by making the necessary modifications and approximate the integral of $f(x) = \cos x$ in the interval $0 \le x \le \pi/2$. Remember that you need to change how the rectangles are drawn. How does the magnitude of the error compare with the results shown in Table 11.1? What is the approximate dependence of the error on n?

(d) Use the midpoint approximation to determine the definite integrals considered in part (b). What is the approximate n-dependence of the error in each case? □

Problem 11.2. The trapezoidal approximation

(a) Modify your program so that the trapezoidal approximation is computed and determine the n-dependence of the error for the same functions as in Problem (11.1). What is the approximate n dependence of the error in each case? Which approximation yields the best results for the same computation time?

(b) It is possible to double the number of intervals without losing the benefit of the previous calculations. For $n = 1$, the trapezoidal approximation is proportional to the average of $f(a)$ and $f(b)$. In the next approximation, the value of f at the midpoint is added to this average. The next refinement adds the values of f at the 1/4 and 3/4 points. Modify your program so that the number of intervals is doubled each time and the results of previous calculations are used. The following should be helpful:

```
if (n == 1) {
   double midpoint = a + 0.5*(b-a);
   sum = (b - a)*(0.5*f(a) + 0.5*f(b) + f(midpoint));
   n = 2;
} else {
   double delta = (b - a)/n;  // separation between new points
   double x = a + 0.5*delta;
   double intermediateSum = 0.0;
   for (int i = 0; i < n; i++) {
      intermediateSum +=  f(x);
      x += delta;
   }
   // 0.5*delta = separation between points
   sum = 0.5*(intermediateSum*delta + sum);
   n *= 2;
}
```

□

The rectangular and trapezoidal algorithms converge relatively slowly and are therefore not recommended in general. In practice, Simpson's rule is adequate for functions that are reasonably well behaved; that is, functions that can be adequately represented by a polynomial. If $f(x)$ is such a function, we can adopt the strategy of evaluating the area for a given number of intervals n and then doubling the number of intervals and evaluating the area again. If the two evaluations are sufficiently close to one another, we stop. Otherwise, we again double n until we achieve the desired accuracy. Of course, this strategy will fail if $f(x)$ is not well behaved. An example of a poorly behaved function is $f(x) = x^{-1/3}$ at $x = 0$, where $f(x)$ diverges. Another example where this strategy might fail is when a limit of integration is equal to $\pm\infty$. In many cases we can eliminate the problem by a change of variables.

Problem 11.3. Simpson's rule

(a) Write a class that implements Simpson's rule. Either adapt your program so that it uses Simpson's rule directly or convince yourself that the result of Simpson's rule can be expressed as

$$S_n = (4T_{2n} - T_n)/3 \tag{11.8}$$

where T_n is the result from the trapezoidal approximation for n intervals. It is not necessary to provide a visualization of the area. Use your program to evaluate the integral of $f(x) = (2\pi)^{-1/2}\,e^{-x^2}$ for $|x| \leq 1$. Do you obtain the same result by choosing the interval $[0,1]$ and then multiplying by two?

(b) Determine the same integrals as in Problem 11.2 and discuss the relative merits of the various approximations.

(c) Evaluate the integral of the function $f(x) = 4\sqrt{1-x^2}$ for $|x| \leq 1$. What value of n is needed for four decimal accuracy? The reason for the slow convergence can be understood by reading Appendix 11A.

(c)* Our strategy for approximating the value of the definite integrals we have considered has been to compute F_n and F_{2n} for reasonable values of n. If the difference $|F_{2n} - F_n|$ is too large, then we double n until the desired accuracy is reached. The success of this strategy is based on the implicit assumption that the sequence $F_n, F_{2n}, \ldots$ converges to the true integral F. Is there a way of extrapolating this sequence to the limit? Explore this idea by using the trapezoidal approximation. Because the error for this approximation decreases approximately as n^{-2}, we can write $F = F_n + Cn^{-2}$ and plot F_n as a function of n^{-2} to obtain the extrapolated result F. Apply this procedure to the integrals considered in some of the previous problems and compare your results to those found from the trapezoidal approximation and Simpson's rule alone. A more sophisticated application of this idea is known as *Romberg* integration (see Press et al.). □

Because integration is usually a routine task, we have implemented the integration methods discussed in this section in the Integral class in the numerics package. Listing 11.3 illustrates how the Integral class is used. Method Integral.ode computes the integrals using an ODE solver as discussed in the following. The solver uses an adaptive step size to achieve the desired tolerance.

Listing 11.3: The IntegralCalcApp program tests the Integral class.

```
package org.opensourcephysics.sip.ch11;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.numerics.*;

public class IntegralCalcApp extends AbstractCalculation {
   public void reset() {
      control.setValue("a", 0);
      control.setValue("b", 1);
      control.setValue("tolerance", 1.0e-2);
      control.setValue("f(x)", "sin(2*pi*x)^2");
   }

   public void calculate() {
      Function f;
      String fx = control.getString("f(x)");
      try {
         f = new ParsedFunction(fx);
      } catch(ParserException ex) {
         control.println(ex.getMessage());
         return;
      }
      double a = control.getDouble("a");
      double b = control.getDouble("b");
      double tolerance = control.getDouble("tolerance");
      double area = Integral.ode(f, a, b, tolerance);
      control.println("ODE area = "+area);
      area = Integral.trapezoidal(f, a, b, 2, tolerance);
      control.println("Trapezoidal area = "+area);
      area = Integral.simpson(f, a, b, 2, tolerance);
      control.println("Simpson area = "+area);
      area = Integral.romberg(f, a, b, 2, tolerance);
      control.println("Romberg area ="+area);
   }

   public static void main(String[] args) {
      CalculationControl.createApp(new IntegralCalcApp());
   }
}
```

The algorithms in the `Integral` class are implemented using static methods so that they are easy to invoke. These methods accept a *tolerance* parameter that allows the user to specify the acceptable relative precision. Because computers store floating point numbers using a fixed number of decimal places, we use relative precision to determine the accuracy of the integration method. However, relative precision is meaningful only if the result is different from zero. If the result is zero, the only possible check is for absolute precision. Because numerical integration algorithms should check the precision of their output, the `Util` class in the numerics package defines the following general purpose method for computing the relative precision from an absolute precision and a numerical result.

```
public static double relativePrecision(final double epsilon,
        final double result) {
   return (result > defaultNumericalPrecision)
      ? epsilon/result    // return epsilon/result if (...) is true
      : epsilon;          // else return epsilon
}
```

The `defaultNumericalPrecision` is a named constant that is equal to `Math.sqrt(Double.MIN_VALUE)`.

Problem 11.4. The `Integral` class

Add a static counter to keep track of the number of times the test function is evaluated in the `IntegralCalcApp` class. Use this counter to compare the efficiencies of the various integration algorithms.

(a) How many function calls are required for each numerical method if the tolerance is set to 10^{-3} or 10^{-12}?

(b) How does the execution time depend on the numerical method? You can compute the time using the statement `System.currentTimeMillis()`, which is the time between the current time and January 1, 1970 measured in milliseconds. Note that the return value is `long` and not `int`.

(c) The method `Integral.ode` determines when it has reached the input tolerance differently than the other integration algorithms in the `Integral` class. Compare the results from these other integrators. Is the accuracy of each method always within the tolerance set by the user? □

Another way of evaluating one-dimensional integrals is to recast them as differential equations. Consider an indefinite integral of the form

$$F(x) = \int_a^x f(t)\,dt \tag{11.9}$$

where $F(a) = 0$. If we differentiate $F(x)$ with respect to x, we obtain the first-order differential equation:

$$\frac{dF(x)}{dx} = f(x) \tag{11.10}$$

with the boundary condition

$$F(a) = 0. \tag{11.11}$$

Because the function $f(x)$ is known, (11.10) can be solved for $F(x)$ using the numerical algorithms that we introduced earlier for obtaining the numerical solutions of first-order differential equations.

In general, the methods for solving ordinary differential equations and evaluating numerical integrals are not equivalent. For example, we cannot use Simpson's rule to obtain the numerical solution of an differential equation of the form $dy/dx = f(x,y)$. Why?

Simpson's rule fails (or is slowly convergent) if the function has regions of rapid change. In this case, an adaptive step-size ODE algorithm is usually effective. The `RK45MultiStep` solver adapts the integration step to the behavior of the integrand and allows the user to set the tolerance. In Problem 11.5 we use it to examine an approximation to the Dirac delta function.

Problem 11.5. Delta function approximation

The Dirac delta function can be approximated by the function

$$\delta(x) = \frac{1}{\pi} \lim_{\epsilon \to 0} \frac{\epsilon}{x^2 + \epsilon^2}. \tag{11.12}$$

Test this approximation by integrating (11.12) over a suitable range of x using various values of ϵ. Try adaptive ODE solvers with a tolerance of 10^{-8}. How small a value of ϵ produces an error of 10^{-4}? 10^{-5}? □

As computers become more powerful and software to perform mathematical operations and numerical analysis becomes more prevalent, there is a strong temptation to use libraries written by others. This use is appropriate because usually these libraries are well written, and there is no reason why a user should re-invent them. However, the downside is that users do not always know or care what the libraries are doing and sometimes can obtain puzzling or even incorrect results. In Problem 11.6 we explore this issue.

Problem 11.6. Understanding errors in integration routines

(a) Use the `ode`, `simpson`, `trapezoid`, and `rhomberg` methods in the `Integral` class of the Open Source Physics library to estimate the integral of $\sin^2(2\pi x)$ between $x = 0$ and 1 with a tolerance of 0.01. The exact answer is 0.5. Do all four methods return results within the tolerance? Are some results much more accurate than the tolerance? Change the tolerance to 0.1. How do the results change? Notice that for some of the methods, the results are much better than might be expected because the positive and negative errors cancel. Why does the trapezoid method always give the exact answer?

(b) Integrate the same function from $x = 0.2$ to 1.0. How accurate are the results now? Explore how the results change with the input tolerance. Does the behavior of the `ode` integrator differ from the others. Why? How do you think the tolerance parameter is used for each of the methods?

(c) Integrate $f(x) = x^n$ for various values of n. How do the different integrators compare? Why does the trapezoid integrator do worse than the others for $n = 2$? □

11.2 Simple Monte Carlo Evaluation of Integrals

We now explore a much different method of estimating integrals. Consider the following example. Suppose an irregularly shaped pond is in a field of known area A. The area of the pond can be estimated by throwing stones so that they land at random within the boundary of the field and counting the number of splashes that occur when a stone lands in a pond. The area of the pond is approximately the area of the field times the fraction of stones that make a splash. This simple procedure is an example of a *Monte Carlo* method.

To be more specific, imagine a rectangle of height h, width $b-a$, and area $A = h(b-a)$ such that the function $f(x)$ is within the boundaries of the rectangle (see Figure 11.3). Compute n pairs of random numbers x_i and y_i with $a \leq x_i \leq b$ and $0 \leq y_i \leq h$. The fraction of points x_i, y_i that satisfy the condition $y_i \leq f(x_i)$ is an estimate of the ratio of the integral of $f(x)$ to the area of the rectangle. Hence, the estimate F_n in the *hit or miss* method is given by

$$F_n = A\frac{n_s}{n} \qquad \text{(hit or miss method)}, \tag{11.13}$$

where n_s is the number of points below the curve or "splashes," and n is the total number of points. The number of points chosen at random in (11.13) should not be confused with the number of intervals used in the numerical methods discussed in Section 11.1.

Another Monte Carlo integration method is based on a mean-value theorem of integral calculus, which states that the definite integral (11.1) is determined by the average value of the integrand $f(x)$ in the range $a \leq x \leq b$:

$$F = \int_a^b f(x)\,dx = (b-a)\langle f\rangle. \tag{11.14}$$

To determine $\langle f\rangle$, we choose the x_i at random instead of at regular intervals and *sample* the values of $f(x)$. For the one-dimensional integral (11.1), the estimate F_n of the integral in the *sample mean* method is given by

$$F_n = (b-a)\frac{1}{n}\sum_{i=1}^{n} f(x_i) \approx (b-a)\langle f\rangle \qquad \text{(sample mean method)}. \tag{11.15}$$

The x_i are random numbers distributed uniformly in the interval $a \leq x_i \leq b$, and n is the number of samples. Note that the forms of (11.3) and (11.15) are formally identical except that the n points are chosen with equal spacing in (11.3) and with random spacing in (11.15). We will find that for low-dimensional integrals, (11.3) is more accurate, but for higher-dimensional integrals, (11.15) does better.

A simple program that implements the hit or miss method is given in Listing 11.4. Note the use of the `Random` class and the methods `setSeed` and `nextDouble()`. As discussed in Chapter 7, the primary reason that it is desirable to specify the seed rather than to choose it more or less at random from the time (as is done by `Math.random()`) is that it is convenient to use the same random number sequence when testing a Monte Carlo program. Suppose that your program gives a strange result for a particular run. If we find an error in the program, we can use the same random number sequence to test whether our program changes make any difference. Another reason for specifying the seed is so that other users can reproduce your results.

Listing 11.4: `MonteCarloEstimatorApp` displays the estimate of the integral for the number of samples equal to 2^p.

```
package org.opensourcephysics.sip.ch11;
import java.util.Random;
import org.opensourcephysics.controls.*;
```

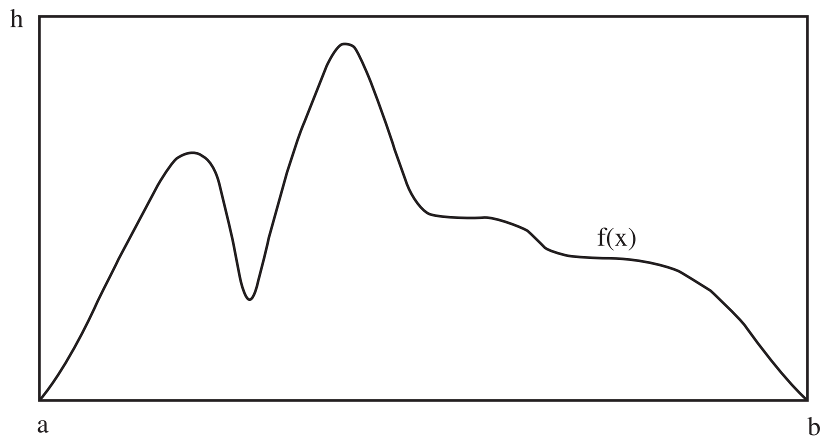


Figure 11.3: The function $f(x)$ is in the domain determined by the rectangle of height H and width $(b-a)$.

```
public class MonteCarloEstimatorApp extends AbstractSimulation {
   Random rnd = new Random();
   int n;       // current trial number
   int nTotal;  // total number of trials
   long seed;
   double a, b; // interval limits
   double ymax;
   int hits = 0;

   public void reset() {
      control.setValue("lower limit a", 0);
      control.setValue("upper limit b", 1.0);
      control.setValue("upper limit on y", 1.0);
      control.setValue("seed", 137933);
   }

   public double evaluate(double x) {
      return Math.sqrt(1-x*x);
   }

   public void initialize() {
      a = control.getDouble("lower limit a");
      b = control.getDouble("upper limit b");
      ymax = control.getDouble("upper limit on y");
      n = 0;
      nTotal = 2;
      seed = (long) control.getInt("seed");
      hits = 0;
      rnd.setSeed(seed);
   }

   public void doStep() {
      // nextDouble returns random double between
      // 0 (inclusive) and 1 (exclusive)
      for(int i = n;i<nTotal;i++) {
         double x = a+rnd.nextDouble()*(b-a);
         double y = rnd.nextDouble()*ymax;
         if(y<=evaluate(x)) {
            hits++;
         }
      }
      control.println("nTotal = "+nTotal+" estimated area = "
              +(hits*(b-a)*ymax)/nTotal);
      n = nTotal;
      nTotal *= 2; // increase number of trials by a factor of 2
   }

   public static void main(String[] args) {
      SimulationControl.createApp(new MonteCarloEstimatorApp());
   }
}
```

Problem 11.7. Monte Carlo integration in one dimension

(a) Use `MonteCarloEstimatorApp` to estimate F_n, the integral of $f(x) = 4\sqrt{1-x^2}$ in the interval $0 \leq x \leq 1$, using the hit or miss method. Choose $a = 0$, $b = 1$, $h = 1$ and compute the mean value of the function $\sqrt{1-x^2}$. Multiply the estimate by 4 to determine F_n. Calculate the difference between F_n and the exact result of π. This difference is a measure of the error associated with the Monte Carlo estimate. Make a log-log plot of the error as a function of n. What is the approximate dependence of the error on n for large n, for example, $n \geq 10^6$?

(b) Estimate the same integral using the sample mean method (11.15) and compute the error as a function of the number of samples n for $n \geq 10^6$. How many samples are needed to determine F_n to two decimal places? What is the approximate dependence of the error on n for large n?

(c) Determine the computational time per sample using the two Monte Carlo methods. Which Monte Carlo method is preferable? □

11.3 Multidimensional Integrals

Many problems in physics involve averaging over many variables. For example, suppose we know the position and velocity dependence of the total energy of ten interacting particles. In three dimensions each particle has three velocity components and three position components. Hence the total energy is a function of 60 variables, and a calculation of the average energy per particle involves computing a $d = 60$ dimensional integral. (More accurately, the total energy is a function of $60-6 = 54$ variables if we use center of mass and relative coordinates.) If we divide each coordinate into p intervals, there would be p^{60} points to sum. Clearly, standard numerical methods such as Simpson's rule would be impractical for this example.

A discussion of the n-dependence of the error associated with the standard numerical methods for d-dimensional integrals is given in Appendix 11A. We show that if the error decreases as n^{-a} for $d = 1$, then the error decreases as $n^{-a/d}$ in d dimensions. In contrast, we find (see

Section 11.4) that the error for all Monte Carlo integration methods decreases as $n^{-1/2}$ *independently* of the dimension of the integral. Because the computational time is roughly proportional to n in both the classical and Monte Carlo methods, we conclude that for low dimensions, the classical numerical methods such as Simpson's rule are preferable to Monte Carlo methods unless the domain of integration is very complicated. However, Monte Carlo methods are essential for higher dimensional integrals.

To illustrate the evaluation of multidimensional integrals, we consider the two-dimensional integral

$$F = \int_R f(x,y)\,dx\,dy \tag{11.16}$$

where R denotes the region of integration. The extension to higher dimensions is straightforward but tedious. Form a rectangle that encloses the region R and divide this rectangle into squares of length h. Assume that the rectangle runs from x_a to x_b in the x direction and from y_a to y_b in the y direction. The total number of squares is $n_x n_y$, where $n_x = (x_b - x_a)/h$ and $n_y = (y_b - y_a)/h$. If we use the midpoint approximation, the integral F is estimated by

$$F \approx \sum_{i=1}^{n_x} \sum_{j=1}^{n_y} f(x_i, y_j) H(x_i, y_j)\, h^2 \tag{11.17}$$

where $x_i = x_a + (i - \frac{1}{2})h$, $y_j = y_a + (j - \frac{1}{2})h$, and the (Heaviside) function $H(x,y) = 1$ if (x,y) is in R and equals 0 otherwise.

A simple Monte Carlo method for evaluating a two-dimensional integral uses the same rectangular region as in (11.17), but the n points (x_i, y_i) are chosen at random within the rectangle. The estimate for the integral is then

$$F_n = \frac{A}{n} \sum_{i=1}^{n} f(x_i, y_i) H(x_i, y_i) \tag{11.18}$$

where A is the area of the rectangle. Note that if $f(x,y) = 1$ everywhere, then (11.18) is equivalent to the hit or miss method of calculating the area of the region R. In general, (11.18) represents the area of the region R multiplied by the average value of $f(x,y)$ in R.

Problem 11.8. Two-dimensional numerical integration

(a) Write a program to implement the midpoint approximation in two dimensions and integrate the function $f(x,y) = x^2 + 6xy + y^2$ over the region defined by the condition $x^2 + y^2 \le 1$. Use $h = 0.1$, 0.05, 0.025, and 0.0125.

(b) Repeat part (a) using a Monte Carlo method and the same number of points n. For each value of n, repeat the calculation several times to obtain a crude estimate of the random error. □

Problem 11.9. Volume of a hypersphere

(a) The interior of a d-dimensional hypersphere of unit radius is defined by the condition $x_1^2 + x_2^2 + \cdots + x_d^2 \le 1$. Write a program that finds the volume of a hypersphere using the midpoint approximation. If you are clever, you can write a program that applies to any dimension using a recursive method. Test your program for $d = 2$ and $d = 3$ and then find the volume for $d = 4$ and $d = 5$. Begin with $h = 0.2$, and decrease h until your results do not change by more than 1%.

(b) Repeat part (a) using a Monte Carlo method. For each value of n, repeat the calculation several times to obtain a rough estimate of the random error. Is your program applicable for any d easier to write than in part (a)? □

11.4 Monte Carlo Error Analysis

Both the classical numerical integration methods and the Monte Carlo methods yield approximate answers whose accuracy depends on the number of intervals or on the number of samples, respectively. So far, we have used our knowledge of the exact values of various integrals to determine that the error in the Monte Carlo methods approaches zero as $n^{-1/2}$ for large n, where n is the number of samples. In the following, we will learn how to estimate the error when the exact answer is unknown. Our main result is that the $n^{-1/2}$-dependence of the error is a general result and is independent of the nature of the integrand and, most importantly, independent of the number of dimensions.

As before, we first determine the error for an explicit example. Consider the Monte Carlo evaluation of the integral of $f(x) = 4\sqrt{1-x^2}$ in the interval $[0,1]$ (see Problem 11.7). Our result for a particular sequence of $n = 10^4$ random numbers using the sample mean method is $F_n = 3.1489$. How does this result for F_n compare with your result found in Problem 11.7 for the same value of n? By comparing F_n to the exact result of $F = \pi \approx 3.1416$, we find that the error associated with $n = 10^4$ samples is approximately 0.0073. How do we know if $n = 10^4$ samples are sufficient to achieve the desired accuracy? We cannot answer this question definitively because if the actual error were known, we could correct F_n by the required amount and obtain F. The best we can do is to calculate the *probability* that the true value F is within a certain range centered about F_n.

We know that if the integrand is a constant, then the error would be zero; that is, F_n would equal F for any n. This limiting behavior suggests that a possible measure of the error is the *sample variance* $\tilde{\sigma}^2$ defined by

$$\tilde{\sigma}^2 = \frac{1}{n-1}\sum_{i=1}^{n}[f(x_i) - \langle f\rangle]^2 \tag{11.19}$$

where

$$\langle f\rangle = \frac{1}{n}\sum_{i=1}^{n} f(x_i). \tag{11.20}$$

The reason for the factor of $1/(n-1)$ in (11.19) rather than $1/n$ is similar to the reason for the expression $1/\sqrt{n-2}$ in the error estimates of the least squares fits [see (7.43)]. To compute $\tilde{\sigma}^2$, we need to use n samples to compute the mean, $\langle f\rangle$, and, loosely speaking, we have only $n-1$ independent samples remaining to calculate $\tilde{\sigma}^2$. Because we will always be considering values of $n >> 1$, we will replace $\tilde{\sigma}^2$ by the *variance* σ^2, which is given by

$$\sigma^2 = \langle f^2\rangle - \langle f\rangle^2 \tag{11.21}$$

where

$$\langle f^2\rangle = \frac{1}{n}\sum_{i=1}^{n}[f(x_i)]^2. \tag{11.22}$$

For our example and the same sequence of random numbers that we used to obtain $F_n = 3.1489$, we obtain the standard deviation $\sigma = 0.8850$. Because this value of σ is two orders of magnitude larger than the actual error, we conclude that σ is not a direct measure of the error.

n	$\mathbf{F_n}$	**actual error**	σ	$\sigma/\sqrt{n}$
10^2	3.0692	0.0724	0.8550	0.0855
10^3	3.1704	0.0288	0.8790	0.0270
10^4	3.1489	0.0073	0.8850	0.0089

Table 11.2: Examples of Monte Carlo measurements of the mean value of $f(x) = 4\sqrt{1-x^2}$ in the interval $[0,1]$. The actual error is given by the difference $|F_n - \pi|$. The standard deviation σ is estimated using (11.21).

Another clue to finding an appropriate measure of the error can be found by increasing n and seeing how the actual error decreases as n increases. In Table 11.2 we see that as n is increased from 10^2 to 10^4, the actual error decreased by a factor of approximately 10; that is, as $\sim 1/n^{1/2}$. We also see that the actual error is approximately given by $\sigma/\sqrt{n}$. In Appendix 11B we show that the *standard error of the means* σ_m is given by

$$\sigma_m = \frac{\sigma}{\sqrt{n-1}} \tag{11.23a}$$

$$\approx \frac{\sigma}{\sqrt{n}}. \tag{11.23b}$$

The interpretation of σ_m is that if we make many independent measurements of F_n, each with n data points, then the *probable error* associated with any single measurement is σ_m. The more precise interpretation of σ_m is that F_n, our estimate for the mean, has a 68% chance of being within σ_m of the "true" mean and a 97% chance of being within $2\sigma_m$. This interpretation assumes a Gaussian distribution of the various measurements.

The quantity F_n is an estimate of the average value of the data points. As we increase n, the number of data points, we do not expect our estimate of the mean to change much. What changes, as we increase n, is our *confidence* in our estimate of the mean. Similar considerations hold for our estimate of σ, which is why σ is not a direct measure of the error.

The error estimate in (11.23) assumes that the data points are independent of each other. However, in many situations the data is correlated, and we have to be careful about how we estimate the error. For example, suppose that instead of choosing n random values for x, we instead start with a particular value x_0 and then randomly add increments such that the ith value of x is given by $x_i = x_{i-1} + (2r-1)\delta$, where r is uniformly distributed between 0 and 1, and $\delta = 0.01$. Clearly, the x_i are now correlated. We can still obtain an estimate for the integral, but we cannot use $\sigma/\sqrt{n}$ as the estimate for the error because this estimate would be smaller than the actual error. However, we expect that two data points, x_i and x_j, will become uncorrelated if $|j-i|$ is sufficiently large. How can we tell when $|j-i|$ is sufficiently large? One way is to group the data by averaging over m data points. We take $f_1^{(m)}$ to be the average of the first m values of $f(x_i)$, $f_2^{(m)}$ to be the average of the next m values, and so forth. Then we compute $\sigma_s/\sqrt{s}$, where $s = n/m$ is the number of $f_i^{(m)}$ data points, each of which is an average over m of the original data points, and σ_s is the standard deviation of the s data points, We do this grouping for different values of m (and s) and find the value of m for which $\sigma_s/\sqrt{s}$ becomes approximately independent of m. This ratio is our estimate of the error of the mean.

We see that we can make the probable error as small as we wish by either increasing n, the number of data points, or by reducing the variance σ^2. Several *reduction of variance* methods are introduced in Sections 11.6 and 11.7.

Problem 11.10. Estimate of the Monte Carlo error

(a) Estimate the integral of $f(x) = e^{-x}$ in the interval $0 \leq x \leq 1$ using the sample mean Monte Carlo method with $n = 10^4$, $n = 10^6$, and $n = 10^8$. Determine the exact integral analytically and estimate the n dependence of the actual error. How does your estimated error compare with the error estimate obtained from the relation (11.23)?

(b) Generate 19 additional measurements of the integral each with $n = 10^6$ samples. Compute the standard deviation of the 20 measurements. Is the magnitude of this standard deviation of the means consistent with your estimates of the error obtained in part (a)? Compute the histogram of the additional measurements and confirm that the distribution of the measurements is consistent with a Gaussian distribution.

(c) Divide your first measurement of $n = 10^6$ samples into $s = 10$ subsets of 10^5 samples each. Is the value of $\sigma_s/\sqrt{s}$ consistent with your previous error estimates?

(d) Estimate the integral

$$\int_0^1 e^{-x^2}\,dx \tag{11.24}$$

to two decimal places using $\sigma/\sqrt{n}$ as an estimate of the probable error.

(e) Estimate the integral $\int_0^{2\pi} \cos^2\theta\,d\theta$ using $n = 10^6$, where $\theta_i = \theta_{i-1} + (2r-1)\delta$, r is uniformly distributed between 0 and 1, and $\delta = 0.1$. Note that because $\cos\theta = \cos\theta + 2k\pi$ for any integer k, we do not have to restrict the range of θ_i. Estimate the error using (11.23). Is this error estimate accurate? Also, estimate the error by grouping the data into $m = 10, 10^2, 10^3$, 10^4, and 10^5 data points and compute $\sigma_s/\sqrt{s}$ for $s = 10^5, 10^4, 10^3, 10^2$, and 10, respectively. How large must m be so that the error estimates for different values of m are approximately the same? Discuss the relation between this result and the correlation of the data points. □

***Problem 11.11.** Importance of randomness

We learned in Chapter 7 that the random number generator included with many programming languages is based on the linear congruential method. In this method each term in the sequence can be found from the preceding one by the relation

$$x_{n+1} = (ax_n + c) \bmod m \tag{11.25}$$

where x_0 is the seed, and a, c, and m are nonnegative integers. The random numbers r in the unit interval $0 \leq r < 1$ are given by $r_n = x_n/m$. To examine the effect of a poor random number generator, we choose values of x_0, m, a, and c such that (11.25) has poor statistical properties, for example, a short period. What is the period for $x_0 = 1$, $a = 5$, $c = 0$, and $m = 32$? Estimate the integral in Problem 11.10 by making a single measurement of $n = 10^4$ samples using the linear congruential method (11.25) with these values of x_0, a, c, and m. Analyze your measurement by computing $\sigma_s/s^{1/2}$ for $s = 20$ subsets. Then divide your data into $s = 10$ subsets. Is the value of $\sigma_s/s^{1/2}$ consistent with what you obtained for $s = 20$? □

***Problem 11.12.** Error estimating by bootstrapping

Suppose that we have made a series of measurements but do not know the underlying probability distribution of the data. How can we estimate the errors of the quantities of interest in an unbiased way? One way is to use a method known as *bootstrapping*, a method that uses random sampling to estimate the errors.

Consider a set of n measurements, such as n values of the pairs (x_i, y_i), and suppose we want to fit this data to the best straight line. If we label the original set of measurements $M = \{m_1, m_2, \dots, m_n\}$, then the kth *resampled* data set M_k consists of n measurements that are randomly chosen from the original set. This procedure means that some of the m_i may not appear in M_k and some may appear more than once. We then compute the quantity G_k from the resampled data set. For example, G_k could be the slope found from a least squares calculation. If we do this resampling n_r times, a measure of the error in the quantity G is given by σ_G^2, where

$$\sigma_G = \frac{1}{n_r - 1} \sum_{k=1}^{n_r} \Big[G_k - \langle G_k \rangle \Big]^2 \tag{11.26}$$

with

$$\langle G_k \rangle = \frac{1}{n_r} \sum_{k=1}^{n_r} G_k. \tag{11.27}$$

(a) To see how this procedure works, consider $n = 15$ pairs of points x_i randomly distributed between 0 and 1, with the corresponding values of y given by $y_i = 2x_i + 3 + s_i$, where s_i is a uniform random number between -1 and $+1$. First compute the slope m and the intercept b using the least squares method and their corresponding errors using (7.42).

(b) Resample the same set of data 200 times, computing the slope and intercept each time using the least squares method. From your results estimate the probable error for the slope and intercept using (11.26). How well do the estimates from bootstrapping compare with the direct error estimates found in part (a)? Does the average of the bootstrap values for the slope and intercept equal m and b, respectively, from the least squares fits. If not, why not? Do your conclusions change if you resample 800 times? □

11.5 Nonuniform Probability Distributions

In Sections 11.2 and 11.4, we learned how uniformly distributed random numbers can be used to estimate definite integrals. We will find that it is more efficient to sample the integrand $f(x)$ more often in regions of x where the magnitude of $f(x)$ is large or rapidly varying. Because *importance sampling* methods require nonuniform probability distributions, we first consider several methods for generating random numbers that are not distributed uniformly. In the following, we will denote r as a member of a uniform random number sequence in the unit interval $0 \le r < 1$.

Suppose that two discrete events 1 and 2 occur with probabilities p_1 and p_2 such that $p_1 + p_2 = 1$. How can we choose the two events with the correct probabilities using a uniform probability distribution? For this simple case, it is clear that we choose event 1 if $r < p_1$; otherwise, we choose event 2. If there are three events with probabilities p_1, p_2, and p_3, then if $r < p_1$, we choose event 1; else if $r < p_1 + p_2$, we choose event 2; else we choose event 3. We can visualize these choices by dividing a line segment of unit length into three pieces whose lengths are shown in Figure 11.4.

Now consider n discrete events. How do we determine which event i to choose given the value of r? The generalization of the procedure we have followed for $n = 2$ and $n = 3$ is to find the value of i that satisfies the condition

$$\sum_{j=0}^{i-1} p_j \le r \le \sum_{j=0}^{i} p_j \tag{11.28}$$

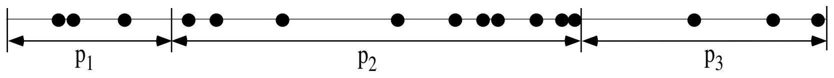


Figure 11.4: The unit interval is divided into three segments of lengths $p_1 = 0.2$, $p_2 = 0.5$, and $p_3 = 0.3$. Sixteen random numbers are represented by the filled circles uniformly distributed on the unit interval. The fraction of circles within each segment is approximately equal to the value of p_i for that segment.

where we have defined $p_0 \equiv 0$. Verify that (11.28) reduces to the correct procedure for $n = 2$ and $n = 3$.

We now consider a continuous nonuniform probability distribution. One way to generate such a distribution is to take the limit of (11.28) and associate p_i with $p(x)\Delta x$, where the *probability density* $p(x)$ is defined such that $p(x)\Delta x$ is the probability that the event x is in the interval between x and $x + \Delta x$. The probability density $p(x)$ is normalized such that

$$\int_{-\infty}^{+\infty} p(x)\,dx = 1. \tag{11.29}$$

In the continuum limit, the two sums in (11.28) become the same integral, and the inequalities become equalities. Hence, we can write

$$P(x) \equiv \int_{-\infty}^{x} p(x')\,dx' = r. \tag{11.30}$$

From (11.30) we see that the uniform random number r corresponds to the *cumulative probability distribution* function $P(x)$, which is the probability of choosing a value less than or equal to x. The function $P(x)$ should not be confused with the probability density $p(x)$ or the probability $p(x)\Delta x$. In many applications the meaningful range of values of x is positive. In that case we have $p(x) = 0$ for $x < 0$.

The relation (11.30) leads to the *inverse transform* method for generating random numbers distributed according to the function $p(x)$. This method involves generating a random number r and solving (11.30) for the corresponding value of x. As an example of the method, we use (11.30) to generate a random number sequence according to the uniform probability distribution on the interval $a \le x \le b$. The desired probability density $p(x)$ is

$$p(x) = \begin{cases} (1/(b-a) & a \le x \le b \\ 0 & \text{otherwise.} \end{cases} \tag{11.31}$$

The cumulative probability distribution function $P(x)$ for $a \le x \le b$ can be found by substituting (11.31) into (11.30) and performing the integral. The result is

$$P(x) = \frac{x-a}{b-a}. \tag{11.32}$$

If we substitute the form (11.32) for $P(x)$ into (11.30) and solve for x, we find the desired relation

$$x = a + (b-a)r. \tag{11.33}$$

The variable x given by (11.33) is distributed according to the probability distribution $p(x)$ given by (11.31). Of course, the relation (11.33) is obvious, and we already have used it in our programs.

We next apply the inverse transform method to the probability density function

$$p(x) = \begin{cases} (1/\lambda)\,e^{-x/\lambda} & 0 \le x \le \infty \\ 0 & x < 0. \end{cases} \tag{11.34}$$

If we substitute (11.34) into (11.30) and integrate, we find the relation

$$r = P(x) = 1 - e^{-x/\lambda}. \tag{11.35}$$

The solution of (11.35) for x yields $x = -\lambda \ln(1-r)$. Because $1-r$ is distributed in the same way as r, we can write

$$x = -\lambda \ln r. \tag{11.36}$$

The variable x found from (11.36) is distributed according to the probability density $p(x)$ given by (11.34). Because the computation of the natural logarithm in (11.36) is relatively slow, the inverse transform method might not be the most efficient method to use in this case.

From the above examples, we see that two conditions must be satisfied in order to apply the inverse transform method: the form of $p(x)$ must allow us to perform the integral in (11.30) analytically, and it must be practical to invert the relation $P(x) = r$ for x.

The Gaussian probability density

$$p(x) = \frac{1}{(2\pi\sigma^2)^{1/2}} e^{-x^2/2\sigma^2} \tag{11.37}$$

is an example of a probability density for which the cumulative distribution $P(x)$ cannot be obtained analytically. However, we can generate the two-dimensional probability $p(x,y)\,dx\,dy$ given by

$$p(x,y)\,dx\,dy = \frac{1}{2\pi\sigma^2} e^{-(x^2+y^2)/2\sigma^2}\,dx\,dy. \tag{11.38}$$

First, we make a change of variables to polar coordinates:

$$r = (x^2+y^2)^{1/2}, \qquad \theta = \tan^{-1}\frac{y}{x}. \tag{11.39}$$

We let $\rho = r^2/2$ and write the two-dimensional probability as

$$p(\rho,\theta)\,d\rho\,d\theta = \frac{1}{2\pi} e^{-\rho}\,d\rho\,d\theta \tag{11.40}$$

where we have set $\sigma = 1$. If we generate ρ according to the exponential distribution (11.34) and generate θ uniformly in the interval $0 \le \theta < 2\pi$, then the quantities

$$x = (2\rho)^{1/2}\cos\theta \quad \text{and} \quad y = (2\rho)^{1/2}\sin\theta \qquad \text{(Box–Muller method)} \tag{11.41}$$

will each be generated according to (11.37) with zero mean and $\sigma = 1$. (Note that the two-dimensional density (11.38) is the product of two independent one-dimensional Gaussian distributions.) This way of generating a Gaussian distribution is known as the *Box–Muller* method. We discuss other methods for generating the Gaussian distribution in Problems 11.14 and 11.17 and in Appendix 11C.

Problem 11.13. Nonuniform probability densities

(a) Write a program to simulate the simultaneous rolling of two dice. In this case the events are discrete and occur with nonuniform probability.

(b) Write a program to verify that the sequence of random numbers $\{x_i\}$ generated by (11.36) is distributed according to the exponential distribution (11.34).

(c) Generate random variables according to the probability density function

$$p(x) = \begin{cases} 2(1-x) & 0 \le x \le 1 \\ 0 & \text{otherwise.} \end{cases} \tag{11.42}$$

(d) Verify that the variables x and y in (11.41) are distributed according to the Gaussian distribution. What are the mean value and the standard deviation of x and y?

(e) How can you use the relations (11.41) to generate a Gaussian distribution with arbitrary mean and standard deviation? □

Problem 11.14. Generating normal distributions

Fernández and Criado have suggested another method of generating normal distributions that is much faster than the Box–Muller method. We will just summarize the algorithm; the proof that the algorithm leads to a normal distribution is given in their paper.

(a) Begin with N numbers v_i in an array. Set all the $v_i = \sigma$, where σ is the desired standard deviation for the normal distribution.

(b) Update the array by randomly choosing two different entries v_i and v_j from the array. Then let $v_i = (v_i + v_j)/\sqrt{2}$ and use the new v_i to set $v_j = -v_i + v_j\sqrt{2}$.

(c) Repeat step (b) many times to bring the array of numbers to "equilibrium."

(d) After equilibration, the entries v_i will have a normal distribution with the desired standard deviation and zero mean.

Write a program to produce a series of random numbers according to this algorithm. Your program should allow the user to enter N and σ, and a button should be implemented to allow for equilibration before various averages are computed. The desired output is the probability distribution of the random numbers that are produced as well as their mean and standard deviation. First make sure that the standard deviation of the probability distribution approaches the desired input σ for sufficiently long times. What is the order of magnitude of the equilibration time? Does it depend on N? Plot the natural log of the probability distribution versus v^2 and check that you obtain a straight line with the appropriate slope. □

11.6 Importance Sampling

In Section 11.4 we found that the error associated with a Monte Carlo estimate is proportional to the standard deviation σ of the integrand and inversely proportional to the square root of the number of samples. Hence, there are only two ways of reducing the error in a Monte Carlo estimate—either increase the number of samples or reduce the variance. Clearly the latter choice is desirable because it does not require much more computer time. In this section we introduce *importance sampling* techniques that reduce σ and improve the efficiency of each sample.

To do importance sampling in the context of numerical integration, we introduce a positive function $p(x)$ such that

$$\int_a^b p(x)\,dx = 1 \tag{11.43}$$

and rewrite the integral (11.1) as

$$F = \int_a^b \left[\frac{f(x)}{p(x)}\right] p(x)\,dx. \tag{11.44}$$

We can evaluate the integral (11.44) by sampling according to the probability distribution $p(x)$ and constructing the sum

$$F_n = \frac{1}{n}\sum_{i=1}^{n} \frac{f(x_i)}{p(x_i)}. \tag{11.45}$$

The sum (11.45) reduces to (11.15) for the uniform case $p(x) = 1/(b-a)$.

The idea is to choose a form for $p(x)$ that minimizes the variance of the ratio $f(x)/p(x)$. To do so we choose a form of $p(x)$ that mimics $f(x)$ as much as possible, particularly where $f(x)$ is large. A suitable choice of $p(x)$ would make the integrand $f(x)/p(x)$ slowly varying, and hence reduce the variance. Because we cannot evaluate the variance analytically in general, we determine σ *a posteriori*.

As an example, we again consider the integral (see Problem 11.10d)

$$F = \int_0^1 e^{-x^2}\,dx. \tag{11.46}$$

The estimate of F with $p(x) = 1$ for $0 \le x \le 1$ is shown in the second column of Table 11.3. A simple choice for the weight function is $p(x) = Ae^{-x}$, where A is chosen such that $p(x)$ is normalized on the unit interval. Note that this choice of $p(x)$ is positive definite and is qualitatively similar to $f(x)$. The results are shown in the third column of Table 11.3. We see that although the computation time per sample for the nonuniform case is larger, the smaller value of σ makes the use of the nonuniform probability distribution more efficient.

Problem 11.15. Importance sampling

(a) Choose $f(x) = \sqrt{1-x^2}$ and consider $p(x) = A(1-x)$ for $x \ge 0$. What is the value of A that normalizes $p(x)$ in the unit interval $[0,1]$? What is the relation for the random variable x in terms of r for this form of $p(x)$? What is the variance of $f(x)/p(x)$ in the unit interval? Evaluate the integral $\int_0^1 f(x)\,dx$ using $n = 10^6$ and estimate the probable error of your result.

	$\mathbf{p(x) = 1}$	$\mathbf{p(x) = Ae^{-x}}$
n (samples)	5×10^6	4×10^5
F_n	0.74684	0.74689
σ	0.2010	0.0550
$\sigma/\sqrt{n}$	0.00009	0.00009
Total CPU time (s)	20	2.5
CPU time per sample (s)	4×10^{-6}	6×10^{-6}

Table 11.3: Comparison of the Monte Carlo estimates of the integral (11.46) using the uniform probability density $p(x) = 1$ and the nonuniform probability density $p(x) = Ae^{-x}$. The normalization constant A is chosen such that $p(x)$ is normalized on the unit interval. The value of the integral to five decimal places is 0.74682. The estimate F_n, variance σ of f/p, and the probable error $\sigma/n^{1/2}$ are shown. The CPU time (in seconds) is shown for comparison only. (The number of samples was chosen so that the error estimates are comparable.)

(b) Choose $p(x) = Ae^{-\lambda x}$ and evaluate the integral

$$\int_0^{\pi} \frac{1}{x^2 + \cos^2 x}\, dx. \tag{11.47}$$

Determine the value of λ that minimizes the variance of the integrand. □

Problem 11.16. An adaptive approach to importance sampling

An alternative approach is to use the known values of $f(x)$ at regular intervals to sample more often where $f(x)$ is relatively large. Because the idea is to use $f(x)$ itself to determine the probability of sampling, we only consider integrands that are nonnegative. To compute a rough estimate of the relative values of $f(x)$, we first compute its average value by taking k equally spaced points s_i and computing the sum

$$S = \sum_{i=1}^{k} f(s_i). \tag{11.48}$$

This sum divided by k gives an estimate of the average value of f in the interval. The approximate value of the integral is given by $F \approx Sh$, where $h = (b-a)/k$. This approximation of the integral is equivalent to the rectangular or midpoint approximation depending on where we compute the values of $f(x)$. We then generate n random samples as follows. The probability of choosing subinterval (bin) i is given by the probability

$$p_i = \frac{f(s_i)}{S}. \tag{11.49}$$

Note that the sum of p_i over all subintervals is normalized to unity.

To choose a subinterval with the desired probability, we generate a random number ¡ uniformly in the interval $[a, b]$ and determine the subinterval i that satisfies the inequality (11.28). Now that the subinterval has been chosen with the desired probability, we generate a random number x_i in the subinterval $[s_i, s_i + h]$ and compute the ratio $f(x_i)/p(x_i)$. The estimate of the integral is given by the following considerations. The probability p_i in (11.49) is the probability of choosing the subinterval i, not the probability $p(x)\Delta x$ of choosing a value of x between x and $x+\Delta x$. The latter is p_i times the probability of picking the particular value of x in subinterval i:

$$p(x_i)\Delta x = p_i \frac{\Delta x}{h}. \tag{11.50}$$

Hence, we have that

$$F_n = \frac{1}{n}\sum_{i=1}^{n}\frac{f(x_i)}{p(x_i)} = \frac{h}{n}\sum_{i=1}^{n}\frac{f(x_i)}{p_i}. \tag{11.51}$$

Apply this method to estimate the integral of $f(x) = \sqrt{1-x^2}$ in the unit interval. Under what circumstances would this approach be most useful? □

11.7 Metropolis Algorithm

Another way of generating an arbitrary nonuniform probability distribution was introduced by Metropolis, Rosenbluth, Rosenbluth, Teller, and Teller in 1953. The *Metropolis* algorithm is a special case of an importance-sampling procedure in which certain possible sampling attempts are rejected (see Appendix 11C). The Metropolis method is useful for computing averages of the form

$$\langle f\rangle = \frac{\int f(x)p(x)\,dx}{\int p(x)\,dx} \tag{11.52}$$

where $p(x)$ is an arbitrary function that need not be normalized. In Chapter 15 we will discuss the application of the Metropolis algorithm to problems in statistical mechanics.

For simplicity, we introduce the Metropolis algorithm in the context of estimating one-dimensional definite integrals. Suppose that we wish to use importance sampling to generate random variables according to $p(x)$. The Metropolis algorithm produces a random walk of points $\{x_i\}$ whose asymptotic probability distribution approaches $p(x)$ after a large number of steps. The random walk is defined by specifying a *transition probability* $T(x_i \to x_j)$ from one value x_i to another value x_j such that the distribution of points $x_0, x_1, x_2, \ldots$ converges to $p(x)$. It can be shown that it is sufficient (but not necessary) to satisfy the *detailed balance* condition

$$p(x_i)T(x_i \to x_j) = p(x_j)T(x_j \to x_i). \tag{11.53}$$

The relation (11.53) does not specify $T(x_i \to x_j)$ uniquely. A simple choice of $T(x_i \to x_j)$ that is consistent with (11.53) is

$$T(x_i \to x_j) = \min\Big[1, \frac{p(x_j)}{p(x_i)}\Big]. \tag{11.54}$$

If the "walker" is at position x_i and we wish to generate x_{i+1}, we can implement this choice of $T(x_i \to x_{i+1})$ by the following steps:

1. Choose a trial position $x_{\text{trial}} = x_i + \delta_i$, where δ_i is a uniform random number in the interval $[-\delta, \delta]$.
2. Calculate $w = p(x_{\text{trial}})/p(x_i)$.
3. If $w \geq 1$, accept the change and let $x_{i+1} = x_{\text{trial}}$.

4. If $w < 1$, generate a random number r.
5. If $r \leq w$, accept the change and let $x_{i+1} = x_{\text{trial}}$.
6. If the trial change is not accepted, then let $x_{i+1} = x_i$.

It is necessary to sample many points of the random walk before the asymptotic probability distribution $p(x)$ is attained. How do we choose the maximum step size δ? If δ is too large, only a small percentage of trial steps will be accepted, and the sampling of $p(x)$ will be inefficient. On the other hand, if δ is too small, a large percentage of trial steps will be accepted, but again the sampling of $p(x)$ will be inefficient. A rough criterion for the magnitude of δ is that approximately one third to one half of the trial steps should be accepted. We also wish to choose the value of x_0 such that the distribution $\{x_i\}$ will approach the asymptotic distribution as quickly as possible. An obvious choice is to begin the random walk at a value of x at which $p(x)$ is a maximum. A code fragment that implements the Metropolis algorithm is given below.

```
double xtrial = x + (2*rnd.nextDouble() - 1.0)*delta;
double w = p(xtrial)/p(x);
if (w > 1 || w > rnd.nextDouble()) {
   x = xtrial;
   naccept++;        // number of acceptances
}
```

Problem 11.17. Generating the Gaussian distribution

(a) Use the Metropolis algorithm to generate the Gaussian distribution $p(x) = Ae^{-x^2/2}$. Is the value of the normalization constant A relevant? Determine the qualitative dependence of the acceptance ratio and the equilibration time on the maximum step size δ. One possible criterion for equilibrium is that $\langle x^2 \rangle \approx 1$. What is a reasonable choice for δ? How many trials are needed to reach equilibrium for your choice of δ?

(b) Modify your program so that it plots the asymptotic probability distribution generated by the Metropolis algorithm.

(c)* Calculate the autocorrelation function $C(j)$ defined by (see Problem 7.31)

$$C(j) = \frac{\langle x_{i+j} x_i \rangle - \langle x_i \rangle^2}{\langle x_i^2 \rangle - \langle x_i \rangle^2} \tag{11.55}$$

where $\langle \ldots \rangle$ indicates an average over the random walk. What is the value of $C(j = 0)$? What would be the value of $C(j \neq 0)$ if x_i were completely random? Calculate $C(j)$ for different values of j and determine the value of j for which $C(j)$ is close to zero. □

Problem 11.18. Application of the Metropolis algorithm

(a) Although the Metropolis algorithm is not the most efficient method in this case, write a program to estimate the average

$$\langle x \rangle = \frac{\int_0^\infty x e^{-x}\, dx}{\int_0^\infty e^{-x}\, dx} \tag{11.56}$$

with $p(x) = Ae^{-x}$ for $x \geq 0$ and $p(x) = 0$ for $x < 0$. Compute the histogram $H(x)$ showing the number of points in the random walk in the region x to $x + \Delta x$ with $\Delta x = 0.2$. Begin with

$n \geq 1000$ and maximum step size $\delta = 1$. Allow the system to equilibrate for at least 200 steps before computing averages. Is the integrand sampled uniformly? If not, what is the approximate region of x where the integrand is sampled more often?

(b) Calculate analytically the exact value of $\langle x \rangle$ in (11.56). How do your Monte Carlo results compare with the exact value for $n = 100$ and $n = 1000$ with $\delta = 0.1$, 1, and 10? Estimate the standard error of the mean. Does this error give a reasonable estimate of the error?

(c) In part (b) you should have found that the estimated error is much smaller than the actual error. The reason is that the $\{x_i\}$ are not statistically independent. The Metropolis algorithm produces a random walk whose points are correlated with each other over short times (measured by the number of steps of the random walker). The correlation of the points decays exponentially with time. If τ is the characteristic time for this decay, then only points separated by approximately 2 to 3τ can be considered statistically independent. Compute $C(j)$ as defined in (11.55) and make a rough estimate of τ. Rerun your program with the data grouped into 20 sets of 50 points each and 10 sets of 100 points each. If the sets of 50 points each are statistically independent (that is, if τ is significantly smaller than 50), then your estimate of the error for the two groupings should be approximately the same. The importance of correlations between sampled points is discussed further in Section 15.7. □

11.8 *Neutron Transport

We consider the application of a nonuniform probability distribution to the simulation of the transmission of neutrons through bulk matter, one of the original applications of a Monte Carlo method. Suppose that a neutron is incident on a plate of thickness t. We assume that the plate is infinite in the x and y directions and the z-axis is normal to the plate. At any point within the plate, the neutron can either be captured with probability p_c or scattered with probability p_s. These probabilities are proportional to the capture cross section and scattering cross section, respectively. If the neutron is scattered, we need to find its new direction as specified by the polar angle θ (see Figure 11.5). Because we are not interested in how far the neutron moves in the x or y direction, the value of the azimuthal angle ϕ is irrelevant.

If the neutrons are scattered equally in all directions, then the probability $p(\theta,\phi)\,d\theta d\phi$ equals $d\Omega/4\pi$, where $d\Omega$ is an infinitesimal solid angle and 4π is the total solid angle. Because $d\Omega = \sin\theta\, d\theta d\phi$, we have

$$p(\theta,\phi) = \frac{\sin\theta}{4\pi}. \tag{11.57}$$

We can find the probability density for θ and ϕ separately by integrating over the other angle. For example,

$$p(\theta) = \int_0^{2\pi} p(\theta,\phi)\, d\phi = \frac{1}{2}\sin\theta \tag{11.58}$$

and

$$p(\phi) = \int_0^{\pi} p(\theta,\phi)\, d\theta = \frac{1}{2\pi}. \tag{11.59}$$

Because the point probability $p(\theta,\phi)$ is the product of the probabilities $p(\theta)$ and $p(\phi)$, θ and ϕ are independent variables. Although we do not need to generate a random angle ϕ, we note

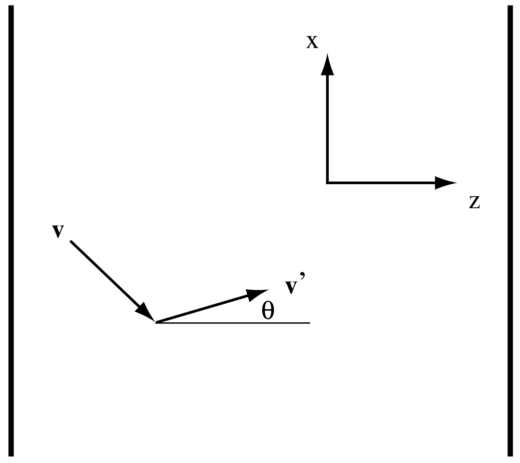


Figure 11.5: The definition of the scattering angle θ. The velocity before scattering is $\mathbf{v}$ and the velocity after scattering is $\mathbf{v}'$. The scattering angle θ is independent of $\mathbf{v}$ and is defined relative to the z-axis.

that because $p(\phi)$ is a constant, ϕ can be found from the relation

$$\phi = 2\pi r. \tag{11.60}$$

To find θ according to the distribution (11.58), we substitute (11.58) in (11.30) and obtain

$$r = \frac{1}{2}\int_0^\theta \sin x\, dx. \tag{11.61}$$

The integration in (11.61) gives

$$\cos\theta = 1 - 2r. \tag{11.62}$$

Note that (11.60) implies that ϕ is uniformly distributed between 0 and 2π, and (11.62) implies that $\cos\theta$ is uniformly distributed between -1 and $+1$. We could invert the cosine in (11.62) to solve for θ. However, to find the z-component of the path of the neutron through the plate, we need to multiply the path length ℓ by $\cos\theta$, and hence we need $\cos\theta$ rather than θ.

The path length, which is the distance traveled between subsequent scattering events, is obtained from the exponential probability density $p(\ell) \propto e^{-\ell/\lambda}$ [see (11.34)]. From (11.36), we have

$$\ell = -\lambda \ln r \tag{11.63}$$

where λ is the mean free path.

Now we have all the necessary ingredients for calculating the probabilities for a neutron to pass through the plate, to be reflected off the plate, or to be captured and absorbed in the plate. The input parameters are the thickness of the plate t, the capture probability p_c and the mean free path λ. The scattering probability is $p_s = 1 - p_c$. We begin with $z = 0$ and implement the following steps:

1. Determine if the neutron is captured or scattered. If it is captured, then add one to the number of captured neutrons and go to step 5.

2. If the neutron is scattered, compute $\cos\theta$ from (11.62) and ℓ from (11.63). Change the z-coordinate of the neutron by $\ell\cos\theta$.
3. If $z < 0$, add one to the number of reflected neutrons. If $z > t$, add one to the number of transmitted neutrons. In either case skip to step 5 below.
4. Repeat steps 1–3 until the fate of the neutron has been determined.
5. Repeat steps 1–4 with additional incident neutrons until sufficient data has been obtained.

Problem 11.19. Elastic neutron scattering

(a) Write a program to implement the above algorithm for neutron scattering through a plate. Assume $t = 1$ and $p_c = p_s/2$. Find the transmission, reflection, and absorption probabilities for $\lambda = 0.01$, 0.05, 0.1, 1, and 10. Begin with 1000 incident neutrons and increase this number until satisfactory statistics are obtained. Give a qualitative explanation of your results.

(b) Choose $t = 1$, $p_c = p_s$, and $\lambda = 0.05$ and compare your results with the analogous case considered in part (a).

(c) Repeat part (b) with $t = 2$ and $\lambda = 0.1$. Do the various probabilities depend on λ and t separately or only on their ratio? Answer this question before doing the simulation.

(d) Draw some typical paths of the neutrons. From the nature of these paths, explain the results in parts (a)–(c). For example, how does the number of scattering events change as the absorption probability changes? □

Problem 11.20. Inelastic neutron scattering

(a) In Problem 11.19 we assumed elastic scattering; that is, no energy is lost during scattering. Here we assume that some of the neutron energy E is lost, and that the mean free path is proportional to the speed and hence to $\sqrt{E}$. Modify your program so that a neutron loses a fraction f of its energy at each scattering event and assume that $\lambda = \sqrt{E}$. Consider $f = 0.05, 0.1$, and 0.5 and compare your results with those found in Problem 11.19a using the values for λ in Problem 11.19a to determine the initial values for E.

(b) Make a histogram for the path lengths between scattering events and plot the path length distribution function for $f = 0.1$, 0.5, and 0 (elastic scattering). □

This procedure for simulating neutron scattering and absorption is more computer intensive than necessary. Instead of considering a single neutron at a time, we can consider a collection of M neutrons. Then, instead of determining whether one neutron is captured or scattered, we determine the number that is captured and the number that is scattered. For example, at the first scattering site, p_cM of the neutrons are captured and p_sM are scattered. We also assume that all the scattered neutrons move in the same direction with the same path length, both of which are generated at random as before. At the next scattering site, there are p_s^2M scattered neutrons and p_sp_cM captured neutrons. At the end of m steps, the number of neutrons remaining is $w = p_s^mM$, and the number of captured neutrons is $(p_c + p_cp_s + p_cp_s^2 + \cdots + p_cp_s^{m-1})M$. If the new position at the mth step is at $z < 0$, we add w to the sum for the reflected neutrons; if $z > t$, we add w to the neutrons transmitted. When the neutrons are reflected or transmitted, we start over again at $z = 0$ with another collection of neutrons.

Problem 11.21. More efficient neutron scattering method

Apply the improved Monte Carlo method to neutron transmission through a plate. Repeat the simulations suggested in Problem 11.19 and compare your new and previous results. Also, compare the computational times for the two approaches to obtain comparable statistics. □

The power of the Monte Carlo method becomes apparent for more complicated geometries or when the material is spatially nonuniform so that the cross sections vary from point to point. A problem of current interest is the absorption of various forms of radiation in the human body.

Problem 11.22. Transmission through layered materials

Consider two plates with the same thickness $t = 1$ that are stacked on top of one another with no space between them. For one plate $p_c = p_s$, and for the other $p_c = 2p_s$; that is, the top plate is a better absorber. Assume that $\lambda = 1$ in both plates. Find the transmission, reflection, and absorption probabilities for elastic scattering. Does it matter which plate receives the incident neutrons? □

Appendix 11A: Error Estimates for Numerical Integration

We derive the dependence of the truncation error on the number of intervals for the numerical integration methods considered in Sections 11.1 and 11.3. These estimates are based on the assumed adequacy of the Taylor series expansion of the integrand $f(x)$:

$$f(x) = f(x_i) + f'(x_i)(x - x_i) + \frac{1}{2}f''(x_i)(x - x_i)^2 + \cdots \tag{11.64}$$

and the integration of (11.1) in the interval $x_i \le x \le x_{i+1}$:

$$\int_{x_i}^{x_{i+1}} f(x)\,dx = f(x_i)\Delta x + \frac{1}{2}f'(x_i)(\Delta x)^2 + \frac{1}{6}f''(x_i)(\Delta x)^3 + \cdots. \tag{11.65}$$

We first estimate the error associated with the rectangular approximation with $f(x)$ evaluated at the left side of each interval. The error Δ_i in the interval $[x_i, x_{i+1}]$ is the difference between (11.65) and the estimate $f(x_i)\Delta x$:

$$\Delta_i = \Big[\int_{x_i}^{x_{i+1}} f(x)\,dx\Big] - f(x_i)\Delta x \approx \frac{1}{2}f'(x_i)(\Delta x)^2. \tag{11.66}$$

We see that to leading order in Δx, the error in each interval is order $(\Delta x)^2$. Because there are a total of n intervals and $\Delta x = (b - a)/n$, the total error associated with the rectangular approximation is $n\Delta_i \sim n(\Delta x)^2 \sim n^{-1}$.

The estimated error associated with the trapezoidal approximation can be found in the same way. The error in the interval $[x_i, x_{i+1}]$ is the difference between the exact integral and the estimate $\frac{1}{2}[f(x_i) + f(x_{i+1})]\Delta x$:

$$\Delta_i = \Big[\int_{x_i}^{x_{i+1}} f(x)\,dx\Big] - \frac{1}{2}[f(x_i) + f(x_{i+1})]\Delta x. \tag{11.67}$$

If we use (11.65) to estimate the integral and (11.64) to estimate $f(x_{i+1})$ in (11.67), we find that the term proportional to f' cancels, and that the error associated with one interval is order

$(\Delta x)^3$. Hence, the total error in the interval $[a,b]$ associated with the trapezoidal approximation is order n^{-2}.

Because Simpson's rule is based on fitting $f(x)$ in the interval $[x_{i-1},x_{i+1}]$ to a parabola, error terms proportional to f'' cancel. We might expect that error terms of order $f'''(x_i)(\Delta x)^4$ contribute, but these terms cancel by virtue of their symmetry. Hence the $(\Delta x)^4$ term of the Taylor expansion of $f(x)$ is adequately represented by Simpson's rule. If we retain the $(\Delta x)^4$ term in the Taylor series of $f(x)$, we find that the error in the interval $[x_i,x_{i+1}]$ is of order $f''''(x_i)(\Delta x)^5$, and that the total error in the interval $[a,b]$ associated with Simpson's rule is $O(n^{-4})$.

The error estimates can be extended to two dimensions in a similar manner. The two-dimensional integral of $f(x,y)$ is the volume under the surface determined by $f(x,y)$. In the "rectangular" approximation, the integral is written as a sum of the volumes of parallelograms with cross sectional area $\Delta x\Delta y$ and a height determined by $f(x,y)$ at one corner. To determine the error, we expand $f(x,y)$ in a Taylor series:

$$f(x,y) = f(x_i,y_i) + \frac{\partial f(x_i,y_i)}{\partial x}(x - x_i) + \frac{\partial f(x_i,y_i)}{\partial y}(y - y_i) + \cdots, \tag{11.68}$$

and write the error as

$$\Delta_i = \Big[\int\!\!\int f(x,y)\,dx dy\Big] - f(x_i,y_i)\Delta x\,\Delta y. \tag{11.69}$$

If we substitute (11.68) into (11.69) and integrate each term, we find that the term proportional to f cancels and the integral of $(x-x_i)\,dx$ yields $\frac{1}{2}(\Delta x)^2$. The integral of this term with respect to dy gives another factor of Δy. The integral of the term proportional to $(y-y_i)$ yields a similar contribution. Because Δy is also order Δx, the error associated with the intervals $[x_i,x_{i+1}]$ and $[y_i,y_{i+1}]$ is to leading order in Δx:

$$\Delta_i \approx \frac{1}{2}[f_x'(x_i,y_i) + f_y'(x_i,y_i)](\Delta x)^3. \tag{11.70}$$

We see that the error associated with one parallelogram is order $(\Delta x)^3$. Because there are n parallelograms, the total error is order $n(\Delta x)^3$. However, in two dimensions, $n = A/(\Delta x)^2$, and hence the total error is order $n^{-1/2}$. In contrast, the total error in one dimension is order n^{-1}, as we saw earlier.

The corresponding error estimates for the two-dimensional generalizations of the trapezoidal approximation and Simpson's rule are order n^{-1} and n^{-2}, respectively. In general, if the error goes as order n^{-a} in one dimension, then the error in d dimensions goes as $n^{-a/d}$. In contrast, Monte Carlo errors vary as $n^{-1/2}$, independent of d. Hence, for large enough d, Monte Carlo integration methods will lead to smaller errors for the same choice of n.

Appendix 11B: The Standard Deviation of the Mean

In Section 11.4 we found empirically that the probable error associated with a single measurement consisting of n samples is $\sigma/\sqrt{n}$, where σ is the standard deviation associated with n data points. We derive this relation in the following. The quantity of experimental interest is denoted as x. Consider m sets of measurements each with n samples for a total of mn samples. For simplicity, we will assume that $n >> 1$. We use the index α to denote a particular measurement and the index i to designate the ith sample within a measurement. We denote $x_{\alpha,i}$ as sample i

in the measurement α. The value of a measurement is given by

$$M_\alpha = \frac{1}{n}\sum_{i=1}^{n} x_{\alpha,i}. \tag{11.71}$$

The mean $\overline{M}$ of the *total mn* individual samples is given by

$$\overline{M} = \frac{1}{m}\sum_{\alpha=1}^{m} M_\alpha = \frac{1}{mn}\sum_{\alpha=1}^{m}\sum_{i=1}^{n} x_{\alpha,i}. \tag{11.72}$$

The difference between measurement α and the mean of all the measurements is given by

$$e_\alpha = M_\alpha - \overline{M}. \tag{11.73}$$

We can write the variance of the means as

$$\sigma_m^2 = \frac{1}{m}\sum_{\alpha=1}^{m} {e_\alpha}^2. \tag{11.74}$$

We now wish to relate σ_m to the variance of the individual measurements. The discrepancy $d_{\alpha,i}$ between an individual sample $x_{\alpha,i}$ and the mean is given by

$$d_{\alpha,i} = x_{\alpha,i} - \overline{M}. \tag{11.75}$$

Hence, the variance σ^2 of the mn individual samples is

$$\sigma^2 = \frac{1}{mn}\sum_{\alpha=1}^{m}\sum_{i=1}^{n} {d_{\alpha,i}}^2. \tag{11.76}$$

We write

$$e_\alpha = M_\alpha - \overline{M} = \frac{1}{n}\sum_{i=1}^{m}\left(x_{\alpha,i} - \overline{M}\right) \tag{11.77a}$$

$$= \frac{1}{n}\sum_{i=1}^{n} d_{\alpha,i}. \tag{11.77b}$$

If we substitute (11.77b) into (11.74), we find

$$\sigma_m^2 = \frac{1}{m}\sum_{\alpha=1}^{m}\Big(\frac{1}{n}\sum_{i=1}^{m} d_{\alpha,i}\Big)\Big(\frac{1}{n}\sum_{j=1}^{m} d_{\alpha,j}\Big). \tag{11.78}$$

The sum in (11.78) over samples i and j in set α contains two kinds of terms—those with $i = j$ and those with $i \neq j$. We expect that $d_{\alpha,i}$ and $d_{\alpha,j}$ are independent and equally positive or negative on the average. Hence in the limit of a large number of measurements, we expect that only the terms with $i = j$ in (11.78) survive, and we write

$$\sigma_m^2 = \frac{1}{mn^2}\sum_{\alpha=1}^{m}\sum_{i=1}^{m} {d_{\alpha,i}}^2. \tag{11.79}$$

If we combine (11.79) with (11.76), we arrive at the result

$$\sigma_m^2 = \frac{\sigma^2}{n}. \tag{11.80}$$

We intepret σ_m as the error in the original n measurments because σ_m provides an estimate of how much an average over n measurments will deviate from the exact mean.

Appendix 11C: The Acceptance-Rejection Method

Although the inverse transform method discussed in Section 11.5 can be used in principle to generate any desired probability distribution, in practice the method is limited to functions for which the equation $r = P(x)$ can be solved analytically for x or by simple numerical approximation. Another method for generating nonuniform probability distributions is the *acceptance-rejection* method due to von Neumann. Suppose that $p(x)$ is the (normalized) probability density function that we wish to generate. For simplicity, we assume $p(x)$ is nonzero in the unit interval. Consider a positive definite *comparison function* $w(x)$ such that $w(x) > p(x)$ in the entire range of interest. A simple, although not generally optimum, choice of w is a constant greater than the maximum value of $p(x)$. Because the area under the curve $p(x)$ in the range x to $x + \Delta x$ is the probability of generating x in that range, we can follow a procedure similar to that used in the hit or miss method. Generate two numbers at random to define the location of a point in two dimensions which is distributed uniformly in the area under the comparison function $w(x)$. If this point is outside the area under $p(x)$, the point is rejected; if it lies inside the area, we accept it. This procedure implies that the accepted points are uniform in the area under the curve $p(x)$, and that their x values are distributed according to $p(x)$. One procedure for generating a uniform random point (x, y) under the comparison function $w(x)$ is as follows:

1. Choose a form of $w(x)$. One convenient choice would be to choose $w(x)$ such that the values of x distributed according to $w(x)$ can be generated by the inverse transform method. Let the total area under the curve $w(x)$ be equal to A.

2. Generate a uniform random number in the interval $[0, A]$ and use it to obtain a corresponding value of x distributed according to $w(x)$.

3. For the value of x generated in step 2, generate a uniform random number y in the interval $[0, w(x)]$. The point (x, y) is uniformly distributed in the area under the comparison function $w(x)$. If $y \le p(x)$, then accept x as a random number distributed according to $p(x)$.

Repeat steps 2 and 3 many times. Note that the acceptance-rejection method is efficient only if the comparison function $w(x)$ is close to $p(x)$ over the entire range of interest.

Appendix 11D: Polynomials and Interpolation

Interpolation is a technique that allows us to estimate a function within the range of a tabulated set of *sample points*. For example, Fourier analysis (see Chapter 9) generates a trigonometric series that can be evaluated between the points that are used to calculate the coefficients. We now describe how polynomials are implemented and used to interpolate between sample points.

A polynomial is a function that is expressed as

$$p(x) = \sum_{i=0}^{n} a_i x^i \tag{11.81}$$

where n is the *degree* of the polynomial and the n constants a_i are the *coefficients*. The evaluation of (11.81) as written is very inefficient because x is repeatedly multiplied by itself and the entire sum requires $\mathcal{O}(N^2)$ multiplications. A more efficient algorithm was published in 1819 by W.

Polynomial Methods	
`add(double a)`	Adds a scalar and returns a new polynomial.
`add(Polynomial p)`	Adds a polynomial and returns a new polynomial.
`deflate(double r)`	Reduces the degree by removing the given root `r`.
`derivative()`	Returns the derivative.
`integral(double a)`	Returns the integral having the value a at $x = 0$.
`roots(double tol)`	Gets the roots using Newton's method.
`subtract(double a)`	Subtracts a scalar and returns a new polynomial.
`subtract(Polynomial p)`	Subtracts a polynomial and returns a new polynomial.

Table 11.4: Some of the methods for manipulating polynomials in the Open Source Physics library.

G. Horner.[1] It uses a factored polynomial and requires only n multiplications and n additions and is known as *Horner's rule*. It is written as follows:

$$p(x) = a_0 + x\Big[a_1 + x\Big[a_2 + x[a_3 + \cdots]\Big]\Big]. \tag{11.82}$$

This algorithm can dramatically reduce processor time if large polynomials are repeatedly evaluated.

Polynomials are important computationally because most analytic functions can be approximated as a polynomial using a Taylor series expansion. Polynomials can be added, multiplied, integrated, and differentiated analytically and the result is still a polynomial. The `Polynomial` class in the numerics package implements many of these algebraic operations (see Table 11.4). Listing 11.5 shows how this class is used to calculate and display a polynomial's roots.

Listing 11.5: The `PolynomialApp` class tests the `Polynomial` class.

```java
package org.opensourcephysics.sip.ch11;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.display.*;
import org.opensourcephysics.frames.*;
import org.opensourcephysics.numerics.*;

public class PolynomialApp extends AbstractCalculation {
   PlotFrame plotFrame = new PlotFrame("x", "f(x)",
         "Polynomial Visualization");
   double xmin, xmax;
   Polynomial p;

   public void resetCalculation() {
      control.setValue("coefficients", "-2,0,1");
      control.setValue("xmin", -10);
      control.setValue("xmax", 10);
   }

   public void calculate() {
      xmin = control.getDouble("xmin");
      xmax = control.getDouble("xmax");
      String[] coefficients =
```

[1] This method of evaluating polynomials by factoring was already known to Newton.

```
            control.getString("coefficients").split(",");
      p = new Polynomial(coefficients);
      plotAndCalculateRoots();
   }

   void plotAndCalculateRoots() {
      plotFrame.clearDrawables();
      plotFrame.addDrawable(new FunctionDrawer(p));
      double[] range = Util.getDomain(p, xmin, xmax, 100);
      plotFrame.setPreferredMinMax(xmin, xmax, range[0], range[1]);
      plotFrame.repaint();
      double[] roots = p.roots(0.001);
      control.clearMessages();
      control.println("polynomial = "+p);
      for(int i = 0, n = roots.length;i<n;i++) {
         control.println("root = "+roots[i]);
      }
   }

   public void derivative() {
      p = p.derivative();
      plotAndCalculateRoots();
   }

   public static void main(String[] args) {
      CalculationControl control =
            CalculationControl.createApp(new PolynomialApp());
      control.addButton("derivative", "Derivative",
            "The derivative of the polynomial.");
   }
}
```

Exercise 11.23. Taylor series

Use the PolynomialApp class to plot the first three nonzero terms of the Taylor series expansion of the sine function. How accurate is this expansion in the interval $|x| < \pi/2$? □

Exercise 11.24. Polynomials

Write a test program to do the following:

(a) Create the polynomial $x^4 - 5x^3 + 5x^2 + 5x - 6$ and divide this polynomial by $x - 2$. Is 2 a root of the original polynomial?

(b) Find the roots of $x^5 - 6x^4 + x^3 - 7x^2 - 7x + 12$. □

Problem 11.25. Chebyshev polynomials

Orthogonal polynomials often can be written in terms of simple recurrence relations. For example, the Chebyshev polynomials of the first kind $T_n(x)$ can be written as

$$T_n(x) = 2xT_{n-1}(x) - T_{n-2}(x) \tag{11.83}$$

where $T_0(x) = 1$ and $T_1(x) = x$. Write and test a class using a static method that creates the Chebyshev polynomials. To improve efficiency, your class should store the polynomials as they are created during recursion. □

It is always possible to construct a polynomial that passes through a set of n data points (x_i, y_i) by creating a *Lagrange interpolating polynomial* as follows:

$$p(x) = \sum_{i=0}^{n} \frac{\prod_{i \neq j}(x - x_j)}{\prod_{i \neq j}(x_{\_}x_j)} y_i. \tag{11.84}$$

For example, three data points generate the second-degree polynomial (see (11.5)):

$$p(x) = \frac{(x-x_1)(x_0-x_2)}{(x_0-x_1)(x_0-x_2)} y_0 + \frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)} y_1 + \frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)} y_3. \tag{11.85}$$

Note that terms multiplying the y values are zero at the sample data points except for the term multiplying the sample data point's abscissa y_i. Various computational tricks can be used to speed the evaluation of (11.84), but these will not be discussed here (see Besset or Press et al.). We have implemented Lagrange's polynomial interpolation formula using a generalized Horner expansion in the LagrangeInterpolator class in the numerics package. Listing 11.6 tests this class.

Listing 11.6: The LagrangeInterpolatorApp class tests the LagrangeInterpolator class by sampling an arbitrary function and fitting the samples by a polynomial.

```java
package org.opensourcephysics.sip.ch11;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.display.*;
import org.opensourcephysics.frames.*;
import org.opensourcephysics.numerics.*;

public class LagrangeInterpolatorApp extends AbstractCalculation {
   PlotFrame plotFrame = new PlotFrame("x", "f(x)",
         "Legendre Interpolation");

   public void resetCalculation() {
      control.setValue("f(x)", "sin(x)");
      control.setValue("sample start", -2);
      control.setValue("sample stop", 2);
      control.setValue("n", 5);
      control.setValue("random y-error", 0);
      calculate();
   }

   public void calculate() {
      String fstring = control.getString("f(x)");
      double a = control.getDouble("sample start");
      double b = control.getDouble("sample stop");
      double err = control.getDouble("random y-error");
      int n = control.getInt("n"); // number of intervals
      double[] xData = new double[n];
      double[] yData = new double[n];
      double dx = (n>1) ? (b-a)/(n-1) : 0;
      Function f;
      try {
         f = new ParsedFunction(fstring);
      } catch(ParserException ex) {
         control.println(ex.getMessage());
```

```
            return;
         }
         plotFrame.clearData();
         double[] range = Util.getDomain(f, a, b, 100);
         plotFrame.setPreferredMinMax(a-(b-a)/4, b+(b-a)/4, range[0],
               range[1]);
         FunctionDrawer func = new FunctionDrawer(f);
         func.color = java.awt.Color.RED;
         plotFrame.addDrawable(func);
         double x = a;
         for(int i = 0;i<n;i++) {
            xData[i] = x;
            yData[i] = f.evaluate(x)*(1+err*(-0.5+Math.random()));
            plotFrame.append(0, xData[i], yData[i]);
            x += dx;
         }
         LagrangeInterpolator interpolator =
              new LagrangeInterpolator(xData, yData);
         plotFrame.addDrawable(new FunctionDrawer(interpolator));
         double[] coef = interpolator.getCoefficients();
         for(int i = 0;i<coef.length;i++) {
            control.println("c["+i+"]="+coef[i]);
         }
      }

      public static void main(String[] args) {
         CalculationControl.createApp(new LagrangeInterpolatorApp());
      }
   }
```

Problem 11.26. Lagrange interpolation

Use the LagrangeInterpolatorApp class to answer the following questions.

(a) How do the interpolating polynomial's coefficients compare to the series expansion coefficients of the sine and exponential functions?

(b) How well does an interpolating polynomial match a unit step function? You will need to write a step function class that implements the Function interface and thus contains an evaluate method.

(c) Do your answers depend on the number of sample points?

(d) Add random error to the sample data points for each function. How sensitive is the fit to random errors? □

Lagrange polynomials should be used cautiously. If the degree of the polynomial is high, the distance between points is large; or if the points are subject to experimental error, the resulting polynomial can oscillate wildly. Press et al. recommend that interpolating polynomials be small. If the data is accurate but the amount of data is large, we often use a polynomial constructed from a small number of nearest neighbors. *Cubic spline* interpolation uses polynomials in this way.

A cubic spline is a third-order polynomial that is required to have a continuous second derivative with neighboring splines at its end points. Because it would be inefficient to store a large number of Polynomial objects, the CubicSpline class in the numerics package stores the coefficients for the multiple polynomials needed to fit a data set in a single array. The CubicSplineApp program tests this class, but it is not shown here because it is similar to LagrangeInterpolatorApp.

Exercise 11.27. Cubic splines

Compare the cubic spline interpolating function to the Lagrange polynomial interpolating function using the same samples as were used in Problem 11.26. □

If the sample data is inaccurate, we often compute the coefficients for a polynomial of lower degree that passes as close as possible to the sample points. This fitting procedure is often used to construct an *ad hoc* function that describes the experimental data. The PolynomialLeastSquareFit class in the numerics package implements such a fitting algorithm (see Besset), and the PolynomialFitApp program tests this class. It is not shown because it is similar to LagrangeInterpolatorApp.

Exercise 11.28. Polynomial fitting

The PolynomialFitApp simulates experimental data from a particle trajectory near the Earth. How large a relative error in the y-values can be tolerated if we wish to determine the acceleration of gravity to within ten%? How does this answer change if the number of samples is increased by a factor of two? four? Discuss the effects of changing the the degree of the fitting polynomial. □

Suppose you are given a table of $y_i = f(x_i)$ and are asked to determine the value of x that corresponds to a given y. In other words, how do you find the inverse function $x = f^{-1}(x)$? An interpolation routine that does not require evenly spaced ordinates, such as the CubicSpline class, provides an easy and effective solution. The following code uses this technique to define the arcsin function.

Listing 11.7: The Arcsin class demonstrates how to use interpolation to define an inverse function.

```java
package org.opensourcephysics.sip.ch11;
import org.opensourcephysics.numerics.CubicSpline;
import org.opensourcephysics.numerics.Function;

public class Arcsin {
   static Function arcsin;

   // prohibit instantiation because all methods are static
   private Arcsin() {}

   static public double evaluate(double x) {
      if((x<-1)||(x>1)) {
         return Double.NaN;
      } else {
         return arcsin.evaluate(x);
      }
   }
```

```
    static { // creates a static function.
        int n = 10;
        double[] xd = new double[n];
        double[] yd = new double[n];
        double
            x = -Math.PI/2, dx = Math.PI/(n-1);
        for(int i = 0;i<n;i++) {
            xd[i] = x;
            yd[i] = Math.sin(x);
            x += dx;
        }
        arcsin = new CubicSpline(yd, xd);
    }
}
```

Problem 11.29. Inverse functions

(a) How accurate is the $\arcsin x$ function shown in Listing 11.7 in the interval $|x| < 0.5$?

(b) Compare the number of tabulated points needed to produce relative accuracies of $1 : 10^2$, $1 : 10^3$, and $1 : 10^4$ in the interval $-0.5 < x < 0.5$.

(c) Is polynomial interpolation more or less efficient than spline interpolation for evaluating inverse functions?

(d) Discuss the accuracy of the inverse interpolation of $\sin x$ if the interval is extended to $|x| \leq 1$. □

References and Suggestions for Further Reading

Forman S. Acton, *Numerical Methods That Work* (Harper & Row, 1970); corrected edition, Mathematical Association of America (1990). A delightful book on numerical methods.

Didier H. Besset, *Object-Oriented Implementation of Numerical Methods* (Morgan Kaufmann, 2001).

Isabel Beichl and Francis Sullivan, "The importance of importance sampling," Computing in Science and Engineering **1** (2), 71–73 (1999).

P. H. Borcherds, "Importance sampling: An illustrative introduction," Eur. J. Phys. **21**, 405–411 (2000).

Bradley Efron and Robert J. Tibshirani, *An Introduction to the Bootstrap* (Chapman and Hall, 1993).

Julio Fernández and Carlos Criado, "Algorithm for normal random numbers," Phys. Rev. E **60**, 3361–3365 (1999).

Steven E. Koonin and Dawn C. Meredith, *Computational Physics* (Addison–Wesley, 1990). Chapter 8 covers much of the same material on Monte Carlo methods as discussed in this chapter.

Malvin H. Kalos and Paula A. Whitlock, *Monte Carlo Methods, Vol. 1: Basics* (John Wiley & Sons, 1986). The authors are well-known experts on Monte Carlo methods.

William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery, *Numerical Recipes*, 2nd ed. (Cambridge University Press, 1992).

Reuven Y. Rubinstein, *Simulation and the Monte Carlo Method* John Wiley & Sons, 1981). An advanced, but clearly written treatment of Monte Carlo methods.

I. M. Sobol, *The Monte Carlo Method* (Mir Publishing, 1975). A very readable short text with excellent sections on nonuniform probability densities and the neutron transport problem.

Chapter 12

Percolation

Christian

We introduce several geometrical concepts associated with percolation, including the percolation threshold, clusters, and cluster finding algorithms. We also introduce the ideas of critical phenomena in the context of the percolation transition, including critical exponents, scaling relations, and the renormalization group.

12.1 Introduction

If a container is filled with small glass beads, and a battery is connected to the ends of the container, no current would pass and the system would be an insulator. Suppose that we choose a glass bead at random and replace it by a small steel ball. Clearly, the system would still be an insulator. If we continue randomly replacing glass beads with steel balls, eventually a current would pass. What percentage of steel balls is needed for the container to become a conductor? The change from the insulating to the conducting state that occurs as the percentage of steel balls is increased is an example of a *percolation phase transition*.

Another example of percolation is from the kitchen. Imagine a large metal sheet on which we randomly place drops of cookie dough. Assume that each drop of cookie dough spreads while the cookies are baking in an oven. If two cookies touch, they coalesce to form one cookie. If we are not careful, we might find a very large cookie that spans from one edge of the sheet to the opposite edge (see Figure 12.1). If such a spanning cookie exists, we say that there has been a percolation transition. As we will discuss in more detail, percolation has to do with *connectivity*.

Our discussion of percolation will require little background in physics, for example, no classical or quantum mechanics and little statistical physics. All that is required is some understanding of geometry and probability. Much of the appeal of percolation is its game-like aspects and intuitive simplicity. If you have a background in physics, this chapter will be more meaningful and can serve as an introduction to phase transitions and to important ideas such as scaling relations, critical exponents, and the renormalization group.

We first discuss a simple model of the cookie example to make the concept of percolation more explicit. We represent the cookie sheet by a lattice where each site can be in one of two states, occupied or empty. Each site is occupied independently of its neighbors with probability

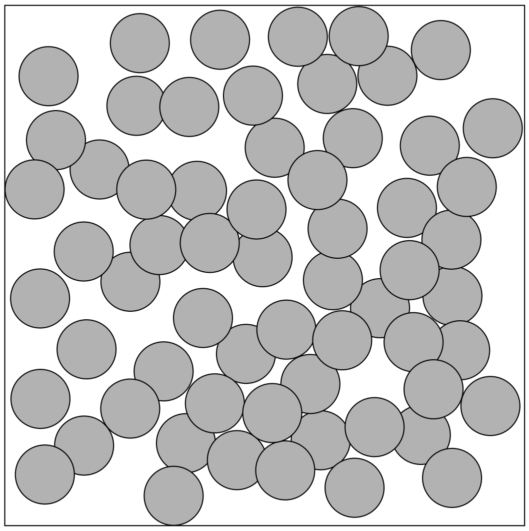


Figure 12.1: Cookies (circles) placed at random on a large sheet. Note that in this case there is a path of overlapping circles that connects the bottom and top edges of the cookie sheet. If such a path exists, we say that the cookies percolate, and there is a spanning path. See Problem 12.4e for a discussion of the algorithm used to generate this configuration.

p. This model of percolation is called *site* percolation. The occupied sites form *clusters*, which are groups of occupied nearest neighbor lattice sites (see Figure 12.2).

An easy way to study site percolation is to generate a uniform random number r in the unit interval $0 < r \leq 1$ for each site in the lattice. A site is occupied if its random number satisfies the condition $r \leq p$. If p is small, we expect that only small isolated clusters will be present (see Figure 12.3a). If p is near unity, we expect that most of the lattice will be occupied, and the occupied sites will form a large cluster that extends from one end of the lattice to the other (see Figure 12.3c). Such a cluster is said to be a *spanning cluster.* Because there is no spanning cluster for small p, and there is a spanning cluster for p near unity, there must be an intermediate value of p at which a spanning cluster first exists (see Figure 12.3b). We shall see that in the limit of an infinite lattice, there exists a well-defined threshold probability p_c such that:

For $p < p_c$, no spanning cluster exists and all clusters are finite.

For $p > p_c$, a spanning cluster exists.

For $p = p_c$, a spanning cluster exists with a probability greater than zero and less than unity.

We emphasize that the defining characteristic of percolation is *connectedness*. Because the connectedness exhibits a qualitative change at a well defined value of a continuous parameter, we shall see that the transition from a state with no spanning cluster to a state with a spanning cluster is an example of a *phase transition*.

An example of percolation that can easily be observed in the laboratory has been done with a wire mesh. Watson and Leath measured the electrical conductivity of a uniform metallic screen as the nodes connecting wire links were removed. The coordinates of the nodes to be removed were determined by a random number generator. The measured electrical conductivity is a

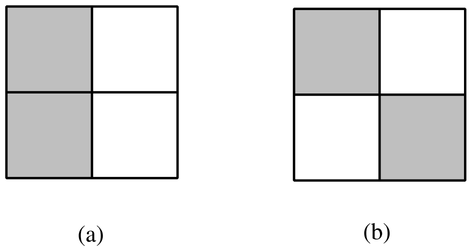


Figure 12.2: Example of a site percolation cluster on a square lattice of linear dimension $L = 2$. The two nearest neighbor occupied sites (shaded) in (a) are part of a cluster of size two; the two occupied sites in (b) are not nearest neighbor sites and do not belong to the same cluster; each occupied site is a cluster of size one.

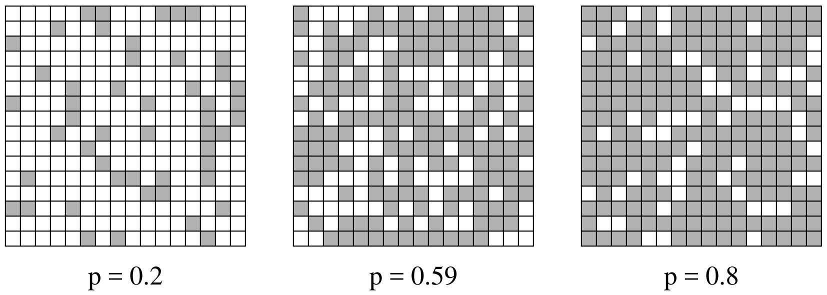


Figure 12.3: Examples of site percolation clusters on a square lattice of linear dimension $L = 16$ for $p = 0.2$, 0.59, and 0.8. On average, the fraction of occupied sites (shaded squares) is equal to p. Note that in this example, there exists a cluster that spans the lattice horizontally and vertically for $p = 0.59$.

rapidly decreasing function of the fraction of nodes p that are still present and vanishes below a critical threshold. A related conductivity measurement on a sheet of conducting paper with random holes has been performed (see Mehr et al.).

The applications of percolation phenomena go beyond metal-insulator transitions and the conductivity of wire mesh and include the spread of disease in a population, the behavior of magnets diluted by nonmagnetic impurities, the flow of oil through porous rock, the microstructure of fiber-reinforced concrete, and the characterization of gels. Percolation ideas have also been used to understand clusters in such diverse systems as granular matter and social networks. We concentrate on understanding several simple models of percolation that have an intuitive appeal of their own. Some of the applications of percolation phenomena are discussed in the references.

12.2 The Percolation Threshold

`PercolationApp` in Listing 12.1 generates site percolation configurations and initially shows the occupied sites as red squares of unit area. The state of each site is stored in `lattice`, which is an object of type `LatticeFrame`. Method `LatticeFrame.resizeLattice` is used to initialize the lattice to the desired size; `LatticeFrame.setAtIndex(site,value)` sets `site` to `value`. Although the lattice is two-dimensional, it is easier to represent the lattice as a one-dimensional array. (The site index at (x,y) is x + L*y.) An unoccupied site has value −2, and an occupied site that has not yet been assigned to a cluster has value −1. Values 0–6 are used to color the clusters. The InteractiveMouseHandler interface is used to allow the user to click on an occupied site. Then the colorCluster method iteratively colors all the sites in the associated cluster. That is, after a site is added to the cluster, we add it to the array sitesToTest so that we can test the site's neighbors for cluster membership. When sitesToTest is empty, all the possible sites have been added to the cluster.

Listing 12.1: The PercolationApp class.

```
package org.opensourcephysics.sip.ch12;
import java.awt.Color;
import java.awt.event.MouseEvent;
import java.util.Random;
import org.opensourcephysics.display.*;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.frames.LatticeFrame;

public class PercolationApp extends AbstractCalculation implements
     InteractiveMouseHandler {
   LatticeFrame lattice = new LatticeFrame("Percolation");
   Random random = new Random();
   int L;
   int clusterNumber; // used to color clusters

   public PercolationApp() {
      lattice.setInteractiveMouseHandler(this);
      // unoccupied sites are black and have value -2
      lattice.setIndexedColor(-2, Color.BLACK);
      // occupied sites that are not part of an identified cluster are red
      and have value -1
      lattice.setIndexedColor(-1, Color.RED);
      // following colors used to identify clusters when user clicks on
      // an occupied site
      lattice.setIndexedColor(0, Color.GREEN);
      lattice.setIndexedColor(1, Color.YELLOW);
      lattice.setIndexedColor(2, Color.BLUE);
      lattice.setIndexedColor(3, Color.CYAN);
      lattice.setIndexedColor(4, Color.MAGENTA);
      lattice.setIndexedColor(5, Color.PINK);
      // recycles cluster colors starting from green
      lattice.setIndexedColor(6, Color.LIGHT_GRAY);
   }

   // uses mouse click events to identify an occupied site
   // and identify its cluster
   public void handleMouseAction(InteractivePanel panel, MouseEvent evt) {
      panel.handleMouseAction(panel, evt);
      if(panel.getMouseAction()==InteractivePanel.MOUSE_PRESSED) {
         int site = lattice.indexFromPoint(panel.getMouseX(), panel.getMouseY());
         // test if a valid site was clicked (index non-negative),
         // and if site is occupied, but not yet cluster colored (value -1).
         if(site>=0&&lattice.getAtIndex(site)==-1) {
            // color cluster to which site belongs
            colorCluster(site);
            // cycle through 7 cluster colors
            clusterNumber = (clusterNumber+1)%7;
            lattice.repaint(// display lattice with colored cluster
         }
      }
   }

   // Occupies all sites with probability p
   public void calculate() {
      L = control.getInt("Lattice size");
      lattice.resizeLattice(L, L); // resize lattice
      // same seed will generate same set of random numbers
      random.setSeed(control.getInt("Random seed"));
      double p = control.getDouble("Site occupation probability");
      // occupy lattice sites with probability p
      for(int i = 0;i<L*L;i++) {
         lattice.setAtIndex(i, random.nextDouble()<p ? -1 : -2);
      }
      // first cluster will have color green (value 0)
      clusterNumber = 0;
   }

   // returns jth neighbor of site s, where j can be 0 (left),
   // 1 (right), 2 (down), or 3 (above). If no neighbor exists
   // because of boundary, return -1.
   // Change this method for periodic boundary conditions.
   int getNeighbor(int s, int j) {
      switch(j) {
      case 0 :                          // left
         if(s%L==0) {
            return -1;
         } else {
            return s-1;
         }
      case 1 :                          // right
         if(s%L==L-1) {
            return -1;
         } else {
            return s+1;
         }
      case 2 :                          // down
         if(s/L==0) {
            return -1;
         } else {
            return s-L;
         }
      case 3 :                          // above
         if(s/L==L-1) {
            return -1;
         } else {
            return s+L;
         }
      default :
         return -1;
      }
   }

   void colorCluster(int initialSite) { // color all sites in cluster
      // cluster sites whose neighbors have not yet been examined
      int[] sitesToTest = new int[L*L];
      int numSitesToTest = 0;    // number of sites in sitesToTest[]
      // color initialSite according to clusterNumber
      lattice.setAtIndex(initialSite, clusterNumber);
      // add initialSite to sitesToTest[]
      sitesToTest[numSitesToTest++] = initialSite;
      // grow cluster until numSitesToTest = 0
      while(numSitesToTest>0) {
      // get next site to test and remove it from list
         int site = sitesToTest[--numSitesToTest];
         for(int j = 0;j<4;j++) {       // visit four possible neighbors
            int neighborSite = getNeighbor(site, j);
            // test if neighborSite is occupied, and not yet added to cluster
            if(neighborSite>=0&&lattice.getAtIndex(neighborSite)==-1) {
               // color neighborSite according to clusterNumber
               lattice.setAtIndex(neighborSite, clusterNumber);
               // add neighborSite to sitesToTest[]
               sitesToTest[numSitesToTest++] = neighborSite;
            }
         }
      }
   }

   public void reset() {
      control.setValue("Lattice size", 32);
      control.setValue("Site occupation probability", 0.5927);
      control.setValue("Random seed", 100);
      calculate();
   }

   public static void main(String args[]) {
      CalculationControl control =
            CalculationControl.createApp(new PercolationApp());
   }
}
```

The percolation threshold p_c is defined as the site occupation probability p at which a spanning cluster first appears in an infinite lattice. However, for a finite lattice, there is a nonzero probability of a spanning cluster connecting one side of the lattice to the opposite side for any value of $p > 0$. For small p, this probability is order p^L (see Figure 12.4), and the probability of spanning goes to zero as L becomes large. Hence, for small p and sufficiently large L, only finite clusters exist.

For a finite lattice, the definition of spanning is arbitrary. For example, we can define a spanning cluster as one that (i) spans the lattice either horizontally or vertically; (ii) spans the lattice in a fixed direction, for example, vertically; or (iii) spans the lattice both horizontally and

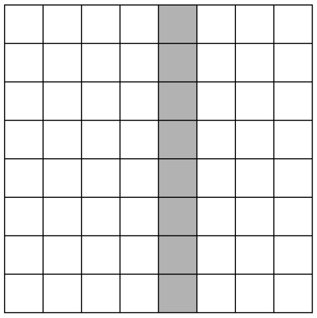


Figure 12.4: An example of a spanning cluster with a probability proportional to p^L on a $L = 8$ lattice. The probability of a spanning cluster with more sites will be proportional to a higher power of p.

vertically. These spanning rules are based on open (nonperiodic) boundary conditions, which we will use because the resulting clusters are easier to visualize and determine. The criterion for defining $p_c(L)$ for a finite lattice is also somewhat arbitrary. One possibility is to define $p_c(L)$ as the mean value of p at which a spanning cluster first appears. Another possibility is to define $p_c(L)$ as the value of p for which half of the configurations generated at random span the lattice. These criteria will lead to the same extrapolated value for p_c in the limit $L \to \infty$. In Problem 12.1 we will find an estimated value for $p_c(L)$ that is accurate to about 10%. A more sophisticated analysis discussed in Project 12.13 allows us to extrapolate our results for $p_c(L)$ to $L \to \infty$. In Project 12.17 we will discuss the use of periodic boundary conditions to define the clusters.

Problem 12.1. Site percolation on the square lattice

(a) Use `PercolationApp` to generate random site percolation configurations on a square lattice. Estimate $p_c(L)$ by finding the mean value of p at which a spanning cluster first occurs. For a given seed, the `calculate` method assigns a random number to each site and determines the occupancy of each site by comparing the sites's random number to p. Choose one of the spanning rules and begin with a value of p for which a spanning cluster is unlikely to be present. Then systematically increase p until you find a spanning cluster. Then choose a new seed and, hence, a new set of random numbers. Repeat this procedure for at least ten configurations and find the average value of $p_c(L)$. (Each configuration corresponds to a different set of random numbers.)

(b) Repeat part (a) for larger values of L. Is $p_c(L)$ better defined for larger L; that is, are the values of $p_c(L)$ spread over a smaller range of values? How quickly can you visually determine the existence of a spanning cluster? Describe your visual algorithm for determining if a spanning cluster exists.

(c) Choose $L \geq 1024$ and generate a configuration of sites at $p = p_c$. For this value of L, you won't be able to distinguish the individual sites. Click on the lattice until you generate some large clusters. Describe their visual appearance. For example, are they compact or ramified? □

The value of p_c depends on the symmetry of the lattice and on its dimension. In addition to the square lattice, the most common two-dimensional lattice is the triangular lattice. As discussed in Chapter 8, the essential difference between the square and triangular lattices is the number of nearest neighbors.

*__Problem 12.2.__ Site percolation on the triangular lattice

Modify `PercolationApp` to simulate random site percolation on a triangular lattice. Assume that a connected path connects the top and bottom sides of the lattice (see Figure 12.5). Do you expect p_c for the triangular lattice to be smaller or larger than the value of p_c for the square lattice? Estimate $p_c(L)$ for increasing values of L. Are your results for p_c consistent with your expectations? As we will discuss in the following, the exact value of p_c for the triangular lattice is $p_c = 1/2$.

In *bond* percolation each lattice site is occupied, but only a fraction of the sites have connections or occupied bonds between them and their nearest neighbor sites (see Figure 12.6). Each bond is either occupied with probability p or not occupied with probability $1-p$. A cluster is a group of sites connected by occupied bonds. The wire mesh described in Section 12.1 is an example of bond percolation if we imagine cutting the bonds between the nodes rather than removing the nodes themselves. An application of bond percolation to the description of gelation is discussed in Problem 12.3.

For bond percolation on the square lattice, the exact value of p_c can be obtained by introducing the *dual* lattice. The nodes of the dual lattice are the centers of the squares between the nodes in the original lattice (see Figure 12.7). The occupied bonds of the dual lattice are those that do not cross an occupied bond of the original lattice. Because every occupied bond on the dual lattice crosses exactly one unoccupied bond of the original lattice, the probability $\tilde{p}$ of an occupied bond on the dual lattice is $1-p$, where p is the probability of an occupied bond on the original lattice. If we assume that the dual lattice percolates if and only if the original lattice does not, and vice versa, then $p_c = 1-p_c$ or $p_c = 1/2$. This assumption holds for bond percolation on a square lattice because if a cluster in the original lattice spans in both directions, then because the occupied dual lattice bonds can only cross unoccupied bonds of the original lattice, the dual lattice clusters are blocked from spanning. An example is shown in Figure 12.7. This argument does not apply to cubic lattices in three dimensions, but it can be used for site percolation on a triangular lattice to yield $p_c = 1/2$.

*__Problem 12.3.__ Bond percolation on a square lattice

Suppose that all the lattice sites of a square lattice are occupied by monomers, each with functionality four; that is, each monomer can form a maximum of four bonds. This model is equivalent to bond percolation on a square lattice. Assume that the presence or absence of a bond between a given pair of monomers is random and is characterized by the probability p. For small p, the system consists of only finite polymers (groups of monomers) and the system is in the *sol* phase. For some threshold value p_c, there will be a single polymer that spans the lattice. We say that for $p \geq p_c$, the system is in the *gel* phase. How does a bowl of jello, an example of a gel, differ from a bowl of broth? Write a program to simulate bond percolation on a square lattice and determine the bond percolation threshold. Are your results consistent with the exact result $p_c = 1/2$?

We can also consider *continuum* percolation models. For example, we can place disks at random into a two-dimensional box. Two disks are in the same cluster if they touch or overlap. A typical continuum (off-lattice) percolation configuration is depicted in Figure 12.8. One quantity of interest is the quantity ϕ, the fraction of the area (volume in three dimensions) in the system that is covered by disks. In the limit of an infinite size box, it can be shown that

$$\phi = 1 - e^{-\rho \pi r^2} \tag{12.1}$$

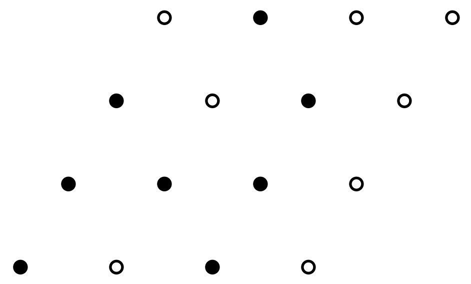


Figure 12.5: Example of a spanning site cluster on a $L = 4$ triangular lattice. The filled circles represent the occupied sites.

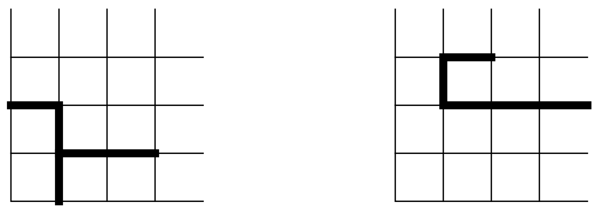


Figure 12.6: Two examples of bond clusters. The occupied bonds are shown as bold lines.

where ρ is the number of disks per unit area, and r is the radius of a disk (see Xia and Thorpe). Equation (12.1) is not accurate for small boxes because disks located near the edge of the box are a significant fraction of the total number of disks.

Problem 12.4. Continuum percolation

(a) Suppose that disks of unit diameter are placed at random on the sites of a square lattice with unit lattice spacing. Define ϕ as the area fraction covered by the disks. Convince yourself that $\phi_c = \pi p_c/4$.

(b) Modify `PercolationApp` to simulate continuum percolation. Instead of placing the disks on regular lattice sites, place their centers at random in a square box of area L^2. The relevant parameter is the density ρ, the number of disks per unit area, instead of the probability p. We can no longer use the `LatticeFrame` class. Instead, two arrays are needed to store the x and y locations of the disks. When the mouse is clicked on a disk, your program will need to determine which disk is at the location of the mouse, and then check all the other disks to see if they overlap or touch the disk you have chosen. This check is recursively continued for all overlapping disks. It is also useful to have an array that keeps track of the `clusterNumber` for each disk. Only disks that have not been assigned a cluster number need to be checked for overlaps.

(c) Estimate the value of the density ρ_c at which a spanning cluster first appears. Given this value of ρ_c, use a Monte Carlo method to estimate the corresponding area fraction ϕ_c (see Section 11.2). Choose points at random in the box and compute the fraction of points that lie within any disk. Explain why ϕ_c is larger for continuum percolation than it is for site percolation. Compare your direct Monte Carlo estimate of ϕ_c with the indirect value of ϕ_c obtained from (12.1) using the value of ρ_c. Explain any discrepancy.

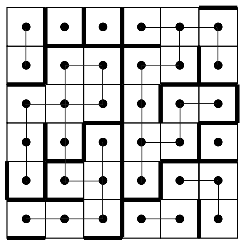


Figure 12.7: Occupied bonds on a bond percolation lattice are shown by heavy dark lines. The dual lattice consists of the open circles. The dashed lines are the occupied bonds on the dual lattice. The original lattice contains a cluster that spans both vertically and horizontally, which prevents the dual lattice from having a spanning cluster.

(d)* Consider the simple model of the cookie problem discussed in Section 12.1. Write a program that places disks at random into a square box and chooses their diameter randomly between 0 and 1. Estimate the value of ρ_c at which a spanning cluster first appears and compare its value to your estimate found in part (c)? Is your value for ϕ_c more or less than what was found in part (c)?

(e)* Another variation of the cookie problem is to place disks with unit diameter at random in a box with the constraint that the disks do not overlap. Continue to add disks until the fraction of successful attempts becomes less than 1%, that is, when one hundred successive attempts at adding a disk are not successful. Does a spanning cluster exist? If not, increase the diameters of all the disks at a constant rate (in analogy to the baking of the cookies) until a spanning cluster is attained. How does ϕ_c for this model compare with the value of ϕ_c found in part (d)? □

A continuum model that is applicable to random porous media is known as the *Swiss cheese* model. In this model the relevant quantity (the cheese) is the space between the disks. For the Swiss cheese model in two dimensions, the cheese area fraction at the percolation threshold, ψ_c, is given by $\psi_c = 1 - \phi_c$, where ϕ_c is the disk area fraction at the percolation threshold of the disks. Does such a relation hold in three dimensions (see Project 12.14)?

So far, we have emphasized the existence of the percolation threshold p_c and the appearance of a spanning cluster or path for $p \geq p_c$. Another quantity that characterizes percolation is $P_\infty(p)$, the probability that an occupied site belongs to the spanning cluster. The probability P_∞ is defined as

$$P_\infty(p) = \frac{\text{number of sites in the spanning cluster}}{\text{total number of occupied sites}}. \tag{12.2}$$

As an example, $P_\infty(p = 0.59) = 140/154$ for the single configuration shown in Figure 12.3b. An accurate estimate of P_∞ involves an average over many configurations for a given value of p. For an infinite lattice, $P_\infty(p) = 0$ for $p < p_c$ and $P_\infty(p) = 1$ for $p = 1$. Between p_c and 1, $P_\infty(p)$ increases monotonically.

More information can be obtained from the *cluster size distribution* $n_s(p)$ defined as

$$n_s(p) = \frac{\text{average number of clusters of size } s}{\text{total number of lattice sites}}. \tag{12.3}$$

For $p \geq p_c$, the spanning cluster is excluded from n_s. (For historical reasons, the *size* of a cluster refers to the *number* of sites in the cluster rather than to its spatial extent.) As an example, we

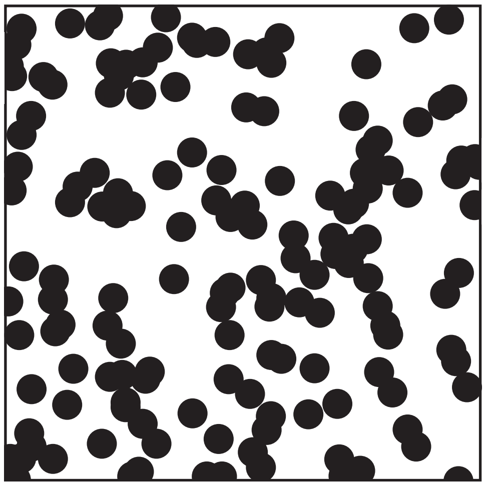


Figure 12.8: A model of continuum (off-lattice) percolation realized by placing disks of unit diameter at random into a square box of linear dimension L. If we concentrate on the voids between the disks rather than the disks, then this model of continuum percolation is known as the Swiss cheese model.

see from Figure 12.3a that $n_s(1) = 20/256$, $n_s(2) = 4/256$, $n_s(3) = 5/256$, and $n_s(7) = 1/256$ for $p = 0.2$ and is zero otherwise.

Because $N\sum_s sn_s$ is the total number of occupied sites (N is the total number of lattice sites), and Nsn_s is the number of occupied sites in clusters of size s, the quantity

$$w_s = \frac{sn_s}{\sum_s sn_s} \tag{12.4}$$

is the probability that an occupied site chosen at random is part of an s-site cluster. The *mean cluster size* S is defined as

$$S(p) = \sum_s sw_s = \frac{\sum_s s^2 n_s}{\sum_s sn_s}. \tag{12.5}$$

The sum in (12.5) is over finite clusters only. As an example, the weights corresponding to the clusters in Figure 12.3a are $w_s(1) = 20/50$, $w_s(2) = 8/50$, $w_s(3) = 15/50$, and $w_s(7) = 7/50$, and hence, $S = 130/50$.

Problem 12.5. Qualitative behavior of $n_s(p)$, $S(p)$, and $P_\infty(p)$

(a) Use PercolationApp to visually determine the cluster size distribution $n_s(p)$ for a square lattice with $L = 16$ and $p = 0.4$, $p = p_c$, and $p = 0.8$. Take $p_c = 0.5927$. Consider at least ten configurations for each value of p and average $n_s(p)$ over the configurations. For each value of p, plot n_s as a function of s and describe the observed s-dependence. Does n_s decrease more rapidly with increasing s for $p = p_c$ or for $p \neq p_c$? Plot $\ln n_s$ versus s and versus $\ln s$. Does either of these plots suggest the form of the s-dependence of n_s? Is there a qualitative change near p_c? You probably will not be able to obtain definitive answers to these questions at this point, but we will discuss a more quantitative approach later. Better results for n_s can also be found if periodic boundary conditions are used (see Project 12.17).

(b) Use the same configurations considered in part (a) to compute the mean cluster size S as a function of p. Remember that for $p > p_c$, the spanning cluster is excluded.

(c) Similarly, compute $P_\infty(p)$ for various values of $p \geq p_c$ and plot $P(p)$ as a function of p and discuss its qualitative behavior.

(d) Verify that $\sum_s s\,n_s(p) = p$ for $p < p_c$ and explain this relation. How is this relation modified for $p \geq p_c$? □

It is useful to associate a characteristic linear dimension or *connectedness length* $\xi(p)$ with the clusters. One way to do so is to define the *radius of gyration* R_s of a single cluster of s particles as

$$R_s^2 = \frac{1}{s}\sum_{i=1}^{s}(\mathbf{r}_i - \overline{\mathbf{r}})^2 \tag{12.6}$$

where

$$\overline{\mathbf{r}} = \frac{1}{s}\sum_{i=1}^{s}\mathbf{r}_i \tag{12.7}$$

and $\mathbf{r}_i$ is the position of the ith site in the same cluster. The quantity $\overline{\mathbf{r}}$ is the familiar definition of the center of mass of the cluster. From (12.6), we see that R_s is the root mean square radius of the cluster measured from its center of mass.

The connectedness length ξ can be defined as an average over the radii of gyration of all the finite clusters. To find the appropriate average for ξ, consider a site in a cluster of s sites. The site is connected to $s-1$ other sites, and the mean square distance to these sites is the order of R_s^2. The probability that a site belongs to a cluster of site s is $w_s = sn_s$. These considerations suggest that a reasonable definition of ξ is

$$\xi^2 = \frac{\sum_s (s-1)w_s\langle R_s^2\rangle}{\sum_s (s-1)w_s} \tag{12.8}$$

where $\langle R_s^2\rangle$ is the average of R_s^2 over all clusters of s sites. To simplify the expression for ξ, we write s instead of $s-1$ and let $w_s = sn_s$:

$$\xi^2 = \frac{\sum_s s^2\,n_s\langle R_s^2\rangle}{\sum_s s^2\,n_s}. \tag{12.9}$$

As before, the sum in (12.9) is over only nonspanning clusters.

Problem 12.6. Simple calculation of the connectedness length

To obtain a feel for how to compute the connectedness length ξ, calculate it for the configuration shown in Figure 12.3a for $p = 0.2$. □

12.3 Finding Clusters

So far we we have visually determined the clusters for a given configuration at a particular value of p. We now discuss an algorithm due to Newman and Ziff for finding clusters at many values of p. This algorithm is based on one that is well known in computer science in the context of the union-find problem. In the Newman–Ziff algorithm we begin with an empty lattice and keep track of the clusters as we randomly occupy new lattice sites. As each site is occupied, we determine whether it becomes a new cluster or whether it is a neighbor of an existing cluster (or clusters). Because $p = n/L^2$, where n is the number of occupied sites, p increases by $1/L^2$ each time we occupy a new site. The algorithm can be summarized as follows (see class `Clusters` in Listing 12.2).

1. *Precompute the occupation order*. One way to occupy the sites is to choose sites at random until we find an unoccupied site. However, this procedure will become inefficient when most of the sites are already occupied. Instead, we store the order in which the sites are to be occupied in `order[]` and generate the order by randomly permuting the integers from 0 to $N - 1$ in method `setOccupationOrder`. For example, `order[0]` = 2 means that we occupy site 2 first.

2. *Add sites according to predetermined order*. When a new site is added, we check all its neighbors to determine if the new site is an isolated cluster (all neighbors empty) or if it joins one or more existing clusters.

3. *Determine the clusters*. The clusters are organized in a tree-like structure, with one site of each cluster designated as the *root*. All sites in a given cluster, other than the root, point to another site in the same cluster, so that the root can be reached by recursively following the pointers. The "pointers"[1] are stored in the `parent` array. To join two clusters, we add a pointer from the root of the smaller cluster to the root of the larger one.

In the following example we use `order` = {2, 6, 8, 4, 5,...} to illustrate the method.

(i) Because `order[0]` = 2, we first occupy site 2 and set `parent[2]` = −1. The negative sign distinguishes site 2 as a root. The size of the cluster is stored as −`parent[root]`. In this case `-parent[2]` = 1, and because no other sites are occupied, there is no possibility of merging clusters.

(ii) For our example, `order[1]` = 6, and we initially set `parent[6]` = −1. We then consider the neighbor sites of site 6. Sites 5 (left), 7 (right), and 10 (up) are unoccupied (`parent[5]` = `parent[7]` = `parent[10]` = `EMPTY`), but site 2 (down) is occupied. Hence, we need to merge the two clusters, and we set `parent[6]` = 2 and `parent[2]` = −2. That is, the value of `parent[6]` points to site 2, and the value of `parent[2]` indicates that site 2 is the root of a cluster of size 2.

(iii) The next sites to be occupied are 8 and 4, as shown in Figure 12.9. These two sites form a size 2 cluster as before. We have `parent[8]` = −1 and then `parent[4]` = 8 and `parent[8]` = −2. We see that the value of each element of the `parent` array has three functions: nonroot occupied sites contain the index for the site's parent in the cluster tree; root sites are negative and equal to the negative of the number of sites in the cluster; and unoccupied sites have the value `EMPTY`.

(iv) We next add site 5 and set `parent[5]` = −1. From Figure 12.9 we see that we have to merge two clusters. We (arbitrarily) check the left neighbor of site 5 first, and hence, we first merge the cluster of size 1 at site 5 with the cluster at site 8 of size 2. Hence, we set `parent[5]` = 8 and `parent[8]` = −3. We next check the right neighbor of site 5 and find that we need to merge two clusters again with root sites at 8 and 2. Because the cluster at site 8 is bigger, we set `parent[2]` = 8 and `parent[8]` = −5.

[1]We use the term "pointer" as it is used by Newman and Ziff, that is, a link to an array index. A true pointer stores a memory address and does not exist in Java.

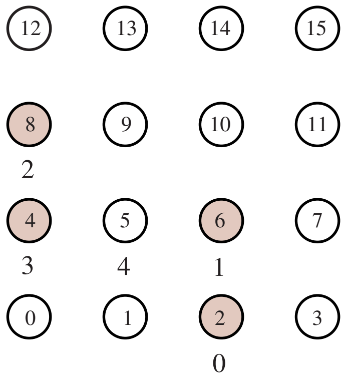


Figure 12.9: Illustration of the Newman–Ziff algorithm. The `order` array is given by $\{2,6,8,4,5,\ldots\}$; the number below a site denotes the order in which that site was occupied. When site 5 is occupied, we have to merge the two clusters as explained in the text.

Listing 12.2: Implementation of Newman–Ziff algorithm for identifying clusters.

```
package org.opensourcephysics.sip.ch12;
public class Clusters {
   static private final int EMPTY = Integer.MIN_VALUE;
   public int L;                        // linear dimension of lattice
   public int N;                        // N = L*L
   public int numSitesOccupied;         // number of occupied lattice sites
   public int[] numClusters;            // number of clusters of size s, n_s
   // secondClusterMoment stores sum{s^2 n_s}, where sum is over
   //  all clusters (not counting spanning cluster)
   // first cluster moment, sum{s n_s} equals numSitesOccupied
   // mean cluster size S is defined as
   // S = secondClusterMoment/numSitesOccupied

   private int secondClusterMoment;

   // spanningClusterSize, number of sites in a spanning cluster; 0 if
   // it doesn't exist. Assume at most one spanning cluster

   private int spanningClusterSize;
   // order[n] gives index of nth occupied site; contains all numbers
   // from [1...N], but in random order. For example, order[0] = 3 means
   // we will occupy site 3 first. An alternative to using order array is
   // to choose sites at random until we find an unoccupied site
   private int[] order;

   // parent[] array serves three purposes: stores cluster size
   // when site is root. Otherwise, it stores index of
   // the site's parent or is EMPTY. The root is found from an
   // occupied site by recursively following the parent array
   // Recursion terminates when we encounter a negative value in the
   // parent array, which indicates we have found the unique cluster
   //  root
   // if (parent[s] >= 0) parent[s] is parent site index
   // if (0 > parent[s] > EMPTY) s is root of size -parent[s]
   // if (parent[s] == EMPTY) site s is empty (unoccupied)
   private int[] parent;

   // A spanning cluster touches both left and right boundaries of the
   // lattice. As  clusters are merged, we maintain this information in
   //  following arrays at roots. For example, if root of a
   //  cluster is at site 7 and this cluster touches the left side,
   // then touchesLeft[7] == true
   private boolean[] touchesLeft, touchesRght;

   public Clusters(int L) {
      this.L = L;
      N = L*L;
      numClusters = new int[N+1];
      order = new int[N];
      parent = new int[N];
      touchesLeft = new boolean[N];
      touchesRght = new boolean[N];
   }

   public void newLattice() {
      setOccupationOrder(); // choose order in which sites are occupied
      // initially all sites are empty, and there are no clusters
      numSitesOccupied = secondClusterMoment = spanningClusterSize = 0;
      for(int s = 0;s<N;s++) {
         numClusters[s] = 0;
         parent[s] = EMPTY;
      }
      // initially left boundary touchesLeft
      // right boundary touchesRight
      for(int s = 0;s<N;s++) {
         touchesLeft[s] = (s%L==0);
         touchesRght[s] = (s%L==L-1);
      }
   }

   // adds site to lattice and updates clusters
   public void addRandomSite() {
      // if all sites are occupied, we can't add anymore
      if(numSitesOccupied==N) {
         return;
      }
      // newSite is index of random site to be occupied
      int newSite = order[numSitesOccupied++];
      // creates a new cluster containing only site newSite
      numClusters[1]++;
      secondClusterMoment++;
      // store new cluster's size in parent[]; negative sign
      // distinguishes newSite as a root, with a size value
      // Positive values correspond to nonroot sites with
      // index pointers
      parent[newSite] = -1;
```

```
      // merge newSite with occupied neighbors. root is index of
      //  merged cluster root at each step
      int root = newSite;
      for(int j = 0;j<4;j++) {
         // neighborSite is jth site neighboring newly added site newSite
         int neighborSite = getNeighbor(newSite, j);
         if((neighborSite!=EMPTY)&&(parent[neighborSite]!=EMPTY)) {
            root = mergeRoots(root, findRoot(neighborSite));
         }
      }
   }

   // gets size of  cluster to which site s belongs
   public int getClusterSize(int s) {
      return(parent[s]==EMPTY) ? 0 : -parent[findRoot(s)];
   }

   // returns size of spanning cluster if it exists, otherwise 0
   public int getSpanningClusterSize() {
      return spanningClusterSize;
   }

   // returns S (mean cluster size); sites belonging to spanning
   // cluster not counted in cluster moments
   public double getMeanClusterSize() {
      int spanSize = getSpanningClusterSize();
      // subtract sites in spanning cluster
      double correctedSecondMoment = secondClusterMoment-spanSize*spanSize;
      double correctedFirstMoment = numSitesOccupied-spanSize;
      if(correctedFirstMoment>0) {
         return correctedSecondMoment/correctedFirstMoment;
      } else {
         return 0;
      }
   }

   // given a site index s, returns site index representing the root
   // of cluster to which s belongs

   private int findRoot(int s) {
      if(parent[s]<0) {
         return s;    // root site (with size -parent[s])
      } else {
         // first link parent[s] to the cluster's root to improve performance
         // (path compression); then return this value
         parent[s] = findRoot(parent[s]);
      }
      return parent[s];
   }

   // returns jth neighbor of site s; j can be 0 (left), 1 (right),
   // 2 (down), or 3 (above). If no neighbor exists because of
   // boundary, return value EMPTY. Change this method for
```

```
// periodic boundary conditions
private int getNeighbor(int s, int j) {
   switch(j) {
   case 0 :
      return(s%L==0) ? EMPTY : s-1;    // left
   case 1 :
      return(s%L==L-1) ? EMPTY : s+1; // right
   case 2 :
      return(s/L==0) ? EMPTY : s-L;    // down
   case 3 :
      return(s/L==L-1) ? EMPTY : s+L; // above
   default :
      return EMPTY;
   }
}

// fills order[] array with random permutation of site indices
// First order[] is set to the identity permutation. Then for
// values of i in {1...N-1}, swap values of order[i] with
// order[r], where r is a random index in {i+1 ...N}
private void setOccupationOrder() {
   for(int s = 0;s<N;s++) {
      order[s] = s;
   }
   for(int s = 0;s<N-1;s++) {
      int r = s+(int) (Math.random()*(N-s));
      int temp = order[s];
      order[s] = order[r];
      order[r] = temp;
   }
}

// utility method to square an integer
private int sqr(int x) {
   return x*x;
}

// merges two root sites into one to represent cluster merging
// use heuristic that root of smaller cluster points to
// root of larger cluster to improve performance
// parent[root] stores negative cluster size
private int mergeRoots(int r1, int r2) {
   // clusters are uniquely identified by their root sites. If they
   // are the same, clusters are already merged, and we
   // need do nothing
   if(r1==r2) {
      return r1;
      // if r1 has smaller cluster size than r2, reverse (r1,r2) labels
   } else if(-parent[r1]<-parent[r2]) {
      return mergeRoots(r2, r1);
   } else { // (-parent[r1] > -parent[r2])
      // update cluster count, and second cluster moment to account
      // for loss of two small clusters and gain of
```

```
         // one bigger cluster
         numClusters[-parent[r1]]--;
         numClusters[-parent[r2]]--;
         numClusters[-parent[r1]-parent[r2]]++;
         secondClusterMoment += sqr(parent[r1]+parent[r2])
               -sqr(parent[r1])-sqr(parent[r2]);
         // cluster at r1 now includes sites of old cluster at r2
         parent[r1] += parent[r2];
         // make r1 new parent of r2
         parent[r2] = r1;
         // if r2 touched left or right, then so does merged cluster r1
         touchesLeft[r1] |= touchesLeft[r2];
         touchesRght[r1] |= touchesRght[r2];
         // if cluster at r1 spans lattice, then remember its size
         if(touchesLeft[r1]&&touchesRght[r1]) {
            spanningClusterSize = -parent[r1];
         }
         // return new root site r1
         return r1;
      }
   }
}
```

Listing 12.3: ClustersApp.

```
package org.opensourcephysics.sip.ch12;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.frames.*;

public class ClustersApp extends AbstractSimulation {
   Scalar2DFrame grid = new Scalar2DFrame("Newman-Ziff cluster algorithm");
   PlotFrame plot1 = new PlotFrame("p", "Mean Cluster Size",
         "Mean cluster size");
   PlotFrame plot2 = new PlotFrame("p", "P_?", "P_?");
   PlotFrame plot3 = new PlotFrame("p", "P_span", "P_span");
   PlotFrame plot4 = new PlotFrame("s", "<n_s>",
         "Cluster size distribution");
   Clusters lattice;
   double pDisplay;
   double[] meanClusterSize;
   double[] P_infinity;
   double[] P_span;           // probability of a spanning cluster
   double[] numClustersAccum; // number of clusters of size s
   int numberOfTrials;

   public void initialize() {
      int L = control.getInt("Lattice size L");
      grid.resizeGrid(L, L);
      lattice = new Clusters(L);
      pDisplay = control.getDouble("Display lattice at this value of p");
      grid.setMessage("p = "+pDisplay);
      plot4.setMessage("p = "+pDisplay);
      plot4.setLogScale(true, true);
      meanClusterSize = new double[L*L];
      P_infinity = new double[L*L];
      P_span = new double[L*L];
      numClustersAccum = new double[L*L+1];
      numberOfTrials = 0;
   }

   public void doStep() {
      control.clearMessages();
      control.println("Trial "+numberOfTrials);
      // adds sites to new cluster, and accumulate results
      lattice.newLattice();
      for(int i = 0;i<lattice.N;i++) {
         lattice.addRandomSite();
         meanClusterSize[i] += (double) lattice.getMeanClusterSize();
         P_infinity[i] += (double) lattice.getSpanningClusterSize()
                        /lattice.numSitesOccupied;
         P_span[i] += (lattice.getSpanningClusterSize()==0 ? 0 : 1);
         if((int) (pDisplay*lattice.N)==i) {
            for(int j = 0;j<lattice.N;j++) {
               numClustersAccum[j] += lattice.numClusters[j];
            }
            displayLattice();
         }
      }
      // display accumulated results
      numberOfTrials++;
      plotAverages();
   }

   private void plotAverages() {
      plot1.clearData();
      plot2.clearData();
      plot3.clearData();
      plot4.clearData();
      for(int i = 0;i<lattice.N;i++) {
         double p = (double) i/lattice.N; // occupation probability
         plot1.append(0, p, meanClusterSize[i]/numberOfTrials);
         plot2.append(0, p, P_infinity[i]/numberOfTrials);
         plot3.append(0, p, P_span[i]/numberOfTrials);
         if(numClustersAccum[i+1]>0) {
            plot4.append(0, i+1, numClustersAccum[i+1]/numberOfTrials);
         }
      }
   }

   private void displayLattice() {
      double display[] = new double[lattice.N];
      for(int s = 0;s<lattice.N;s++) {
         display[s] = lattice.getClusterSize(s);
      }
      grid.setAll(display);
   }
```

```
    public void reset() {
        control.setValue("Lattice size L", 128);
        control.setValue("Display lattice at this value of p", 0.5927);
    }

    public static void main(String args[]) {
        SimulationControl.createApp(new ClustersApp());
    }
}
```

Problem 12.7. Qualitative behavior of various percolation quantities

(a) Read the code for class `Clusters` and explain how the Newman–Ziff algorithm is implemented.

(b) Collect data for $P_\infty(p)$, the probability that an occupied site belongs to the spanning cluster, $S(p)$, the mean cluster size, and $P_{\text{span}}(p)$, the probability of a spanning cluster. Consider L = 8, 32, 128, and 256, and average over at least 100 configurations. How does the qualitative behavior of these quantities change with increasing L? Discuss the qualitative dependence of P_∞ and $S(p)$ on p for the largest lattice that you can simulate in a reasonable time.

(c) At what value of p is $P_{\text{span}} \approx 0.5$ for each value of L? Call this value $p_c(L)$. How strongly does $p_c(L)$ depend on L? Extrapolate your results for $p_c(L)$ to $L \to \infty$. For example, try fitting your data for $p_c(L)$ to the form $p_c(L) = p_c - cL^{-x}$, where p_c, c, and x are fitting parameters. Because you will likely have insufficient data to determine three parameters with reasonable accuracy, take $x = 3/4$ and plot $p_c(L)$ versus $L^{-3/4}$. How sensitive is your result for the intercept p_c on the assumed value of x? A more sophisticated analysis is discussed in Project 12.13.

(d) Consider the cluster distribution $n_s(p)$. Why is n_s a decreasing function of s? Does n_s decrease more quickly for $p = p_c$ or for $p \neq p_c$? Why is there so much scatter in n_s for large s? Plot $\ln n_s$ versus s and $\ln n_s$ versus $\ln s$ for each value of p. Which form fits best? Assume that a power law (straight line on a log-log plot) works for s less than some cutoff. Estimate the cutoff as a function of p and show that this cutoff diverges as $p \to p_c$. □

12.4 Critical Exponents and Finite Size Scaling

We are familiar with different phases of matter from our everyday experience. The most familiar example is water which can exist as a gas, liquid, or solid. It is well known that water changes from one phase to another at a well-defined temperature and pressure, for example, the transition from ice to liquid water occurs at 0 °C at atmospheric pressure. Such a change of phase is an example of a *thermodynamic phase transition.* Most substances also exhibit a *critical point*. For example, beyond a particular temperature and pressure, it is not possible to distinguish between the liquid and gaseous phases, and the phase boundary terminates.

Another example of a critical point occurs in magnetic systems at the Curie temperature T_c and zero magnetic field. We know that at low temperatures some substances such as iron exhibit ferromagnetism, a spontaneous magnetization in the absence of an external magnetic field. If we raise the temperature of a ferromagnet, the spontaneous magnetization decreases and vanishes continuously at a critical temperature T_c. For $T > T_c$, the system is a paramagnet. In Chapter 15 we will use Monte Carlo methods to investigate the behavior of a magnetic system near the magnetic critical point.

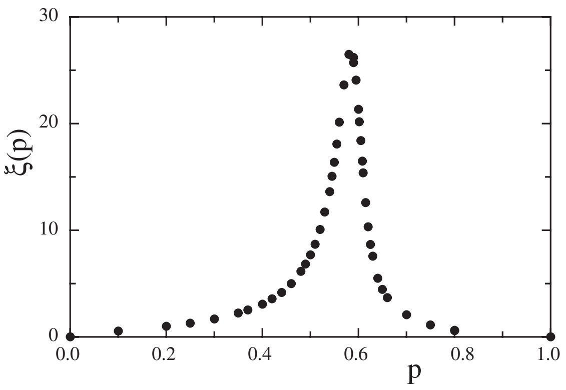


Figure 12.10: The qualitative p-dependence of the connectedness length $\xi(p)$ for a square lattice with $L = 128$. The results were averaged over approximately 2000–6000 configurations for each value of p. Note that ξ is finite for a finite lattice.

In the following, we will find that the properties of the *geometrical* phase transition in percolation are qualitatively similar to the properties of the critical point in thermodynamic transitions. We will see that in the vicinity of a critical point, the qualitative behavior of the system is governed by the occurrence of long-range correlations.

We have found that the essential physics near the percolation threshold is associated with the existence of large clusters. For example, for $p \neq p_c$, we found in Problem 12.7 that n_s decays rapidly with s. However for $p = p_c$, the s-dependence of n_s is qualitatively different, and n_s decreases much more slowly. This different behavior of n_s at $p = p_c$ is due to the presence of clusters of all length scales, for example, the "infinite" spanning cluster and the finite clusters of all sizes. In Figure 12.10 we show the mean connectedness length $\xi(p)$ for a lattice with $L = 128$. We see that ξ is finite, and an increasing function of p for $p < p_c$, and a decreasing function of p for $p > p_c$. Moreover, we know that $\xi(p = p_c)$ is approximately equal to L and hence, diverges as $L \to \infty$. These qualitative considerations lead us to conjecture that in the limit $L \to \infty$, $\xi(p)$ grows rapidly in the *critical region*, $|p - p_c| \ll 1$.

We can describe the quantitative behavior of $\xi(p)$ for p near p_c by introducing the *critical exponent* ν defined by the relation

$$\xi(p) \sim |p - p_c|^{-\nu}. \tag{12.10}$$

Of course, there is no *a priori* reason why the divergence of $\xi(p)$ can be characterized by a simple power law. Note that the exponent ν is assumed to be the same above and below p_c.

How do the other quantities that we have considered behave in the critical region in the limit $L \to \infty$? According to the definition (12.2) of P_∞, $P_\infty = 0$ for $p < p_c$ and is an increasing function of p for $p > p_c$. We conjecture that in the critical region, the increase of P_∞ with increasing p is characterized by the exponent β defined by the relation

$$P_\infty(p) \sim (p - p_c)^\beta. \tag{12.11}$$

Note that P_∞ is assumed to approach zero continuously as p approaches p_c from above; that is, the percolation transition is an example of a *continuous* phase transition. In the language of critical phenomena, P_∞ is an example of an *order parameter*; that is, P_∞ is nonzero in the ordered phase, $p > p_c$ and zero in the disordered phase $p < p_c$. We will see that at $p = p_c$, the spanning cluster is *fractal* and approaches zero density as the size of the system becomes larger.

Quantity	**Functional form**	**Exponent**	$d = 2$	$d = 3$
Percolation				
order parameter	$P_\infty \sim (p - p_c)^\beta$	β	5/36	0.41
mean size of finite clusters	$S(p) \sim \|p - p_c\|^{-\gamma}$	γ	43/18	1.80
connectedness length	$\xi(p) \sim \|p - p_c\|^{-\nu}$	ν	4/3	0.88
cluster numbers	$n_s \sim s^{-\tau}$ $(p = p_c)$	τ	187/91	2.19
Ising model				
order parameter	$M(T) \sim (T_c - T)^\beta$	β	1/8	0.32
susceptibility	$\chi(T) \sim \|T - T_c\|^{-\gamma}$	γ	7/4	1.24
correlation length	$\xi(T) \sim \|T - T_c\|^{-\nu}$	ν	1	0.63

Table 12.1: Several of the critical exponents for the percolation and magnetism phase transitions in $d = 2$ and $d = 3$ dimensions. Ratios of integers correspond to known exact results. The critical exponents for the Ising model are discussed in Chapter 15.

The mean number of sites in the finite clusters $S(p)$ also diverges in the critical region. Its critical behavior is written as

$$S(p) \sim |p - p_c|^{-\gamma}, \tag{12.12}$$

which defines the critical exponent γ. The common critical exponents for percolation are summarized in Table 12.1. The analogous critical exponents of a magnetic critical point are also shown.

Because we can simulate only finite lattices, a direct fit of the measured quantities ξ, P_∞, and $S(p)$ to their assumed critical behavior for an infinite lattice would not yield good estimates for the corresponding exponents ν, β, and γ (see Problem 12.8). The problem is that if p is close to p_c, the connectedness length of the largest cluster becomes comparable to L, and the nature of the clusters is affected by the finite size of the system. In contrast, for p far from p_c, $\xi(p)$ is small in comparison to L, and the measured values of ξ, and hence, the values of other physical quantities, are not appreciably affected by the finite size of the lattice. Hence, for $p \ll p_c$ and $p \gg p_c$, the properties of the system are indistinguishable from the corresponding properties of a truly macroscopic system ($L \to \infty$). However, if p is close to p_c, $\xi(p)$ is comparable to L and the nature of the system differs from that of an infinite system. In particular, a finite lattice cannot exhibit a true phase transition characterized by divergent physical quantities. Instead, ξ reaches a finite maximum at $p = p_c(L)$.

The effects of the finite system size can be made more quantitative by the following argument. Consider, for example, the critical behavior (12.11) of P_∞. If $\xi \gg 1$ but is much less than L, the power law behavior given by (12.11) is expected to hold. However, if ξ is comparable to L, ξ cannot change appreciably and (12.11) is no longer applicable. This qualitative change in the behavior of P_∞ and other physical quantities occurs for

$$\xi(p) \sim L \sim |p - p_c|^{-\nu}. \tag{12.13}$$

We invert (12.13) and write

$$|p - p_c| \sim L^{-1/\nu}. \tag{12.14}$$

The difference $|p - p_c|$ in (12.14) is the "distance" from the percolation threshold point at which finite size effects occur. Hence, if ξ and L are approximately the same size, we can replace (12.11) by the relation

$$P_\infty(p = p_c) \sim L^{-\beta/\nu}. \qquad (L \to \infty). \tag{12.15}$$

The relation (12.15) between P_∞ and L at $p = p_c$ is consistent with the fact that a phase transition is defined only for infinite systems.

One implication of (12.15) is that we can use it to determine the ratio β/ν. This method of analysis is known as *finite size scaling*. Suppose that we generate percolation configurations at $p = p_c$ for different values of L and analyze P_∞ as a function of L. If our values of L are sufficiently large, we can use the asymptotic relation (12.15) to estimate the ratio β/ν. A similar analysis can be used for $S(p)$ and other quantities of interest. We use this method in Problem 12.8.

Problem 12.8. Finite size scaling analysis of critical exponents

(a) Compute P_∞ at $p = p_c$ for at least 100 configurations for $L = 10$, 20, 40, and 80. Include in your average only those configurations that have a spanning cluster. Best results are obtained using the value of p_c for the infinite square lattice, $p_c \approx 0.5927$. Plot $\ln P_\infty$ versus $\ln L$ and estimate the ratio β/ν.

(b) Use finite size scaling to determine the dependence of the mean cluster size S on L at $p = p_c$. Average S over the same configurations considered in part (a). Remember that S is the mean number of sites in the nonspanning clusters.

(c) Find the size (number of particles) M in the spanning cluster at $p = p_c$ as a function of L. Use the same configurations as in part (a). Determine an exponent from the slope of a plot of $\ln M$ versus $\ln L$. This exponent is called the fractal dimension and is discussed in Chapter 13. □

Finite size scaling is particularly useful at the percolation threshold in comparison to thermal critical points where, as we will learn in Chapter 15, *critical slowing down* occurs. Critical slowing down makes it very time consuming to sample statistically independent configurations. No such slowing down occurs at the percolation threshold because we can easily create new configurations at any value of p by simply using a new set of random numbers.

We found in Section 12.2 that the numerical value of the percolation threshold p_c depends on the symmetry and dimension of the lattice, for example, $p_c \approx 0.5927$ for the square lattice and $p_c = 1/2$ for the triangular lattice. A remarkable feature of the power law dependencies summarized in Table 12.1 is that the values of the critical exponents do not depend on the symmetry of the lattice and are independent of the existence of the lattice itself, for example, they are identical for site percolation, bond percolation, and continuum percolation. Moreover, it is not necessary to distinguish between the exponents for site and bond percolation. In the vocabulary of critical phenomena, we say that site, bond, and continuum percolation all belong to the same *universality class* and that their critical exponents are identical for the same spatial dimension.

Another important idea in critical phenomena is the existence of relations between the critical exponents. An example of such a *scaling law* is

$$2\beta + \gamma = \nu d \tag{12.16}$$

where d is the spatial dimension of the lattice. The scaling law (12.16) indicates that the universality class depends on the spatial dimension. A more detailed discussion of finite size scaling and the scaling laws can be found in Chapter 15 and in the references.

12.5 The Renormalization Group

In Section 12.4 we studied the properties of various quantities on different length scales to determine the values of the critical exponents. The idea of examining physical quantities near the critical point on different length scales can be extended beyond finite size scaling and is the basis of the *renormalization group* method, one of the more important new methods in theoretical physics developed in the past several decades.[2] Although the method originated in the theory of elementary particles and was first applied to thermodynamic critical points, it is simpler to understand the method in the context of the percolation transition. We will find that the renormalization group method yields the critical exponents directly and in combination with Monte Carlo methods, is more powerful than Monte Carlo methods alone.

The basic idea of the renormalization group method is the following. Imagine a percolation configuration generated at $p = p_0$. What would happen if we average the configuration over groups of sites to obtain a configuration of occupied and empty cells? For example, the cells could be groups of four sites such that each cell is occupied or empty according to a mapping rule from the sites to the cell. If the original group of 2×2 sites spans, the cell would be occupied; otherwise, the cell would be empty. What value of $p = p_1$ would describe the new configuration of cells? If $p_0 < p_c$, we would expect $p_1 < p_0$. To understand why, consider a value of p_0 near $p = 0$ where almost all the clusters are of size one. Clearly, the occupied sites would be mapped into empty cells, and there would be a lower percentage of occupied cells than before. For $p_0 > p_c$ we would find $p_1 > p_0$ because the rare isolated unoccupied sites would be grouped into occupied cells. At $p = p_c$ we might expect that this blocking procedure would lead to configurations that look like they were generated at the same value of p because of the existence of clusters of all length scales.

Given the new configuration of cells at probability p_1, we can group the cells according to the same mapping rule, leading to a new $p = p_2$. The sequence p_0, p_1, p_2,... is called a renormalization group flow. We expect for $p_0 < p_c$ the flow will move to the trivial *fixed point* $p = 0$, and for $p > p_c$ the flow will move to the other trivial fixed point $p = 1$. At $p = p_c$, there is a *nontrivial* fixed point. We will see that by analyzing the renormalization group flow, we can determine the location of the critical point and the critical exponent ν.

We now consider a way of using a computer to change the configurations in a way that is similar to the procedure that we have just described. Consider a square lattice that is partitioned into *cells* or *blocks* that cover the lattice (see Figure 12.11). Note that we have defined the cells so that the new lattice of cells has the same symmetry as the original lattice. However, the replacement of sites by the cells has changed the length scale— all distances are now smaller by a factor of b, where b is the linear dimension of the cell. Hence, the effect of a "renormalization" is to replace each group of sites by a single renormalized site and to rescale the connectedness length for the renormalized lattice by a factor of b.

Because we want to preserve the main features of the original lattice and hence its connectedness (and its symmetry), we assume that a renormalized site is occupied if the original group of sites spans the cell. For simplicity, we adopt the vertical spanning criterion. The effect of performing a renormalization transformation on typical percolation configurations for p above and below p_c is illustrated in Figures 12.12 and 12.13, respectively. In both cases the effect of the successive transformations is to move the system away from p_c. We see that for $p = 0.7$, the effect of the transformations is to drive the system toward $p = 1$. For $p = 0.5$, the trend is to

[2]Kenneth Wilson was honored with the Nobel prize in physics in 1981 for his contributions to the development of the renormalization group method.

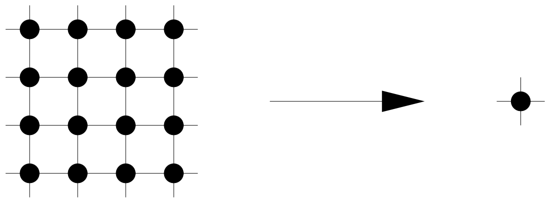


Figure 12.11: An example of a $b = 4$ cell used on the square lattice. The cell contains b^2 sites which are rescaled to a single supersite or cell after a renormalization group transformation.

drive the system toward $p = 0$. Because we began with a finite lattice, we cannot continue the renormalization transformation indefinitely.

The class RGApp implements a visual interpretation of the renormalization group. This class creates four windows with the original lattice in the first window and three renormalized lattices in the other three windows.

Listing 12.4: The visual renormalization group.

```java
package org.opensourcephysics.sip.ch12;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.frames.*;
import java.awt.Color;

public class RGApp extends AbstractCalculation {
   LatticeFrame originalLattice = new LatticeFrame("Original Lattice");
   LatticeFrame block1 = new LatticeFrame("First Blocked Lattice");
   LatticeFrame block2 = new LatticeFrame("Second Blocked Lattice");
   LatticeFrame block3 = new LatticeFrame("Third Blocked Lattice");

   public RGApp() {
      setLatticeColors(originalLattice);
      setLatticeColors(block1);
      setLatticeColors(block2);
      setLatticeColors(block3);
   }

   public void calculate() {
      int L = control.getInt("L");
      double p = control.getDouble("p");
      newLattice(L, p, originalLattice);
      block(originalLattice, block1, L/2); // block original lattice
      block(block1, block2, L/4);          // next blocking
      block(block2, block3, L/8);          // final blocking
      originalLattice.setVisible(true);
      block1.setVisible(true);
      block2.setVisible(true);
      block3.setVisible(true);
   }

   public void reset() {
      control.setValue("L", 64);
      control.setValue("p", 0.6);
```

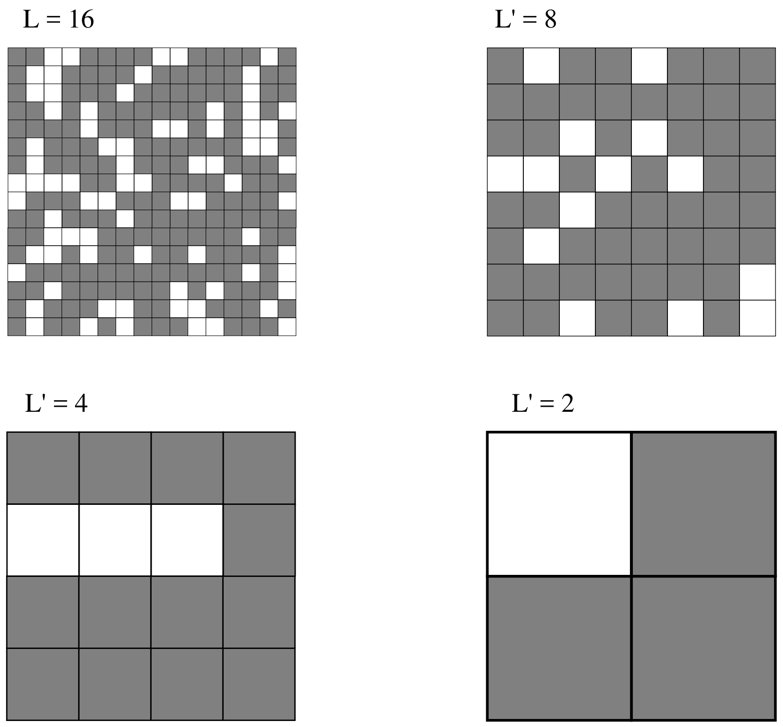


Figure 12.12: A percolation configuration generated at $p = 0.7$. The original configuration has been renormalized three times by transforming cells of four sites into one new supersite. What would be the effect of an additional transformation?

```
}

// new lattice
public void newLattice(int L, double p, LatticeFrame lattice) {
   lattice.resizeLattice(L, L);
   for(int i = 0;i<L;i++) {
      for(int j = 0;j<L;j++) {
         if(Math.random()<p) {
            lattice.setValue(i, j, 1);
         }
      }
   }
}

public void block(LatticeFrame lattice, LatticeFrame blockedLattice,
         int Lb) {
   blockedLattice.resizeLattice(Lb, Lb);
   for(int ib = 0;ib<Lb;ib++) {
      for(int jb = 0;jb<Lb;jb++) {
         int leftCellsProduct = lattice.getValue(2*ib, 2*jb)*
         lattice.getValue(2*ib, 2*jb+1);
         int rightCellsProduct = lattice.getValue(2*ib+1, 2*jb)*
         lattice.getValue(2*ib+1, 2*jb+1);
```

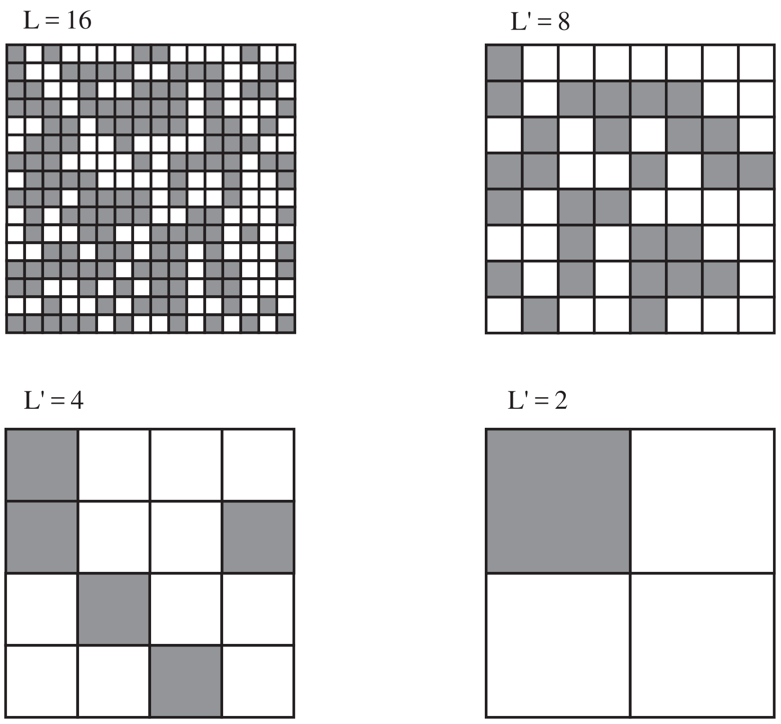


Figure 12.13: A percolation configuration generated at $p = 0.5$ (shaded cells are occupied). The original configuration has been renormalized three times by transforming blocks of four sites into one new site and rescaling all lengths by a factor of $b = 2$. What would be the effect of an additional transformation?

```
              if(leftCellsProduct==1||rightCellsProduct==1) {
                 // vertical spanning rule
                 blockedLattice.setValue(ib, jb, 1);
              }
           }
        }
     }

     public void setLatticeColors(LatticeFrame lattice) {
        lattice.setIndexedColor(0, Color.WHITE);
        lattice.setIndexedColor(1, Color.BLUE);
     }

     public static void main(String[] args) {
        CalculationControl.createApp(new RGApp());
     }
  }
```

Problem 12.9. Visual renormalization group

Use RGApp with $L = 64$ to estimate the value of the percolation threshold. For example, confirm that for small p, such as $p = 0.4$, the renormalized lattice almost always renormalizes to a

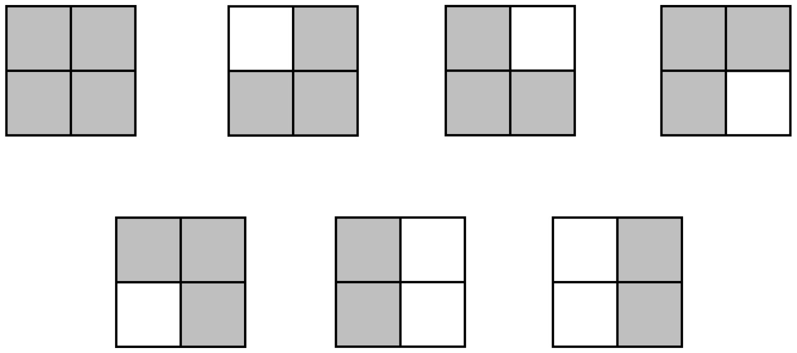


Figure 12.14: The seven (vertically) spanning configurations on a $b = 2$ cell.

nonspanning cluster. What happens for $p = 0.8$? How can you use the properties of the renormalized lattices to estimate p_c? □

Although a visual implementation of the renormalization group allows us to estimate p_c, it does not allow us to estimate the critical exponents. In the following, we present a renormalization group method that allows us to obtain p_c and the critical exponent ν associated with the connectedness length. This analysis follows closely the method presented by Reynolds et al.

We adopt the same procedure as before; that is, we replace the b^d sites within a cell of linear dimension b by a single site that represents whether or not the original lattice sites span the cell. The second step is to determine the parameters that specify the new renormalized configuration. We make the simple approximation that each cell is independent of all the other cells and is characterized only by the probability p' that the cell is occupied. The relation between p and p' reflects the fact that the basic physics of percolation is connectedness, because we define a cell to be occupied only if it contains a set of sites that span the cell. If the sites are occupied with probability p, then the cells are occupied with probability p', where p' is given by a *renormalization transformation* or a *recursion relation* of the form

$$p' = R(p). \tag{12.17}$$

The quantity $R(p)$ is the total probability that the sites form a spanning path.

An example will make the formal relation (12.17) more clear. In Figure 12.14, we show the seven vertically spanning site configurations for a $b = 2$ cell. The probability p' that the renormalized site is occupied is given by the sum of the probabilities of all the spanning configurations:

$$p' = R(p) = p^4 + 4p^3(1-p) + 2p^2(1-p)^2. \tag{12.18}$$

(Note that $q = 1 - p$ is the probability that a site is empty.) In general, the probability p' of the occupied renormalized sites is different than the occupation probability p of the original sites. For example, suppose that we begin with $p = p_0 = 0.5$. After a single renormalization transformation, the value of p' from (12.18) is $p_1 = p' = R(p_0 = 0.5) = 0.44$. If we perform a second renormalization transformation, we have $p_2 = R(p_1) = 0.35$. It is easy to see that further transformations drive the system to the fixed point $p = 0$. Similarly, if we begin with $p = p_0 = 0.7$, we find that successive transformations drive the system to the fixed point $p = 1$. This behavior is qualitatively similar to what we observed in the visual renormalization group.

To find the nontrivial fixed point associated with the critical threshold p_c, we need to find the special value of p such that

$$p^* = R(p^*). \tag{12.19}$$

For the recursion relation (12.18), we find that the solution of the fourth–degree equation for p^* yields the two trivial fixed points, $p^* = 0$ and $p^* = 1$, and the nontrivial fixed point $p^* = 0.61804$ which we associate with p_c. This calculated value of p^* for $b = 2$ should be compared with $p_c \approx 0.5927$.

To calculate the critical exponent ν, we recall that all lengths are reduced on the renormalized lattice by a factor of b in comparison to the lengths in the original system. Hence, the connectedness length transforms as

$$\xi' = \xi/b. \tag{12.20}$$

Because $\xi(p) = A|p - p_c|^{-\nu}$ for p near p_c, where A is a constant, we have

$$|p' - p^*|^{-\nu} = b^{-1}|p - p^*|^{-\nu} \tag{12.21}$$

where we have identified p_c with p^*. To find the relation between p' and p near p_c, we expand the renormalization transformation (12.17) in a Taylor series about p^* and obtain to first order in $(p - p^*)$

$$p' - p^* = R(p) - R(p^*) \approx \lambda(p - p^*) \tag{12.22}$$

where

$$\lambda = \frac{dR}{dp}\Big|_{p=p^*}. \tag{12.23}$$

We need to do a little algebra to obtain an explicit expression for ν. We first raise both sides of (12.22) to the νth power and write

$$|p' - p^*|^{\nu} = \lambda^{\nu}(p - p^*)^{\nu}. \tag{12.24}$$

We then compare (12.24) and (12.21) and obtain

$$b = \lambda^{\nu}. \tag{12.25}$$

Finally, we take the logarithm of both sides of (12.25) and obtain the desired relation for the critical exponent ν:

$$\nu = \frac{\log b}{\log \lambda}. \tag{12.26}$$

As an example, let us calculate λ for a square lattice with $b = 2$. We write (12.18) in the form $R(p) = -p^4 + 2p^2$. The derivative of $R(p)$ with respect to p yields $\lambda = 4p(1 - p^2) = 1.5279$ at $p = p^* = 0.61804$. We then use the relation (12.26) to obtain

$$\nu = \frac{\log 2}{\log 1.5279} \approx 1.635. \tag{12.27}$$

A comparison of (12.27) with the exact result $\nu = 4/3$ (see Table 12.1) for two dimensions shows reasonable agreement for such a simple calculation. (What would we be able to conclude if we were to measure $\xi(p)$ directly on a 2×2 lattice?)

Our calculation of ν does not give us an estimate of the error. What is the nature of our approximations? Our major assumption has been that the occupancy of each cell is independent of all other cells. This assumption is correct for the original sites, but after one renormalization,

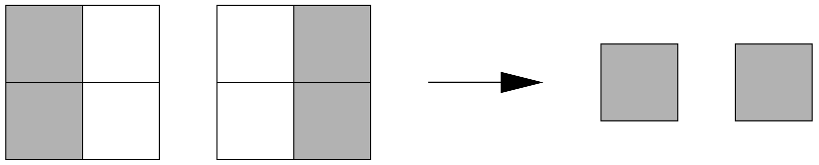


Figure 12.15: Example of the interface problem between cells. Two cells that are not connected at the original site level but that are connected at the cell level.

we lose some of the original connecting paths and gain connecting paths that are not present in the original lattice. An example of this interface problem is shown in Figure 12.15. Because this surface effect becomes less important with increasing cell size, one way to improve the renormalization group calculation is to consider larger cells. In Project 12.12 we consider a cell-to-cell method that does not require large cells and yields comparable accuracy.

Problem 12.10. Renormalization group method for small cells

(a) Enumerate the spanning configurations for a $b = 2$ cell assuming that a cell is occupied if a spanning path exists in either the vertical or the horizontal directions. Obtain the recursion relation and solve for the fixed point p^*. Use either a root finding algorithm or simple trial and error to find the value of $p = p^*$ such that $R(p) - p$ is zero. How do p^* and ν compare to their values using the vertical spanning criterion?

(b) Repeat the simple renormalization group calculation in part (a) using the criterion that a cell is occupied only if a spanning path exists in both directions.

(c)* The association of p_c with p^* is not the only possible one. Two alternatives involve the derivative $R'(p) = dR/dp$. For example, we could let $p_c = \int_0^1 pR'(p)\,dp$. Alternatively, we could choose $p_c = p_{\max}$, where $p_{\max}$ is the value of p at which $R'(p)$ has its maximum value. Compute p_c using these two alternative definitions and the various spanning criteria. In the limit of large cells, all three definitions should lead to the same values of p_c.

(d)* Enumerate the possible spanning configurations of a $b = 3$ cell, assuming that a cell is occupied if a cluster spans the cell vertically. Determine the probability of each configuration, and verify the renormalization transformation $R(p) = p^9 + 9p^8q + 36p^7q^2 + 67p^6q^3 + 59p^5q^4 + 22p^4q^5 + 3p^3q^6$, where $q = 1 - p$ is the probability of an empty site. Solve the recursion relation (12.19) for p^*. Use this value of p^* to find the slope λ and the exponent ν. Then assume a cell is occupied if a cluster spans the cell both vertically and horizontally and obtain $R(p)$. Determine $p^*(b = 3)$ and $\nu(b = 3)$ for the two spanning criteria. Are your results for p^* and ν closer to their known values than for $b = 2$ for the same spanning criteria? □

Problem 12.11. Renormalization group method for the triangular lattice

(a) For the triangular lattice, a cell can be formed by grouping three sites that form a triangle into one renormalized site. The only reasonable spanning criterion is that the cell spans if any two sites are occupied. Verify that $R(p) = p^3 + 3p^2(1 - p)$ and find $p_c = p^*$. How does p^* compare to the exact result $p_c = 1/2$?

(b) Calculate the critical exponent ν and compare its value with the exact result. Explain why b is given by $b^2 = 3$. Give a qualitative argument as to why the renormalization group argument might work better for small cells on a triangular lattice than on a square lattice. □

It is possible to improve our renormalization group results for p_c and ν by enumerating the spanning clusters for larger b. However, because the 2^N possible configurations for a $N = b^2$ cell increase rapidly with b, exact enumeration is not practical for $b > 7$, and we must use Monte Carlo methods if we wish to proceed further. Two Monte Carlo approaches are discussed in Project 12.13. The combination of Monte Carlo and renormalization group methods provides a powerful tool for obtaining information on phase transitions and other properties of materials.

As summarized in Table 12.1, the various critical exponents for percolation in two dimensions are known exactly. For example, the exponent ν, corresponding to the divergence of the connectedness length, is $\nu = 4/3$. It is interesting that the theory for this result is based on algebraic reasoning (too abstract to be summarized here), even though percolation is a geometrical phenomena. The most accurate estimate of p_c for the square lattice is $p_c = 0.59274621(13)$. We note that although there has been much work on percolation, only numerical estimates for p_c are known for most lattices.

12.6 Projects

Most of the following projects require larger systems and more computer resources than the problems that we have considered so far, but most are not much more difficult conceptually. More ideas for projects can be obtained from the references.

Project 12.12. Cell-to-cell renormalization group method

In Section 12.5 we discussed the cell-to-site renormalization group transformation for a system of cells of linear dimension b. An alternative transformation is to go from cells of linear dimension b_1 to cells of linear dimension b_2. For this cell-to-cell transformation, the rescaling length b_1/b_2 can be made close to unity. Many errors in a cell-to-cell renormalization group transformation cancel, resulting in a transformation that is more accurate in the limit in which the change in length scale is infinitesimal. We can use the fact that the connectedness lengths of the two systems are related by $\xi(p_2) = (b_1/b_2)^{-1}\xi(p_1)$ to derive the relation

$$\nu = \frac{\ln b_1/b_2}{\ln \lambda_1/\lambda_2} \tag{12.28}$$

where $\lambda_i = dR(p^*, b_i)/dp$ is evaluated at the solution to the fixed point equation, $R(b_2, p^*) = R(b_1, p^*)$. Note that (12.28) reduces to (12.26) for $b_2 = 1$. Use the results you found in Problem 12.10d for one of the spanning criteria to estimate ν from a $b_1 = 3$ to $b_2 = 2$ transformation. □

Project 12.13. Estimates for two-dimensional percolation

One way to estimate $R_L(p)$, the total probability of all the spanning clusters on a lattice of linear dimension L, can be understood by writing $R_L(p)$ in the form

$$R_L(p) = \sum_{n=1}^{N} S(n) P_N(n, p) \tag{12.29}$$

where

$$P_N(n, p) = \binom{N}{n} p^n q^{(N-n)} \tag{12.30}$$

and $N = L^2$. The binomial coefficient $\binom{N}{n} = N!/\big[(N-n)!\,n!\big]$ represents the number of possible configurations of n occupied sites and $N - n$ empty sites; $P_N(n, p)$ is the probability that n sites out of N are occupied with probability p. The quantity $S(n)$ is the probability that a random configuration of n occupied sites spans the lattice. A comparison of (12.18) and (12.29) shows that for $L = 2$ and the vertical spanning criterion, $S(1) = 0$, $S(2) = 2/6$, $S(3) = 1$, and $S(4) = 1$. What are the values of $S(n)$ for $L = 3$?

We can estimate the probability $S(n)$ by Monte Carlo methods. One way to sample $S(n)$ is to add a particle at random to an unoccupied site and check if a spanning cluster exists. If a spanning cluster does not exist, add another particle at random to a previously unoccupied site. If a spanning cluster exists after s particles are added, then let $S(n) = S(n) + 1$ for all $n \geq s$, and generate a new configuration. After a reasonable number of configurations, the results for $S(n)$ can be normalized. Of course, this procedure can be made more efficient by checking for a spanning cluster only after the total number of particles added is near $s \sim p_c N$.

(a) Write a Monte Carlo program to sample $S(n)$. Store the location of the unoccupied sites in a separate array. To check your program, first sample $S(n)$ for $L = 2$ and $L = 3$ and compare your results to the exact results for $S(n)$. Consider larger values of L and determine $S(n)$ for $L = 4$, 5, 8, 16, and 32. Because the number of sites in the lattice can become very large, the direct evaluation of the binomial coefficients using factorials is not possible. One way to proceed is to approximate the probability of a configuration of n occupied sites by a Gaussian:

$$P_N(n,p) \approx \binom{N}{n} p^n q^{(N-n)} \approx (2\pi Npq)^{-\frac{1}{2}}\, e^{-(n-pN)^2/2Npq}. \tag{12.31}$$

(b) As pointed out by Newman and Ziff, the Gaussian approximation for $P_N(n,p)$ is not sufficiently accurate for high precision studies. Instead, they used the following method. The binomial distribution is a maximum for a given N and p when $n = n_{\max} = pN$. Set this value to 1 for the moment. Then compute $P_N(n)$ iteratively for all other n using

$$P_N(n,p) = \begin{cases} P_N(N, n-1, p)\frac{N-n+1}{n}\frac{p}{1-p} & (n > n_{\max}) \\ P_N(N, n+1, p)\frac{n+1}{N-n}\frac{1-p}{p}. & (n < n_{\max}). \end{cases} \tag{12.32}$$

Then calculate the normalization coefficient $C = \sum_n P_N(n,p)$ and divide all the $P_N(n,p)$ by C to normalize the probability distribution.

(c) Compute ν from the cell-to-cell transformation discussed in Project 12.13 for $b_1 = 5$ and $b_2 = 4$.

(d) The article by Ziff and Newman discusses the convergence of various estimates of the percolation threshold in two dimensions. Some examples of these estimates include:

(i) The cell-to-site renormalization group fixed point:

$$R_L(p) = p \tag{12.33}$$

where p^* is the solution to (12.33).

(ii) The average value of p at which spanning first occurs:

$$\langle p\rangle = \int_0^1 p\frac{dR_L(p)}{dp}dp = 1 - \int_0^1 R_L(p)dp \tag{12.34}$$

where we have integrated by parts to obtain the second integral.

(iii) The estimate $p_{\max}$, which is the value of p at which dR_L/dp reaches a maximum:

$$\frac{d^2R_L(p)}{dp^2} = 0. \tag{12.35}$$

(iv) The cell-to-cell renormalization group fixed point:

$$R_L(p) = R_{L-1}(p) \tag{12.36a}$$

or

$$R_L(p) = R_{L/2}(p). \tag{12.36b}$$

(v) The value of p for which $R_L(p) = R_\infty(p_c)$. For a square lattice, $R_\infty(p_c) = 1/2$.

Verify that the various estimates of the percolation threshold converge to the infinite lattice value p_c either as

$$p_{\text{est}}(L) - p_c \approx cL^{-1/\nu} \tag{12.37a}$$

or

$$p_{\text{est}}(L) - p_c \approx cL^{-1-1/\nu} \tag{12.37b}$$

where the constant c is a fit parameter that depends on the criterion and $\nu = 4/3$ for percolation in two dimensions. Determine which estimates converge more quickly. □

Project 12.14. Percolation in three dimensions

(a) The value of p_c for site percolation on the simple cubic lattice is approximately 0.3112. Do a simulation to verify this value. Compute ϕ_c, the volume fraction occupied at p_c, if a sphere with a diameter equal to the lattice spacing is placed at each occupied site.

(b) Consider continuum percolation in three dimensions where spheres of unit diameter are placed at random in a cubical box of linear dimension L. Two spheres that overlap are in the same cluster. The volume fraction occupied by the spheres is given by

$$\phi = 1 - e^{-\rho 4\pi r^3/3} \tag{12.38}$$

where ρ is the number density of the spheres, and r is their radius. Write a program to simulate continuum percolation in three dimensions and find the percolation threshold ρ_c. Use the Monte Carlo procedure discussed in Problem 12.4 to estimate ϕ_c and compare its value with the value determined from (12.38). How does ϕ_c for continuum percolation compare with the value of ϕ_c found for site percolation in part (a)? Which do you expect to be larger and why?

(c) In the Swiss cheese model in three dimensions, we are concerned with the percolation of the space between the spheres. This model is appropriate for porous rock with the spheres representing solid material and the space between the spheres representing the pores. Because we need to compute the connectivity properties of the space between the spheres, we superimpose a regular grid with lattice spacing equal to $0.1r$ on the system, where r is the radius of the spheres. If a point on the grid is not within any sphere, it is "occupied." The use of the grid allows us to determine the connectivity between different regions of the pore space. Use a cluster labeling algorithm to label the clusters and determine $\tilde{\phi}_c$, the volume fraction occupied by the pores at threshold. You might be surprised to find that $\tilde{\phi}_c$ is relatively small. If time permits, use a finer grid and repeat the calculation to improve the accuracy of your results.

(d)* Use finite-size scaling to estimate the critical percolation exponents for the three models presented in parts (a)–(c). Are they the same within the accuracy of your calculation? □

Project 12.15. Fluctuations of the stock market

Although the fluctuations of the stock market are believed to be Gaussian for long time intervals, they are not Gaussian for short time intervals. The model of Cont and Bouchaud assumes that percolation clusters act as groups of traders who influence each other. The sites are occupied with probability p as usual. Each occupied site is a trader, and clusters are groups of traders (agents) who buy and sell together an amount proportional to the number s of traders in the cluster. At each time step, each cluster is independently active with probability $2p_a$ and is inactive with probability $1 - 2p_a$. If a cluster is active, it buys with probability p_b and sells with probability $p_s = 1 - p_b$. In the simplest version of the model the change in the price of a stock is proportional to the difference between supply and demand; that is,

$$R = \sum_{\text{buy}} s n_s - \sum_{\text{sell}} s n_s, \tag{12.39}$$

where the constant of proportionality is taken to be one. If the probability p_a is small, at most one cluster trades at a time, and the distribution $P(R)$ of relative price changes or "returns" scales as $n_s(p)$. In contrast, for large p_a, the relative price variation is the sum of many clusters (not counting the spanning cluster), and the central limit theorem implies that $P(R)$ converges to a Gaussian for large systems (except at $p = p_c$). Confirm these statements and find the shape of $P(R)$ for $p = p_c$ and $p_a = 0.25$. Variations of the Cont–Bouchaud model can be found in the references. The application of methods of statistical physics and simulations to economics and finance is now an active area of research and is commonly known as *econophysics*. □

Project 12.16. The connectedness length

(a) Modify class `Clusters` so that the connectedness length ξ defined in (12.9) is computed. One way to do so is to introduce four additional arrays, `xAccum`, `yAccum`, `xSquaredAccum`, and `ySquaredAccum`, with the data stored at indices corresponding to the root sites. We visit each occupied site in the lattice and determine its root site. For example, if the site x, y is occupied and its root is `root`, we set `xAccum[root] += x`, `xSquaredAccum[root] += x*x`, `yAccum[root] += y`, and `ySquaredAccum[root] += y*y`. Then R_s^2 for an individual cluster is given by

$$\begin{aligned} R_s^2 \;=\; & \texttt{xSquaredAccum[root]}/s + \texttt{ySquaredAccum[root]}/s \\ & - (\texttt{xAccum[root]}/s)^2 - (\texttt{xAccum/s[root]})^2 \end{aligned} \tag{12.40}$$

where s is the number of sites in the cluster, which is given by `-parent[root]`.

(b) What is the qualitative behavior of $\xi(p)$ as a function of p for different size lattices? Is $\xi(p)$ a monotonically increasing or decreasing function of p for $p < p_c$ and $p > p_c$? Remember that ξ does not include the spanning cluster. □

Project 12.17. Spanning clusters and periodic boundary conditions

For simplicity, we have used open boundary conditions, partly for historical reasons and partly because a spanning cluster is easier to visualize for open boundary conditions. An alternative

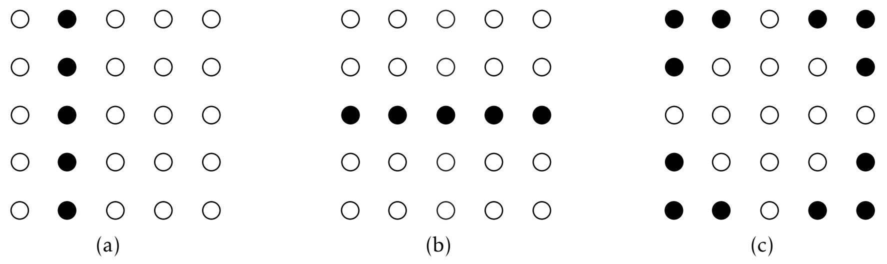


Figure 12.16: (a) Example of a cluster that wraps vertically. (b) Example of a cluster that wraps vertically and horizontally. (c) Example of a single cluster that does not wrap. Periodic boundary conditions are used in each case.

is to use periodic boundary conditions and define a spanning cluster as one that wraps all the way around the lattice (see Figure 12.16).

A method for detecting cluster wrapping has been proposed by Machta et al. In addition to the `parent` array introduced on page 457, we define two integer arrays that give the net displacement in the x and y direction of each site to its parent site. When we traverse a site's cluster tree, we sum these displacements to find the total displacement to the root site. When an added site neighbors two (or more) sites that belong to the same cluster, we compare the total displacements to the root site for these two sites. If these displacements differ by an amount that does not equal the minimum displacement between these two sites, then cluster wrapping has occurred (see Figure 12.17).

Modify the Newman–Ziff algorithm so that periodic boundary conditions are used to define the clusters and the existence of a spanning cluster. Use your program to estimate p_c and n_s and show that periodic boundary conditions give better results for the percolation threshold p_c and the cluster size distribution n_s for the same size lattice. □

Project 12.18. Conductivity in a random resistor network

(a) An important critical exponent for percolation is the conductivity exponent t defined by

$$\sigma \sim (p - p_c)^t \tag{12.41}$$

where σ is the conductance (or inverse resistance) per unit length in two dimensions. Consider bond percolation on a square lattice where each occupied bond between two neighboring sites is a resistor of unit resistance. Unoccupied bonds have infinite resistance. Because the total current into any node must equal zero by Kirchhoff's law, the voltage at any site (node) is equal to the average of the voltages of all nearest neighbor sites connected by resistors (occupied bonds). Because this relation for the voltage is the same as the algorithm for solving Laplace's equation on a lattice, the voltage at each site can be computed using the relaxation method discussed in Chapter 10. To compute the conductivity for a given $L \times L$ resistor network, we fix the voltage $V = 0$ at sites for which $x = 0$ and fix $V = 1$ at sites for which $x = L + 1$. In the y direction we use periodic boundary conditions. We then compute the voltage at all sites using the relaxation method. The current through each resistor connected to a site at $x = 0$ is $I = \Delta V/R = (V - 0)/1 = V$. The conductivity is the sum of the currents through all the resistors connected to $x = 0$ divided by L. In a similar way, the conductivity can be computed from the resistors attached to the $x = L + 1$ boundary. Write

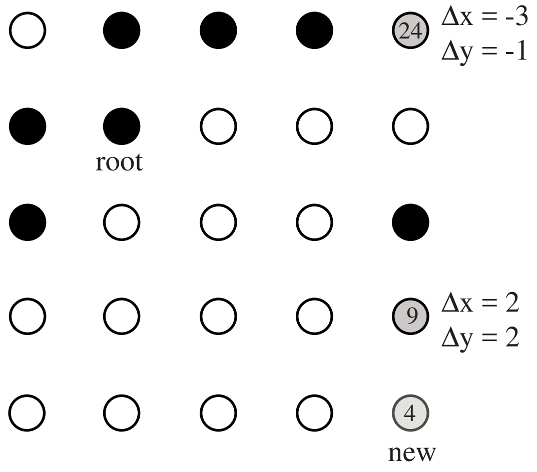


Figure 12.17: Example of cluster wrapping for periodic boundary conditions. When site 4 is occupied, it is a neighbor of sites 9 and 24 which belong to a single cluster. We compare the horizontal and vertical displacements of neighbors 9 and 24 to their root. If the difference between these displacements is not equal to the minimum displacement between them ($\Delta x_{\text{min}} = 0$, $\Delta y_{\text{min}} = 2$), then wrapping has occurred, as is the case here.

a program to implement the relaxation method for the conductivity of a random resistor network on a square lattice. An indirect, but easier way of computing the conductivity, is considered in Problem 13.8.

(b) The bond percolation threshold on a square lattice is $p_c = 0.5$. Use your program to compute the conductivity for a $L = 30$ square lattice. Average over at least ten spanning configurations for $p = 0.51, 0.52$, and 0.53. Note that you can eliminate all bonds that are not part of the spanning cluster and all occupied bonds connected to only one other occupied bond. Why? If possible, consider more values of p. Estimate the critical exponent t defined in (12.41).

(c) Fix p at $p = p_c = 1/2$ and use finite size scaling to estimate the conductivity exponent t.

(d)* Use larger lattices and the multigrid method (see Project 10.26) to improve your results. If you have sufficient computing resources, compute t for a simple cubic lattice for which $p_c \approx 0.249$. (In general, t is not the same for lattice and continuum percolation.) □

References and Suggestions for Further Reading

Joan Adler, "Series expansions," Computers in Physics **8**, 287–295 (1994). The critical exponents and the value of p_c can also be determined by doing exact enumeration.

I. Balberg, "Recent developments in continuum percolation," Phil. Mag. **56**, 991–1003 (1987). An earlier paper on continuum percolation is by Edward T. Gawlinski and H. Eugene Stanley "Continuum percolation in two dimensions: Monte Carlo tests of scaling and universality for non-interacting discs," J. Phys. A: Math. Gen. **14**, L291–L299 (1981). These workers divide the system into cells and use the Poisson distribution to place the appropriate number of disks in each cell.

Jean–Philippe Bouchaud and Marc Potters, *Theory of Financial Risk and Derivative Pricing: From Statistical Physics to Risk Management*, 2nd ed. (Cambridge University Press, 2003); Rosario N. Mantegna and H. Eugene Stanley, *An Introduction to Econophysics: Correlations and Complexity in Finance* (Cambridge University Press, 2000); Johannes Voit, *The Statistical Mechanics of Financial Markets*, 2nd ed. (Springer, 2004). These texts introduce the general field of econophysics.

Armin Bunde and Shlomo Havlin, editors, *Fractals and Disordered Systems*, revised edition (Springer-Verlag, 1996). Chapter 2 by the editors is on percolation.

R. Cont and J.-P. Bouchaud, "Herd behavior and aggregate fluctuations in financial markets," Macroeconomic Dynamics **4**, 170–196 (2000).

P. M. C. deOliveira, R. A. Nobrega, and D. Stauffer, "Are the tails of percolation thresholds Gaussians?," J. Phys. A **37**, 3743–3748 (2004). The authors compute the probability that there is a spanning cluster at $p = p_c$.

C. Domb, E. Stoll, and T. Schneider, "Percolation clusters," Contemp. Phys. **21**, 577–592 (1980). This review paper discusses the nature of the percolation transition using illustrations from a film of a Monte Carlo simulation of a percolation process.

J. W. Essam, "Percolation theory," Reports Progress Physics **53**, 833–912 (1980). A mathematically oriented review paper.

Jens Feder, *Fractals* (Plenum Press, 1988). See Chapter 7 on percolation. We discuss the fractal properties of the spanning cluster at the percolation threshold in Chapter 13.

J. P. Fitzpatrick, R. B. Malt, and F. Spaepen, "Percolation theory of the conductivity of random close-packed mixtures of hard spheres," Phys. Lett. A **47**, 207–208 (1974). The authors describe a demonstration experiment done in a first year physics course.

J. Hoshen and R. Kopelman, "Percolation and cluster distribution. I. Cluster multiple labeling technique and critical concentration algorithm," Phys. Rev. B **14**, 3438–3445 (1976). The original paper on an efficient cluster labeling algorithm. The Hoshen–Kopelman algorithm is well suited for very large lattices in two dimensions, but, in general, the Newman–Ziff algorithm is easier to use.

Chin-Kun Hu, Chi-Ning Chen, and F. Y. Wu, "Histogram Monte Carlo position-space renormalization group: Applications to site percolation," J. Stat. Phys. **82**, 1199–1206 (1996). The authors use a histogram Monte Carlo method that is similar to the method discussed in Project 12.13. A similar Monte Carlo method was used by M. Ahsan Khan, Harvey Gould, and J. Chalupa, "Monte Carlo renormalization group study of bootstrap percolation," J. Phys. C **18**, L223–L228 (1985).

J. Machta, Y. S. Choi, A. Lucke, T. Schweizer, and L. M. Chayes, "Invaded cluster algorithm for Potts models," Phys. Rev. E **54**, 1332–1345 (1996). The authors discuss the definition of a spanning cluster for periodic boundary conditions.

P. H. L. Martins and J. A. Plascak, "Percolation on two- and three-dimensional lattices," Phys. Rev. E **67**, 046119-1–6 (2003). The authors use the Newman–Ziff algorithm to compute various quantities.

Ramit Mehr, Tal Grossman, N. Kristianpoller, and Yuval Gefen,"Simple percolation experiment in two dimensions," Am. J. Phys. **54**, 271–273 (1986). The authors discuss a simple experiment on a sheet of conducting silver paper. This type of experiment is much easier to do than the insulator-conductor transition discussed in Section 12.1. In the latter case, the results are difficult to interpret because the current depends on the contact area between two spheres and thus on the applied pressure.

M. E. J. Newman and R. M. Ziff, "Fast Monte Carlo algorithm for site or bond percolation," Phys. Rev. E **64**, 016706-1–16 (2001). Our discussion of the Newman–Ziff algorithm in Section 12.3 closely follows this well-written paper.

Peter J. Reynolds, H. Eugene Stanley, and W. Klein, "Large-cell Monte Carlo renormalization group for percolation," Phys. Rev. B **21**, 1223 (1980). Another especially well written research paper. Our discussion on the renormalization group in Section 12.5 is based upon this paper.

Muhammad Sahimi, *Applications of Percolation Theory* (Taylor & Francis, 1994). The emphasis is on modeling various phenomena in disordered media.

Lev N. Shchur, "Incipient spanning clusters in square and cubic percolation," in *Studies in Condensed Matter Physics*, Vol. 85, edited by D. P. Landau, S. P. Lewis, and H. B. Schuettler (Springer–Verlag, 2000). Not many years ago, it was commonly believed that only one spanning cluster could exist at the percolation threshold. In this paper the probability of the simultaneous occurrence of at least k spanning clusters was studied by extensive Monte Carlo simulations and found to be in agreement with theoretical predictions.

Dietrich Stauffer, "Percolation models of financial market dynamics," Advances in Complex Systems **4**, 19–27 (2001).

D. Stauffer, "Percolation clusters as teaching aid for Monte Carlo simulation and critical exponents," Am. J. Phys. **45**, 1001–1002 (1977); D. Stauffer, "Scaling theory of percolation clusters," Physics Reports **54**, 1–74 (1979).

Dietrich Stauffer and Amnon Aharony, *Introduction to Percolation Theory*, 2nd ed. (Taylor & Francis, 1994). A delightful book by two of the leading workers in the field. An efficient Fortran implementation of the Hoshen-Kopelman algorithm is given in Appendix A.3.

B. P. Watson and P. L. Leath, "Conductivity in the two-dimensional site percolation problem," Phys. Rev. B **9**, 4893–4896 (1974). A research paper on the conductivity of chicken wire.

John C. Wierman and Dora Passen Naor, "Criteria for evaluation of universal formulas for percolation thresholds," Phys. Rev. E **71**, 036143-1–7 (2005). Wierman and Naor evaluate several universal formulas that predict approximate values for p_c for various lattices. Percolation was first conceived by chemistry Nobel laureate P. J. Flory as a model for polymer gelation, for example, the sol-gel transition of jello [P. J. Flory, "Molecular size distribution in three dimensional polymers," J. Am. Chem. Soc. **63**, 3083–3100 (1941)]. Further important work in this context was done by another chemist, Walter H. Stockmayer. The term "percolation" was coined by mathematicians Broadbent and Hammersley in 1957 who considered percolation on a lattice for the first time. See S. R. Broadbent and J. M. Hammersley, "Percolation processes I. Crystals and mazes," Proceedings Cambridge Philosophical Society **53**, 629–641 (1957).

Kenneth G. Wilson, "Problems in physics with many scales of length," Sci. Am. **241** (8), 158–179 (1979). An accessible article on the renormalization group method and its applications in particle and condensed matter physics. See also K. G. Wilson, "The renormalization group and critical phenomena," Rev. Mod. Phys. **55**, 583–600 (1983). The latter article is the text of Wilson's lecture on the occasion of the presentation of the 1982 Nobel Prize in Physics. In this lecture he claims that he "… found it very helpful to demand that a correctly formulated field theory be soluble by computer, the same way an ordinary differential equation can be solved on a computer …" .

W. Xia and M. F. Thorpe, "Percolation properties of random ellipses," Phys. Rev. A **38**, 2650–2656 (1988). The authors consider continuum percolation and show that the area fraction remaining after punching out holes at random is given by $\phi = e^{-A\rho}$, where A is the area of a hole and ρ is the number density of the holes. This relation does not depend on the shape of the holes.

Richard Zallen, *The Physics of Amorphous Solids* (Wiley–Interscience, 1983). Chapter 4 discusses many of the applications of percolation concepts to realistic systems.

R. M. Ziff and M. E. J. Newman, "Convergence of threshold estimates for two-dimensional percolation," Phys. Rev. E **66**, 016129-1–10 (2002).

Chapter 13

Fractals and Kinetic Growth Models

We introduce the concept of fractal dimension and discuss several processes that generate fractal objects.

13.1 The Fractal Dimension

One of the more interesting geometrical properties of objects is their shape. As an example, we show in Figure 13.1 a spanning cluster generated at the percolation threshold. Although the visual description of such a cluster is subjective, such a cluster can be described as ramified, airy, tenuous, and stringy, rather than compact or space-filling.

In the 1970s a new *fractal* geometry was developed by Mandelbrot and others to describe the characteristics of ramified objects. One quantitative measure of the structure of these objects is their *fractal dimension* D. To define D, we first review some simple ideas of dimension in ordinary Euclidean geometry. Consider a circular or spherical object of mass M and radius R. If the radius of the object is increased from R to $2R$, the mass of the object is increased by a factor of 2^2 if the object is circular or by 2^3 if the object is spherical. We can express this relation between mass and the radius or a characteristic length as

$$M(R) \sim R^D \qquad \text{(mass dimension)}, \tag{13.1}$$

where D is the dimension of the object. Equation (13.1) implies that if the linear dimensions of an object are increased by a factor of b while preserving its shape, then the mass of the object is increased by b^D. This mass-length scaling relation is closely related to our intuitive understanding of spatial dimension.

If the dimension of the object D and the dimension of the Euclidean space in which the object is embedded d are identical, then the mass density $\rho = M/R^d$ scales as

$$\rho(R) \propto M(R)/R^d \sim R^0; \tag{13.2}$$

that is, its density is constant. An example of a two-dimensional object is shown in Figure 13.2. An object whose mass-length relation satisfies (13.1) with $D = d$ is said to be *compact*.

Equation (13.1) can be generalized to define the fractal dimension. We denote objects as fractals if they satisfy (13.1) with a value of D different from the spatial dimension d. If an object satisfies (13.1) with $D < d$, its density is not the same for all R but scales as

$$\rho(R) \propto M/R^d \sim R^{D-d}. \tag{13.3}$$

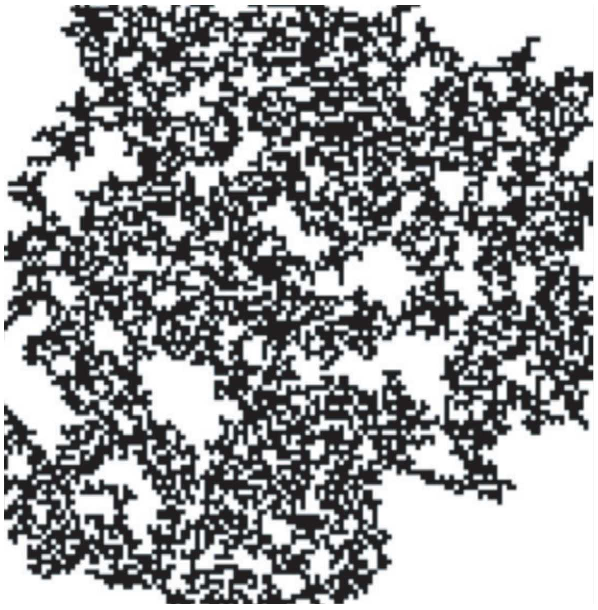


Figure 13.1: Example of a spanning percolation cluster generated at $p = 0.5927$ on a $L = 124$ square lattice. The other occupied sites are not shown.

Because $D < d$, we see that a fractal object becomes less dense at larger length scales. The scale dependence of the density is a quantitative measure of the ramified or stringy nature of fractal objects. In addition, another characteristic of fractal objects is that they have holes of all sizes. This property follows from (13.3) because if we replace R by Rb, where b is some constant, we obtain the same power law dependence for $\rho(R)$. Thus, it does not matter what scale of length is used, and thus all hole sizes must be present.

Another important characteristic of fractal objects is that they look the same over a range of length scales. This property of *self-similarity* or *scale invariance* means that if we take part of a fractal object and magnify it by the same magnification factor in all directions, the magnified picture is similar to the original. This property follows from the scaling argument given for $\rho(R)$.

The percolation cluster shown in Figure 13.1 is an example of a *random* or statistical fractal because the mass-length relation (13.1) is satisfied only on the average, that is, only if the quantity $M(R)$ is averaged over many different origins in a given cluster and over many clusters.

In physical systems, the relation (13.1) does not extend over all length scales but is bounded by both upper and lower cut-off lengths. For example, a lower cut-off length is provided by the lattice spacing or the mean distance between the constituents of the object. In computer simulations, the maximum length is usually the finite system size. The presence of these cut-offs complicates the determination of the fractal dimension.

In Problem 13.1 we compute the fractal dimension of percolation clusters using straightforward Monte Carlo methods. Remember that data extending over several decades is required to obtain convincing evidence for a power law relationship between M and R and to determine accurate estimates for the fractal dimension. Hence, conclusions based on the limited simulations posed in the problems need to be interpreted with caution.

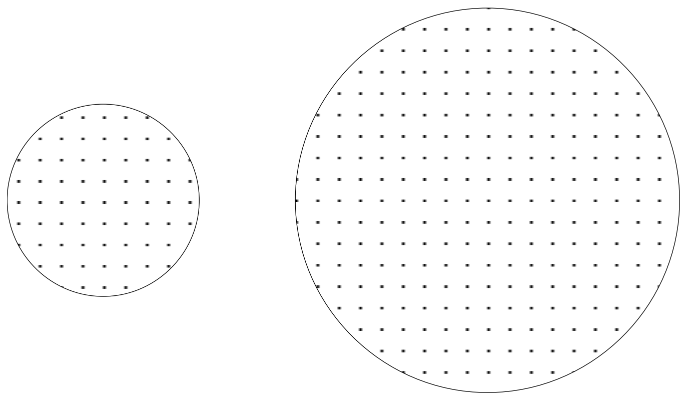


Figure 13.2: The number of dots per unit area in each circle is uniform. How does the total number of dots (mass) vary with the radius of the circle?

Problem 13.1. The fractal dimension of percolation clusters

(a) Generate a site percolation configuration on a square lattice with $L \geq 61$ at $p = p_c \approx 0.5927$. Why might it be necessary to generate several configurations before a spanning cluster is obtained? Does the spanning cluster have many dangling ends?

(b) Choose a point on the spanning cluster and count the number of points in the spanning cluster $M(b)$ within a square of area b^2 centered about that point. Then double b and count the number of points within the larger box. Can you repeat this procedure indefinitely? Repeat this procedure until you can estimate the b-dependence of the number of points. Use the b-dependence of $M(b)$ to estimate D according to the definition $M(b) \sim b^D$, that is, estimate D from a log-log plot of $M(b)$ versus b. Choose another point in the cluster and repeat this procedure. Are your results similar? A better estimate for D can be found by averaging $M(b)$ over several origins in each spanning cluster and averaging over many spanning clusters.

(c) If you have not already done Problem 12.8a, compute D by determining the mean size (mass) M of the spanning cluster at $p = p_c$ as a function of the linear dimension L of the lattice. Consider $L = 11$, 21, 41, and 61 and estimate D from a log-log plot of M versus L. □

***Problem 13.2.** Renormalization group calculation of the fractal dimension
Compute $\langle M^2 \rangle$, the average of the square of the number of occupied sites in the spanning cluster at $p = p_c$, and the quantity $\langle M'^2 \rangle$, the average of the square of the number of occupied sites in the spanning cluster on the renormalized lattice of linear dimension $L' = L/b$. Because $\langle M^2 \rangle \sim L^{2D}$ and $\langle M'^2 \rangle \sim (L/b)^{2D}$, we can obtain D from the relation $b^{2D} = \langle M^2 \rangle / \langle M'^2 \rangle$. Choose the length rescaling factor to be $b = 2$ and adopt the same blocking procedure as was used in Section 12.5. An average over ten spanning clusters for $L = 16$ and $p = 0.5927$ is sufficient for qualitative results. □

In Problems 13.1 and 13.2, we were interested only in the properties of the spanning clusters. For this reason, our algorithm for generating percolation configurations by randomly occupying each site is inefficient because it generates many clusters. A more efficient way of generating single percolation clusters is due independently to Hammersley, Leath, and Alexandrowicz.

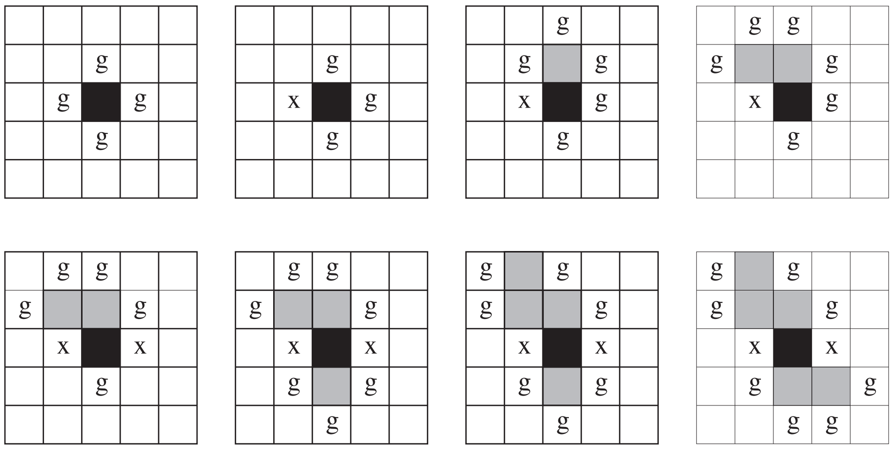


Figure 13.3: An example of the growth of a percolation cluster. Sites are occupied with probability p. Occupied sites are represented by a shaded square, growth or perimeter sites are labeled by g, and tested unoccupied sites are labeled by x. Because the seed site is occupied but not tested, we have represented it differently than the other occupied sites. The growth sites are chosen at random.

This algorithm, commonly known as the Leath or the single cluster growth algorithm, is equivalent to the following steps (see Figure 13.3):

1. Occupy a single seed site on the lattice. The nearest neighbors (four on the square lattice) of the seed represent the *perimeter* sites.
2. For each perimeter site, generate a uniform random number r in the unit interval. If $r \leq p$, the site is occupied and added to the cluster; otherwise, the site is not occupied. In order that sites be unoccupied with probability $1-p$, these sites are not tested again.
3. For each site that is occupied, determine if there are any new perimeter sites, that is, untested neighbors. Add the new perimeter sites to the perimeter list.
4. Continue steps 2 and 3 until there are no untested perimeter sites to test for occupancy. □

Class `SingleCluster` implements this algorithm and computes the number of occupied sites within a radius r of the seed particle. The seed site is placed at the center of a square lattice. Two one-dimensional arrays, `pxs` and `pys`, store the x and y positions of the perimeter sites. The status of a site is stored in the byte array `s` with `s(x,y) = (byte) 1` for an occupied site, `s(x,y) = (byte) 2` for a perimeter site, `s(x,y) = (byte)-1` for a site that has already been tested and not occupied, and `s(x,y) = (byte) 0` for an untested and unvisited site. To avoid checking for the boundaries of the lattice, we add extra rows and columns at the boundaries and set these sites equal to `(byte)-1`. We use a byte array because the array `s` will be sent to the `LatticeFrame` class which uses byte arrays.

Listing 13.1: Class `SingleCluster` generates and analyzes a single percolation cluster

```java
package org.opensourcephysics.sip.ch13.cluster;
public class SingleCluster {
```

```
public byte site[][];
public int[] xs, ys, pxs, pys;
public int L;
public double p;          // site occupation probability
int occupiedNumber;
int perimeterNumber;
// displacement x to nearest neighbors
int nx[] = {1, -1, 0, 0};
// displacement y to nearest neighbors
int ny[] = {0, 0, 1, -1};
// mass of ring, index is distance from center of mass
double mass[];

public void initialize() {
   site = new byte[L+2][L+2]; // gives status of each site
   xs = new int[L*L];         // location of occupied sites
   ys = new int[L*L];
   pxs = new int[L*L];        // location of perimeter sites
   pys = new int[L*L];
   for(int i = 0;i<L+2;i++) {
      site[0][i] = (byte) -1; // don't occupy edge sites
      site[L+1][i] = (byte) -1;
      site[i][0] = (byte) -1;
      site[i][L+1] = (byte) -1;
   }
   xs[0] = 1+(L/2);
   ys[0] = xs[0];
   site[xs[0]][ys[0]] = (byte) 1; // occupy center site
   occupiedNumber = 1;
   for(int n = 0;n<4;n++) { // perimeter sites
      pxs[n] = xs[0]+nx[n];
      pys[n] = ys[0]+ny[n];
      site[pxs[n]][pys[n]] = (byte) 2;
   }
   perimeterNumber = 4;
}

public void step() {
   if(perimeterNumber>0) {
      int perimeter = (int) (Math.random()*perimeterNumber);
      int x = pxs[perimeter];
      int y = pys[perimeter];
      perimeterNumber--;
      pxs[perimeter] = pxs[perimeterNumber];
      pys[perimeter] = pys[perimeterNumber];
      if(Math.random()<p) {         //occupy site
         site[x][y] = (byte) 1;
         xs[occupiedNumber] = x;
         ys[occupiedNumber] = y;
         occupiedNumber++;
         for(int n = 0;n<4;n++) { // find new perimeter sites
            int px = x+nx[n];
            int py = y+ny[n];
```

```
                if(site[px][py]==(byte) 0) {
                   pxs[perimeterNumber] = px;
                   pys[perimeterNumber] = py;
                   site[px][py] = (byte) 2;
                   perimeterNumber++;
                }
             }
          } else {
             site[x][y] = (byte) -1;
          }
       }
    }

    public void massDistribution() {
       mass = new double[L];
       double xcm = 0;
       double ycm = 0;
       for(int n = 0;n<occupiedNumber;n++) {
          xcm += xs[n];
          ycm += ys[n];
       }
       xcm /= occupiedNumber;
       ycm /= occupiedNumber;
       for(int n = 0;n<occupiedNumber;n++) {
          double dx = xs[n]-xcm;
          double dy = ys[n]-ycm;
          int r = (int) Math.sqrt(dx*dx+dy*dy);
          if((r>1)&&(r<L)) {
             mass[r]++;
          }
       }
    }
}
```

The target class is shown in Listing 13.2 Note the use of the Open Source Physics `LatticeFrame` class. When the user stops the cluster growth, a log-log plot of the mass distribution is shown.

Listing 13.2: Class `SingleClusterApp` displays the site percolation cluster and the mass distribution

```
package org.opensourcephysics.sip.ch13.cluster;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.frames.*;
import java.awt.Color;

public class SingleClusterApp extends AbstractSimulation {
   SingleCluster cluster = new SingleCluster();
   PlotFrame plotFrame = new PlotFrame("ln r", "ln M", "Mass distribution");
   LatticeFrame latticeFrame = new LatticeFrame("Percolation cluster");
   int steps;

   public void initialize() {
      // not occupied or tested
      latticeFrame.setIndexedColor(0, Color.BLACK);
      latticeFrame.setIndexedColor(1, Color.BLUE);     // occupied
      // perimeter or growth site
      latticeFrame.setIndexedColor(2, Color.GREEN);
      // permanently not occupied
      latticeFrame.setIndexedColor(-1, Color.YELLOW);
      cluster.L = control.getInt("L");
      cluster.p = control.getDouble("p");
      cluster.initialize();
      latticeFrame.setAll(cluster.site);
   }

   public void doStep() {
      cluster.step();
      latticeFrame.setAll(cluster.site);
      latticeFrame.setMessage("n = "+cluster.occupiedNumber);
      if(cluster.perimeterNumber==0) {
         control.calculationDone("Computation done");
      }
   }

   public void stopRunning() {
      plotFrame.clearData();
      cluster.massDistribution();
      double massEnclosed = 0;
      int rPrint = 2;
      for(int r = 2;r<cluster.L/2;r++) {
         massEnclosed += cluster.mass[r];
         if(r==rPrint) { // use logarithmic scale
            plotFrame.append(0, Math.log(r), Math.log(massEnclosed));
            rPrint *= 2;
         }
      }
      plotFrame.setVisible(true);
   }

   public void reset() {
      control.setValue("L", 61);
      control.setValue("p", 0.5927);
      setStepsPerDisplay(10);
      enableStepsPerDisplay(true);
   }

   public static void main(String[] args) {
      SimulationControl.createApp(new SingleClusterApp());
   }
}
```

We will use the Leath or single cluster growth algorithm in Problem 13.3 to generate a spanning cluster at the percolation threshold. The fractal dimension is determined by counting the number of sites M in the cluster within a distance r of the center of mass of the cluster. The center of mass is defined by

$$\mathbf{r}_{\rm cm} = \frac{1}{N}\sum_i \mathbf{r}_i \tag{13.4}$$

where N is the total number of particles in the cluster. A typical plot of $\ln M(r)$ versus $\ln r$ is

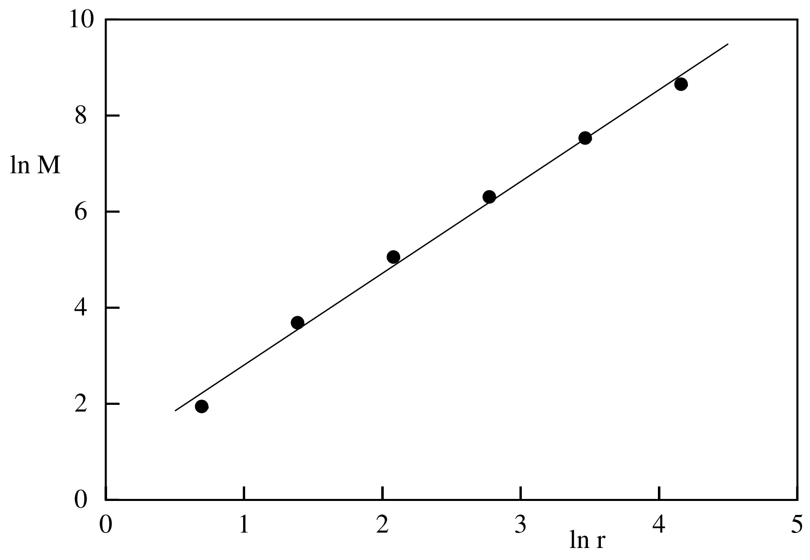


Figure 13.4: Plot of $\ln M$ versus $\ln r$ for a single spanning percolation cluster generated at $p = 0.5927$ on a $L = 129$ square lattice. The straight line is a linear least squares fit to the data. The slope of this line is 1.91 and is an estimate of the fractal dimension D. The exact value of D for a percolation cluster at $p = p_c$ in two dimensions is $D = 91/48 \approx 1.896$.

shown in Figure 13.4. Because the cluster cannot grow past the edge of the lattice, we do not include data for $r \approx L$.

Problem 13.3. Single cluster growth and the fractal dimension

(a) Explain how the Leath algorithm generates single clusters in a way that is equivalent to the multiple clusters that are generated by visiting all sites. More precisely, the Leath algorithm generates percolation clusters with a distribution of cluster sizes equal to sn_s. For example, if you grow 10 clusters of size $s = 2$, then $n_s = 10/2 = 5$. The additional factor of s is due to the fact that each site of the cluster has an equal chance of being the seed of the cluster, and hence the same cluster can be generated in s ways.

(b) Grow as large a spanning cluster as you can and look at it on different length scales. One way to do so is to divide the screen into four windows, each of which magnifies a part of the cluster shown in the previous window. Does the part of the cluster shown in each window look approximately self-similar?

(c) Choose $p = 0.5927$ and $L \geq 61$ and generate at least ten configurations of spanning clusters. Determine the number of occupied sites $M(r)$ within a distance r of the seed site of each cluster. (Better results can be found by choosing the origin to be center of mass of each cluster.) Average $M(r)$ over the spanning clusters. Estimate D from the log-log plot of M versus r (see Figure 13.4). If time permits, generate percolation clusters on larger lattices.

(d) Generate clusters at $p = 0.65$, a value of p greater than p_c, for $L = 101$. Make a log-log plot of $M(r)$ versus r. Is the slope approximately equal to the value of D found in part (c)? Does the slope increase or decrease for larger r? Repeat for $p = 0.80$. Is a spanning cluster generated at $p > p_c$ a fractal?

(e) The fractal dimension of percolation clusters is not an independent exponent but satisfies the scaling relation

$$D = d - \beta/\nu, \tag{13.5}$$

where β and ν are defined in Table 12.1. The relation (13.5) can be understood by the following finite-size scaling argument. The number of sites in the spanning cluster on a lattice of linear dimension L is given by

$$M(L) \sim P_\infty(L)L^d, \tag{13.6}$$

where P_∞ is the probability that an occupied site belongs to the spanning cluster, and L^d is the total number of sites in the lattice. In the limit of an infinite lattice and p near p_c, we know that $P_\infty(p) \sim (p-p_c)^\beta$ and $\xi(p) \sim (p-p_c)^{-\nu}$ independent of L. Hence, for $L \sim \xi$, we have that $P_\infty(L) \sim L^{-\beta/\nu}$ [see (12.11)], and we can write

$$M(L) \sim L^{-\beta/\nu}L^d \sim L^D. \tag{13.7}$$

The relation (13.5) follows. Use the exact values of β and ν from Table 12.1 to find the exact value of D for $d = 2$. Is your estimate for D consistent with this value?

(f)* Rewrite the `SingleCluster` class so that the lattice is stored as a one-dimensional array as is done for class `Clusters` in Chapter 12.

(g)* Estimate the fractal dimension for percolation clusters on a simple cubic lattice. Take $p_c = 0.3117$. □

13.2 Regular Fractals

As we have seen, one characteristic of random fractal objects is that they look the same on a range of length scales. To gain a better understanding of the meaning of self-similarity, consider the following example of a *regular* fractal, a mathematical object that is self-similar on *all* length scales. Begin with a line one unit long (see Figure 13.5a). Remove the middle third of the line and replace it by two lines of length 1/3 each so that the curve has a triangular bump in it and the total length of the curve is 4/3 (see Figure 13.5b). In the next stage, each of the segments of length 1/3 is divided into lines of length 1/9 and the procedure is repeated as shown in Figure 13.5c. What is the length of the curve shown in Figure 13.5c?

The three stages shown in Figure 13.5 can be extended an infinite number of times. The resulting curve is infinitely long and contains an infinite number of infinitesimally small segments. Such a curve is known as the *triadic Koch curve*. A Java class that uses a recursive procedure (see Section 6.3) to draw this curve is given in Listing 13.3. Note that method `iterate` calls itself. Use class KochApp to generate the curves shown in Figure 13.5.

Listing 13.3: Class for drawing the Koch curve

```
package org.opensourcephysics.sip.ch13;
import java.awt.Graphics;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.frames.*;
import org.opensourcephysics.display.*;

public class KochApp extends AbstractCalculation implements Drawable {
```

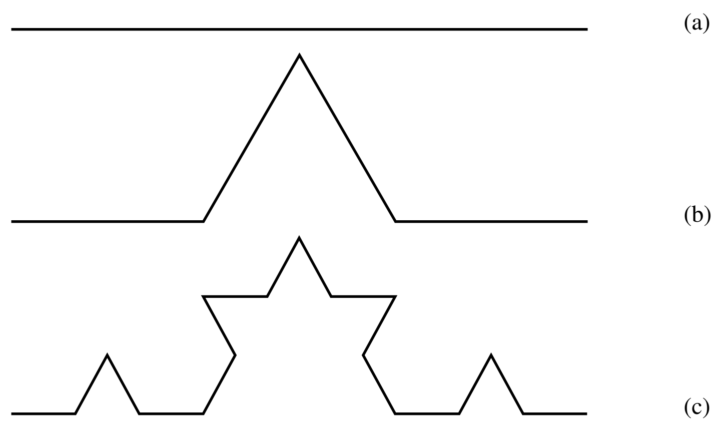


Figure 13.5: The first three stages (a)–(c) of the generation of a self-similar Koch curve. At each stage the displacement of the middle third of each segment is in the direction that increases the area under the curve. The curves were generated using `class KochApp`. The Koch curve is an example of a continuous curve for which there is no tangent defined at any of its points. The Koch curve is self-similar on each length scale.

```
DisplayFrame frame = new DisplayFrame("Koch Curve");
int n = 0;

public KochApp() {
   frame.setPreferredMinMax(-100, 600, -100, 600);
   frame.setSquareAspect(true);
   frame.addDrawable(this);
}

public void calculate() {
   n = control.getInt("Number of iterations");
   frame.setVisible(true);
}

public void iterate(double x1, double y1, double x2, double y2,
        int n, DrawingPanel panel,
                    Graphics g) {
   // draw Koch curve using recursion
   if(n>0) {
      double dx = (x2-x1)/3;
      double dy = (y2-y1)/3;
      double xOneThird = x1+dx;       // new end at 1/3 of line segment
      double yOneThird = y1+dy;
      double xTwoThird = x1+2*dx;     // new end at 2/3 of line segment
      double yTwoThird = y1+2*dy;
      // rotates line segment (dx, dy) by 60 degrees and adds
      // to (xOneThird,yOneThird)
      double xMidPoint = (0.5*dx-0.866*dy+xOneThird);
      double yMidPoint = (0.5*dy+0.866*dx+yOneThird);
      // each line segment generates 4 new ones
      iterate(x1, y1, xOneThird, yOneThird, n-1, panel, g);
```

```
            iterate(xOneThird, yOneThird, xMidPoint, yMidPoint, n-1,
                     panel, g);
            iterate(xMidPoint, yMidPoint, xTwoThird, yTwoThird, n-1, panel, g);
            iterate(xTwoThird, yTwoThird, x2, y2, n-1,
                     panel, g);
         } else {
            int ix1 = panel.xToPix(x1);
            int iy1 = panel.yToPix(y1);
            int ix2 = panel.xToPix(x2);
            int iy2 = panel.yToPix(y2);
            g.drawLine(ix1, iy1, ix2, iy2);
         }
      }

      public void draw(DrawingPanel panel, Graphics g) {
         iterate(0, 0, 500, 0, n, panel, g);
      }

      public void reset() {
         control.setValue("Number of iterations", 3);
      }

      public static void main(String args[]) {
         CalculationControl.createApp(new KochApp());
      }
   }
```

How can we determine the fractal dimension of the Koch and similar mathematical objects? There are several generalizations of the Euclidean dimension that lead naturally to a definition of the fractal dimension (see Section 13.5). Here we consider a definition based on counting boxes. Consider a one-dimensional curve of unit length that has been divided into N equal segments of length ℓ so that $N = 1/\ell$ (see Figure 13.6). As ℓ decreases, N increases linearly, which is the expected result for a one-dimensional curve. Similarly, if we divide a two-dimensional square of unit area into N equal subsquares of length ℓ, we have $N = 1/\ell^2$, the expected result for a two-dimensional object (see Figure 13.6). In general, we have $N = 1/\ell^D$, where D is the fractal dimension of the object. If we take the logarithm of both sides of this relation, we can express the fractal dimension as

$$D = \frac{\log N}{\log(1/\ell)} \qquad \text{(box dimension)}. \tag{13.8}$$

Now let us apply this definition to the Koch curve. Each time the length ℓ of our measuring unit is reduced by a factor of 3, the number of segments is increased by a factor of 4. If we use the size of each segment as the size of our measuring unit, then at the nth iteration we have $N = 4^n$ and $\ell = (1/3)^n$, and the fractal dimension of the triadic Koch curve is given by

$$D = \frac{\log 4^n}{\log 3^n} = \frac{n \log 4}{n \log 3} \approx 1.2619 \qquad \text{(triadic Koch curve)}. \tag{13.9}$$

From (13.9) we see that the Koch curve has a fractal dimension between that of a line and a plane. Is this statement consistent with your visual interpretation of the degree to which the triadic Koch curve fills space?

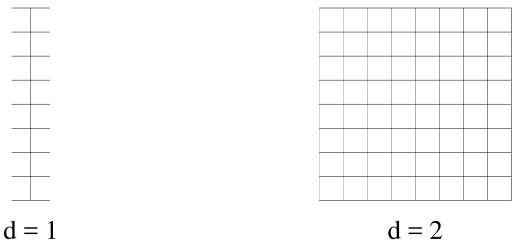


Figure 13.6: Examples of one-dimensional and two-dimensional objects.

Problem 13.4. The recursive generation of regular fractals

(a) Recursion is used in method `iterate` in KochApp and is one of the more difficult programming concepts. Explain the nature of recursion and the way it is implemented.

(b) Regular fractals can be generated from a pattern that is used in a self-replicating manner. Write a program to generate the quadric Koch curve shown in Figure 13.7a. What is its fractal dimension?

(c) What is the fractal dimension of the Sierpiński gasket shown in Figure 13.7b? Write a program that generates the next several iterations.

(d) What is the fractal dimension of the Sierpiński carpet shown in Figure 13.7c? How does the fractal dimension of the Sierpiński carpet compare to the fractal dimension of a percolation cluster? Are the two fractals visually similar? □

13.3 Kinetic Growth Processes

Many systems in nature exhibit fractal geometry. Fractals have been used to describe the irregular shapes of such varied objects as coastlines, clouds, coral reefs, and the human lung. Why are fractal structures so common? How do fractal structures form? In this section we discuss several growth models that generate structures that show a remarkable similarity to forms observed in nature. The first two models are already familiar to us and exemplify the flexibility and utility of kinetic growth models.

Epidemic model. In the context of the spread of disease, we usually want to know the conditions for an epidemic. A simple lattice model of the spread of a disease can be formulated as follows. Suppose that an occupied site corresponds to an infected person. Initially there is a single infected person and the four nearest neighbor sites (on the square lattice) correspond to susceptible people. At the next time step, we visit the four susceptible sites and occupy (infect) each site with probability p. If a susceptible site is not occupied, we say that the site is immune and we do not test it again. We then find the new susceptible sites and continue until either the disease is controlled or reaches the boundary of the lattice. Convince yourself that this growth model of a disease generates a cluster of infected sites that is identical to a percolation cluster at probability p. The only difference is that we have introduced a discrete time step into the model. Some of the properties of this model are explored in Problem 13.5.

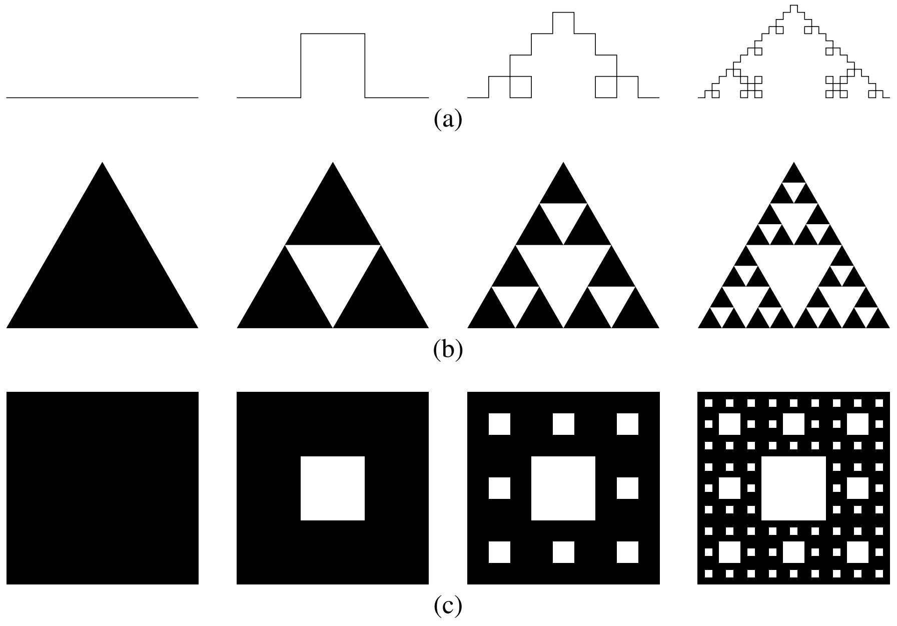


Figure 13.7: (a) The first few iterations of the quadric Koch curve. (b) The first few iterations of the Sierpiński gasket. (c) The first few iterations of the Sierpiński carpet.

Problem 13.5. A simple epidemic model

(a) Explain why the simple epidemic model discussed in the text generates the same clusters as in the percolation model. What is the minimum value of p necessary for an epidemic to occur? Recall that in one time step, all susceptible sites are visited simultaneously and infected with probability p. Determine how n, the number of infected sites, depends on the time t (the number of time steps) for various values of p. A straightforward way to proceed is to modify class `SingleCluster` so that all susceptible sites are visited and occupied with probability p before new susceptible sites are found. In Chapter 14 we will learn that this model is an example of a cellular automaton.

(b) What are some ways that you could modify the model to make it more realistic? For example, the infected sites might recover after a certain time. □

Eden model. An even simpler example of a growth model was proposed by Eden in 1958 to simulate the growth of tumors or a bacterial colony. Although we will find that the resultant mass distribution is not a fractal, the description of the Eden model illustrates the general nature of the fractal growth models we will discuss.

Choose a seed site at the center of the lattice for simplicity. The unoccupied nearest neighbors of the occupied sites are the perimeter or *growth* sites. In the simplest version of the model, a growth site is chosen at random and occupied. The newly occupied site is removed from the list of growth sites and the new growth sites are added to the list. This process is repeated many times until a large cluster of occupied sites is formed. The difference between this model and the simple epidemic model is that all tested sites are occupied. In other words, no growth sites ever become "immune." Some of the properties of Eden clusters are investigated in Problem 13.6.

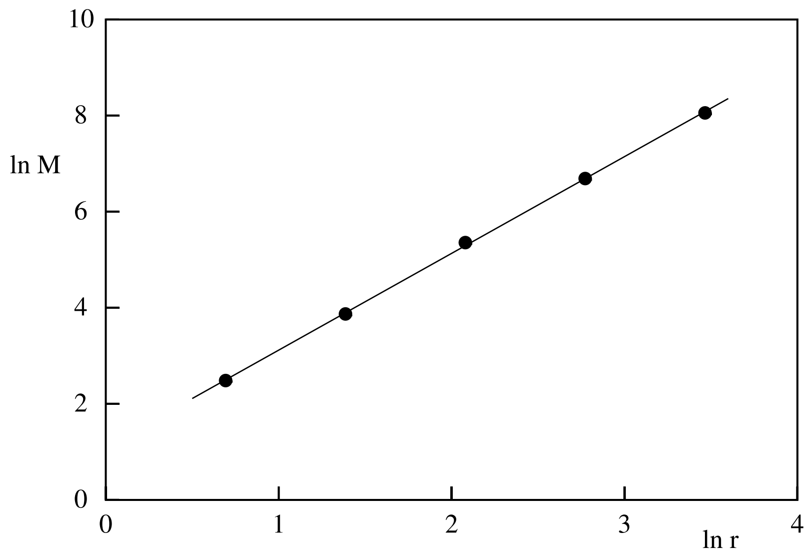


Figure 13.8: Plot of $\ln M$ versus $\ln r$ for a single Eden cluster generated on a $L = 61$ square lattice. A least squares fit from $r = 2$ to $r = 32$ yields a slope of approximately 2.01.

Problem 13.6. The Eden model

(a) Modify class `SingleCluster` so that clusters are generated on a square lattice according to the Eden model. A straightforward procedure is to occupy perimeter sites with probability $p = 1$. The simulation should be stopped when the cluster just reaches the edge of the lattice. What would happen if we were to occupy perimeter sites indefinitely? Follow the procedure of Problem 13.3 and determine the number of occupied sites $M(r)$ within a distance r of the seed site. Assume that $M(r) \sim r^D$ for sufficiently large r and estimate D from the slope of a log-log plot of M versus r. A typical log-log plot is shown in Figure 13.8 for $L = 61$. Can you conclude from your data that Eden clusters are compact?

(b) Modify your program so that only the perimeter or growth sites are shown. Where are the majority of the perimeter sites relative to the center of the cluster? Grow as big a cluster as time permits. □

Invasion percolation. A dynamical process known as *invasion percolation* has been used to model the shape of the oil-water interface that occurs when water is forced into a porous medium containing oil. The goal is to use the water to recover as much oil as possible. In this process a water cluster grows into the oil through the path of least resistance. Consider a lattice of size $L_x \times L_y$, with the water (the invader) initially occupying the left edge (see Figure 13.9). The resistance to the invader is given by assigning to each lattice site a uniformly distributed random number between 0 and 1; these numbers are fixed throughout the invasion. Sites that are nearest neighbors of the invader sites are the perimeter sites. At each time step, the perimeter site with the lowest random number is occupied by the invader and the oil (the defender) is displaced. The invading cluster grows until a path of occupied sites connects the left and right edges of the lattice. After this path forms, there is no need for the water to occupy any additional sites. To minimize boundary effects, periodic boundary conditions are used for the top and bottom edges, and all quantities are measured over only a central region for from the left and right edges of the lattice.

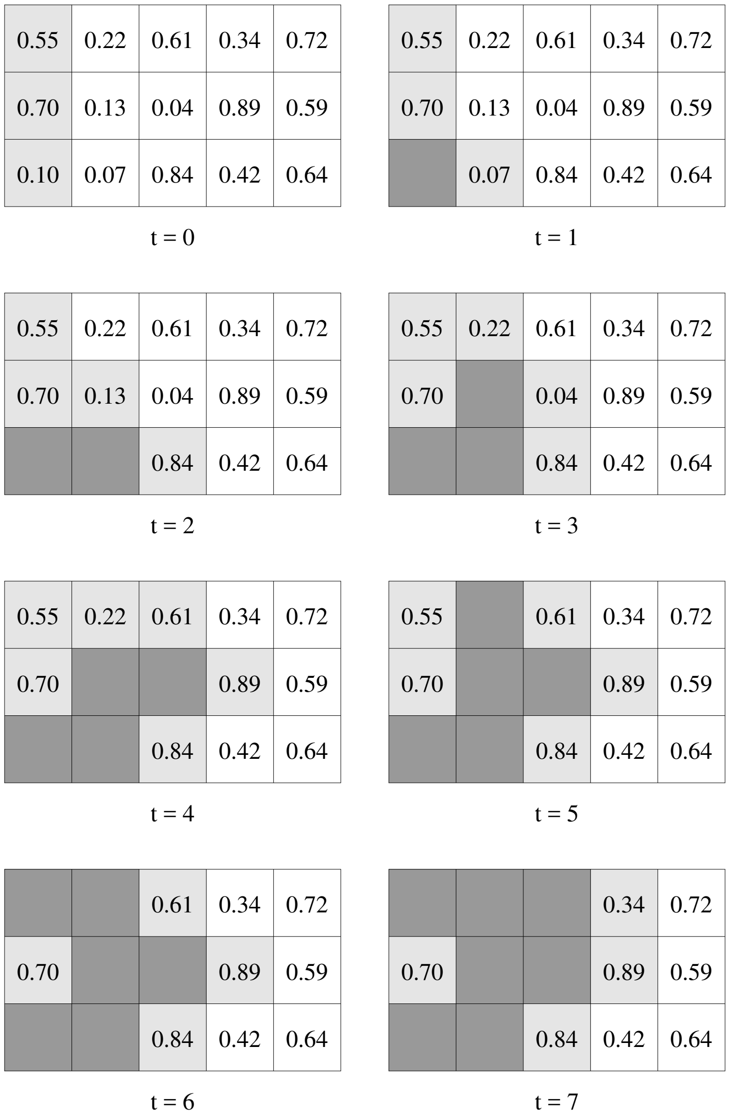


Figure 13.9: Example of a cluster formed by invasion percolation on a 5×3 lattice. The lattice at $t = 0$ shows the random numbers that have been assigned to the sites. The darkly shaded sites are occupied by the invader that occupies the perimeter site (lightly shaded) with the smallest random number. The cluster continues to grow until a site in the right-most column is occupied.

Class `Invasion` implements the invasion percolation algorithm. The two-dimensional array element `site[i][j]` initially stores a random number for the site at `(i,j)`. If the site at `(i,j)` is occupied, then `site[i][j]` is set equal to 1. If the site at `(i,j)` is a perimeter site, then `site[i][j]` is increased by 2. In this way we know which sites are perimeter sites, and the value of the random number is associated with the perimeter site. A new perimeter site is inserted into its proper ordered position in the lists `perimeterListX` and `perimeterListY`. The perimeter lists are ordered so that the site with the largest random number is at the beginning.

Two search methods are provided for determining the position of a new perimeter site in the perimeter lists. In a *linear search* we go through the list in order until the random number associated with the new perimeter site is between two random numbers in the list. In a *binary search* we divide the list in two, and determine in which half the new random number belongs. Then we divide this half into half again and so on until the correct position is found. The linear and binary search methods are compared in Problem 13.7d. The binary search is the default method used in class Invasion.

The main quantities of interest are the fraction of sites occupied by the invader and the probability $P(r)\Delta r$ that a site with a random number between r and $r + \Delta r$ is occupied. The properties of invasion percolation are explored in Problem 13.7.

Listing 13.4: Class for simulating invasion percolation

```
package org.opensourcephysics.sip.ch13.invasion;
import java.awt.Color;
import org.opensourcephysics.frames.*;

public class Invasion {
   public int Lx, Ly;
   public double site[][];
   public int perimeterListX[], perimeterListY[];
   public int numberOfPerimeterSites;
   public boolean ok = true;
   public LatticeFrame lattice;

   public Invasion(LatticeFrame latticeFrame) {
      lattice = latticeFrame;
      lattice.setIndexedColor(0, Color.blue);
      lattice.setIndexedColor(1, Color.black);
   }

   public void initialize() {
      Lx = 2*Ly;
      site = new double[Lx][Ly];
      perimeterListX = new int[Lx*Ly];
      perimeterListY = new int[Lx*Ly];
      for(int y = 0;y<Ly;y++) {
         site[0][y] = 1; // occupy first column
         lattice.setValue(0, y, 1);
      }
      for(int y = 0;y<Ly;y++) {
         for(int x = 1;x<Lx;x++) {
            site[x][y] = Math.random();
            lattice.setValue(x, y, 0);
         }
      }
      numberOfPerimeterSites = 0;
      for(int y = 0;y<Ly;y++) { // second column is perimeter sites
         site[1][y] += 2;        // perimeter sites have site > 2;
         numberOfPerimeterSites++;
         // inserts site in perimeter list in order
         insert(1, y);
      }
      ok = true;
   }
```

```
public void insert(int x, int y) {
   int insertionLocation = binarySearch(x, y);
   for(int i = numberOfPerimeterSites-1;i>insertionLocation;i--) {
      perimeterListX[i] = perimeterListX[i-1];
      perimeterListY[i] = perimeterListY[i-1];
   }
   perimeterListX[insertionLocation] = x;
   perimeterListY[insertionLocation] = y;
}

public int binarySearch(int x, int y) {
   int firstLocation = 0;
   int lastLocation = numberOfPerimeterSites-2;
   if(lastLocation<0) {
      lastLocation = 0;
   }
   int middleLocation = (firstLocation+lastLocation)/2;
   // determine which half of list new number is in
   while(lastLocation-firstLocation>1) {
      int middleX = perimeterListX[middleLocation];
      int middleY = perimeterListY[middleLocation];
      if(site[x][y]>site[middleX][middleY]) {
         lastLocation = middleLocation;
      } else {
         firstLocation = middleLocation;
      }
      middleLocation = (firstLocation+lastLocation)/2;
   }
   return lastLocation;
}

// goes in order looking for location to insert
public int linearSearch(int x, int y) {
   if(numberOfPerimeterSites==1) {
      return 0;
   } else {
      for(int i = 0;i<numberOfPerimeterSites-1;i++) {
         if(site[x][y]>site[perimeterListX[i]][perimeterListY[i]]) {
            return i;
         }
      }
   }
   return numberOfPerimeterSites-1;
}

public void step() {
   if(ok) {
      int nx[] = {1, -1, 0, 0};
      int ny[] = {0, 0, 1, -1};
      int x = perimeterListX[numberOfPerimeterSites-1];
      int y = perimeterListY[numberOfPerimeterSites-1];
      if(x>Lx-3) {
            // if cluster gets near the end, stop simulation
            ok = false;
         }
         numberOfPerimeterSites--;
         site[x][y] -= 1;
         lattice.setValue(x, y, 1);
         for(int i = 0;i<4;i++) { // finds new perimeter sites
            int perimeterX = x+nx[i];
            int perimeterY = (y+ny[i])%Ly;
            if(perimeterY==-1) {
               perimeterY = Ly-1;
            }
            if(site[perimeterX][perimeterY]<1) { // new perimeter site
               site[perimeterX][perimeterY] += 2;
               numberOfPerimeterSites++;
               insert(perimeterX, perimeterY);
            }
         }
      }
   }

   public void computeDistribution(PlotFrame data) {
      int numberOfBins = 20;
      int numberOccupied = 0;
      double occupied[] = new double[numberOfBins];
      double number[] = new double[numberOfBins];
      double binSize = 1.0/numberOfBins;
      int minX = Lx/3;
      int maxX = 2*minX;
      for(int x = minX;x<=maxX;x++) {
         for(int y = 0;y<Ly;y++) {
            int bin = (int) (numberOfBins*(site[x][y]%1));
            number[bin]++;
            if((site[x][y]>1)&&(site[x][y]<2)) {
               numberOccupied++;
               occupied[bin]++;
            }
         }
      }
      data.setMessage("Number occupied = "+numberOccupied);
      for(int bin = 0;bin<numberOfBins;bin++) {
         data.append(0, (bin+0.5)*binSize, occupied[bin]/number[bin]);
      }
   }
}
```

Problem 13.7. Invasion percolation

(a) Use class Invasion to generate an invasion percolation cluster on a 20×40 lattice and describe the qualitative nature of the cluster.

(b) Compute $M(L)$, the number of sites occupied by the invader in the central $L \times L$ region of the $L \times 2L$ lattice when the invader first reaches the right edge. Average over at least twenty configurations. Assume that $M(L) \sim L^D$ and estimate D from a plot of $\ln M$ versus $\ln L$. Compare your estimate for D with the fractal dimension of site percolation clusters at $p = p_c$. (The first published results for $M(L)$ by Wilkinson and Willemsen were for 2000 realizations each for L in the range 20 to 100.)

(c) Determine the probability $P(r)\Delta r$ that a site with a random number between r and $r + \Delta r$ is occupied. Choose $\Delta r = 0.01$. Plot $P(r)$ versus r for $L = 20$ and for values of L up to about $L \geq 50$. Is there a value of r near which $P(r)$ changes rapidly? How does this value of r compare to the value of p_c for site percolation on the square lattice? On the basis of your numerical estimate for the exponent D found in part (b) and the qualitative behavior of $P(r)$, make a hypothesis about the relation between the nature of the geometrical properties of the invasion percolation cluster and the spanning percolation cluster at $p = p_c$.

(d) Explain the nature of the two search algorithms given in class `Invasion`. Which method yields the fastest results on a 30×60 lattice? Verify that the CPU time for a linear and binary search is proportional to n and $\log n$, respectively, where n is the number of items in the list to be searched. Hence, for sufficiently large n, a binary search usually is preferred.

(e)* Modify your program so that the invasion percolation clusters are grown from a seed at the origin. Grow a cluster until it reaches a boundary of the lattice. Estimate the fractal dimension as you did for the spanning percolation clusters in Problem 13.3 and compare your two estimates. On the basis of this estimate and your results from parts (b) and (c), can you conclude that the spanning cluster in invasion percolation is a fractal? □

Diffusion in disordered media. In Chapter 7 we considered random walks on perfect lattices. We found that the mean square displacement of a random walker $\langle R^2(t)\rangle$ is proportional to the time t for sufficiently large t. (For a simple random walk, this relation holds for all t.) Now let us suppose that the random walker is restricted to a disordered lattice, for example, the occupied sites of a percolation cluster. What is the asymptotic t-dependence of $\langle R^2(t)\rangle$ in this case? This model of a random walk on a percolation cluster is known as the "ant in the labyrinth."

Just as a random walk on a lattice is a simple example of diffusion, a random walk on a disordered lattice is a simple example of the general problem of diffusion and transport in disordered media. Because many materials of interest are noncrystalline and disordered, there are many physical phenomena that can be related to the motion of an ant in the labyrinth.

In the usual formulation of the ant in the labyrinth, we place a walker (ant) at random on one of the occupied sites of a percolation cluster that has been generated with probability p. At each time step, the ant tosses a coin with four possible outcomes (for a square lattice). If the outcome corresponds to a step to an occupied site, the ant moves; otherwise, it remains at its present position. In either case, the time t is increased by one unit.

The main quantity of interest is $R^2(t)$, the square of the distance between the ant's position at $t = 0$ and its position at time t. We can generate many walks with different initial positions on the same cluster and average over many percolation clusters to obtain the ant's mean square displacement $\langle R^2(t)\rangle$. How does $\langle R^2(t)\rangle$ depend on p and t? We consider this question in Problem 13.8.

Problem 13.8. The ant in the labyrinth

(a) For $p = 1$, the ant walks on a perfect lattice, and hence, $\langle R^2(t)\rangle = 2dDt$. Suppose that an ant does a random walk on a spanning cluster with $p > p_c$ on a square lattice. Assume that $\langle R^2(t)\rangle \to 4D_s(p)t$ for $p > p_c$ and sufficiently long times. We have denoted the diffusion

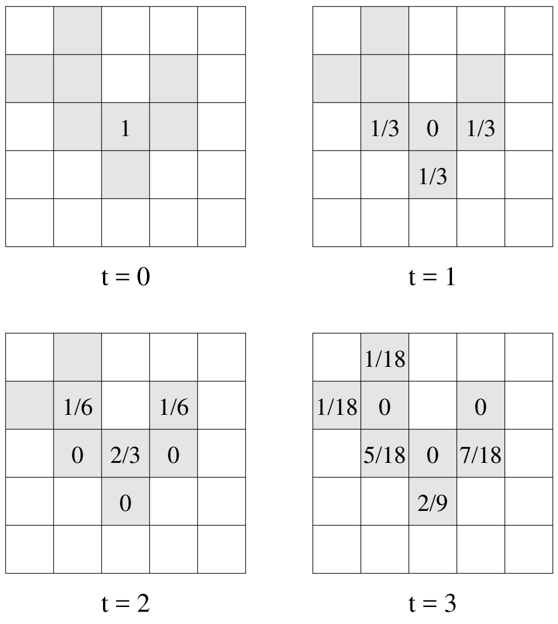


Figure 13.10: The evolution of the probability distribution function $W_t(i)$ for three successive time steps.

coefficient by D_s because we are considering random walks only on spanning clusters and are not considering walks on the finite clusters that also exist for $p > p_c$. Generate a cluster at $p = 0.7$ using the single cluster growth algorithm considered in Problem 13.3. Choose the initial position of the ant to be the seed site and modify your program to observe the motion of the ant on the screen. Use $L \geq 101$ and average over at least 100 walkers for t up to 500. Where does the ant spend much of its time? If $\langle R^2(t)\rangle \propto t$, what is $D_s(p)/D(p = 1)$?

(b) As in part (a) compute $\langle R^2(t)\rangle$ for $p = 1.0$, 0.8, 0.7, 0.65, and 0.62 with $L = 101$. If time permits, average over several clusters. Make a log-log plot of $\langle R^2(t)\rangle$ versus t. What is the qualitative t-dependence of $\langle R^2(t)\rangle$ for relatively short times? Is $\langle R^2(t)\rangle$ proportional to t for longer times? (Remember that the maximum value of $\langle R^2\rangle$ is bounded by the finite size of the lattice.) If $\langle R^2(t)\rangle \propto t$, estimate $D_s(p)$. Plot $D_s(p)/D(p = 1)$ as a function of p and discuss its qualitative dependence.

(c) Compute $\langle R^2(t)\rangle$ for $p = 0.4$ and confirm that for $p < p_c$, the clusters are finite, $\langle R^2(t)\rangle$ is bounded, and diffusion is impossible.

(d) Because there is no diffusion for $p < p_c$, we might expect that D_s vanishes as $p \to p_c$ from above, that is, $D_s(p) \sim (p - p_c)^{\mu_s}$ for $p \gtrsim p_c$. Extend your calculations of part (b) to larger L, more walkers (at least 1000), and more values of p near p_c and estimate the dynamical exponent μ_s.

(e) At $p = p_c$, we might expect $\langle R^2(t)\rangle$ to exhibit a different type of t-dependence, for example, $\langle R^2(t)\rangle \to t^{2/z}$ for large t. Do you expect the exponent z to be greater or less than two? Do a simulation of $\langle R^2(t)\rangle$ at $p = p_c$ and estimate z. Choose $L \geq 201$ and average over several spanning clusters.

(f) The algorithm we have been using corresponds to a "blind" ant because the ant chooses from four outcomes even if some of these outcomes are not possible. In contrast, the "myopic" ant can look ahead and see the number q of nearest neighbor occupied sites. The ant then chooses one of the q possible outcomes and thus always takes a step. Redo the simulations in part (b). Does $\langle R^2(t)\rangle$ reach its asymptotic linear dependence on t earlier or later compared to the blind ant?

(g)* The limitation of approach we have taken so far is that we have to average over different random walks $R^2(t)$ on a given cluster and also average over different clusters. A more efficient way of treating random walks on a random lattice is to use an exact enumeration approach and to consider all possible walks on a given cluster. The idea of the exact enumeration method is that $W_{t+1}(i)$, the probability that the ant is at site i at time $t+1$, is determined solely by the probabilities of the ant being at the neighbors of site i at time t. Store the positions of the occupied sites in an array and introduce two arrays corresponding to $W_{t+1}(i)$ and $W_t(i)$ for all sites i in the cluster. Use the probabilities $W_t(i)$ to obtain $W_{t+1}(i)$ (see Figure 13.10). Spatial averages such as the mean square displacement can be calculated from the probability distribution function at different times. The details of the method and the results are discussed in Majid et al., who used walks of 5000 steps on clusters with $\sim 10^3$ sites and averaged their results over 1000 different clusters.

(h)* Another reason for the interest in diffusion in disordered media is that the diffusion coefficient is proportional to the electrical conductivity of the medium. One of Einstein's many contributions was to show that the mobility, the ratio of the mean velocity of the particles in a system to an applied force, is proportional to the self-diffusion coefficient in the absence of the applied force (see Reif). For a system of charged particles, the mean velocity of the particles is proportional to the electrical current and the applied force is proportional to the voltage. Hence, the mobility and the electrical conductivity are proportional, and the conductivity is proportional to the self-diffusion coefficient.

The electrical conductivity σ vanishes near the percolation threshold as $\sigma \sim (p-p_c)^\mu$ with $\mu \approx 1.30$ (see Section 12.1). The difficulty of doing a direct Monte Carlo calculation of σ was considered in Project 12.18. We measured the self-diffusion coefficient D_s by always placing the ant on a spanning cluster rather than on *any* cluster. In contrast, the conductivity is measured for the entire system including all finite clusters. Hence, the self-diffusion coefficient D that enters into the Einstein relation should be determined by placing the ant at random anywhere on the lattice, including sites that belong to the spanning cluster and sites that belong to the many finite clusters. Because only those ants that start on the spanning cluster can contribute to D, D is related to D_s by $D = P_\infty D_s$, where P_∞ is the probability that the ant would land on a spanning cluster. Because P_∞ scales as $P_\infty \sim (p-p_c)^\beta$, we have that $(p-p_c)^\mu \sim (p-p_c)^\beta (p-p_c)^{\mu_s}$ or $\mu = \mu_s + \beta$. Use your result for μ_s found in part (d) and the exact result $\beta = 5/36$ (see Table 12.1) to estimate μ and compare your result to the critical exponent μ for the dc electrical conductivity.

(i)* We can also derive the scaling relation $z = 2 + \mu_s/\nu = 2 + (\mu - \beta)\nu$, where z is defined in part (e). Is it easier to determine μ_s or z accurately from a Monte Carlo simulation on a finite lattice? That is, if your real interest is estimating the best value of the critical exponent μ for the conductivity, should you determine the conductivity directly or should we measure the self-diffusion coefficient at $p = p_c$ or at $p > p_c$? What is your best estimate of the conductivity exponent μ? □

Diffusion limited aggregation. Many objects in nature grow by the random addition of subunits. Examples include snow flakes, lightning, crack formation along a geological fault,

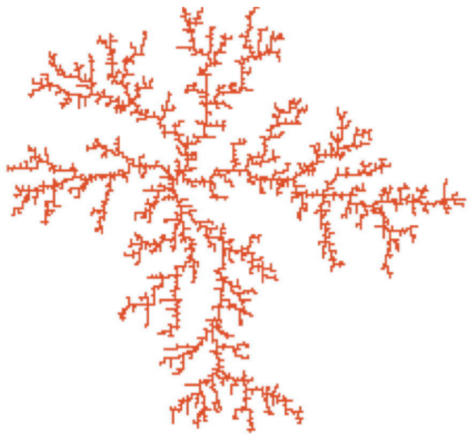


Figure 13.11: A DLA cluster of 4284 particles on a square lattice with $L = 300$.

and the growth of bacterial colonies. Although it might seem unlikely that such phenomena have much in common, the behavior observed in many models gives us clues that these and many other natural phenomena can be understood in terms of a few unifying principles. A popular model that is a good example of how random motion can give rise to beautiful self-similar clusters is known as *diffusion limited aggregation* or DLA.

The first step is to occupy a site with a seed particle. Next, a particle is released at random from a point on the circumference of a large circle whose center coincides with the seed. The particle undergoes a random walk until it reaches a perimeter site of the seed and sticks. Then another random walker is released from the circumference of a large circle and walks until it reaches a perimeter site of one of the two particles in the cluster and sticks. This process is repeated many times (typically on the order of several thousand to several million) until a large cluster is formed. A typical DLA cluster is shown in Figure 13.11. Some of the properties of DLA clusters are explored in Problem 13.9.

The following class provides a reasonably efficient simulation of DLA. Walkers begin just outside a circle of radius `startRadius` enclosing the existing cluster and centered at the seed site. If the walker moves away from the cluster, the step size for the random walker increases. If the walker wanders too far away (further than `maxRadius`), the walk is restarted.

Listing 13.5: Class for simulating diffusion limited aggregation

```java
package org.opensourcephysics.sip.ch13;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.frames.LatticeFrame;
import java.awt.Color;

public class DLAApp extends AbstractSimulation {
   LatticeFrame latticeFrame = new LatticeFrame("DLA");
   byte s[][];                          // lattice on which cluster lives
   int xOccupied[], yOccupied[];  // location of occupied sites
   int L;                               // linear dimension of lattice
   int halfL;                           // L/2
   int ringSize;             // ring size in which walkers can move
   int numberOfParticles;  // number of particles in cluster
   // radius of cluster at which walkers are started
```

```
int startRadius;
// maximum radius walker can go before a new walk is started
int maxRadius;

public void initialize() {
   latticeFrame.setMessage(null);
   numberOfParticles = 1;
   L = control.getInt("lattice size");
   startRadius = 3;
   halfL = L/2;
   ringSize = L/10;
   maxRadius = startRadius+ringSize;
   s = new byte[L][L];
   s[halfL][halfL] = Byte.MAX_VALUE;
   latticeFrame.setAll(s);
}

public void reset() {
   latticeFrame.setIndexedColor(0, Color.BLACK);
   control.setValue("lattice size", 300);
   setStepsPerDisplay(100);
   enableStepsPerDisplay(true);
   initialize();
}

public void stopRunning() {
   control.println("Number of particles = "+numberOfParticles);
   // add code to compute the mass distribution here
}

public void doStep() {
   int x = 0, y = 0;
   if(startRadius<halfL) {
      // find random initial position of new walker
      do {
         double theta = 2*Math.PI*Math.random();
         x = halfL+(int) (startRadius*Math.cos(theta));
         y = halfL+(int) (startRadius*Math.sin(theta));
         // random walk, returns true if new walk is needed
      } while(walk(x, y));
   }
   if(startRadius>=halfL) { // stop the simulation
      control.calculationDone("Done");
      latticeFrame.setMessage("Done");
   }
   latticeFrame.setMessage("n = "+numberOfParticles);
}

public boolean walk(int x, int y) {
   do {
      double rSquared = (x-halfL)*(x-halfL)+(y-halfL)*(y-halfL);
      int r = 1+(int) Math.sqrt(rSquared);
      if(r>maxRadius) {
            return true;                    // start new walker
         }
         if((r<halfL)&&(s[x+1][y]+s[x-1][y]+s[x][y+1]+s[x][y-1]>0)) {
            numberOfParticles++;
            s[x][y] = 1;
            latticeFrame.setValue(x, y, Byte.MAX_VALUE);
            if(r>=startRadius) {
               startRadius = r+2;
            }
            maxRadius = startRadius+ringSize;
            return false;                   // walk is finished
         } else {                           // take a step
            // select direction randomly
            switch((int) (4*Math.random())) {
            case 0 :
               x++;
               break;
            case 1 :
               x--;
               break;
            case 2 :
               y++;
               break;
            case 3 :
               y--;
            }
         }                              // end else if
      } while(true);                    // end do loop
   }

   public static void main(String[] args) {
      SimulationControl.createApp(new DLAApp());
   }
}
```

Problem 13.9. Diffusion limited aggregation

(a) DLAApp generates diffusion limited aggregation clusters on a square lattice. Each walker begins at a random site on a launching circle of radius $r = R_{\text{max}} + 2$, where R_{max} is the maximum distance of any particle in the cluster from the origin. To save computer time, we remove a walker that reaches a distance $2R_{\text{max}}$ from the seed site and place a new walker at random on the circle of radius r. If the clusters appear to be fractals, make a visual estimate of the fractal dimension. Choose a lattice of linear dimension $L \geq 61$. (Experts can make a visual estimate of D to within a few percent.) Modify DLAApp by color coding the sites in the cluster according to their time of arrival; for example, color the first group of sites white, the next group blue, the next group red, and the last group sites green. (Your choice of the size of the group depends in part on the total size of your cluster.) Which parts of the cluster grow faster? Do any of the late arriving green particles reach the center?

(b) At $t = 0$, the four perimeter (growth) sites on the square lattice each have a probability $p_i = 1/4$ of becoming part of the cluster. At $t = 1$, the cluster has mass two and six perimeter sites. Identify the perimeter sites and convince yourself that their growth probabilities are not the same. Do a Monte Carlo simulation and verify that two perimeter sites have growth probabilities $p = 2/9$ and the other four have $p = 5/36$. We discuss a more direct way of determining the growth probabilities in Problem 13.10.

(c) DLAApp generates clusters inefficiently because most of the CPU time is spent while the random walker is wandering far from the perimeter sites of the cluster. There are several ways of making your program more efficient. One way is to let the walker take bigger steps the further it is from the cluster. For example, if the walker is a distance $R > R_{\max}$, a step of length greater than or equal to $R - R_{\max} - 1$ may be permitted if this distance is greater than one lattice unit. If the walker is very close to the cluster, the step length is one lattice unit. Make this modification to class DLA and estimate the fractal dimension of diffusion limited clusters generated on a square lattice by computing $M(r)$, the number of sites in the cluster within a radius r centered at the seed site. Because very large clusters are needed to accurately estimate the fractal dimension, you will obtain only approximate results. Other possible modifications to make the implementation of the algorithm are discussed in Project 13.17 and by Meakin (see references).

(d)* Each time we grow a DLA cluster (and other clusters in which a perimeter site is selected at random), we obtain a slightly different cluster if we use a different random number sequence. One way of reducing this "noise" is to use "noise reduction," that is, a perimeter site not occupied until it has been visited m times. Each time the random walker lands on a perimeter site, the number of visits for this site is increased by one until the number of visits equals m and the site is occupied. The idea is that noise reduction accelerates the approach to the asymptotic scaling behavior. Consider $m = 2$, 3, 4, and 5 and grow DLA clusters on the square lattice. Are there any qualitative differences between the clusters for different values of m?

(e)* In Chapter 12 we found that the exponents describing the percolation transition are independent of the symmetry of the lattice; for example, the exponents for the square and triangular lattices are the same. We might expect that the fractal dimension of DLA clusters would also show such universal behavior. However, the presence of a lattice introduces a small anisotropy that becomes apparent only when very large clusters with the order of 10^6 sites are grown. Modify your program so that DLA clusters are generated on a triangular lattice. Do the clusters have the same visual appearance as on the square lattice? Estimate the fractal dimension and compare your estimate to your result for the square lattice. The best estimates of D for the square and triangular lattices are $D \approx 1.5$ and $D \approx 1.71$, respectively. We are reminded of the difficulty of extrapolating the asymptotic behavior from finite clusters. We consider the growth of diffusion limited aggregation clusters in the continuum in Project 13.16. □

***Laplacian growth model**. As we discussed in Section 10.6, we can formulate the solution of Laplace's equation in terms of a random walk. We now do the converse and formulate the DLA algorithm in terms of a solution to Laplace's equation. Consider the probability $P(\mathbf{r})$ that a random walker reaches a site $\mathbf{r}$ starting from the external boundary. This probability satisfies the relation

$$P(\mathbf{r}) = \frac{1}{4}\sum_{\mathbf{a}} P(\mathbf{r}+\mathbf{a}), \tag{13.10}$$

where the sum in (13.10) is over the four nearest neighbor sites (on a square lattice). If we set $P = 1$ on the boundary and $P = 0$ on the cluster, then (13.10) also applies to sites that are neighbors of the external boundary and the cluster. A comparison of the form of (13.10) with the form of (10.12) shows that the former is a discrete version of Laplace's equation $\nabla^2 P = 0$. Hence, $P(\mathbf{r})$ has the same behavior as the electrical potential between two electrodes connected to the outer boundary and the cluster, and the growth probability at a perimeter site of the cluster is proportional to the value of the potential at that site.

*__Problem 13.10.__ Laplacian growth models

(a) Solve the discrete Laplace equation (13.10) by hand for the growth probabilities of a DLA cluster of mass 1, 2, and 3. Set $P = 1$ on the boundary and $P = 0$ on the cluster. Compare your results to your results in Problem 13.9b for mass 1 and 2.

(b) You are probably familiar with the random nature of electrical discharge patterns that occur in atmospheric lightning. Although this phenomenon, known as *dielectric breakdown*, is complicated, we will see that a simple model leads to discharge patterns that are similar to those that are observed in nature. Because lightning occurs in an inhomogeneous medium with differences in the density, humidity, and conductivity of air, we will develop a model of electrical discharge in an inhomogeneous insulator. We know that when an electrical discharge occurs, the electrical potential ϕ satisfies Laplace's equation $\nabla^2\phi = 0$. One version of the model (see Family et al.) is specified by the following steps:

   (i) Consider a large boundary circle of radius R and place a charge source at the origin. Choose the potential $\phi = 0$ at the origin (an occupied site) and $\phi = 1$ for sites on the circumference of the circle. The radius R should be larger than the radius of the growing pattern.

   (ii) Use the relaxation method (see Section 10.5) to compute the values of the potential ϕ_i for (empty) sites within the circle.

   (iii) Assign a random number r to each empty site within the boundary circle. The random number r_i at site i represents a breakdown coefficient and the random inhomogeneous nature of the insulator.

   (iv) The growth sites are the nearest neighbor sites of the discharge pattern (the occupied sites). Form the product $r_i\phi_i^a$ for each growth site i, where a is an adjustable parameter. Because the potential for the discharge pattern is zero, ϕ_i for growth site i can be interpreted as the magnitude of the potential gradient at site i.

   (v) The perimeter site with the maximum value of the product $r\phi^a$ breaks down; that is, set ϕ for this site equal to zero.

   (vi) Use the relaxation method to recompute the values of the potential at the remaining unoccupied sites and repeat steps (iv) and (v).

   Choose $a = 1/4$ and analyze the structure of the discharge pattern. Does the pattern appear qualitatively similar to lightning? Does the pattern appear to have a fractal geometry? Estimate the fractal dimension by counting $M(b)$, the average number of sites belonging to the discharge pattern that are within a $b\times b$ box. Consider other values of a, for example, $a = 1/6$ and $a = 1/3$, and show that the patterns have a fractal structure with a tunable fractal dimension that depends on the parameter a. Published results (Family et al.) are for patterns with 800 occupied sites.

(c) Another version of the dielectric breakdown model associates a growth probability $p_i = \phi_i^a/\sum_j \phi_j^a$ with each growth site i, where the sum is over all the growth sites. One of the growth sites is occupied with probability p_i. That is, choose a growth site at random and generate a random number r between 0 and 1. If $r \leq p_i$, the growth site i is occupied. As before, the exponent a is a free parameter. Convince yourself that $a = 1$ corresponds to diffusion limited aggregation. (The boundary condition used in the latter corresponds to a zero potential at the growth sites.) To what type of cluster does $a = 0$ correspond? Consider $a = 1/2$, 1, and 2 and explore the dependence of the visual appearance of the clusters on a. Estimate the fractal dimension of the clusters.

(d) Consider a deterministic growth model for which *all* growth sites are tested for occupancy at each growth step. Adopt the same geometry and boundary conditions as in part (b) and use the relaxation method to solve Laplace's equation for ϕ_i. Then find the perimeter site with the largest value of ϕ and set $\phi_{\max}$ equal to this value. Only those perimeter sites for which the ratio $\phi_i/\phi_{\max}$ is larger than a parameter p become part of the cluster; ϕ_i is set equal to unity for these sites. After each growth step, the new growth sites are determined and the relaxation method is used to recompute the values of ϕ_i at each unoccupied site. Choose $p = 0.35$ and determine the nature of the regular fractal pattern. What is the fractal dimension? Consider other values of p and determine the corresponding fractal dimension. These patterns have been termed *Laplace fractal carpets* (see Family et al.). □

Surface growth models. The fractal objects we have discussed so far are self-similar' that is, if we look at a small piece of the object and magnify it isotropically to the size of the original, the original and the magnified object look similar (on the average). In the following, we introduce some simple models that generate a class of fractals that are self-similar only for scale changes in certain directions.

Suppose that we have a flat surface at time $t = 0$. How does the surface grow as a result of vapor deposition and sedimentation? For example, consider a surface that is initially a line of L occupied sites. Growth is in the vertical direction only (see Figure 13.12).

As before, we simply choose a growth site at random and occupy it (the Eden model again). The average height of the surface is given by

$$\overline{h} = \frac{1}{N_s}\sum_{i=1}^{N_s} h_i, \tag{13.11}$$

where h_i is the distance of the ith surface site from the substrate, and the sum is over all surface sites N_s. (The precise definition of a surface site is discussed in Problem 13.11.)

Each time a particle is deposited, the time t is increased by unity. Our main interest is how the width of the surface changes with t. We define the width of the surface by

$$w^2 = \frac{1}{N_s}\sum_{i=1}^{N_s}(h_i - \overline{h})^2. \tag{13.12}$$

In general, the width w, which is a measure of the surface roughness, depends on L and t. For short times we expect that

$$w(L,t) \sim t^{\beta}. \tag{13.13}$$

The exponent β describes the growth of the correlations with time along the vertical direction.

Figure 13.12 illustrates the evolution of the surface generated according to the Eden model. After a characteristic time, the length over which the fluctuations are correlated becomes comparable to L, and the width reaches a steady-state value that depends only on L. We write

$$w(L,t \gg 1) \sim L^{\alpha}, \tag{13.14}$$

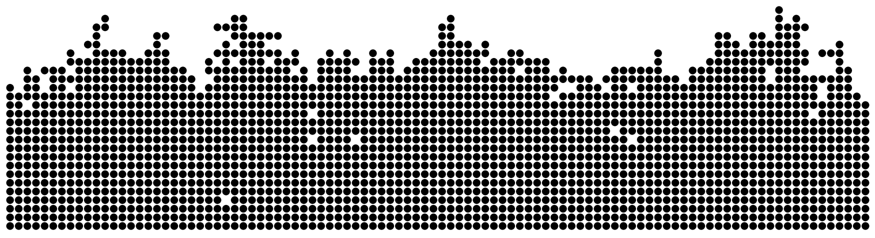


Figure 13.12: Example of surface growth according to the Eden model. The surface site in column i is the perimeter site with the maximum value of h_i. In the figure the average height of the surface is 20.46 and the width is 2.33.

where α is known as the roughness exponent.

From (13.14) we see that in the steady state, the width of the surface in the direction perpendicular to the substrate grows as L^α. This scaling behavior of the width is characteristic of a *self-affine fractal*. Such a fractal is invariant (on the average) under anisotropic scale changes; that is, different scaling relations exist along different directions. For example, if we rescale the surface by a factor b in the horizontal direction, then the surface must be rescaled by a factor of b^α in the direction perpendicular to the surface to preserve the similarity along the original and rescaled surfaces.

Note that on short length scales, that is, lengths shorter than the width of the interface, the surface is rough and its roughness can be characterized by the exponent α. (Imagine an ant walking on the surface.) For length scales much larger than the width of the surface, the surface appears to be flat and, in our example, it is a one-dimensional object. The properties of the surface generated by several growth models are explored in Problem 13.11.

Problem 13.11. Growing surfaces

(a) In the Eden model a perimeter site is chosen at random and occupied. The growth rule is the same as the usual Eden model, but the growth is started from a line of length L rather than a single site. Hence, there can be "overhangs" as shown in Figure 13.12. Use periodic boundary conditions in the horizontal direction to determine the perimeter sites. The height h_i corresponds to the height of column i. Consider $L = 64$. Describe the visual appearance of the surface as the surface grows. Is the surface well defined visually? Where are most of the perimeter sites?

(b) To estimate the exponents α and β, plot the width $w(t)$ as a function of t for $L = 32$, 64, and 128 on the same graph. What type of plot is most appropriate? Does the width initially grow as a power law? If so, estimate the exponent β. Is there a L-dependent crossover time after which the width of the surface approaches its steady-state value? How can you estimate the exponent α? The best numerical estimates for β and α are consistent with the exact values $\beta = 1/3$ and $\alpha = 1/2$.

(c)* The dependence of $w(L,t)$ on t and L can be combined into the scaling form

$$w(L,t) \approx L^\alpha f(t/L^{\alpha/\beta}), \tag{13.15}$$

where

$$f(x) = \begin{cases} Ax^\beta & x \ll 1 \\ \text{constant} & x \gg 1, \end{cases} \tag{13.16}$$

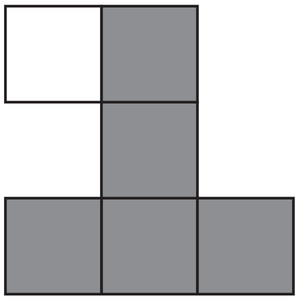


Figure 13.13: Example of the growth of a surface according to the ballistic deposition model. Note that if column one is chosen, the next site that would be occupied (not shaded) would leave an unoccupied site below it.

where A is a constant. Verify the existence of the scaling form (13.15) by plotting the ratio $w(L,t)/L^\alpha$ versus $t/L^{\alpha/\beta}$ for the different values of L considered in part (b). If the scaling form holds, the results for w for the different values of L should fall on a universal curve. Use either the estimated values of α and β that you found in part (b) or the exact values.

(d) The Eden model is not really a surface growth model, because any perimeter site can become part of the cluster. In the simplest *random deposition* model, a column is chosen at random and a particle is deposited at the top of the column of already deposited particles. There is no horizontal correlation between neighboring columns. Do a simulation of this growth model and visually inspect the surface of the interface. Show that the heights of the columns follow a Poisson distribution [see (7.31)] and that $\overline{h} \sim t$ and $w \sim t^{1/2}$. This structure does not depend on L and hence $\alpha = 0$.

(e) In the *ballistic deposition* model, a column is chosen at random and a particle is assumed to fall vertically until it reaches the first perimeter site that is a nearest neighbor of a site that already is part of the surface. This condition allows for growth parallel to the substrate. Only one particle falls at a time. How do the rules for this growth model differ from those of the Eden model? How does the surface compare to that of the Eden model? Suppose that instead of the particle falling vertically, we let it do a random walk as in DLA. Would the resultant surface be the same? □

13.4 Fractals and Chaos

In Chapter 6 we explored dynamical systems that exhibited chaos under certain conditions. We found that after an initial transient, the trajectory of such a dynamical system consists of a set of points in phase space called an attractor. For chaotic motion this attractor is often an object that can be described as a fractal. Such attractors are called *strange attractors*.

We first consider the familiar logistic map [see (6.1)] $x_{n+1} = 4rx_n(1-x_n)$. For most values of the control parameter $r > r_\infty = 0.892486417967\ldots$, the trajectories are chaotic. Are these trajectories fractals?

To calculate the fractal dimension for dynamical systems, we use the *box counting* method introduced in Section 13.2 in which space is divided into d-dimensional boxes of length ℓ. Let

$N(\ell)$ equal the number of boxes that contain a piece of the trajectory. The fractal dimension is defined by the relation

$$N(\ell) \sim \lim_{\ell \to 0} \ell^{-D} \qquad \text{(box dimension).} \tag{13.17}$$

Equation (13.17) holds only when the number of boxes is much larger than $N(\ell)$ and the number of points on the trajectory is sufficiently large. If the trajectory moves through many dimensions, that is, the phase space is very large, box counting becomes too memory intensive because we need an array of size $\propto \ell^{-d}$. This array becomes very large for small ℓ and large d.

A more efficient approach is to compute the *correlation dimension*. In this approach we store in an array the position of N points on the trajectory. We compute the number of points $N_i(r)$, and the fraction of points $f_i(r) = N_i(r)/(N-1)$ within a distance r of the point i. The correlation function $C(r)$ is defined by

$$C(r) \equiv \frac{1}{N} \sum_i f_i(r), \tag{13.18}$$

and the *correlation dimension* D_c is defined by

$$C(r) \sim \lim_{r \to 0} r^{D_c} \qquad \text{(correlation dimension).} \tag{13.19}$$

From (13.19) we see that the slope of a log-log plot of $C(r)$ versus r yields an estimate of the correlation dimension. In practice, small values of r must be discarded because we cannot sample all of the points on the trajectory, and hence there is a cutoff value of r below which $C(r) = 0$. In the large r limit, $C(r)$ saturates to unity if the trajectory is localized as it is for chaotic trajectories. We expect that for intermediate values of r, there is a scaling regime where (13.19) holds.

In Problems 13.12–13.14, we consider the fractal properties of some of the dynamical systems that we considered in Chapter 6.

Problem 13.12. Strange attractor of the logistic map

(a) Write a program that uses box counting to determine the fractal dimension of the attractor for the logistic map. Compute $N(\ell)$, the number of boxes of length ℓ that have been visited by the trajectory. Test your program for $r < r_\infty$. How does the number of boxes containing a piece of the trajectory change with ℓ? What does this dependence tell you about the dimension of the trajectory for $r < r_\infty$?

(b) Compute $N(\ell)$ for $r = 0.9$ using at least five different values of ℓ, for example, $1/\ell = 100$, 300, 1000, 3000, ... Iterate the map at least 10,000 times before determining $N(\ell)$. What is the fractal dimension of the attractor? Repeat for $r \approx r_\infty$, $r = 0.95$, and $r = 1$.

(c) Generate points at random in the unit interval and estimate the fractal dimension using the same method as in part (b). What do you expect to find? Use your results to estimate the accuracy of the fractal dimension that you found in part (b).

(d) Write a program to compute the correlation dimension for the logistic map and repeat the calculations for parts (b) and (c). □

Problem 13.13. Strange attractor of the Hénon map

(a) Use two-dimensional boxes of linear dimension ℓ to estimate the fractal dimension of the strange attractor of the Hénon map [see (6.32)] with $a = 1.4$ and $b = 0.3$. Iterate the map at least 1000 times before computing $N(\ell)$. Does it matter what initial condition you choose?

(b) Compute the correlation dimension for the same parameters used in part (a) and compare D_c with the box dimension computed in part (a).

(c) Iterate the Hénon map and view the trajectory on the screen by plotting x_{n+1} versus x_n in one window and y_n versus x_n in another window. Do the two ways of viewing the trajectory look similar? Estimate the correlation dimension, where the ith data point is defined by (x_i, x_{i+1}) and the distance r_{ij} between the ith and jth data point is given by ${r_{ij}}^2 = (x_i - x_j)^2 + (x_{i+1} - x_{j+1})^2$.

(d) Estimate the correlation dimension with the ith data point defined by x_i and ${r_{ij}}^2 = (x_i - x_j)^2$. What do you expect to obtain for D_c? Repeat the calculation for the ith data point given by (x_i, x_{i+1}, x_{i+2}) and ${r_{ij}}^2 = (x_i - x_j)^2 + (x_{i+1} - x_{j+1})^2 + (x_{i+2} - x_{j+2})^2$. What do you find for D_c? □

*__Problem 13.14.__ Strange attractor of the Lorenz model

(a) Use three-dimensional graphics or three two-dimensional plots of $x(t)$ versus $y(t)$, $x(t)$ versus $z(t)$, and $y(t)$ versus $z(t)$ to view the structure of the Lorenz attractor. Use $\sigma = 10$, $b = 8/3$, $r = 28$, and the time step $\Delta t = 0.01$. Compute the correlation dimension for the Lorenz attractor.

(b) Repeat the calculation of the correlation dimension using $x(t)$, $x(t+\tau)$, and $x(t+2\tau)$ instead of $x(t)$, $y(t)$, and $z(t)$. Choose the delay time τ to be at least ten times greater than the time step Δt.

(c) Compute the correlation dimension in the two-dimensional space of $x(t)$ and $x(t+\tau)$. Do the same calculation in four dimensions using $x(t)$, $x(t+\tau)$, $x(t+2\tau)$, and $x(t+3\tau)$. What can you conclude about the results for the correlation dimension using two-, three-, and four-dimensional spaces? What do you expect to see for $d > 4$? □

Problems 13.13 and 13.14 illustrate a practical method for determining the underlying structure of systems when, for example, the data consists only of a single time series, that is, measurements of a single quantity over time. The dimension $D_c(d)$ computed by increasing the dimension of the space d using the delayed coordinate τ eventually saturates when d is approximately equal to the number of variables that actually determine the dynamics. Hence, if we have extensive data for a single variable, for example, the atmospheric pressure or a stock market index, we can use this method to determine the number of independent variables that determine the dynamics of the variable. This information can then be used to help create models of the dynamics.

13.5 Many Dimensions

So far we have discussed three ways of defining the fractal dimension: the mass dimension (13.1), the box dimension (13.17), and the correlation dimension (13.19). These methods do not always give the same results for the fractal dimension. Indeed, there are many other dimensions that we could compute. For example, instead of just counting the boxes that contain a part of an object, we can count the number of points of the object in each box n_i and compute $p_i = n_i/N$, where N is the total number of points. A generalized dimension D_q can be defined as

$$D_q = \frac{1}{q-1} \lim_{\ell \to 0} \frac{\ln \sum_{i=1}^{N(\ell)} p_i^q}{\ln \ell}. \tag{13.20}$$

The sum in (13.20) is over all the boxes and involves the probabilities raised to the qth power. For $q = 0$, we have

$$D_0 = -\lim_{\ell \to 0} \frac{\ln N(\ell)}{\ln \ell}. \tag{13.21}$$

If we compare the form of (13.21) with (13.17), we can identify D_0 with the box dimension. For $q = 1$, we need to take the limit of (13.20) as $q \to 1$. Let

$$u(q) = \ln \sum_i {p_i}^q, \tag{13.22}$$

and do a Taylor-series expansion of $u(q)$ about $q = 1$. We have

$$u(q) = u(1) + (q-1)\frac{du}{dq} + \cdots. \tag{13.23}$$

The quantity $u(1) = 0$ because $\sum_i p_i = 1$. The first derivative of $u(q)$ is given by

$$\frac{du}{dq} = \frac{\sum_i {p_i}^q \ln p_i}{\sum_i {p_i}^q} = \sum_i p_i \ln p_i, \tag{13.24}$$

where the last equality follows by setting $q = 1$. If we use the above relations, we find that D_1 is given by

$$D_1 = \lim_{\ell \to 0} \frac{\sum_i p_i \ln p_i}{\ln \ell} \qquad \text{(information dimension).} \tag{13.25}$$

D_1 is called the *information dimension* because of the similarity of the $p \ln p$ term in the numerator of (13.24) to the information form of the entropy.

It is possible to show that D_2 as defined by (13.20) is the same as the mass dimension defined in (13.1) and the correlation dimension D_c. That is, box counting gives D_0 and correlation functions give D_2 (cf. Sander et al.).

There are many objects in nature that differ in appearance but have similar fractal dimension. An example is the different visual appearance in three dimensions of diffusion limited aggregation clusters and the percolation clusters at the percolation threshold. (Both objects have a fractal dimension of approximately 2.5.) In some cases this difference can be accounted for by the *multifractal* properties of an object. For *multifractals* the various D_q are different, in contrast to *monofractals* for which the different measures are the same. Percolation clusters are an example of a monofractal because $p_i \sim \ell^{D_0}$, the number of boxes $N(\ell) \sim \ell^{-D_0}$, and from (13.20) $D_q = D_0$ for all q. Multifractals occur when the growth quantities are not the same throughout the object as frequently happens for the strange attractors produced by chaotic dynamics. Diffusion limited aggregation is an example of a multifractal.

13.6 Projects

Although the kinetic growth models we have considered yield beautiful pictures, there is much we do not understand. For example, the fractal dimension of DLA clusters can be calculated only by approximate theories whose accuracy is unknown. Why do the fractal dimensions have the values that we estimated by various simulations? Can we trust our numerical estimates of the various exponents, or is it necessary to consider much larger systems to obtain their true asymptotic values? Can we find unifying features for the many kinetic growth models that presently exist? What is the relation of the various kinetic growth models to physical systems? What are the essential quantities needed to characterize the geometry of an object?

One of the reasons that kinetic growth models are difficult to understand is that the final cluster typically depends on the history of the growth. We say that these models are examples of "nonequilibrium behavior." The combination of simplicity, beauty, complexity, and relevance to many experimental systems suggests that the study of fractal objects will continue to involve a wide range of workers in many disciplines.

Project 13.15. The percolation cluster size distribution

Use the Leath algorithm to determine the critical exponent τ of the cluster size distribution n_s for percolation clusters at $p = p_c$:

$$n_s \sim s^{-\tau}. \qquad (s \gg 1) \tag{13.26}$$

Modify class `SingleCluster` so that many clusters are generated and n_s is computed for a given probability p. Remember that the number of clusters of size s that are grown from a seed is the product sn_s, rather than n_s itself (see Problem 13.3a). Grow at least 100 clusters on a square lattice with $L \geq 61$. If time permits, use bigger lattices and average over more clusters and also estimate the accuracy of your estimate of τ. See Grassberger for a discussion of an extension of this approach to estimating the value of p_c in higher dimensions. □

Project 13.16. Continuum DLA

(a) In the continuum (off-lattice) version of diffusion limited aggregation. the diffusing particles are assumed to be disks of radius a. A disk executes a random walk until its center is within a distance $2a$ of the center of a disk that is already a part of the DLA cluster. At each step the walker changes its position by $(r\cos\theta, r\sin\theta)$, where r is the step size, and θ is a random variable between 0 and 2π. Modify your DLA program or class `DLAApp` to simulate continuum DLA.

(b) Compare the appearance of a continuum DLA cluster with a DLA cluster generated on a square lattice. It is necessary to grow very large clusters (approximately 10^6 particles) to see the differences.

(c) Use the mass dimension to estimate the fractal dimension of continuum DLA clusters and compare its value with the value you found for the square lattice. □

Project 13.17. More efficient simulation of DLA

To improve the efficiency of the algorithm, the walker in class `DLAApp` is restarted if it wanders too far from the existing cluster. When the walker is within the distance `startRadius` of the seed, no optimization is used. Because there can be many unoccupied sites within this distance, it is desirable to use an additional optimization technique (see Ball and Brady). The idea is to choose a simple geometrical object (a circle or square) centered at the walker such that none of the cluster is within the object. The walker moves in one step to a site on the boundary of the object. For a circle the walker can move with equal probability to any location on the circumference. For the square we need the probability of moving to various locations on the boundary. To find the largest object that does not contain a part of the DLA cluster, consider coarse grained lattices. For example, each 2×2 group of sites on the original lattice corresponds to one site on the coarser lattice; each 2×2 group of sites on the coarse lattice corresponds to a site on an even coarser lattice, etc. If a site is occupied, then any coarse grained site containing this site is also occupied.

(a) Because we have considered DLA clusters on a square lattice, we use squares centered at the walker. We first find the probability $p(\Delta x, \Delta y, s)$ that a walker centered on a square of length $l = 2s + 1$, will be displaced by the $(\Delta x, \Delta y)$. This probability can be computed by simulating a random walk starting at the origin and ending at a boundary site of the square. Repeat this simulation for many walkers and then for various values of s. The fraction of walkers that reach the position $(\Delta x, \Delta y)$ is $p(\Delta x, \Delta y, s)$. Determine $p(\Delta x, \Delta y, s)$ for $s = 1$ to 16. Store your results in a file.

(b) We next determine the arrays such that for a given value of s and a uniform random number r, we can quickly find $(\Delta x, \Delta y)$. One way to do so is to create four arrays. The first array lists the probability determined from part (a) such that the values for $s = 1$ are listed first. Call this array `p`. For example, `p[1]` $= p(-1,-1,1)$, `p(2)` $=$ `p(1)` $+ p(-1,0,1)$, `p[3]` $=$ `p[2]` $+ p(-1,1,1)$, etc. The array `start` tells us where to start in the array `p` for each value of `s`. The arrays `dx(i)` and `dy(i)` give the values of Δx and Δy corresponding to `p[i]`. To see how these arrays are used, consider a walker located at (x, y) centered on a square of linear dimension $2s + 1$. Generate a random number r and find `i = start(s)`. If $r <$ `p[i]`, then the walker moves to `(x + dx(i),y + dy(i))`. If not, increment `i` by unity and check again. Repeat until $r \leq$ `p[i]`. Write a program to create these four arrays and store them in a file.

(c) Write a method to determine the maximum value of the parameter s such that a square of size $2s + 1$ centered at the position of the walker does not contain any part of the DLA cluster. Use coarse grained lattices to do this determination more efficiently. Modify class `DLA` to incorporate this method and the arrays defined in part (b). How much faster is your modified program than the original class `DLA` for clusters of 500 and 5000 particles?

(d) What is the largest cluster you can grow on your computer in a reasonable time? Does the cluster show any evidence for anisotropy? For example, does the cluster tend to extend further along the axes or along any other direction? □

Project 13.18. Cluster-cluster aggregation

In DLA all the particles that stick to a cluster are the same size (the growth occurs by the addition of one particle at a time), and the cluster that is formed is motionless. In the following, we consider a cluster-cluster aggregation (CCA) model in which the clusters do a random walk as they aggregate.

Suppose we begin with a dilute collection of N particles. Each of these particles is initially a cluster of unit mass and does a random walk until two particles become nearest neighbors. They then stick together to form a cluster of two particles. This new cluster now moves as a single random walker with a smaller diffusion coefficient. As this process continues, the clusters become larger and fewer in number. For simplicity, we assume a square lattice with periodic boundary conditions. The CCA algorithm can be summarized as follows:

(i) Place N particles at random positions on the lattice. Do not allow a site to be occupied by more than one particle. Identify the ith particle with the ith cluster.

(ii) Check if any two clusters have particles that are nearest neighbors. If so, join these two clusters to form a single cluster.

(iii) Choose a cluster at random. Decide whether to move the cluster as discussed. If so, move it at random to one of the four possible directions. The details will be discussed in the following paragraphs.

(iv) Repeat steps (ii) and (iii) for the desired number of steps or until there is only a single cluster.

What rule should we use to decide whether to move a cluster? One possibility is to select a cluster at random and simply move it. This possibility corresponds to all clusters having the same diffusion coefficient, regardless of their mass. A more realistic rule is to assume that the diffusion coefficient of a cluster is inversely related to its mass s, for example, $D_s \propto s^{-x}$ with $x \neq 0$. A common assumption is $x = 1$. If we assume that D_s is inversely proportional to the linear dimension (radius) of the cluster, an assumption consistent with the Stokes–Einstein relation, then $x = 1/d$, where d is the spatial dimension. However, because the resultant clusters are fractals, we really should take $x = 1/D$, where D is the fractal dimension of the cluster.

To implement the cluster-cluster aggregation algorithm, we need to store the position of each particle and the cluster to which each particle belongs. In class `CCA`, which can be downloaded from ch13 directory, the position of a particle is given by its x- and y-coordinates and stored in the arrays `x` and `y`, respectively. The array element `site[x][y]` equals zero if there is no particle at (x, y); otherwise, the element equals the label of the cluster to which the particle at (x, y) belongs.

The labels of the clusters are found as follows. The array element `firstParticle(k)` gives the particle label of the first particle in cluster `k`. To determine all the particles in a given cluster, we use a data structure called a *linked list*. We implement the linked list using the array `nextParticle`, so that the value of an element of this array is the index for the next element in the linked list. The array `nextParticle` contains a series of linked lists, one for each cluster, such that `nextParticle[i]` equals the particle label of another particle in the same cluster as particle `i`. If `nextParticle[i]` = −1, there are no more particles in the cluster. To see how these arrays work, consider three particles 5, 9, and 16 which constitute cluster 4. We have `firstParticle[4]` = 5, `nextParticle[5]` = 9, `nextParticle[9]` = 16, and `nextParticle[16]` = -1.

As the clusters undergo a random walk, we need to check if any pair of particles in different clusters have become nearest neighbors. If such a situation occurs, their respective clusters have to be merged. The check for nearest neighbors is done in method `checkNeighbors`. If `site[x][y]` and `site[x+1][y]` are both nonzero and are not equal, then the two clusters associated with these sites need to be combined. To do so, we add the particles of the smaller cluster to those of the larger cluster. We use another array, `lastParticle`, to keep track of the last particle in a cluster. The merger can be accomplished by the following statements:

```
// link last particle of larger cluster to first particle
//  of smaller cluster
nextParticle[lastparticle[largerClusterLabel]] =
    firstParticle[smallerClusterLabel];
// sets the last particle of larger cluster to the last particle
// of smaller cluster
```

```
lastParticle[largerClusterLabel] = lastParticle[smallerClusterLabel];
// adds mass of smaller cluster to the larger cluster
mass[largerClusterLabel] += mass[smallerClusterLabel];
```

To complete the merger, all the entries in `site[x][y]` corresponding to the smaller cluster are relabeled with the label for the larger cluster, and the last cluster in the list is relabeled by the label of the small cluster, so that if there are n clusters they are labeled by $0, 1, \ldots, n-1$.

(a) Write a target class for class `CCA`. The class assumes that the diffusion coefficient is independent of the cluster mass. Choose $L = 50$ and $N = 500$ and describe the qualitative appearance of the clusters as they form. Do they appear to be fractals? Compare their appearance to DLA clusters.

(b) Compute the fractal dimension of the final cluster. Use the center of mass $\mathbf{r}_{\rm cm}$ as the origin of the cluster, where $\mathbf{r}_{\rm cm} = (1/N)\big(\sum_i x_i, \sum_i y_i\big)$ and (x_i, y_i) is the position of the ith particle. Average your results over at least ten final clusters. Do the same for other values of L and N. Are the clusters formed by cluster-cluster aggregation more or less space filling than DLA clusters?

(c) Assume that the diffusion coefficient of a cluster of s particles varies as $D_s \propto s^{-1/2}$ in two dimensions. Let $D_{\max}$ be the diffusion coefficient of the largest cluster. Choose a random number r between 0 and 1 and move the cluster if $r < D_s/D_{\max}$. Repeat the simulations in part (a) and discuss any changes in your results. What effect does the dependence of D on s have on the motion of the clusters? □

References and Suggestions for Further Reading

We have considered only a few of the models that lead to self-similar patterns. Use your imagination to design your own model of real-world growth processes. We encourage you to read the research literature and the many books on fractals.

R. C. Ball and R. M. Brady, "Large scale lattice effect in diffusion-limited aggregation," J. Phys. A **18**, L809–L813 (1985). The authors discuss the optimization algorithm used in Project 13.17.

Albert–László Barabási and H. Eugene Stanley, *Fractal Concepts in Surface Growth*, Cambridge University Press (1995).

J. B. Bassingthwaighte, L. S. Liebovitch, and B. J. West, *Fractal Physiology* Oxford University Press (1994).

D. Ben–Avraham and S. Havlin, *Diffusion and Reactions in Fractals and Disordered Systems*, Cambridge University Press (2005).

K. S. Birdi, *Fractals in Chemistry, Geochemistry, and Biophysics*, Plenum Press (1993).

Armin Bunde and Shlomo Havlin, editors, *Fractals and Disordered Systems*, revised edition, Springer–Verlag (1996).

Fereydoon Family and David P. Landau, editors, *Kinetics of Aggregation and Gelation*, North–Holland (1984). A collection of research papers that give a wealth of information, pictures, and references on a variety of growth models.

Fereydoon Family, Daniel E. Platt, and Tamás Vicsek, "Deterministic growth model of pattern formation in dendritic solidification," J. Phys. A **20**, L1177–L1183 (1987). The authors discuss the nature of Laplace fractal carpets.

Fereydoon Family and Tamás Vicsek, editors, *Dynamics of Fractal Surfaces*, World Scientific (1991). A collection of reprints.

Fereydoon Family, Y. C. Zhang, and Tamás Vicsek, "Invasion percolation in an external field: Dielectric breakdown in random media," J. Phys. A. **19**, L733–L737 (1986).

Jens Feder, *Fractals*, Plenum Press (1988). This text discusses the applications as well as the mathematics of fractals.

Gary William Flake, *The Computational Beauty of Nature*, MIT Press (2000).

J.–M. Garcia–Ruiz, E. Louis, P. Meakin, and L. M. Sander, editors, *Growth Patterns in Physical Sciences and Biology*, NATO ASI Series B304, Plenum (1993).

Peter Grassberger, "Critical percolation in high dimensions," Phys. Rev. E **67**, 036101-1–4 (2003). The author uses the Leath algorithm to estimate the value of p_c.

Thomas C. Halsey, "Diffusion limited aggregation: A model for pattern formation," Physics Today **53** (11), 36 (2000).

J. M. Hammersley and D. C. Handscomb, *Monte Carlo Methods*, Methuen (1964). The chapter on percolation processes discusses a growth algorithm for percolation.

H. J. Herrmann, "Geometrical cluster growth models and kinetic gelation," Physics Reports **136**, 153–224 (1986).

Robert C. Hilborn, *Chaos and Nonlinear Dynamics*, second edition, Oxford University Press (2000).

Ofer Malcai, Daniel A. Lidar, Ofer Biham, and David Avnir, "Scaling range and cutoffs in empirical fractals," Phys. Rev. E, **56**, 2817–2828 (1997). The authors show that experimental reports of fractal behavior are typically based on a scaling range that spans only 0.5–2 decades and discuss the possible implications of this limited scaling range.

Benoit B. Mandelbrot, *The Fractal Geometry of Nature*, W. H. Freeman (1983). An influential and beautifully illustrated book on fractals.

Imtiaz Majid, Daniel Ben-Avraham, Shlomo Havlin, and H. Eugene Stanley, "Exact-enumeration approach to random walks on percolation clusters in two dimensions," Phys. Rev. B **30**, 1626 (1984).

Paul Meakin, *Fractals, Scaling and Growth Far From Equilibrium*, Cambridge University Press (1998). Also see P. Meakin, "The growth of rough surfaces and interfaces," Physics Reports **235**, 189–289 (1993). The author has written many seminal articles on DLA and similar models.

L. Niemeyer, L. Pietronero, and H. J. Wiesmann, "Fractal dimension of dielectric breakdown," Phys. Rev. Lett. **52**, 1033 (1984).

H. O. Peitgen and P. H. Richter, *The Beauty of Fractals*, Springer-Verlag (1986).

Luciano Pietronero and Erio Tosatti, editors, *Fractals in Physics*, North–Holland (1986). A collection of research papers, many of which are accessible to the motivated reader.

Raissa M. D'Souza, "Anomalies in simulations of nearest neighbor ballistic deposition," Int. J. Mod. Phys. C **8** (4), 941–951 (1997). The author finds that ballistic deposition is a sensitive physical test for correlations present in pseudorandom sequences.

F. Reif, *Fundamentals of Statistical and Thermal Physics*, McGraw-Hill (1965). Einstein's relation between the diffusion and mobility is discussed in Chapter 15.

John C. Russ, *Fractal Surfaces*, Plenum Press (1994).

Evelyn Sander, Leonard M. Sander, and Robert M. Ziff, "Fractals and Fractal Correlations," Computers in Physics **8**, 420–425 (1994). An introduction to fractal growth models and the calculation of their properties.

H. Eugene Stanley and Nicole Ostrowsky, editors, *On Growth and Form*, Martinus Nijhoff Publishers, Netherlands (1986). A collection of research papers at the same level as the 1984 Family and Landau collection.

Hideki Takayasu, *Fractals in the Physical Sciences*, John Wiley & Sons (1990).

David D. Thornburg, *Discovering Logo*, Addison–Wesley (1983). The book is more accurately described by its subtitle, *An Invitation to the Art and Pattern of Nature*. The nature of recursive procedures and fractals are discussed using many simple examples.

Donald L. Turcotte, *Fractals and Chaos in Geology and Geophysics*, Cambridge University Press (1992).

Tamás Vicsek, *Fractal Growth Phenomena*, second edition, World Scientific Publishing (1992). This book contains an accessible introduction to diffusion limited and cluster-cluster aggregation.

Bruce J. West, *Fractal Physiology and Chaos in Medicine*, World Scientific Publishing (1990).

David Wilkinson and Jorge F. Willemsen, "Invasion percolation: A new form of percolation theory," J. Phys. A **16**, 3365–3376 (1983).

Yu-Xia Zhang, Jian-Ping Sang, Xian-Wu Zou, and Zhun-Zhi Jin, "Random walk on percolation under an external field," Physica A **350**, 163–172 (2005). The authors consider random walks with a drift.

Chapter 14

Complex Systems

We introduce cellular automata, neural networks, genetic algorithms, and growing networks to explore the concepts of self-organization and complexity. Applications to sandpiles, fluids, earthquakes, and other areas are discussed.

14.1 Cellular Automata

Part of the fascination of physics is that it allows us to reduce natural phenomena to a few simple laws. It is also fascinating to think about how a few simple laws can produce the enormously rich behavior that we see in nature. In this chapter we will discuss several models that illustrate some of the new ideas that are emerging from the study of *complex systems*.

The first class of models that we will discuss are known as *cellular automata*. Cellular automata were introduced by von Neumann and Ulam in 1948 and are mathematical idealizations of dynamical systems in which space and time are discrete and the quantities of interest have a finite set of discrete values that are updated according to a local rule. A cellular automaton can be thought of as a lattice of sites or a checkerboard with colored squares (the cells). Each cell changes its state at the tick of an external clock according to a rule based on the present configuration of the cells in its neighborhood. Cellular automata are examples of discrete dynamical systems that can be simulated exactly on a digital computer.

Because the original motivation for studying cellular automata was their biological aspects, the discrete locations in space are frequently referred to as cells. More recently, cellular automata have been applied to a wide variety of physical systems ranging from fluids to galaxies. We will usually refer to sites rather then cells, except when we are explicitly discussing biological systems. The important characteristics of cellular automata include the following:

1. Space is discrete and consists of a regular array of sites. Each site has a finite set of values.
2. The rule for the new value of a site depends only on the values of a *local* neighborhood of sites near it.
3. Time is discrete. The variables at each site are updated *simultaneously* based on the values of the variables at the previous time step. Hence, the state of the entire lattice advances in discrete time steps.

t:	111	110	101	100	011	010	001	000
t + 1:	0	1	0	1	1	0	1	0

Figure 14.1: Example of a local rule for the evolution of a one-dimensional cellular automaton. The variable at each site can have values 0 or 1. The top row shows the $2^3 = 8$ possible combinations of three sites. The bottom row gives the value of the central site at the next iteration. For example, if the value of a site is 0 and its left neighbor is 1 and its right neighbor is 0, the central site will have the value 1 in the next time step. This rule is termed 01011010 in binary notation (see the second row), the modulo-two rule or rule 90. Note that 90 is the base ten (decimal) equivalent of the binary number 01011010; that is, $90 = 2^1 + 2^3 + 2^4 + 2^6$.

We first consider one-dimensional cellular automata and assume that the neighborhood of a given site is the site itself and the sites immediately to the left and right of it. Each site is assumed to have two states (a Boolean automaton). An example of such a rule is illustrated in Figure 14.1, where we see that a rule can be labeled by the binary representation of the update rule for each of the eight possible neighborhoods and by the base ten equivalent of the binary representation. Because any eight digit binary number specifies a one-dimensional cellular automaton, there are $2^8 = 256$ possible rules.

Class `OneDimensionalAutomatonApp` takes the decimal representation of the rule as input and produces the rule array `update`, which is used to update each lattice site using periodic boundary conditions. The `OneDimensionalAutomatonApp` class manipulates numbers using their binary representation. Note the use of the bit manipulation operators `>>` and `& (AND)` in method `setRule`. To understand how the right shift operator `>>` works, consider the expression `13 >> 1`. In this case the result of the shift operator is to shift the bits of the binary representation of the integer 13 to the right by one. Because the binary representation of 13 is 1101, the result of the shift operator is 0110. (The left-hand bits are filled with 0s as needed.) To understand the nature of the `&` operator, consider the expression `0110 & 1`, which we can write as `0110 & 0001`. In this case the result is `0000` because the `&` operator sets each of the the resulting bits to 1 if the corresponding bit in both operands is 1; otherwise, the bit is zero.

We use the `LatticeFrame` class to represent the sites and their evolution. At a given time, the sites are drawn in the horizontal direction; time increases in the vertical direction. In method `iterate`, the `%` operator is used to determine the left and right neighbors of a site using periodic boundary conditions. Also note the use of the left shift operator `<<` in method `iterate`. A more complete discussion of bit manipulation is given in Section 14.6.

Listing 14.1: One-dimensional cellular automaton class.

```java
package org.opensourcephysics.sip.ch14.ca;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.frames.*;

public class OneDimensionalAutomatonApp extends AbstractCalculation {
   LatticeFrame automaton = new LatticeFrame("");
    // update[] maps neighborhood configurations to 0 or 1
   int[] update = new int[8];

   public void calculate() {
      control.clearMessages();
      int L = control.getInt("Linear dimension");
      int tmax = control.getInt("Maximum time");
```

```
       // default is lattice sites all zero
      automaton.resizeLattice(L, tmax);
      // seed lattice by putting 1 in middle of first row
      automaton.setValue(L/2, 0, 1);
      // choose color of empty and occupied sites
      automaton.setIndexedColor(0, java.awt.Color.YELLOW); // empty
      automaton.setIndexedColor(1, java.awt.Color.BLUE);   // occupied
      setRule(control.getInt("Rule number"));
      for(int t = 1;t<tmax;t++) {
         iterate(t, L);
      }
   }

   public void iterate(int t, int L) {
      for(int i = 0;i<L;i++) {
         // read the neighborhood bits around index i, using periodic b.c's
         int left = automaton.getValue((i-1+L)%L, t-1);
         int center = automaton.getValue(i, t-1);
         int right = automaton.getValue((i+1)%L, t-1);
         // encode left, center, and right bits into one integer value
         // between 0 and 7
         int neighborhood = (left<<2)+(center<<1)+(right<<0);
         // update[neighborhood] gives the new site value for this neighborhood
         automaton.setValue(i, t, update[neighborhood]);
      }
   }

   public void setRule(int ruleNumber) {
      control.println("Rule = "+ruleNumber+"\n");
      control.println("111   110   101   100   011   010   001   000");
      for(int i = 7;i>=0;i--) {
         // (ruleNumber >>> i) shifts the contents of ruleNumber to the right by i
         // bits. In particular, the ith bit of ruleNumber resides in the rightmost
         // position of this expression. After "and"ing with the number 1, we are
         // left with either the number 0 or 1, depending on whether the ith
         // bit of ruleNumber was cleared or set.
         update[i] = ((ruleNumber>>>i)&1);
         control.print("  "+update[i]+"     ");
      }
      control.println();
   }

   public void reset() {
      control.setValue("Rule number", 90);
      control.setValue("Maximum time", 100);
      control.setValue("Linear dimension", 500);
   }

   public static void main(String args[]) {
      CalculationControl.createApp(new OneDimensionalAutomatonApp());
   }
}
```

The properties of all 256 one-dimensional cellular automata have been cataloged (see Wolfram, 1984). We explore some of the properties of one-dimensional cellular automata in Problems 14.1 and 14.3.

Problem 14.1. One-dimensional cellular automata

(a) What is the result of 13 `&` 12, 33 `>>>` 1 (decimal representation) and 1101 & 0111 (binary representation)? Consider rule 90 and work out by hand the values of `update[]` according to method `setRule`.

(b) Use `OneDimensionalAutomatonApp` and consider rule 90 shown in Figure 14.1. This rule is also known as the modulo-two rule because the value of a site at step $t+1$ is the sum modulo 2 of its two neighbors at step t. Choose the initial configuration to be a single nonzero site (the seed) at the midpoint of the lattice. It is sufficient to consider the evolution for approximately twenty iterations. Is the resulting pattern of nonzero sites self-similar? If so, characterize the pattern by a fractal dimension.

(c) Determine the properties of a rule for which the value of a site at step $t+1$ is the sum modulo 2 of the values of its neighbors plus its own value at step t. This rule is equivalent to 10010110 or rule $150 = 2^1 + 2^2 + 2^4 + 2^7$. Start with a single seed site.

(d) Choose a random initial configuration for which the independent probability for each site to have the value 1 is $p = 1/2$; otherwise, the value of the site is 0. Determine the evolution of rule 90, rule 150, rule $18 = 2^1+2^4$ (00010010), rule $73 = 2^0+2^3+2^6$ (01001001), and rule 136 (10001000). How sensitive are the patterns that are formed to the initial conditions? Does the nature of the patterns depend on the use or nonuse of periodic boundary conditions? □

Listing 14.2: A more efficient implementation of method iterate in `OneDimensionalAutomatonApp`.

```
public void iterate(int t, int L) {
   // encodes state(L-1) and state(0) in second and first bits
   // of neighborhood variable
   int neighborhood = (automaton.getValue(L-1,t-1)<<1) +
      automaton.getValue(0,t-1);
   for (int i = 0; i < L; i++) {
     // clear third bit of neighborhood, but keep second and first bits
     neighborhood = neighborhood & 3;
     // shift second and first bits of neighborhood to third
     // and second bits
     neighborhood = neighborhood << 1;
     // encode state(i+1) into first bit of neighborhood using
     // periodic boundary conditions
     neighborhood += automaton.getValue((i+1)%L, t-1);
     // neighborhood now encodes the three bits of state surrounding
     // index i at time t-1. with neighborhood as an index, the
     // update[] table gives us the state at index i and time t.
     automaton.setValue(i,t,update[neighborhood]);
   }
}
```

Method `iterate` in class `OneDimensionalAutomatonApp` is not as efficient as possible because it does not use information about the neighborhood at site i to determine the neighborhood at site $i+1$. A more efficient implementation is given in Listing 14.2. To understand how this version of method `iterate` works, suppose that the lattice at $t = 0$ is 1011, and we want to determine the neighborhood of the site at $i = 0$. The answer is 6 in decimal, corresponding to 110 in binary. Because of periodic boundary conditions, the index to the left of $i = 0$ is $L-1$. The expression `(automaton.getValue(L-1,t-1)«1)` yields `001 « 1 = 010` because « shifts all bits to the left. (Only 3 bits are needed to describe the neighborhood.) The statement

```
int neighborhood = (automaton.getValue(L−1,t−1)<<1) +
     automaton.getValue(0,t−1);
```

yields `010 + 001 = 011`. The effect of the statement

```
neighborhood = neighborhood & 3;
```

is to clear the third bit of the neighborhood but to keep the second and first bits: 011 & 011 = 011. In this case nothing is changed. We then shift the second and first bits of the neighborhood to the third and second bits:

```
neighborhood = neighborhood << 1;
```

and obtain `neighborhood = 110`. Finally the statement

```
neighborhood += automaton.getValue((i+1)%L, t−1);
```

gives `neighborhood = 011 + 000 = 011`, which is 2 in decimal.

*__Problem 14.2.__ Whose time is more important?

(a) Work out another example to make sure that you understand the nature of the bit manipulations that are used in Listing 14.1 and in the more efficient version of method `iterate`.

(b) Which version of method `iterate` would you use, the more efficient but more difficult to understand (and debug) version or the less efficient but easier to understand version? What is more important, computer time or programmer time? In general, the answer depends on the context. □

The dynamical behavior of many of the 256 one-dimensional Boolean cellular automata is uninteresting, and hence we also consider one-dimensional Boolean cellular automata with larger neighborhoods (including the site itself). Because a larger neighborhood implies that there are many more possible update rules, we place some reasonable restrictions on the rules. First, we assume that the rules are symmetrical; for example, the neighborhood 100 produces the same value for the central site as 001. We also require that the zero neighborhood 000 yields 0 for the central site, and that the value of the central site depends only on the sum of the values of the sites in the neighborhood; for example, 011 produces the same value for the central site as 101 (see Wolfram, 1984).

A simple way of coding the rules that is consistent with these requirements is as follows. Each rule is labeled by a sequence of 0s and 1s such that the sequence indicates which sums set the central site equal to 1. If the lowest order digit is 1, then the central site is set to 1 if the sum is 0. If the next digit is 1, then the central site is set to 1 if the sum is 1, etc. For example, the rule 10110 indicates that the central site will be set to 1 if the number of neighbors equal to 1 is 1, 2, or 4.

Problem 14.3. More one-dimensional cellular automata

(a) Modify class `OneDimensionalAutomatonApp` so that it incorporates the possible rules discussed in the text based on the number of sites equal to 1 in a neighborhood of $2z+1$ sites. How many possible rules are there for $z = 1$? Choose $z = 1$ and a random initial configuration and determine if the long time behavior for each rule belongs to one of the following categories:

   (i) A homogeneous state where every site has the same value. An example is rule 1000.

   (ii) A pattern consisting of separate stable or periodic regions. An example is rule 0100.

   (iii) A chaotic, aperiodic pattern. An example is rule 1010.

   (iv) A set of complex, localized structures that may not live forever. There are no examples for $z = 1$.

(b) Modify your program so that $z = 2$. Wolfram (1984) claims that rules 010100 and 110100 are the only examples of complex behavior (category 4). Describe how the behavior of these two rules differs from the behavior of the other rules. Find at least one rule for each of the four categories. □

The results of Problem 14.3 suggests that an important feature of cellular automata is their capability for self-organization. In particular, the class of complex localized structures is distinct from regular as well as aperiodic structures.

An important idea of complexity theory is that simple rules can lead to complex behavior. This complex behavior is not random but has structure. Are there "coarse grained" descriptions that can predict the dynamical behavior of these systems, or do we have to implement the model on a computer using the dynamical rules at the lowest level of description? For example, our understanding of the flow of a fluid through a pipe would be very limited if the only way we could obtain information about the behavior of fluids was to solve the equations of motion for all the individual particles. In this case there is a coarse grained description of fluids where the fundamental fluid variables are not the individual positions and velocities of the molecules, but rather a velocity field which can be interpreted as a spatial average over the velocities of many particles. The resultant partial differential equation of fluid mechanics, known as the Navier–Stokes equation, provides the coarse grained description, which can be solved, in principle, to predict the motion of the fluid.

Is there an analogous coarse grained description of a cellular automaton? Israeli and Goldenfeld have found some examples for which a coarse grained description exists. We first simulate a cellular automaton that produces complex structures. Then we start with the same initial state and create a coarse grained lattice such that each of its cells is a coarse grained description of a group of cells on the original lattice. The idea is to determine a different update rule to evolve the coarse grained lattice such that the configurations of the coarse grained lattice are identical to the coarse grained configurations of the original lattice that were obtained using the original update rule. If it is possible to implement this procedure in general, we would be better able to develop theories of complex macroscopic systems without needing to know the details of the dynamics of the microscopic constituents that make up these systems. We explore two examples in Problem 14.4.

*__Problem 14.4.__ Coarse graining one-dimensional cellular automata

(a) Add methods to `OneDimensionalAutomatonApp` that create a coarse grained lattice such that groups of three cells are coarse grained to 1 if all three cells are 1 and coarse grained to 0 otherwise. Allow the coarse grained lattice to evolve separately using a different update rule than the original lattice. The coarse grained lattice should be updated after every three updates of the original lattice. Draw the coarse grained lattice as a space-time diagram similar to what we have done for the original lattice, such that each cell in the coarse grained lattice is three times the size of a cell on the original lattice in both the space and time directions. Use rule 146 (10010010) for the original lattice and rule 128 (10000000) for the coarse grained lattice. Choose a lattice size L that is a multiple of 3 and run for a time that is a multiple of 3. You should see similar patterns in the two lattices, although the original lattice contains some details that are washed out by the coarse grained lattice. If you coarse grain the original lattice cells at each time step, you will obtain the same pattern as the coarse grained lattice.

(b) Modify your program such that each pair of cells is coarse grained to 1 if two original cells are both 0 or both 1 and coarse grained to 0 otherwise. Use rule 105 (01101001) on the original cells with $L = 120$ for 60 iterations and run the coarse grained system using rule 150 (10100110). You should obtain results similar to those found in part (a). □

Traffic models. Physicists have been at the forefront of the development of a more systematic approach to the characterization and control of traffic. Much of this work was initiated at General Motors by Robert Herman in the late 1950s. The car-following theory of traffic flow that he and Elliott Montroll and others developed during this time is still used today. What has changed is the way we can implement these theories. The continuum approach used by Herman and Montroll is based on partial differential equations. An alternative that is more flexible and easier to understand is based on cellular automata.

We first consider a simple one lane highway where cars enter at one end and exit at the other end. To implement the Nagel–Schreckenberg cellular automaton model, we use integer arrays for the position x_i and velocity v_i, where i indexes a car and not a lattice site. The important input parameters of the simulation are the maximum velocity $v_{\max}$, the density of cars ρ, and the probability p of a car slowing down. This probability adds some randomization to the drivers. The algorithm implemented in class `Freeway` for the motion of each car at each iteration is as follows:

1. If $v_i < v_{\max}$, increase the velocity v_i of car i by one unit; that is, $v_i \to v_i + 1$. This change models the process of acceleration to the maximum velocity.
2. Compute the distance to the next car d. If $v_i \geq d$, then reduce the velocity to $v_i = d - 1$ to prevent crashes.
3. With probability p, reduce the velocity of a moving car by one unit. Thus, $v_i \to v_i - 1$.
4. Update the position x_i of car i so that $x_i(t+1) = x_i(t) + v_i$.

This ordering of the steps ensures that cars do not overlap.

Listing 14.3: One lane freeway class.

```
package org.opensourcephysics.sip.ch14.traffic;
```

```
import java.awt.Graphics;
import org.opensourcephysics.display.*;
import org.opensourcephysics.frames.*;
import org.opensourcephysics.display2d.*;
import org.opensourcephysics.controls.*;

public class Freeway implements Drawable {
   public int[] v, x, xtemp;
   public LatticeFrame spaceTime;
   public double[] distribution;
   public int roadLength;
   public int numberOfCars;
   public int maximumVelocity;
   public double p;              // probability of reducing velocity
   private CellLattice road;
   public double flow;
   public int steps, t;
   // number of iterations before scrolling space-time diagram
   public int scrollTime = 100;

   public void initialize(LatticeFrame spaceTime) {
      this.spaceTime = spaceTime;
      x = new int[numberOfCars];
      xtemp = new int[numberOfCars]; // used to allow parallel updating
      v = new int[numberOfCars];
      spaceTime.resizeLattice(roadLength, 100);
      road = new CellLattice(roadLength, 1);
      road.setIndexedColor(0, java.awt.Color.RED);
      road.setIndexedColor(1, java.awt.Color.GREEN);
      spaceTime.setIndexedColor(0, java.awt.Color.RED);
      spaceTime.setIndexedColor(1, java.awt.Color.GREEN);
      int d = roadLength/numberOfCars;
      x[0] = 0;
      v[0] = maximumVelocity;
      for(int i = 1;i<numberOfCars;i++) {
         x[i] = x[i-1]+d;
         if(Math.random()<0.5) {
            v[i] = 0;
         } else {
            v[i] = 1;
         }
      }
      flow = 0;
      steps = 0;
      t = 0;
   }

   public void step() {
      for(int i = 0;i<numberOfCars;i++) {
         xtemp[i] = x[i];
      }
      for(int i = 0;i<numberOfCars;i++) {
         if(v[i]<maximumVelocity) {
            v[i]++;                                   // acceleration
         }
         // distance between cars
         int d = xtemp[(i+1)%numberOfCars]-xtemp[i];
         // periodic boundary conditions, d = 0 correctly treats one
         // car on road
         if(d<=0) {
            d += roadLength;
         }
         if(v[i]>=d) {
            v[i] = d-1;                   // slow down due to cars in front
         }
         if((v[i]>0)&&(Math.random()<p)) {
            v[i]--;                      // randomization
         }
         x[i] = (xtemp[i]+v[i])%roadLength;
         flow += v[i];
      }
      steps++;
      computeSpaceTimeDiagram();
   }

   public void computeSpaceTimeDiagram() {
      t++;
      if(t<scrollTime) {
         for(int i = 0;i<numberOfCars;i++) {
            spaceTime.setValue(x[i], t, 1);
         }
      } else {                                      // scroll diagram
         for(int y = 0;y<scrollTime-1;y++) {
            for(int i = 0;i<roadLength;i++) {
               spaceTime.setValue(i, y, spaceTime.getValue(i, y+1));
            }
         }
         for(int i = 0;i<roadLength;i++) {
            spaceTime.setValue(i, scrollTime-1, 0);      // zero last row
         }
         for(int i = 0;i<numberOfCars;i++) {
            spaceTime.setValue(x[i], scrollTime-1, 1); // add new row
         }
      }
   }

   public void draw(DrawingPanel panel, Graphics g) {
      if(x==null) {
         return;
      }
      road.setBlock(0, 0, new byte[roadLength][1]);
      for(int i = 0;i<numberOfCars;i++) {
         road.setValue(x[i], 0, (byte) 1);
      }
      road.draw(panel, g);
      g.drawString("Number of Steps = "+steps, 10, 20);
      g.drawString("Flow =
         "+ControlUtils.f3((double) flow/(roadLength*steps)), 10, 40);
      g.drawString("Density =
         "+ControlUtils.f3((double) numberOfCars/(roadLength)), 10, 60);
   }
}
```

The target class FreewayApp shows the movement of the cars and a space-time diagram, with time on the vertical axis and space on the horizontal axis. When the number of iterations equals scrollTime, the diagram scrolls down. The flow rate is the average of the car velocities divided by the length of the highway. Thus, two cars moving at constant velocity will have twice the flow rate of one car moving at the same velocity.

Listing 14.4: FreewayApp Class.

```
package org.opensourcephysics.sip.ch14.traffic;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.frames.*;

public class FreewayApp extends AbstractSimulation {
   Freeway freeway = new Freeway();
   DisplayFrame display = new DisplayFrame("Freeway");
   LatticeFrame spaceTime = new LatticeFrame("space", "time",
       "Space Time Diagram");

   public FreewayApp() {
      display.addDrawable(freeway);
   }

   public void initialize() {
      freeway.numberOfCars = control.getInt("Number of cars");
      freeway.roadLength = control.getInt("Road length");
      freeway.p = control.getDouble("Slow down probability");
      freeway.maximumVelocity = control.getInt("Maximum velocity");
      display.setPreferredMinMax(0, freeway.roadLength, -3, 4);
      freeway.initialize(spaceTime);
   }

   public void doStep() {
      freeway.step();
   }

   public void reset() {
      control.setValue("Number of cars", 10);
      control.setValue("Road length", 50);
      control.setValue("Slow down probability", 0.5);
      control.setValue("Maximum velocity", 2);
      control.setValue("Steps between plots", 1);
      enableStepsPerDisplay(true);
   }

   public void resetAverages() {
      freeway.flow = 0;
      freeway.steps = 0;
   }

   public static void main(String[] args) {
      SimulationControl control =
          SimulationControl.createApp(new FreewayApp());
      control.addButton("resetAverages", "resetAverages");
   }
}
```

Problem 14.5. Cellular automata traffic models

(a) Run FreewayApp for 10 cars on a road of length 50, with $v_{\text{max}} = 2$ and $p = 0.5$. Allow the system to evolve before recording the flow rate. Repeat the simulation with a different initial configuration at least several more times to estimate the uncertainty in the data. Repeat for 1, 2, 5, 20, 30, and 40 cars. Plot the flow rate versus the density. This plot is called the *fundamental diagram*. Explain its qualitative shape. At what density do traffic jams begin to occur?

(b) Repeat part (a) with a road of length 500 and the same car densities. Use other road lengths to determine the minimum road length needed to obtain results that are independent of the length of the road.

(c) Add methods to your classes to compute the velocity and gap distributions, where the gap is defined as the distance between two cars.

(d) For a fixed road length, compare your results for $v_{\text{max}} = 1$ with your results for $v_{\text{max}} = 2$. Also consider $v_{\text{max}} = 5$. Are there any qualitative differences in the behavior of the cars?

(e) Explore the effect of the speed reduction probability by considering $p = 0.2$ and $p = 0.8$.

(f) Add on- and off-ramps separated by a fixed distance. One way to do so is to choose a car at random and have it slow down as it approaches the off-ramp and exits. To maintain a constant density, allow a car to enter the on-ramp whenever a car leaves the highway. What is the effect of adding the on- and off-ramps?

(g) Modify your program to simulate a two-lane highway. You will need to choose rules for moving from one lane to the other. Some possibilities to explore include the following. One reason for a car to move to the left lane is that the car is moving at less than the maximum speed and cannot increase its speed due to the car in front of it. Such a car could move to the left lane if there were a free space to the left. One reason for a car to move to the right lane is that there is a car immediately behind it. How does the behavior of the two lane highway differ from that of the one-lane highway?

(h) Modify your two-lane simulation so that there are two kinds of vehicles (for example, cars and trucks) with different values of v_{max}. How do the gap and velocity distributions change? Compute separate values for the truck and car flows as well as the total flow. Compute the average speed of the trucks and compare it with that of cars. □

Because one-dimensional cellular automata models are limited, we consider several two-dimensional models. The philosophy is the same except that the neighborhood contains more sites. For the eight neighbor sites shown in Figure 14.2a, there are $2^9 = 512$ possible configurations for the eight neighbors and the center site, and 2^{512} possible rules. Clearly, we cannot

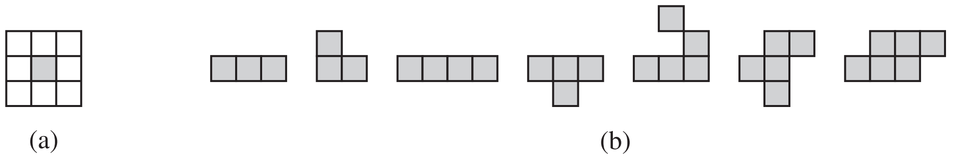


Figure 14.2: (a) The local neighborhood of a site in the Game of Life is given by the sum of its eight neighbors. (b) Examples of initial configurations for the Game of Life, some of which lead to interesting patterns. Live cells are shaded.

go through all these rules in any systematic fashion as we did for one-dimensional cellular automata. For this reason, we will choose our rules based on other considerations.

The Game of Life. The rules used in `LifeApp` implement a popular two-dimensional cellular automaton known as the *Game of Life*. This model, invented in 1970 by the mathematician John Conway, produces many fascinating patterns. The rules of the game are simple. For each cell determine the sum of the values of its four nearest and four next-nearest neighbors (see Figure 14.2a). A "live" cell (value 1) remains alive only if this sum equals 2 or 3. If the sum is greater than 3, the cell will "die" (become 0) at the next iteration due to overcrowding. If the sum is less than 2, the cell will die due to isolation. A dead cell will come to life only if the sum equals 3.

Listing 14.5: Implementation of the Game of Life.

```
package org.opensourcephysics.sip.ch14.ca;
import org.opensourcephysics.frames.*;
import org.opensourcephysics.controls.*;
import java.awt.Color;

public class LifeApp extends AbstractSimulation {
   LatticeFrame latticeFrame = new LatticeFrame("Game of Life");
   byte[][] newCells;
   int size = 16;

   public LifeApp() {
      latticeFrame.setToggleOnClick(true, 0, 1);
      latticeFrame.setIndexedColor(0, Color.RED);
      latticeFrame.setIndexedColor(1, Color.BLUE);
   }

   public void initCells(int size) {
      this.size = size;
      newCells = new byte[size][size];
      latticeFrame.setAll(newCells, 0, size, 0, size);
      latticeFrame.setValue(size/2, size/2, 1);
      latticeFrame.setValue(size/2-1, size/2, 1);
      latticeFrame.setValue(size/2+1, size/2, 1);
      latticeFrame.setValue(size/2, size/2-1, 1);
      latticeFrame.setValue(size/2, size/2+1, 1);
   }

   public void clear() {
      latticeFrame.setAll(new byte[size][size]);
      latticeFrame.repaint();
   }

   public void reset() {
      control.println("Click in drawingPanel to toggle cells.");
      control.setValue("grid size", 16);
      initCells(16);
   }

   public void initialize() {
      initCells(control.getInt("grid size"));
   }

   private int calcNeighborsPeriodic(int row, int col) {
      // do not count self
      int neighbors = -latticeFrame.getValue(row, col);
      // add the size so that the mod operator works for row = 0
      // and col = 0
      row += size;
      col += size;
      for(int i = -1;i<=1;i++) {
         for(int j = -1;j<=1;j++) {
            neighbors += latticeFrame.getValue((row+i)%size, (col+j)%size);
         }
      }
      return neighbors;
   }

   public void doStep() {
      for(int i = 0;i<size;i++) {
         for(int j = 0;j<size;j++) {
            newCells[i][j] = 0;
         }
      }
      for(int i = 0;i<size;i++) {
         for(int j = 0;j<size;j++) {
            switch(calcNeighborsPeriodic(i, j)) {
            case 0 :
            case 1 :
               newCells[i][j] = 0;                   // dies
               break;
            case 2 :
               // life goes on
               newCells[i][j] = (byte) latticeFrame.getValue(i, j);
               break;
            case 3 :
               newCells[i][j] = 1;         // condition for birth
               break;
            default :
               newCells[i][j] = 0;       // dies of overcrowding if >3
            }
         }
      }
```

```
         latticeFrame.setAll(newCells);
      }

      public static void main(String[] args) {
         OSPControl control = SimulationControl.createApp(new LifeApp());
         control.addButton("clear", "Clear"); // optional custom action
      }
   }
```

Problem 14.6. The Game of Life

(a) LifeApp allows the user to determine the initial configuration interactively by clicking on a cell to change its value before hitting the Start button. Choose several initial configurations with a small number of live cells and determine the different types of patterns that emerge. Some suggested initial configurations are shown in Figure 14.2b. Does it matter whether you use fixed or periodic boundary conditions? Use a 16×16 lattice.

(b) Modify LifeApp so that each cell is initially alive with a 50% probability. Use a 32×32 lattice. What types of patterns typically result after a long time? What happens for 20% live cells? What happens for 70% live cells?

(c) Assume that each cell is initially alive with probability p. Given that the density of live cells at time t is $\rho(t)$, what is $\rho(t+1)$, the expected density at time $t+1$? Do the simulation and plot $\rho(t+1)$ versus $\rho(t)$. If $p = 0.5$, what is the steady-state density of live cells?

(d)* LifeApp has not been optimized for the Game of Life and is written so that other rules can be implemented easily. Rewrite LifeApp so that it uses bit manipulation (see Section 14.6).

□

The Game of Life is an example of a universal computing machine. That is, we can choose an initial configuration of live cells to represent any possible program and any set of input data, run the Game of Life, and the output data will appear in some region of the lattice. The proof of this result (see Berlekamp et al.) involves showing how various configurations of cells represent the components of a computer, including wires, storage, and the fundamental components of a CPU – the digital logic gates that perform *and*, *or*, and other logical and arithmetic operations. Other cellular automata can also be shown to be universal computing machines.

14.2 Self-Organized Critical Phenomena

Very large events such as a magnitude eight earthquake, an avalanche on a snow covered mountain, the sudden collapse of an empire (for example, the Soviet Union), or the crash of the stock market are rare. When such events occur, are they due to some special set of circumstances or are they part of a more general pattern of events that would occur without any specific external intervention? The idea of *self-organized criticality* is that in many cases the occurrence of very large events does not depend on special conditions or external forces and is due to the intrinsic dynamics of the system.

If s represents the magnitude of an event, such as the energy released in an earthquake or the amount of snow in an avalanche, then a system is said to be *critical* if the number of events, $N(s)$, follows a power law:

$$N(s) \sim s^{-\alpha} \qquad \text{(no characteristic scale).} \tag{14.1}$$

If $\alpha \approx 1$, the form (14.1) implies that there would be one large event of size 1000 for every 1000 events of size one. One implication of the power law form (14.1) is that there is no characteristic scale, and the system is said to be *scale invariant*. This terminology reflects the fact that power laws look the same on all scales. For example, the replacement $s \to bs$ in the function $N(s) = As^{-\alpha}$ yields a function $\widetilde{N}(s)$ that is indistinguishable from $N(s)$, except for a change in the amplitude A by the factor $b^{-\alpha}$.

Contrast the nature of the power law dependence of $N(s)$ in (14.1) to the result of combining a large number of independently acting random events. In this case we know that the distribution of the sum is a Gaussian (see Problem 7.15), and $N(s)$ has the form

$$N(s) \sim e^{-(s/s_0)^2} \qquad \text{(characteristic scale)}. \tag{14.2}$$

Scale invariance does not hold for functions that decay as in (14.2), because the replacement $s \to bs$ in the function $e^{-(s/s_0)^2}$ changes s_0 (the characteristic scale or size of s) by the factor b. Note that for a power law distribution, there are events of all sizes, but for a Gaussian distribution, there are, practically speaking, no events much larger than the characteristic scale s_0. For example, if we take $s_0 = 100$, there would be one large event of size 1000 for every 2.7×10^{43} events of size one!

A simple example of self-organized critical phenomena is an idealized sandpile. Suppose that we construct a sandpile by randomly adding one grain at a time onto a flat surface with open edges. Initially, the grains will remain where they land, but after we add more grains, there will be small avalanches during which the grains move so that the local slope of the pile is not too big. Eventually, the pile will reach a statistically stationary (time-independent) state, and the amount of sand added will balance the sand that falls off the edge (on the average). When a single grain of sand is added to such a configuration, a rearrangement might occur that triggers an avalanche of any size (up to the size of the system), so that the mean slope again equals the critical value. We say that the statistically stationary state is critical because there are avalanches of all sizes. The stationary state is *self-organized* because no external parameter (such as the temperature) needs to be tuned to force the system to this state. In contrast, the concentration of fissionable material in a nuclear chain reaction has to be carefully controlled for the nuclear chain reaction to become critical.

We consider a two-dimensional model of a sandpile and represent the height at site i by the array element `height[i]`. One grain of sand is added to a random site j, `height[j]++`, at each iteration. If `height[j]` = 4, then we remove the four grains from site j and distribute them equally to its nearest neighbors. A site whose height is equal to four is said to *topple*. If any of the neighbors now have four grains of sand, they topple as well. This process continues until all sites have less than four grains of sand. Grains that fall outside the lattice are lost forever.

Class `Sandpile` implements this idealized model. The lattice is stored in a `LatticeFrame` and the arrays `toppleSiteX` and `toppleSiteY` store the coordinates of the sites with four grains of sand. The array `distribution` accumulates the data for the number of sites that topple at each addition of a grain of sand to the pile. It is possible, though rare, that a site will topple more than once in one step. Hence, the number of toppled sites may be greater than the number of sites in the lattice.

Physically, it is not the actual height that determines toppling but the mean local slope between a site and its nearest neighbors. Thus, what we call the "height" really should be called the "slope." However, in the literature many authors use the term "height."

Listing 14.6: Implementation of the two-dimensional sandpile model.

```
package org.opensourcephysics.sip.ch14.sandpile;
```

```
import java.awt.Graphics;
import org.opensourcephysics.frames.*;

public class Sandpile {
   int[] distribution; // distribution of number of sites toppling
   int[] toppleSiteX, toppleSiteY;
   LatticeFrame height;
   int L, numberToppledMax;
   int numberToppled, numberOfSitesToTopple, numberOfGrains;

   public void initialize(LatticeFrame height) {
      this.height = height;
      height.resizeLattice(L, L); // create new lattice
      // size of distribution array
      numberToppledMax = 2*L*L+1;
      // could use histogramframe instead
      distribution = new int[numberToppledMax];
      toppleSiteX = new int[L*L];
      toppleSiteY = new int[L*L];
      numberOfGrains = 0;
      resetAverages();
   }

   public void step() {
      numberOfGrains++;
      numberToppled = 0;
      int x = (int) (Math.random()*L);
      int y = (int) (Math.random()*L);
      int h = height.getValue(x, y)+1;
      height.setValue(x, y, h); // add grain to random site
      height.render();
      if(h==4) { // topple grain
         numberOfSitesToTopple = 1;
         boolean unstable = true;
         int[] siteToTopple = {x, y};
         while(unstable) {
            unstable = toppleSite(siteToTopple);
         }
      }
      distribution[numberToppled]++;
   }

   public boolean toppleSite(int siteToTopple[]) { // topple site
      numberToppled++;
      int x = siteToTopple[0];
      int y = siteToTopple[1];
      numberOfSitesToTopple--;
       // remove grains from site
      height.setValue(x, y, height.getValue(x, y)-4);
      height.render();
      // add grains to neighbors
      // if (x,y) is on the border of the lattice, then
      // some grains will be lost.
      if(x+1<L) {
         addGrain(x+1, y);
      }
      if(x>0) {
         addGrain(x-1, y);
      }
      if(y+1<L) {
         addGrain(x, y+1);
      }
      if(y>0) {
         addGrain(x, y-1);
      }
      if(numberOfSitesToTopple>0) {
         // next site to topple
         siteToTopple[0] = toppleSiteX[numberOfSitesToTopple-1];
         siteToTopple[1] = toppleSiteY[numberOfSitesToTopple-1];
         return true;
      } else {
         return false;
      }
   }

   public void addGrain(int x, int y) {
      int h = height.getValue(x, y)+1;
      height.setValue(x, y, h); // add grain to site
      height.render();
      if(h==4) { // new site to topple
         toppleSiteX[numberOfSitesToTopple] = x;
         toppleSiteY[numberOfSitesToTopple] = y;
         numberOfSitesToTopple++;
      }
   }

   public void resetAverages() {
      distribution = new int[numberToppledMax];
      numberOfGrains = 0;
   }
}
```

Listing 14.7: The target class for the two-dimensional sandpile model.

```
package org.opensourcephysics.sip.ch14.sandpile;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.frames.*;

public class SandpileApp extends AbstractSimulation {
   Sandpile sandpile = new Sandpile();;
   LatticeFrame height = new LatticeFrame("x", "y", "Sandpile");
   PlotFrame plotFrame = new PlotFrame("ln s", "ln N",
        "Distribution of toppled sites");

   public SandpileApp() {
      height.setIndexedColor(0, java.awt.Color.WHITE);
      height.setIndexedColor(1, java.awt.Color.BLUE);
      height.setIndexedColor(2, java.awt.Color.GREEN);
      height.setIndexedColor(3, java.awt.Color.RED);
      height.setIndexedColor(4, java.awt.Color.BLACK);
   }

   public void initialize() {
      sandpile.L = control.getInt("L");
      height.setPreferredMinMax(0, sandpile.L, 0, sandpile.L);
      sandpile.initialize(height);
   }

   public void doStep() {
      sandpile.step();
   }

   public void stop() {
      plotFrame.clearData();
      for(int s = 1;s<sandpile.distribution.length;s++) {
         double f = (double) sandpile.distribution[s];
         double N = (double) sandpile.numberOfGrains;
         if (f > 0) {
             plotFrame.append(0, Math.log(s), Math.log(f/N));;
         }
      }
      plotFrame.render();
   }

   public void reset() {
      control.setValue("L", 10);
      enableStepsPerDisplay(true);
   }

   public void resetAverages() {
      sandpile.resetAverages();
   }

   public static void main(String[] args) {
      SimulationControl control = SimulationControl.createApp(new SandpileApp());
      control.addButton("resetAverages", "resetAverages");
   }
}
```

Problem 14.7. A two-dimensional sandpile model

(a) Use the classes Sandpile and SandpileApp to simulate a two-dimensional sandpile with linear dimension L. Run the simulation with $L = 10$ and stop it once toppling starts to occur. When this behavior occurs, black cells (with four grains) will momentarily appear. Use the Step button to watch individual toppling events and obtain a qualitative sense of the dynamics of the sandpile model.

(b) Comment out the `height.render()` statements in Sandpile and add a statement to SandPileApp so that the number of grains added to the system is displayed. (The number of grains added is a measure of the number of configurations that are included in the various averages.) Now you will not be able to see individual toppling events, but you can more quickly collect data on the toppling distribution, the frequency of the number of sites that topple when a grain is added. The program outputs a log-log plot of the distribution. Estimate the slope of the log-log distribution from the part of the plot that is linear and thus determine the power law exponent α. Reset the averages and repeat your calculation to obtain another estimate of α. If your two estimates of α are within a few percent, you have added enough grains of sand. Compute α for $L = 10$, 20, 40, and 80. As you make the lattice size larger, the range over which the log-log plot is linear should increase. Explain why the plot is not linear for large values of the number of toppled sites. □

Of course, the model of a sandpile in Problem 14.7 is over simplified. Laboratory experiments indicate that real sandpiles show power law behavior if the piles are small, but that larger sandpiles do not (see Jaeger et al.).

Earthquakes. The empirical Gutenberg–Richter law for $N(E)$, the number of earthquakes with energy release E, is consistent with power law behavior:

$$N(E) \sim E^{-b}, \tag{14.3}$$

with $b \approx 1$. The magnitude of earthquakes on the Richter scale is approximately the logarithm of the energy release. This power law behavior does not necessarily hold for individual fault systems, but holds reasonably accurately when all fault systems are considered. One implication of the power law dependence in (14.3) is that there is nothing special about large earthquakes. In Problems 14.8 and 14.9 and Project 14.26 we explore some earthquake models.

Given the long time scales between earthquakes, there is considerable interest in simulating models of earthquakes. The Burridge–Knopoff model considered in Project 14.26 consists of a system of coupled masses in contact with a rough surface. The masses are subjected to static and dynamic friction forces due to the surface, and are also pulled by an external force corresponding to slow tectonic plate motion. The major difficulty with this model is that the numerical solution of the corresponding equations of motion is computationally intensive. For this reason we consider several cellular automaton models that retain some of the basic physics of the Burridge–Knopoff model.

Problem 14.8. A simple earthquake model

Define the real variable $F(i,j)$ on a square lattice, where F represents the force or stress on the block at position (i,j). The initial state of the lattice at time $t = 0$ is found by assigning small random values to $F(i,j)$. The lattice is updated according to the following rules:

(i) Increase F at every site by a small amount ΔF, for example, $\Delta F = 10^{-3}$, and increase the time t by 1. This increase represents the effect of the driving force due to the slow motion of the tectonic plate.

(ii) Check if $F(i,j)$ is greater than F_c, the threshold value of the force. If not, the system is stable and step 1 is repeated. If the system is unstable, go to step 3. Choose $F_c = 4$ for convenience.

(iii) The release of stress due to the slippage of a block is represented by letting $F(i,j) = F(i,j) - F_c$. The transfer of stress is represented by updating the stress at the sites of the four neighbors at $(i, j \pm 1)$ and $(i \pm 1, j)$: $F \to F + 1$. Periodic boundary conditions are not used.

These rules are equivalent to the Bak–Tang–Wiesenfeld model. What is the relation of this model to the sandpile model considered in Problem 14.7?

As an example, choose $L = 10$. Do the simulation and show that the system eventually comes to a statistically stationary state, where the average value of the stress at each site stops growing. Monitor $N(s)$, the number of earthquakes of size s, where s is the total number of sites (blocks) that are affected by the instability. Then consider $L = 30$ and repeat your simulations. Are your results for $N(s)$ consistent with scaling? □

Problem 14.9. A dissipative earthquake model

The Bak–Tang–Wiesenfeld earthquake model discussed in Problem 14.8 displays power law scaling due to the inherent conservation of the dynamical variable, the stress. It is easy to modify the model so that the stress is not conserved and the model is more realistic. The Rundle–Jackson–Brown/Olami–Feder–Christensen model of an earthquake fault is a simple example of such a nonconservative system.

(a) Modify the toppling rule in Problem 14.8 so that when the stress on site (i, j) exceeds F_c, not all the excess stress is given to the neighbors. In particular, assume that when site (i, j) topples, $F(i, j)$ is reduced to the residual stress $F_r(i, j)$. The amount $\alpha(F_{ij} - F_r)$ is dissipated leaving $(F_{ij} - F_r)(1 - \alpha)$ to be distributed equally to the neighbors. If $\alpha = 0$, the model is equivalent to the model considered in Problem 14.8. Choose $\alpha = 0.2$ and determine if $N(s)$ exhibits power law scaling. For simplicity, choose $F_c = 4$ and $F_r = 1$ (see Grassberger).

(b) Make the model more realistic by adding a small amount of noise to F_r so that F_r is uniformly distributed between $1 - \delta, 1 + \delta$ with $\delta = 0.05$. Also run the model in what is called the "zero-velocity limit" by finding the site with the maximum stress $F_{\max}$ and then increasing the stress on all sites by $F_c - F_{\max}$ so that only one site initially becomes unstable. Determine $N(s)$ and see if your results differ from what you found in part (a). Do you still observe power law scaling?

(c) The model can be made more realistic still by assuming that the interaction between the blocks is long range due to the existence of elastic forces. Distribute the excess stress equally to all z neighbors that are within a distance of radius R of an unstable site. Each of the z neighbors receives a stress equal to $(F_{ij} - F_r)(1 - \alpha)/z$. First choose $R = 3$ and see if the qualitative behavior of $N(s)$ changes as R becomes larger. Lattices with $L \geq 256$ are typically considered with $R \simeq 30$ (see the papers by W. Klein and J. B. Rundle and collaborators.). □

The behavior of other simple models of natural phenomena is explored in the following.

Problem 14.10. Forest fire model

(a) Consider the following model of the spread of a forest fire. Suppose that at $t = 0$ the $L \times L$ sites of a square lattice either have a tree or are empty with probability p and $1 - p$, respectively. The sites that have a tree are on fire with probability f. At each iteration an empty site grows a tree with probability g, a tree that has a nearest neighbor site on fire catches fire, and a site that is already on fire dies and becomes empty. This model is an example of a probabilistic cellular automaton. Write a program to simulate this model and color code the three types of sites. Use periodic boundary conditions.

(b) Choose $L \geq 30$ and determine the values of g for which the forest maintains fires indefinitely. Note that as long as $g > 0$, new trees will always grow.

(c) Use the value of g that you found in part (b) and compute the distribution of the number of sites s_f on fire. If the distribution is critical, determine the exponent α that characterizes this distribution. Also compute the distribution for the number of trees s_t. Is there any relation between these two distributions?

(d)* To obtain reliable results it is frequently necessary to average over many initial configurations. However, the behavior of many systems is independent of the initial configuration and averaging over many initial configurations is unnecessary. This latter possibility is called *self-averaging*. Repeat parts (b) and (c), but average your results over ten initial configurations. Is this forest fire model self-averaging? □

Problem 14.11. Another forest fire model

Consider a simple variation of the model discussed in Problem 14.10. At $t = 0$ each site is occupied by a tree with probability p; otherwise, it is empty. The system is updated in successive iterations as follows:

(i) Randomly grow new trees at time t with a small probability g from sites that are empty at time $t-1$;

(ii) A tree that is not on fire at $t-1$ catches fire due to lightning with probability f.

(iii) Trees on fire ignite neighboring trees, which in turn ignite their neighboring trees, etc. The spreading of the fire occurs instantaneously.

(iv) Trees on fire at time $t-1$ die (become empty sites) and are removed at time t (after they have set their neighbors on fire).

As in Problem 14.10, the changes in each site occur synchronously.

(a) Determine $N(s)$, the number of clusters of trees of size s that catch fire in each iteration. Two trees are in the same cluster if they are nearest neighbors. Is the behavior of $N(s)$ consistent with $N(s) \sim s^{-\alpha}$? If so, estimate the exponent α for several values of g and f.

(b)* The balance between the mean rate of birth and burning of trees in the steady state suggests a value for the ratio f/g at which this model is likely to be scale invariant. If the average steady state density of trees is ρ, then at each iteration the mean number of new trees appearing is $gN(1-\rho)$, where $N = L^2$ is the total number of sites. In the same spirit, we can say that for small f, the mean number of trees destroyed by lightning is $f\rho N\langle s\rangle$, where $\langle s\rangle$ is the mean number of trees in a cluster. Is this reasoning consistent with the results of your simulation? If we equate these two rates, we find that $\langle s\rangle \sim [(1-\rho)/\rho](g/f)$. Because $0 < \rho < 1$, it follows that $\langle s\rangle \to \infty$ in the limit $f/g \to 0$. Given the relation $\langle s\rangle = \sum_{s=1}^{\infty} sN(s)/\sum_s N(s)$ and the divergent behavior of $\langle s\rangle$, why does it follow that $N(s)$ must decay more slowly than exponentially with s? This reasoning suggests that $N(s) \sim s^{-\alpha}$ with $\alpha < 2$. Is this expectation consistent with the results that you obtained in part (a)?

In this model there are three well-separated time scales; that is, the time for lightning to strike ($\propto f^{-1}$), the time for trees to grow ($\propto g^{-1}$), and the instantaneous spreading of fire through a connected cluster. This separation of time scales seems to be an essential ingredient for self-organized criticality (see Grinstein and Jayaprakash). □

Problem 14.12. Model of punctuated equilibrium

(a) The idea of *punctuated equilibrium* is that biological evolution occurs episodically rather than as a steady, gradual process. That is, most of the major changes in life forms occur in relatively short periods of time. Bak and Sneppen have proposed a simple model that exhibits some of the behavior of punctuated equilibrium. The model consists of a one-dimensional cellular automaton of linear dimension L, where cell i represents the biological fitness of species i. Initially, all cells receive a random fitness f_i between 0 and 1. Then the cell with the lowest fitness and its two nearest neighbors are randomly given new fitness values. This update rule is repeated indefinitely. Write a program to simulate the behavior of this model. Use periodic boundary conditions and display the fitness of each cell as a column of height f_i. Begin with $L = 64$ and describe what happens to the distribution of fitness values after a long time.

(b) We can crudely think of the update process as replacing a species and its neighbors by three new species. In this sense the fitness represents a barrier to creating a new species. If the barrier is low, it is easier to create a new species. Do the low fitness species die out? What is the average value of fitness of the species after the model is run for a long time (10^4 or more iterations)? Compute the distribution of fitness values $N(f)$ averaged over all cells and over many iterations. Allow the system to come to a fluctuating steady state before computing $N(f)$. Plot $N(f)$ versus f. Is there a critical value f_c below which $N(f)$ is much less than the values above f_c? Is the update rule reasonable from a evolutionary point of view?

(c) Modify your program to compute the distance x between successive fitness changes and the distribution of these distances $P(x)$. Make a log-log plot of $P(x)$ versus x. Is there any evidence of self-organized criticality (power law scaling)?

(d) Another way to visualize the results is to make a plot of the time at which a cell is changed versus the position of the cell. Is the distribution of the plotted points approximately uniform? We might expect that the survival time of a species depends exponentially on its fitness, and hence each update corresponds to an elapsed time of e^{-cf_i}, where the constant c sets the time scale, and f_i is the fitness of the cell that has been changed. Choose $c = 100$ and make a similar plot with the time axis replaced by the logarithm of the time; that is, the quantity $100f_i$. Is this plot more meaningful?

(e) Another way of visualizing punctuated equilibrium is to plot the number of times groups of cells change as a function of time. Divide the time into units of 100 updates and compute the number of fitness changes for cells $i = 1$ to 10 as a function of time. Do you see any evidence of punctuated equilibrium? □

14.3 The Hopfield Model and Neural Networks

Neural network models have been motivated in part by how neurons in the brain collectively store and recall memories. Usually, a neuron is in one of two states, a resting potential (not firing) or firing at the maximum rate. A neuron "fires" once it receives electrical inputs from other neurons whose strength reaches a certain threshold. An important characteristic of a neuron is that its output is a nonlinear function of the sum of its inputs. The assumption is that when memories are stored in the brain, the strengths of the connections between neurons change.

One of the uses of neural network models is pattern recognition. If we see someone more than once, the person's face provides input that helps us to recall the person's name. In the same spirit, a neural network can be given a pattern, for example, a string of ±1s, that partially reflect a previously memorized pattern. The idea is to store memories so that a computer can recall them when the inputs are close to a particular memory.

We now consider an example of a neural network due to Hopfield. The network consists of N neurons and the state of the network is defined by the state of each neuron S_i, which in the Hopfield model takes on the values -1 (not firing) and $+1$ (firing). The strength of the connection between neuron i and neuron j is denoted by w_{ij}, which is determined by the M stored memories:

$$w_{ij} = \sum_{\alpha=1}^{M} S_i^\alpha S_j^\alpha \tag{14.4}$$

where S_i^α represents the state of neuron i in stored memory α. Given the initial state of all the neurons, the dynamics of the network is simple. We choose a neuron i at random and change its state according to its input, which is $\sum_{i\neq j} w_{ij}S_j$, where S_j represents the current state of neuron j. Then we change the state of neuron i by setting

$$S_i = \begin{cases} +1 & \text{for } \sum_{i\neq j} w_{ij}S_j > 0 \\ -1 & \text{for } \sum_{i\neq j} w_{ij}S_j \leq 0. \end{cases} \tag{14.5}$$

The threshold value of the input has been set equal to zero, but other values could be used as well.

The HopfieldApp class in Listing 14.8 implements this model of a neural network and stores memories based on user input. The state of the network is stored in the array S[i] and the connections between the neurons are stored in the array w[i][j]. The user initially clicks on various cells to toggle their values between -1 and $+1$ and presses the Remember button to store a pattern. Then the user presses the Randomize button to initialize the S_i by setting S_i to ±1 at random. After the memories are stored, press the Start button to update the neurons using the Hopfield algorithm to try to recall one of the stored memories.

Listing 14.8: HopfieldApp class.

```java
package org.opensourcephysics.sip.ch14;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.frames.*;

// Hopfield model of a neural network
public class HopfieldApp extends AbstractSimulation {
   LatticeFrame lattice;
   int N;          // total number of neurons
   double[][] w; // connection array (N by N elements)
   int numberOfStoredMemories;

   public HopfieldApp() {
      lattice = new LatticeFrame("Hopfield state");
      lattice.setToggleOnClick(true, -1, 1);
      lattice.setIndexedColor(-1, java.awt.Color.blue);
      lattice.setIndexedColor(0, java.awt.Color.blue);
      lattice.setIndexedColor(1, java.awt.Color.green);
      lattice.setSize(600, 120);
   }

   public void doStep() {
      int[] S = lattice.getAll();
      for(int counter = 0;counter<N;counter++) {
         // chooses random neuron index
         int i = (int) (N*Math.random());
         double sum = 0;
         for(int j = 0;j<N;j++) {
            sum += w[i][j]*S[j];
         }
         S[i] = (sum>0) ? 1 : -1;
      }
      lattice.setAll(S);
   }

   public void initialize() {
      N = control.getInt("Lattice size");
      w = new double[N][N];
      lattice.resizeLattice(N, 1);
      for(int i = 0;i<N;i++) {
         lattice.setAtIndex(i, -1);
      }
      lattice.setMessage("# memories = "+(numberOfStoredMemories = 0));
   }

   public void reset() {
      control.setValue("Lattice size", 8);
      lattice.setMessage("# memories = "+(numberOfStoredMemories = 0));
   }

   public void addMemory() {
      int[] S = lattice.getAll();
      for(int i = 0;i<N;i++) {
         for(int j = i+1;j<N;j++) {
            w[i][j] += S[i]*S[j];
            w[j][i] += S[i]*S[j];
         }
      }
      lattice.setMessage("# memories = "+(++numberOfStoredMemories));
   }

   public void randomizeState() {
      for(int i = 0;i<N;i++) {
         lattice.setAtIndex(i, Math.random()<0.5 ? -1 : 1);
      }
      lattice.repaint();
   }

   public static void main(String args[]) {
      SimulationControl control =
           SimulationControl.createApp(new HopfieldApp());
      control.addButton("addMemory", "Remember");
```

```
        control.addButton("randomizeState", "Randomize");
    }
}
```

Problem 14.13. Memory recall in the Hopfield model

(a) Use the `HopfieldApp` class to explore the ability of the Hopfield neural network to store and recall memories. Begin with $N = 10$ neurons and click on the cells to choose a pattern to remember. Then click on the Randomize button to randomize the spins. Does the neural network find a pattern similar to the one you saved? Consider other values of N and various patterns to obtain a feel for how the algorithm works.

(b) Store two memories of 20 bits, for example,

$$11111\bar{1}\bar{1}\bar{1}1\bar{1}\bar{1}\bar{1}\bar{1}\bar{1}\bar{1}\bar{1}11111 \text{ and } 11\bar{1}\bar{1}11\bar{1}\bar{1}11\bar{1}\bar{1}11\bar{1}\bar{1}11\bar{1}\bar{1}$$

where we have written -1 as $\bar{1}$. Try to recall a memory using the input

$$1111111\bar{1}\bar{1}\bar{1}\bar{1}\bar{1}\bar{1}1111111.$$

This input is similar to the first memory. Record the Hamming distance between the final state and the closest memory, where the Hamming distance is the number of bits that differ between two strings. Repeat this procedure for several different values of the number of neurons and the memory length.

(c) Estimate how many memories can be stored for a given number of neurons before recall becomes severely reduced. Make estimates for $N = 10$, 20, and 40. What is your criteria for the recall to be considered correct?

(d) In the Hopfield model every neuron is linked to every other neuron. (The value of the links w_{ij} is determined by the stored memories.) Is the spatial dimension of the system relevant? Describe how the `HopfieldApp` class can store two-dimensional patterns. □

Neural networks can also be used for difficult optimization problems. In Problem 14.14 we consider the problem of finding the minimum energy of a model *spin glass*. The latter is a magnetic analog of an ordinary glass in which the positions of the molecules are not ordered as in a crystal. In a spin glass the local magnetic moment is disordered because random magnetic interactions are "frozen in" and do not change. The simplest model of a spin glass is based on the simplest model of magnetism, the Ising model (see Section 15.5). In the Ising model the magnetic moment is represented by a spin s_i which can take on two values, ± 1. The spins are located on the sites of a lattice. Each spin is assumed to interact with all other spins, and the total energy of the system is given by

$$E = -\sum_{i,j \neq i} J_{ij} V_i V_j \tag{14.6}$$

where the sum is over all pairs of spins. We have let $w \to J$ so that the notation is the same as the Ising model. If $J_{ij} > 0$, the spins i and j lower their energy by lining up in the same direction. If $J_{ij} < 0$, the spins lower their energy by lining up in opposite directions (see Figure 15.1).

We are interested in finding the ground state when the coupling constant J_{ij} randomly takes on the values $\pm J_0/N$, where N is the number of spins and J_0 is an arbitrary constant. To find the ground state, we need to find the configurations of spins that give the lowest value of the energy. Finding the ground state of a spin glass is particularly difficult because there are many configurations that correspond to local minima of the energy. In fact the problem of finding the exact ground state is an example of a computationally difficult problem called *NP-complete*. (Another example of such a problem is considered in Problem 15.31.) In Problem 14.14 we explore if the Hopfield algorithm can find a good approximation to the global minimum.

Problem 14.14. Minimum energy of an Ising spin glass

(a) Choose $J_0 = 4$ in (14.6) and modify the `HopfieldApp` class so that it applies to a model spin glass. Display the output string and the energy after every N attempts to change a spin. Begin with $N = 20$.

(b) What happens to the energy after a long time? For different initial states, but the same set of the J_{ij}, is the value of the energy the same after the system has evolved for a long time? Explain your results in terms of the number of local energy minima.

(c) What is the behavior of the system? Do you find periodic behavior, random behavior, or does the system evolve to a state that does not change? □

14.4 Growing Networks

A network is a collection of points called nodes that are connected by lines called links. Mathematicians refer to networks as graphs, and graph theory has been an active field of mathematics for many years. A mathematical network can represent an actual network by defining what a node represents and the kind of relationship represented by a link. For example, in an airline network the nodes represent airports and the links represent flights between airports. In an acquaintance network, the nodes represent individuals, and the links represent the state of two people knowing each other. In a biochemical network, the nodes represent various molecular types, and the links represent a reaction between molecules.

One reason for the recent interest in networks is that data on existing networks is now more readily available due to the widespread use of computers. Indeed, one of the networks of current interest is the network of websites. Another reason for the interest in networks is that some new models of networks have been developed.

We first discuss one of the original network models, the Erdös–Rényi model. In this model we start with N nodes and then form n links between pairs of nodes such that each pair has either one link or no links. The probability of a link between any pair of nodes is $p = n/(N(N-1)/2)$. One quantity of interest is the degree distribution $D(\ell)$, which is the fraction of nodes that have ℓ links. An example of the determination of $D(\ell)$ is shown in Figure 14.3. In the Erdös-Rényi model this distribution is a Poisson distribution for large N. Thus, there is a peak in $D(\ell)$, and for large ℓ, $D(\ell)$ decreases exponentially.

In some network models there is a path between any pair of nodes. In other models, such as the Erdös–Rényi model, there are some nodes that cannot be reached from other nodes (see Figure 14.3). In these networks there are other quantities of interest that are analogous to those in percolation theory. The main difference is that in network models the position of the nodes is irrelevant, and only their connectivity is relevant. In particular, there is no spanning cluster as can exist in percolation models. Instead, there can be a cluster that is significantly larger than the other clusters. In the Erdös–Rényi model, the transition at which such a "giant" cluster appears depends on the probability p that any pair of nodes is connected. In the large N limit this transition occurs at $p = 1/N$.

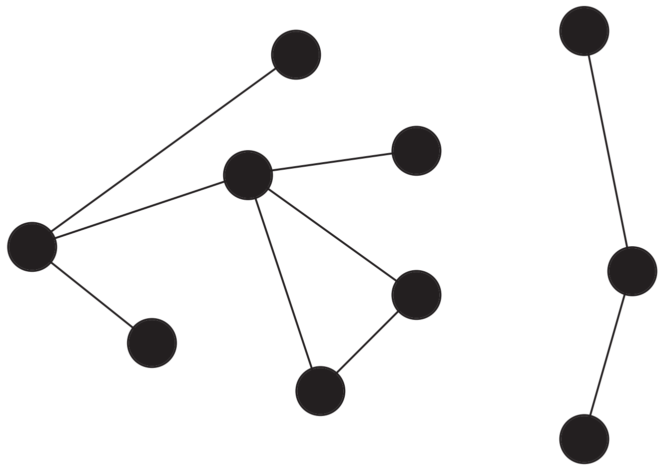


Figure 14.3: Example of a disconnected network with 10 nodes and 9 links. The degree distribution for this network is $D(1) = 5/10 = 0.5$, $D(2) = 3/10 = 0.3$, $D(3) = 1/10 = 0.1$, and $D(4) = 1/10 = 0.1$. The cluster coefficient or transitivity is defined as 3 times the number of triangles divided by the number of possible triples of connected nodes. In this case we have 1 triangle and 12 triples. Thus, the clustering coefficient equals $3 \times 1/12 = 0.25$. If a node has ℓ links, then the number of triples centered at that node is $\ell/(2!(\ell - 2)!)$.

Problem 14.15. The Erdös–Rényi model

(a) Write a program to create networks based on the Erdös–Rényi model. Choose $N = 100$ and $p \approx 0.01$ and compute $D(\ell)$; average over at least 10 networks. Show that $D(\ell)$ follows a Poisson distribution.

(b) Define a giant cluster as one that has over three times as many nodes as any other cluster and at least 10% of the nodes. Find the value of p at which the giant cluster first appears for $N = 64$, 128, and 256. Average over 10 networks for each value of N. The cluster distribution should be updated after every link is added using the labeling procedure used in Chapter 12. In this case it is easier because every time we add a link, we either combine two clusters or we make no change in the cluster distribution. □

Some of the networks that we will consider are by definition connected. In these cases one of the important quantities of interest is the mean path length between two nodes, where the path length between two nodes is the shortest number of links from one node to the other. If the mean path length weakly depends on the total number of nodes and is small, then this property of networks is known as the "small world" property. A well-known example of the small world property is what is called "six degrees of separation," which refers to the fact that almost any person is connected through a sequence of six connections to almost any other person.

We wish to understand the structure of different networks. One structural property is the clustering coefficient or transitivity. If node A is linked to B and B to C, the clustering coefficient is the probability that A is linked to C (see Figure 14.3 for a precise definition). If this coefficient is large, then there will be many small loops of nodes in the network. If we think of the nodes as people and the links as friendship connections, then the clustering coefficient is a measure of the tendency of people to form cliques. It is also of interest to see to what extent the network is hierarchically organized. Can we find groups of nodes that are linked together at different levels of organization? Can we produce an organizational chart for the network similar to what is used by many businesses? Algorithms for computing the hierarchical or community structure of a network are discussed in the references.

Two popular network models are the Watts–Strogatz small world model and the Barabasi–Albert preferential attachment model. In the Watts–Strogatz model, a regular lattice of nodes connected by nearest neighbor links is "rewired" so that a link between two neighboring nodes is broken with probability p, and a link is randomly added between one of the nodes and any other node in the system. The small world property shows up as a logarithmic dependence of the mean path length on the system size N for large p. The degree distribution is similar to that of the Erdös-Rényi model.

In the preferential attachment model, we begin with a few connected nodes and then add one node at a time. Each new node is then linked to m existing nodes, with preference given to those nodes that already have many links. The probability of a node with ℓ links being connected to a new node is proportional to ℓ. For example, if we have ten nodes in the network with 1, 1, 3, 2, 7, 3, 4, 7, 10, and 2 links, respectively, then there are a total of 40 links and the probability of getting the next link from a new node is 1/40, 1/40, 3/40, 2/40, 4/40, 3/40, 4/40, 7/40, 10/40, and 2/40, respectively. The result of this growth rule is that some nodes will accumulate many links. The key result is that the link distribution is a power law with $D(\ell) \sim \ell^{-\alpha}$. This *scale-free* behavior is very important because it says thst in the limit of an infinite network, there is a non-negligible probability that a node exists with any particular number of links. Examples of real networks that have this behavior are actor networks where the links correspond to two actors appearing in the same movie, airport networks, the internet, and the links between various websites. In addition to the scale-free degree distribution, the preferential attachment model also has the small world property that the mean path length grows only logarithmically with the number of nodes.

The `PreferentialAttachment` class implements the preferential attachment model. Method `setPosition` is not relevant to the actual growth model. It places the nodes in random positions so that the network can be drawn so they are too close to each other. This drawing method is useful only for networks with less than about 100 nodes.

Listing 14.9: PreferentialAttachment class: Preferential attachment network model.

```
package org.opensourcephysics.sip.ch14.networks;
import java.awt.Color;
import java.awt.Graphics;
import org.opensourcephysics.frames.*;
import org.opensourcephysics.display.Drawable;
import org.opensourcephysics.display.DrawingPanel;

public class PreferentialAttachment implements Drawable {
   int[] node, linkFrom, degree;
   // positions of nodes, only meaningful for display purposes
   double[] x, y;
   int N;                          // maximum number of nodes
   int m = 2;                      // number of attempted links per node
   int linkNumber = 0;             // twice current number of links
   int n = 0;                      // current number of nodes
   boolean drawPositions = true;   // only draw network if true
   int numberOfCompletedNetworks = 0;

   public void initialize() {
      // degree distribution to be averaged over many networks
      degree = new int[N];
      numberOfCompletedNetworks = 0; // will draw many networks
      startNetwork();
   }

   public void addLink(int i, int j, int s) {
      linkFrom[i*m+s] = j;
      node[i]++;
      node[j]++;
      linkNumber += 2; // twice current number of links
   }

   public void startNetwork() {
      n = 0;
      linkFrom = new int[m*N];
      node = new int[N];
      x = new double[N];
      y = new double[N];
      linkNumber = 0;
      for(int i = 0;i<=m;i++) {
         n++;
         setPosition(i);
      }
      for(int i = 1;i<m+1;i++) {
         for(int j = 0;j<i;j++) {
            addLink(i, j, j);
         }
      }
   }

   public void setPosition(int i) {
      double r2min = 1000./N;
      // used to insure two nodes are not drawn too close to each other
      boolean ok = true;
      do {
         ok = true;
         x[i] = Math.random()*100;
         y[i] = Math.random()*100;
         int j = 0;
         while(j<i&&ok) {
            double dx = x[i]-x[j];
            double dy = y[i]-y[j];
            double r2 = dx*dx+dy*dy;
            if(r2<r2min) {
               ok = false;
            }
            j++;
         }
      } while(!ok);
   }

   public int findNode(int i, int s) {
      boolean ok = true;
      int j = 0;
      do {
         ok = true;
         int k = (int) (1+Math.random()*linkNumber);
         j = -1;
         int sum = 0;
         do {
            j++;
            sum += node[j];
         } while(k>sum);
         for(int r = 0;r<s;r++) {
            if(linkFrom[i*m+r]==j) {
               ok = false;
            }
         }
      } while(!ok);
      return j;
   }

   public void addNode(int i) {
      n++;
      if(drawPositions) {
         setPosition(i);
      }
      for(int s = 0;s<m;s++) {
         addLink(i, findNode(i, s), s);
      }
   }

   public void step() {
      if(n<N) {
         addNode(n);
      } else {
         numberOfCompletedNetworks++;
         // accumulate data for degree distribution
         for(int i = 0;i<n;i++) {
            degree[node[i]]++;
         }
         startNetwork();          // start another network
      }
   }

   public void degreeDistribution(PlotFrame plot) {
      plot.clearData();
      for(int i = 1;i<N;i++) {
         if(degree[i]>0) {
            plot.append(0, Math.log(i), Math.log(degree[i]*1.0/
                (N*numberOfCompletedNetworks)));
         }
      }
   }
   public void draw(DrawingPanel panel, Graphics g) {
      if(node!=null&&drawPositions) {
         int pxRadius = Math.abs(panel.xToPix(1.0)-panel.xToPix(0));
         int pyRadius = Math.abs(panel.yToPix(1.0)-panel.yToPix(0));
         g.setColor(Color.green);
         for(int i = 0;i<n;i++) {
            int xpix = panel.xToPix(x[i]);
            int ypix = panel.yToPix(y[i]);
            for(int s = 0;s<m;s++) {
               int j = linkFrom[i*m+s];
               int xpixj = panel.xToPix(x[j]);
               int ypixj = panel.yToPix(y[j]);
               g.drawLine(xpix, ypix, xpixj, ypixj);  // draw link
            }
         }
         g.setColor(Color.red);
         for(int i = 0;i<n;i++) {
            int xpix = panel.xToPix(x[i])-pxRadius;
            int ypix = panel.yToPix(y[i])-pyRadius;
             // draw node
            g.fillOval(xpix, ypix, 2*pxRadius, 2*pyRadius);
         }
      }
   }
}
```

Problem 14.16. Preferential attachment model

(a) Write a target class that uses the `PreferentialAttachment` class and continuously creates new networks until stopped by the user (so we can compute averages over many networks). To speed up the computation, make it possible to optionally display the networks. The program should output the average degree distribution $D(\ell)$.

(b) Estimate the exponent α defined by $D(\ell) \sim \ell^{-\alpha}$ for $N = 100$ and $m = 2$. Repeat for $N = 500$. Does the exponent α change? If time permits, consider $N = 10,000$. Does α depend on m?

(c) Modify `PreferentialAttachmentModel` so that the ℓ links are made randomly so that the number of links a node already has is irrelevant to adding a link. What functional form does the link distribution have now? Is this model equivalent to the Erdös-Rényi model?

(d)* Write a method to compute the clustering coefficient, which is defined in Figure 14.3. Plot $\ln C(N)$ versus $\ln N$ for both the preference attachment model and the Erdös-Rényi model. Compare and discuss your results in terms of the visual appearance of the networks. □

Problem 14.17. Watts–Strogatz network

(a) Write a class to create a Watts–Strogatz network. Begin with $N = 100$ nodes which you can visualize as equally spaced on a circle. (Their actual position is irrelevant.) Place links between the $2m$ nearest neighbors. Thus, if $m = 1$, then only the nearest neighbors are linked. If $m = 2$, then the nearest and next nearest neighbors are linked. Next write a method to go through each link and then with probability p, break the link connection at one end and reconnect it to another node at random.

(b) Compute the degree distribution as a function of m for several values of p. Discuss your results.

(c) As we increase p, the networks becomes more and more random. There is a transition from a network where the path length $\ell \sim N$ to one where $\ell \sim \ln N$. This transition occurs when $Np^{1/d} \sim 1$, where d is the dimension of the original lattice before rewiring ($d = 1$ for a circle). Draw a number of networks with different values of N and p and use this visualization to explain the dependence of ℓ on N.

(d)* Write a method to compute the clustering coefficient C. Plot C versus $\ln p$ for $N = 100$ and $m = 2$. Repeat for larger N. □

Problem 14.18. A model of a social network

In many social situations we notice groups of people who interact closely with each other, but not necessarily with other groups. Usually, those in a group have some common interest or personal attribute. How can we model this situation? People do not usually become friends with other people just because they have many friends already (the preferential attachment mechanism). Instead, they choose someone to interact with and a friendship is established with some probability. A simple model is given by the following rule. As each node is added to a system, choose m existing nodes at random, and with probability p establish a link. This process will create a number of clusters of linked nodes. We can imagine that there is a possibility of a phase transition between the existence of a giant cluster that contains a large fraction of the nodes and a situation where all the clusters are small. This model was analyzed by Zalányi et al.

(a) Write a class to model this random attachment model and compute the degree distribution as well as the cluster distribution. Consider at least $N = 1000$ nodes and measure $D(\ell)$, the degree distribution, for $m = 2$, 3, and 5 and $p = 0.1$ and $p = 0.9$. Average over at least ten trials. You should not find power law behavior for $D(\ell)$. Explain why this behavior is expected.

(b) Compute $D(\ell, t)$, the number of links connected to a node as a function of t, the time when the node is added. We would expect nodes added in the beginning to have more links than those at the end. Describe and discuss the functional form of $D(\ell, t)$.

(c) Consider $m = 5$ and generate many networks for different values of p. Determine the cluster distribution. A giant cluster exists when the largest cluster is at least three times larger than the next largest cluster. Estimate the value of p for which the giant cluster first appears. You should find an approximate power law cluster distribution only at the transition. What is the exponent of the power law?

(d) How does the value of p at the transition change with m? Explain your results.

(e) Consider $m = 1$ and generate networks for many values of p. Determine the cluster distribution. You should find an approximate power law distribution for all values of p. What are the exponents for the power law? Why do you think there is not a phase transition for $m = 1$? Consider the possibility of two clusters merging for different values of m. □

14.5 Genetic Algorithms

Many people find it difficult to accept that evolution is sufficiently powerful to generate the biological complexity observed in nature. Part of this difficulty arises from the inability of humans to intuitively grasp time scales that are much greater than their own lifetimes. Another reason is that it is very difficult to appreciate how random changes can lead to emergent complex structures. Genetic algorithms provide one way of understanding the nature of evolution. Their principal utility at present is in optimization problems, but they are also being used to model biological and social evolution.

Historically, developments in physics, such as x-ray crystallography and quantum mechanics, have lead to developments in biology. In recent years developments in biology as well as in computer science and other areas have had a direct impact on developments in physics. Genetic algorithms are an example of the influence of ideas in biology impacting ideas in physics.

The idea of genetic algorithms is to model the process of evolution by natural selection. This process involves two steps: random changes in the genetic code during reproduction and selection according to some fitness criteria. In biological organisms the genetic code is stored in the DNA. We will store the genetic code as a string of 1s and 0s. The genetic code constitutes the *genotype*. The conversion of this string to the organism or *phenotype* depends on the problem. The selection criteria is applied to the phenotype.

First we describe how change is introduced into the genotype. Typically, nature changes the genetic code in two ways. The most obvious, but less often used method, is mutation. Mutation corresponds to changing a character at random in the genetic code string from 0 to 1 or from 1 to 0. The other much more powerful method is associated with sexual reproduction. We take two strings, remove a piece from one string, and exchange it with the same length piece from the other string. For example, if string $A = 0011001010$ and string $B = 0001110001$, then exchanging the piece from position 4 to position 7 leads to two new strings, $A' = 0011110010$ and $B' = 0001001001$. This type of change is called recombination or crossover.

At each generation we produce changes using recombination and mutation. We then select from the enlarged population of strings (including strings from the previous generation) a new population for the next generation. Usually, a constant population size is maintained from one generation of strings to the next.

We next have to choose a selection criterion. If we want to model an actual ecosystem, we can include a physical environment and other sets of populations corresponding to different species. The fitness could depend on the interaction of the different species with one another, the interaction within each species, and the interaction with the physical environment. In addition, the behavior of the populations might change the environment from one generation to the next. For simplicity, we will consider only a single population of strings, a simple phenotype, and a simple criteria for fitness.

The phenotype we consider is a variant of the Ising model considered in Problem 14.14. We consider a square lattice of linear dimension L occupied by $N = L^2$ spins that have the values $s_i = \pm 1$. The energy of the system is given by

$$E = -\sum_{i,j=\text{nn(i)}} J_{ij} s_i s_j \tag{14.7}$$

where the sum is over all pairs of spins that are nearest neighbors. The energy function in (14.7) assumes that only nearest neighbor spins interact, in contrast to the energy function in (14.6) which assumes that every spin interacts with every other spin. The coupling constants J_{ij} are either +1, −1, or distributed according to some probability distribution. If we assume that $|J_{ij}| = 1$, then the minimum energy equals $-2N$ and the maximum energy is $2N$. Because we want the fitness to be positive, we choose $2N - E$ as the measure of fitness and take the probability of selecting a particular string with energy E for the next generation to be proportional to the fitness $2N - E$.

How does a genotype become "expressed" as a phenotype? A genotype consists of a string of length N with 1s and 0s. The lattice site (i, j) corresponds to the nth position in the string where $n = jL + i$. If the character in the string at position n is 0, then the spin at site (i, j) equals −1. If the character is 1, then the spin equals +1. Note that in this case the representation of the genotype is very similar to that of the phenotype. In particular, they have the same size N, and each "piece" can have only two values. In general, the expression of the genotype in the phenotype is much more complicated. Usually, a sequence within the genotype corresponds to one value in the phenotype, which in biological systems is related to the coding for a specific protein. Such a sequence is what we call a gene.

We now have all the ingredients we need to apply the genetic algorithm. The GeneticApp class obtains the various parameters, initializes the population of genotypes, and calls the various methods needed to evolve the gene pool (see the doStep method). The GenePool class carries out the evolution. In method recombine two genotypes are chosen at random, and a random piece of one is exchanged for the equivalent piece of the other. In method mutate a random position in a randomly selected genotype is changed. We use a boolean array to represent the genotype, so that a change represents converting true to false or vice versa. In both methods we do not replace the original genotype but instead add a new genotype to the population. The Phenotype class determines the fitness of each member of the population by computing the energy of the lattice of spins corresponding to each member of the population. Members of this population are selected for the new generation by generating a discrete nonuniform probability distribution as discussed in Section 11.5.

Listing 14.10: The GeneticApp class.

```java
package org.opensourcephysics.sip.ch14.genetic;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.frames.*;

public class GeneticApp extends AbstractSimulation {
   GenePool genePool = new GenePool();
   Phenotype phenotype = new Phenotype();
   DisplayFrame frame = new DisplayFrame("Gene pool");

   public void initialize() {
      phenotype.L = control.getInt("Lattice size");
      genePool.populationNumber = control.getInt("Population size");
      genePool.recombinationRate = control.getInt("Recombination rate");
      genePool.mutationRate = control.getInt("Mutation rate");
      genePool.genotypeSize = phenotype.L*phenotype.L;
      genePool.initialize(phenotype);
      phenotype.initialize();
      frame.addDrawable(genePool);
      frame.setPreferredMinMax(-1.0, genePool.genotypeSize+5, -1.0,
           genePool.populationNumber+2);
      frame.setSize(phenotype.L*phenotype.L*10,
         genePool.populationNumber*20);
   }
   public void doStep() {
      genePool.evolve();
      phenotype.determineFitness(genePool);
      phenotype.select(genePool);
      control.clearMessages();
      control.println(genePool.generation+
         " generations, best fitness = "+phenotype.bestFitness);
   }

   public void reset() {
      control.setValue("Lattice size", 8);
      control.setValue("Population size", 20);
      control.setValue("Recombination rate", 10);
      control.setValue("Mutation rate", 4);
   }

   public static void main(String args[]) {
      SimulationControl.createApp(new GeneticApp());
   }
}
```

Listing 14.11: The GenePool class.

```
package org.opensourcephysics.sip.ch14.genetic;
import java.awt.Color;
import java.awt.Graphics;
import org.opensourcephysics.display.*;

public class GenePool implements Drawable {
   int populationNumber;
   int numberOfGenotypes;
   int recombinationRate;
   int mutationRate;
   int genotypeSize;
   boolean[][] genotype;
   int generation = 0;
   Phenotype phenotype;

   public void initialize(Phenotype phenotype) {
      this.phenotype = phenotype;
      generation = 0;
      numberOfGenotypes = populationNumber+2*recombinationRate+mutationRate;
      genotype = new boolean[numberOfGenotypes][genotypeSize];
      for(int i = 0;i<populationNumber;i++) {
         for(int j = 0;j<genotypeSize;j++) {
            if(Math.random()>0.5) {
               genotype[i][j] = true;              // sets genes randomly
            }
         }
      }
   }

   public void copyGenotype(boolean a[], boolean b[]) { // copy a to b
```

```
      for(int i = 0;i<genotypeSize;i++) {
         b[i] = a[i];
      }
   }

   public void recombine() {
      for(int r = 0;r<recombinationRate;r += 2) {
         // chooses random genotype
         int i = (int) (Math.random()*populationNumber);
         int j = 0;
         do {
            // chooses second random genotype
            j = (int) (Math.random()*populationNumber);
         } while(i==j);
         // random size to recombine
         int size = 1+(int) (0.5*genotypeSize*Math.random());
         // random location
         int startPosition =
             (int) (genotypeSize*Math.random());
         // index for new genotype
         int r1 = populationNumber+r;
         // index for second new genotype
         int r2 = populationNumber+r+1;
         copyGenotype(genotype[i], genotype[r1]);
         copyGenotype(genotype[j], genotype[r2]);
         for(int position =
             startPosition;position<startPosition+size;position++) {
            int pbcPosition = position%genotypeSize;
            // make new genotypes
            genotype[r1][pbcPosition] = genotype[j][pbcPosition];
            genotype[r2][pbcPosition] = genotype[i][pbcPosition];
         }
      }
   }

   public void mutate() {
       // index for new genotype
      int index = populationNumber+2*recombinationRate;
      for(int m = 0;m<mutationRate;m++) {
         // choice random existing genotype
         int n = (int) (Math.random()*populationNumber);
         // random position to mutate
         int position = (int) (genotypeSize*Math.random());
         // copy genotype
         copyGenotype(genotype[n], genotype[index+m]);
         // mutate
         genotype[index+m][position] = !genotype[n][position];
      }
   }

   public void evolve() {
      recombine();
      mutate();
      generation++;
   }

   public void draw(DrawingPanel panel, Graphics g) {
      // draws genotype as string of red or green squares and lists
      // fitness for each genotype
      if(genotype==null) {
         return;
      }
      if(phenotype.selectedPopulationFitness==null) {
         return;
      }
      int sizeX = Math.abs(panel.xToPix(0.8)-panel.xToPix(0));
      int sizeY = Math.abs(panel.yToPix(0.6)-panel.yToPix(0));
      for(int n = 0;n<populationNumber;n++) {
         int ypix = panel.yToPix(1.5*n)-sizeY;
         for(int position = 0;position<genotypeSize;position++) {
            if(genotype[n][position]) {
               g.setColor(Color.red);
            } else {
               g.setColor(Color.green);
            }
            int xpix = panel.xToPix(position)-sizeX;
            g.fillRect(xpix, ypix, sizeX, sizeY);
         }
         g.setColor(Color.black);
         g.drawString(String.valueOf(phenotype.selectedPopulationFitness[n]),
                      panel.xToPix(genotypeSize+1), ypix+sizeY);
      }
   }
}
```

Listing 14.12: The Phenotype class.

```
// population of phenotypes (random bond Ising model)
package org.opensourcephysics.sip.ch14.genetic;
public class Phenotype {
   int L;
   int[][][] J; // random bonds
   int[] populationFitness, selectedPopulationFitness;
   int totalFitness;
   int highestEnergy;
   int bestFitness;

   public void initialize() {
      J = new int[L][L][2];
      highestEnergy = 2*L*L; // highest possible energy
      bestFitness = 0;
      for(int i = 0;i<L;i++) {
         for(int j = 0;j<L;j++) {
            for(int bond = 0;bond<2;bond++) {
               if(Math.random()>0.5) {
                  J[i][j][bond] = 1;
               } else {
                  J[i][j][bond] = -1;
               }
            }
         }
      }
   }

   public void determineFitness(GenePool genePool) {
      totalFitness = 0;
      int state[][] = new int[L][L];
      populationFitness = new int[genePool.numberOfGenotypes];
      for(int n = 0;n<genePool.numberOfGenotypes;n++) {
         for(int i = 0;i<L;i++) {
            // sets up lattice based on genotype
            for(int j = 0;j<L;j++) {
               int position = i+j*L;
               if(genePool.genotype[n][position]) {
                  state[i][j] = 1;
               } else {
                  state[i][j] = -1;
               }
            }
         }
         for(int i = 0;i<L;i++) {
            // compute energy of lattice configuration
            for(int j = 0;j<L;j++) {
               populationFitness[n] -=
               state[i][j]*(J[i][j][0]*state[(i+1)%L][j]+
               J[i][j][1]*state[i][(j+1)%L]);
            }
         }
         // fitness > 0; low energy implies high fitness
         populationFitness[n] = highestEnergy-populationFitness[n];
         totalFitness += populationFitness[n];
      }
   }

   public void select(GenePool genePool) {
      selectedPopulationFitness = new int[genePool.numberOfGenotypes];
      boolean savedGenotype[][] =
         new boolean[genePool.numberOfGenotypes][genePool.genotypeSize];
      for(int n = 0;n<genePool.numberOfGenotypes;n++) {
         genePool.copyGenotype(genePool.genotype[n], savedGenotype[n]);
      }
      for(int n = 0;n<genePool.populationNumber;n++) {
         int fitnessFraction = (int) (Math.random()*totalFitness);
         int choice = 0;
         int fitnessSum = populationFitness[0];
         while(fitnessSum<fitnessFraction) {
            choice++;
            fitnessSum += populationFitness[choice];
         }
         selectedPopulationFitness[n] = populationFitness[choice];
```

```
            if(selectedPopulationFitness[n]>bestFitness) {
               bestFitness = selectedPopulationFitness[n];
            }
            genePool.copyGenotype(savedGenotype[choice], genePool.genotype[n]);
         }
      }
   }
```

Problem 14.19. Ground state of Ising-like models

(a) Use the genetic algorithm we have discussed to find the ground state of the ferromagnetic Ising model for which $J_{ij} = 1$. In this case the ground state energy is $E = -2L^2$ (all spins up or all spins down). It will be necessary to modify method `Initialize` in class `Phenotype`. Choose $L = 4$ and consider a population of 20 strings, with 10 recombinations and 4 mutations per generation. How long does it take to find the ground state energy? You might wish to modify the program to show each new generation is shown on the screen.

(b) Find the mean number of generations needed to find the ground state for $L = 4$, 6, and 8. Repeat each run several times. Use a population of 100, a recombination rate of 50, and a mutation rate of 20. Are there any general trends as L is increased? How do your results change if you double the population size? What happens if you double the recombination rate or mutation rate? Use larger lattices if you have sufficient computer resources.

(c) Repeat part (b) for the antiferromagnetic model for which $J_{ij} = -1$.

(d) Repeat part (b) for a spin glass for which $J_{ij} = \pm 1$ at random. In this case we do not know the ground state energy in advance. What criterion can you use to terminate a run? □

One of the important features of the genetic algorithm is that the change in the genetic code is selected not in the genotype directly, but in the phenotype. Note that the way we change the strings (particularly with recombination) is not closely related to the two-dimensional lattice of spins. We could have used some other prescription for converting a string of 0s and 1s to a configuration of spins on a two-dimensional lattice. If the phenotype is a three-dimensional lattice, we could use the same procedure for modifying the genotype, but a different prescription for converting the genetic sequence (the string of 0s and 1s) to the phenotype (the three-dimensional lattice of spins). The point is that it is not necessary for the genetic coding to mimic the phenotypic expression. This point becomes distorted in the popular press when a gene is tied to a particular trait, because specific pieces of DNA rarely correspond directly to any explicitly expressed trait in the phenotype.

14.6 Lattice Gas Models of Fluid Flow

We now return to cellular automaton models and discuss one of their more interesting applications—simulations of fluid flow. In general, fluid flow is very difficult to simulate because the partial differential equation describing the flow of incompressible fluids, the Navier–Stokes equation, is nonlinear, and this nonlinearity can lead to the failure of standard numerical algorithms. In addition, there are typically many length scales that must be considered simultaneously. These length scales include the microscopic motion of the fluid particles, the length scales associated with fluid structures such as vortices, and the length scales of macroscopic objects such as pipes

Velocity Vector	**Direction**	**Symbol**	**Abbreviation**	**Decimal**	**Binary**
$\mathbf{v}_0$	$(1,0)$	RIGHT	RI	1	00000001
$\mathbf{v}_1$	$(1,-\sqrt{3})/2$	RIGHT_DOWN	RD	1	00000010
$\mathbf{v}_2$	$-(1,\sqrt{3})/2$	LEFT_DOWN	LD	4	00000100
$\mathbf{v}_3$	$(-1,0)$	LEFT	LE	8	00001000
$\mathbf{v}_4$	$(-1,\sqrt{3})/2$	LEFT_UP	LU	16	00010000
$\mathbf{v}_5$	$(1,\sqrt{3})/2$	RIGHT_UP	RU	32	00100000
$\mathbf{v}_6$	$(0,0)$	STATIONARY	S	64	01000000
		BARRIER		128	10000000

Table 14.1: Summary of the possible velocities and their representations.

or obstacles. Because of these considerations, simulations of fluid flow based on the direct numerical solutions of the Navier–Stokes equation typically require very sophisticated numerical methods (cf. Oran and Boris).

Cellular automaton models of fluids are known as *lattice gas* models. In a lattice gas model the positions of the particles are restricted to the sites of a lattice, and the velocities are restricted to a small number of vectors corresponding to neighbor sites. A time step is divided into two substeps. In the first substep the particles move freely to their corresponding nearest neighbor lattice sites. Then the velocities of the particles at each lattice site are changed according to a collision rule that conserves mass (particle number), momentum, and kinetic energy. The purpose of the collision rules is not to accurately model microscopic collisions, but rather to achieve the correct macroscopic behavior. The idea is that if we satisfy the conservation laws associated with microscopic collisions, then we can find the correct physics at the macroscopic level, including translational and rotational invariance, by averaging over many particles.

We assume a triangular lattice, because it can be shown that this symmetry is sufficient to yield the macroscopic Navier–Stokes equations for a continuum. In contrast, the more limited symmetry of a square lattice is not sufficient. Three-dimensional models are much more difficult to implement and justify theoretically.

All the moving particles are assumed to have the same speed and mass. The possible velocity vectors lie only in the direction of the nearest neighbor sites, and hence there are six possible velocities as summarized in Table 14.1. A rest particle is also allowed. The number of particles at each site moving in a particular direction (channel) is restricted to be zero or one.

In the first substep all particles move in the direction of their velocity to a neighboring site. In the second substep the velocity vectors at each lattice site are changed according to the appropriate collision rule. Examples of the collision rules are illustrated in Figures 14.4–14.6. The rules are deterministic with only one possible set of velocities after a collision for each possible set of velocities before a collision. It is easy to check that momentum conservation for collisions between the particles is enforced by these rules.

As in Section 14.1, we use bit manipulation to efficiently represent a lattice site and the collision rules. Each lattice site is represented by one element of the integer array `lattice`. In Java each `int` stores 32 bits, but we will use only the first 8 bits. We use the first six bits from 0 to 5 to represent particles moving in the six possible directions with bit 0 corresponding to a particle moving with velocity $\mathbf{v}_0$ (see Table 14.1). If there are three particles with velocities $\mathbf{v}_0$, $\mathbf{v}_2$, and $\mathbf{v}_4$ at a site and no barrier, then the value of the lattice array element at this site is 00010101 in binary notation.

Bit 6 represents a possible rest (stationary) particle. If we want a site to act as a barrier that

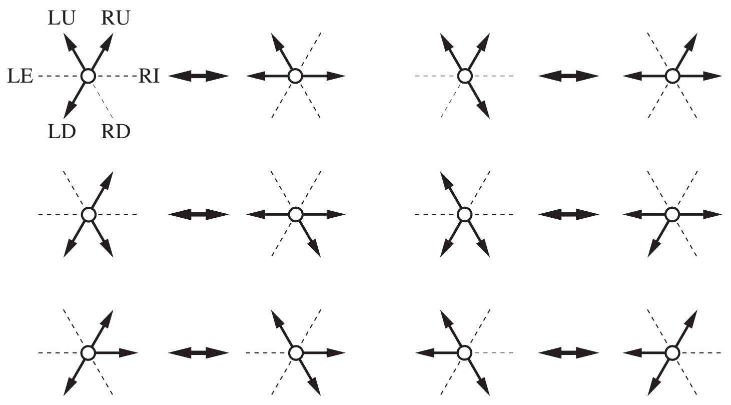


Figure 14.4: Examples of collision rules for three particles, with one particle unchanged and no stationary particles. Each direction or channel is represented by 32 bits, but we need only the first 8 bits. The various channels are summarized in Table 14.1.

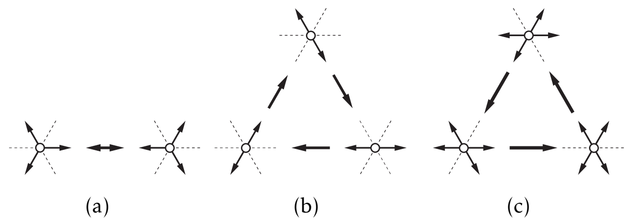


Figure 14.5: (a) Example of collision rule for three particles with zero net momentum. (b) Example of two particle collision rule. (c) Example of four-particle collision rule. The rules for states that are not shown is that the velocities do not change after a collision. An open circle represents a lattice site and the absence of a stationary particle.

blocks incoming particles, we set bit 7. For example, a barrier site containing a particle with velocity $\mathbf{v}_1$ is represented by 10000010.

The rules for the collisions are given in the declaration of the class variables in class `LatticeGas`. Because `rule` is declared `static final`, we cannot normally overwrite its values. However, an exception is made for static initializers that are run when the class is first loaded. To construct the rules, we use the bitwise *or* operator | and use named constants for each of the possible states. As an example, the state corresponding to one particle moving to the right, one moving to the left and down, and one moving to the left and up is given by `LU`+`LD`+`RI`, which we write as `LU|LD|RI` or 00010101. The collision rule in Figure 14.5(a) is that this state transforms to one particle moving to the right and down, one moving left, and one moving to the right and up. Hence, this collision rule is given by `rule[LU|LD|RI]` = `RU|LE|RD`. The other rules are given in a similar way. Stationary particles can also be created or destroyed. For example, what are the states before and after the collision for `rule[LU|RI]` = `RU|S`?

To every rule corresponds a dual rule that flips the bits corresponding to the presence and absence of a particle. This duality means that we need to only specify half of the rules. The dual rules can be constructed by flipping all bits of the input and output. Our convention is to list the rules starting without a stationary particle. Then the corresponding dual rules are those

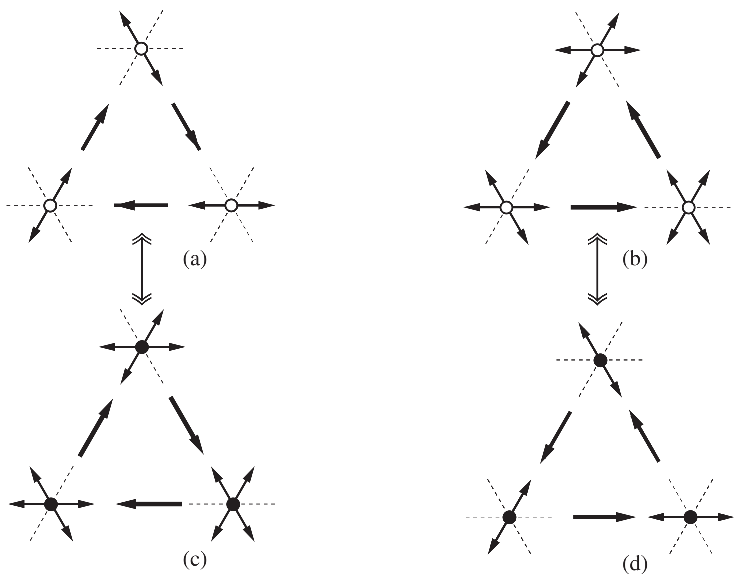


Figure 14.6: (a) and (c) and (b) and (d) are duals of each other. An open circle represents the absence of a stationary particle, and a filled circle represents the presence of a stationary particle. Note that the collision rule in (c) is similar to (b), and the collision rule in (d) is similar to (a), but in the opposite direction.

that start with a stationary particle. The dual rules are implemented by the statement

```
rule[i^(RU|LU|LE|LD|RD|RI|S)] = rule[i]^(RU|LU|LE|LD|RD|RI|S);
```

where ˆ is the bitwise exclusive or operator, which equals 1 if both bits are different and is 0 otherwise. Two examples of dual rules are given in Figure 14.6.

The rules in Figures 14.5(b) and 14.5(c) cycle through the states in a particular direction. Although these rules are straightforward, they are not invariant under reflection. To help eliminate this bias, we cycle in the opposite direction when a stationary particle is present (see Figure 14.6).

We adopt the rule that when a particle moves onto a barrier site, we set the velocity **v** of this particle equal to −**v** (see Figure 14.7). Because of our ordering of the velocities, the rule for updating a barrier can be expressed compactly using bit manipulation. Reflection off a barrier is accomplished by shifting the higher-order bits to the right by three bits (»>3) and shifting the lower-order bits to the left by three bits («3). Check the rules given in Listing 14.13. Other possibilities are to set the angle of incidence equal to the angle of reflection or to set the velocity to an arbitrary direction. The latter case would correspond to a collision off a rough surface.

The `step` method runs through the entire lattice and moves all the particles. The updated values of the sites are placed in the array `newLattice`. We then go through the `newLattice` array, implement the relevant collision rule at each site, and write the results into the array `Lattice`.

The movement of the particles is accomplished as follows. Because the even rows are horizontally displaced one half a lattice spacing from the odd rows, we need to treat odd and even rows separately. In the `step` method we loop through every other row and update `site1` and `site2` at the same time. An example will show how this update works. The statement

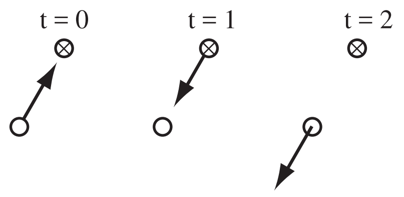


Figure 14.7: Example of a collision from a barrier. At $t = 1$ the particle moves to the barrier site and then reverses its velocity. The symbol ⊗ denotes a barrier site.

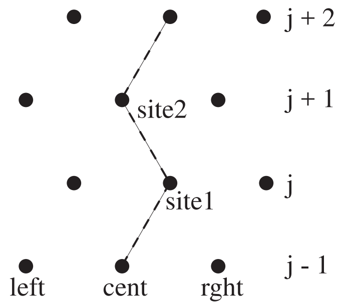


Figure 14.8: We update `site1` and `site2` at the same time. The rows are indexed by `j`. The dotted line connects sites in the same column.

```
rght[j-1] |= site1 & RIGHT_DOWN;
```

means that if there is a particle moving to the right and down at `site1`, then the bit corresponding to `RIGHT_DOWN` is added to the site `rght` (see Figure 14.8). The statement

```
cent[j] |= site1 & (STATIONARY|BARRIER) | site2 & RIGHT_DOWN;
```

means that a stationary particle at `site1` remains there, and if `site1` is a barrier, it remains so. If `site2` has a particle moving in the direction `RD`, then `site1` will receive this particle.

To maintain a steady flow rate, we add the necessary horizonal momentum to the lattice uniformly after each time step. The procedure is to chose a site at random and determine if it is possible to change the sites's horizontal momentum. If so, we then remove the left bit and add the right bit or vice versa. This procedure is accomplished by the statements at the end of the `step` method.

Listing 14.13: Listing of the `LatticeGas` class.

```
package org.opensourcephysics.sip.ch14.latticegas;
import org.opensourcephysics.display.*;
import java.awt.*;
import java.awt.geom.AffineTransform;
import java.awt.geom.Line2D;

public class LatticeGas implements Drawable {
   // input parameters from user
   public double flowSpeed;          // controls pressure
   // size of velocity arrows displayed
   public double arrowSize;
   public int spatialAveragingLength; // spatial averaging of velocity
   public int Lx, Ly;                 // linear dimensions of lattice
   public int[][] lattice, newLattice;
   private double numParticles;
   static final double SQRT3_OVER2 = Math.sqrt(3)/2;
   static final double SQRT2 = Math.sqrt(2);
   static final int
      RIGHT = 1, RIGHT_DOWN = 2, LEFT_DOWN = 4;
   static final int
      LEFT = 8, LEFT_UP = 16, RIGHT_UP = 32;
   static final int
      STATIONARY = 64, BARRIER = 128;
      // maximum number of particles per site
   static final int NUM_CHANNELS = 7;
   // 7 channel bits plus 1 barrier bit per site
   static final int NUM_BITS = 8;
   // total number of possible site configurations = 2^8
   static final int NUM_RULES = 1<<8;
   // 1 << 8 means move the zeroth bit over 8 places to the left to
   // the eighth bit

   static final double ux[] = {
      1.0, 0.5, -0.5, -1.0, -0.5, 0.5, 0
   };
   static final double uy[] = {
      0.0, -SQRT3_OVER2, -SQRT3_OVER2, 0.0, SQRT3_OVER2, SQRT3_OVER2, 0
   };
   // averaged velocities for every site configuration
   static final double[] vx, vy;
   static final int[] rule;

   static { // set rule table
      // default rule is the identity rule
      rule = new int[NUM_RULES];
      for(int i = 0;i<BARRIER;i++) {
         rule[i] = i;
      }
      // abbreviations for channel bit indices
      int RI = RIGHT, RD = RIGHT_DOWN, LD = LEFT_DOWN;
      int LE = LEFT, LU = LEFT_UP, RU = RIGHT_UP;
      int S = STATIONARY;
      // three particle zero momentum rules
      rule[LU|LD|RI] = RU|LE|RD;
      rule[RU|LE|RD] = LU|LD|RI;
      // three particle rules with unperturbed particle
      rule[RU|LU|LD] = LU|LE|RI;
      rule[LU|LE|RI] = RU|LU|LD;
      rule[RU|LU|RD] = RU|LE|RI;
      rule[RU|LE|RI] = RU|LU|RD;
      rule[RU|LD|RD] = LE|RD|RI;
      rule[LE|RD|RI] = RU|LD|RD;
      rule[LU|LD|RD] = LE|LD|RI;
      rule[LE|LD|RI] = LU|LD|RD;
```

```
      rule[RU|LD|RI] = LU|RD|RI;
      rule[LU|RD|RI] = RU|LD|RI;
      rule[LU|LE|RD] = RU|LE|LD;
      rule[RU|LE|LD] = LU|LE|RD;
      // two particle cyclic rules
      rule[LE|RI] = RU|LD;
      rule[RU|LD] = LU|RD;
      rule[LU|RD] = LE|RI;
      // four particle cyclic rules
      rule[RU|LU|LD|RD] = RU|LE|LD|RI;
      rule[RU|LE|LD|RI] = LU|LE|RD|RI;
      rule[LU|LE|RD|RI] = RU|LU|LD|RD;
      // stationary particle creation rules
      rule[LU|RI] = RU|S;
      rule[RU|LE] = LU|S;
      rule[LU|LD] = LE|S;
      rule[LE|RD] = LD|S;
      rule[LD|RI] = RD|S;
      rule[RD|RU] = RI|S;
      rule[LU|LE|LD|RD|RI] = RU|LE|LD|RD|S;
      rule[RU|LE|LD|RD|RI] = LU|LD|RD|RI|S;
      rule[RU|LU|LD|RD|RI] = RU|LE|RD|RI|S;
      rule[RU|LU|LE|RD|RI] = RU|LU|LD|RI|S;
      rule[RU|LU|LE|LD|RI] = RU|LU|LE|RD|S;
      rule[RU|LU|LE|LD|RD] = LU|LE|LD|RI|S;
      // add all rules indexed with a stationary particle (dual rules)
      for(int i = 0;i<S;i++) {
         // ^ is the exclusive or operator
         rule[i^(RU|LU|LE|LD|RD|RI|S)] = rule[i]^(RU|LU|LE|LD|RD|RI|S);
      }
      // add rules to bounce back at barriers
      for(int i = BARRIER;i<NUM_RULES;i++) {
         // & is bitwise and operator
         int highBits = i&(LE|LU|RU);
         int lowBits = i&(RI|RD|LD);
         rule[i] = BARRIER|(highBits>>3)|(lowBits<<3);
      }
   }
   static { // set average site velocities
      // for every particle site configuration i, calculate total
      //  net velocity and place in vx[i], vy[i]
      vx = new double[NUM_RULES];
      vy = new double[NUM_RULES];
      for(int i = 0;i<NUM_RULES;i++) {
         for(int dir = 0;dir<NUM_CHANNELS;dir++) {
            if((i&(1<<dir))!=0) {
               vx[i] += ux[dir];
               vy[i] += uy[dir];
            }
         }
      }
   }
   public void initialize(int Lx, int Ly, double density) {
      this.Lx = Lx;
      this.Ly = Ly-Ly%2;                          // Ly must be even
      // approximate total number of particles
      numParticles = Lx*Ly*NUM_CHANNELS*density;
      // density = number of particles divided by the maximum number possible
      lattice = new int[Lx][Ly];
      newLattice = new int[Lx][Ly];
      int sevenParticleSite = ((1<<NUM_CHANNELS)-1); // equals 127
      for(int i = 0;i<Lx;i++) {
         // wall at top and bottom
         lattice[i][1] = lattice[i][Ly-2] = BARRIER;
         for(int j = 2;j<Ly-2;j++) {
            // occupy site by 0 or 7 particles, average occupation will
            // be about the density
            int siteValue = Math.random()<density ? sevenParticleSite : 0;
            lattice[i][j] = siteValue; // random particle configuration
         }
      }
      for(int j = 3*Ly/10;j<7*Ly/10;j++) {
         lattice[2*Lx/10][j] = BARRIER; // obstruction toward the left
      }
   }

   public void step() {
      // move all particles forward
      for(int i = 0;i<Lx;i++) {
         // define the columns of a 2-dim array
         int[] left = newLattice[(i-1+Lx)%Lx];
         // use abbreviations to align expressions
         int[] cent = newLattice[i];
         int[] rght = newLattice[(i+1)%Lx];
         for(int j = 1;j<Ly-2;j += 2) {
            // loop j in increments of 2 to decrease reads and writes
            // of neighbors
            int site1 = lattice[i][j];
            int site2 = lattice[i][j+1];
            // move all particles in site1 and site2 to their neighbors
            rght[j-1] |= site1&RIGHT_DOWN;
            cent[j-1] |= site1&LEFT_DOWN;
            rght[j] |= site1&RIGHT;
            cent[j] |= site1&(STATIONARY|BARRIER)|site2&RIGHT_DOWN;
            left[j] |= site1&LEFT|site2&LEFT_DOWN;
            rght[j+1] |= site1&RIGHT_UP|site2&RIGHT;
            cent[j+1] |= site1&LEFT_UP|site2&(STATIONARY|BARRIER);
            left[j+1] |= site2&LEFT;
            cent[j+2] |= site2&RIGHT_UP;
            left[j+2] |= site2&LEFT_UP;
         }
      } // handle collisions, find average x velocity
      double vxTotal = 0;
      for(int i = 0;i<Lx;i++) {
         for(int j = 0;j<Ly;j++) {
            int site = rule[newLattice[i][j]]; // use collision rule
            lattice[i][j] = site;
            newLattice[i][j] = 0;      // reset newLattice values to 0
            vxTotal += vx[site];
         }
      }
      int scale = 4;
      int injections = (int) ((flowSpeed*numParticles-vxTotal)/scale);
      for(int k = 0;k<Math.abs(injections);k++) {
         int i = (int) (Math.random()*Lx); // choose site at random
         int j = (int) (Math.random()*Ly);
         // flip direction of horizontally moving particle if possible
         if((lattice[i][j]&(RIGHT|LEFT))==((injections>0) ? LEFT : RIGHT)) {
            lattice[i][j] ^= RIGHT|LEFT;
         }
      }
   }

   public void draw(DrawingPanel panel, Graphics g) {
      if(lattice==null) {
         return;
      }
      // if s = 1 draw lattice and particle details explicitly
      // otherwise average velocity over an s by s square
      int s = spatialAveragingLength;
      Graphics2D g2 = (Graphics2D) g;
      AffineTransform toPixels = panel.getPixelTransform();
      Line2D.Double line = new Line2D.Double();
      for(int i = 0;i<Lx;i++) {
         for(int j = 2;j<Ly-2;j++) {
            double x = i+(j%2)*0.5;
            double y = j*SQRT3_OVER2;
            if(s==1) {
               g2.setPaint(Color.BLACK);
               for(int dir = 0;dir<NUM_CHANNELS;dir++) {
                  if((lattice[i][j]&(1<<dir))!=0) {
                     line.setLine(x, y, x+ux[dir]*0.4, y+uy[dir]*0.4);
                     g2.draw(toPixels.createTransformedShape(line));
                  }
               }
            }
            // draw points at lattice sites
            if((lattice[i][j]&BARRIER)==BARRIER||s==1) {
               Circle c = new Circle(x, y);
               c.pixRadius = ((lattice[i][j]&BARRIER)==BARRIER) ? 2 : 1;
               c.draw(panel, g);
            }
         }
      }
      if(s==1) {
         return;
      }
      for(int i = 0;i<Lx;i += s) {
         for(int j = 0;j<Ly;j += s) {
            double x = i+s/2.0;
            double y = (j+s/2.0)*SQRT3_OVER2;
            double
               wx = 0, wy = 0; // compute coarse grained average velocity
            for(int m = i;m!=(i+s)%Lx;m = (m+1)%Lx) {
               for(int n = j;n!=(j+s)%Ly;n = (n+1)%Ly) {
                  wx += vx[lattice[m][n]];
                  wy += vy[lattice[m][n]];
               }
            }
            Arrow a = new Arrow(x, y, arrowSize*wx/s, arrowSize*wy/s);
            a.setHeadSize(2);
            a.draw(panel, g);
         }
      }
   }
}
```

Listing 14.14: Listing of the LatticeGasApp class.

```
package org.opensourcephysics.sip.ch14.latticegas;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.frames.*;

public class LatticeGasApp extends AbstractSimulation {
   LatticeGas model = new LatticeGas();
   DisplayFrame display = new DisplayFrame("Lattice gas");

   public LatticeGasApp() {
      display.addDrawable(model);
      display.setSize(800, (int) (400*Math.sqrt(3)/2));
   }

   public void initialize() {
      int lx = control.getInt("lx");
      int ly = control.getInt("ly");
      double density = control.getDouble("Particle density");
      model.initialize(lx, ly, density);
      model.flowSpeed = control.getDouble("Flow speed");
      model.spatialAveragingLength = control.getInt("Spatial averaging length");
      model.arrowSize = control.getInt("Arrow size");
      display.setPreferredMinMax(-1, lx, -Math.sqrt(3)/2, ly*Math.sqrt(3)/2);
   }

   public void doStep() {
      model.flowSpeed = control.getDouble("Flow speed");
      model.spatialAveragingLength = control.getInt("Spatial averaging length");
      model.arrowSize = control.getDouble("Arrow size");
      model.step();
   }

   public void reset() {
      control.setValue("lx", 1000);
      control.setValue("ly", 500);
      control.setValue("Particle density", 0.2);
      control.setAdjustableValue("Flow speed", 0.2);
      control.setAdjustableValue("Spatial averaging length", 20);
      control.setAdjustableValue("Arrow size", 2);
      enableStepsPerDisplay(true);
      control.setAdjustableValue("steps per display", 100);
   }

   public static void main(String[] args) {
      SimulationControl.createApp(new LatticeGasApp());
   }
}
```

An important application of lattice gas models is to simulate the flow in and around various geometries. In Problem 14.20 we will see that the fluid velocity field develops vortices, wakes, and other fluid structures near obstacles. Method `initialize` in class `LatticeGas` places an obstacle in the middle of the lattice and provides initial values for each site. Large lattices are required to obtain quantitative results, because it is necessary to average the velocity over many sites. The parameter `density` is the average number of particles divided by the maximum possible. The pressure can be varied by changing the `flowSpeed` parameter.

Problem 14.20. Flow past a barrier

(a) Convince yourself that you understand the collision rules and their implementation in class `LatticeGas`. Then download the class `FastLatticeGas` from the ch14 directory. This latter class uses all 32 bits of an `int` variable and runs about twice as fast. The tradeoff is that the code is more difficult to debug and understand. Use the parameters in Listing 14.14. Describe the flow once a steady-state velocity field begins to appear. Do you see a wake appearing behind the obstacle? Are there vortices?

(b) Repeat part (a) with different size obstacles. Are there any systematic trends? (One limitation of the present program is that it naively redraws a circle to represent each barrier site. This redrawing requires a significant amount of computer resources and limits the size of the obstacles that we can consider.)

(c) Reduce the pressure by reducing the flow speed. Are there any noticeable changes in behavior from parts (a) and (b)? Reduce the pressure still further and describe any changes in the fluid flow. □

Problem 14.21. Approach to equilibrium

(a) Consider the approach of a lattice gas to equilibrium. Modify `LatticeGas` so that the initial configuration has zero net momentum, the particles are localized in a $b \times b$ region, and there are no barrier sites. Choose $L = 30$ and $b = 4$ and place six particles at every site in the localized region. The other sites in the lattice are initially empty. Describe what happens to the particles as a function of time. Approximately how many time steps does it take for the system to come to equilibrium? Do the particles appear to be at random positions with random velocities? What is your visual algorithm for determining when equilibrium has been reached?

(b) Repeat part (a) for $b = 2$, 6, 8, and 10. Estimate the equilibration time in each case. What is the qualitative dependence of the equilibration time on b? How does the equilibration time depend on the number density ρ?

(c) Repeat part (a) with $b = 4$, but with $L = 10$, 20, and 40. Estimate the equilibration time in each case. How does the equilibration time depend on ρ? □

Problem 14.22. Fluid flow in porous media

(a) Modify class `LatticeGas` so that instead of a rectangular barrier, the barrier sites are placed at random in the system. We define the porosity ϕ as the fraction of sites without a barrier. The interesting quantity to measure is the permeability, k, which is a measure of the fluid conductivity. We can compute the permeability using the relation

$$k \propto \frac{\phi \sum_i \langle v_{i,x} \rangle}{\sum_j \langle \Delta p_{j,x} \rangle} \tag{14.8}$$

where the sum in the numerator is over the horizontal velocity of all particles in the pore space (the sites at which there are no barriers), and the sum in the denominator is over the injected momentum at all sites used to maintain the flow. The brackets refer to averages over time. Compute the permeability as a function of the porosity ϕ and display your results on a log-log plot. You should average over at least 10 configurations of random barrier sites for each value of the porosity. What value of ϕ corresponds to the percolation threshold, defined by $k = 0$? See Rothman and Zaleski for a discussion of the comparison of this type of simulation with results for real rocks.

(b)* Vary the size of the lattice and use the finite size scaling procedure discussed in Section 12.4 to estimate the critical exponent μ defined by the dependence of the permeability on the porosity; that is, $k \sim (\phi - \phi_c)^{\mu}$. Assume that you know the value of the percolation exponent ν defined by the critical behavior of the connectedness length $\xi \sim |p - p_c|^{-\nu}$ (see Table 12.1). □

The principal virtues of lattice gas models are their use of simultaneous updating, which makes them very fast on parallel computers, and their use of integer or boolean arithmetic and bit manipulation, which is faster than floating point arithmetic. Their major limitation is that it is necessary to average over many sites to obtain quantitative results. It is not yet clear whether lattice gas models are more efficient than standard simulations of the Navier–Stokes equation. The greatest promise for lattice gas models may not be with simple single component fluids, but with multicomponent fluids such as binary fluids and fluids containing bubbles (see the book by Rothman and Zaleski). A related technique that might hold greater promise is the lattice Boltzmann method (see the references).

14.7 Overview and Projects

The models we have discussed in this chapter have been presented as algorithms rather than in terms of differential equations and are a reflection of the way that technology affects the way we think. Can you discuss the models in this chapter without thinking about their implementation on a computer? Can you imagine understanding these models without the use of computer graphics?

We have given only a brief introduction to cellular automata and other models that are relevant to the rapidly developing study of complex systems. There are many more models and applications that we have not discussed, ranging from aging, the immune system, economic cycles, and pedestrian movements to name just a few.

Models of opinion formation have become popular in recent years. The basic idea is that the opinions of others will influence the opinion of individuals. The first two projects in the following explore some of the popular models.

Project 14.23. Models of opinion formation

(a) *The voter model.* On a regular lattice, assign each site the value ±1. Choose a site (the voter) at random. The voter then adopts the same value as a randomly chosen neighbor. These two steps continue until all sites have the same value; that is, when they have reached consensus. Compute the probability of achieving a consensus of +1 given that the initial density of +1 sites is ρ_0. Use a 10×10 square lattice and make at least 20 runs at each density. Also compute the time to reach consensus as a function of the lattice size. In two dimensions this time scales as $N \ln N$, where N is the number of sites. How does the consensus time scale with N in $d = 1$ and $d = 3$ dimensions? How does it scale on a preferential attachment network (see the article by Sood and Redner)?

(b) *The relative agreement interaction model.* N individuals are initially assigned an opinion that takes on a value between 0 and 1. Choose two individuals, i and j, at random. Assume that the ith opinion O_i is greater than the jth opinion O_j. If their opinions differ by less than the parameter ϵ, then increase O_j by $(m/2)(O_i - O_j)$ and decrease O_i by the same amount, where m is another parameter. This model implements the idea that two people will influence each other only if their opinions are sufficiently close. Write a program to simulate this model. Use a `LatticeFrame` for which each cell can take on one of 256 values. The approximation of the continuum by 256 values is for visualization purposes only, and the 256 values should be sufficiently large to approximate a continuum of values. Choose $\epsilon = 10$, 50, and 100 (out of 256), and $m = 0.3$ and 0.6. To speed up the simulation, include in your program the option to plot configurations only after a certain number of iterations (use `enableStepsPerDisplay(true)`). Choose $N \geq 2500$, begin with a random set of opinions. Discuss whether a single opinion emerges and explain the magnitude of the fluctuations.

(c) *The Sznajd model.* Place individuals on a square lattice with linear dimension L and periodic boundary conditions. Each individual has one of two opinions. At each iteration, an individual and one of the person's neighbors is chosen at random. If the two individuals have the same opinion, the opinion of the six neighbors of the pair is changed to that of the pair. The idea is that people are more likely to change their opinion to those physically near them if more than one person shares the same opinion (peer pressure). Write a program to simulate this model and show that consensus is always reached for all sites if the simulation is run for a sufficiently long time. Discuss the visual appearance of the groups of like-minded individuals. Consider initial configurations where the individuals are randomly assigned the two opinions and initial configurations where one opinion has a majority of 1%, 5%, and 10%. Choose $L \geq 50$.

(d) Generalize the Sznajd model so that an individual may be assigned one of more than two opinions. Is consensus still always reached? What happens if the individuals are not on the sites of a square lattice, but rather are the nodes of a preferential attachment network of at least 5000 nodes? □

Project 14.24. The minority game

In certain situations we wish to be in the minority. For example, we might wish to go to a popular restaurant on an off-night so that we do not have to wait in line. A business might want to sell goods and services that are not being sold by other businesses. The following algorithm, known as the *minority game*, is a model of adaptive competition where each player tries to maximize his gain. We will find that there is a phase transition between states where the players mainly act on their own and states for which cooperative behavior emerges.

There are N players, where N is odd. At each iteration, each player can choose one of two actions which we call 1 or 0 but which we encode as the boolean `true` or `false`. A player's choice is determined by a strategy based on the previous m (memory) iterations. Each strategy is represented by a table of all the possible outcomes of the previous m iterations and a decision on what to do for each outcome. Each player has his own table of strategies. An outcome is defined as the action that was chosen *least* by all the players. For example, suppose $m = 2$. There are four possible pasts: (1,1), (1,0), (0,1), and (0,0). The past (1,1) means that in each of the last two iterations, action 1 was chosen by a minority of the players. A strategy would be encoded by a table such as the following: (1,1,1), (1,0,0), (0,1,0), and (0,0,1). The first two entries in each triple are the possible outcomes of the last two iterations, and the third entry in the triple gives the action that the strategy suggests taking. Thus the triple (1,1,1) means that if (1,1) occurred in the past, the strategy is to choose action 1; (1,0,0) means that if (1,0) occurred in the past, the strategy is to use action 0. At the beginning of the game,each player is assigned at least two strategies that are chosen at random from all possible strategies. As the game is played, the performance of each strategy (whether or not it is used) is updated, such that if a strategy leads to the same action that was in the minority, then this strategy is successful and its performance is incremented by unity; otherwise, it stays the same. At each iteration each player chooses the strategy with the best performance and takes the action determined by his best performing strategy. Then the outcome (which action was in the minority) for that iteration is determined, and the past m outcomes and the performance for all the strategies are updated. Note that the strategies available to each player does not change, but which of each player's strategy is best changes as the game is iterated.

To simplify the code, represent the past outcomes by an integer where each bit represents an outcome. For example, the bit 110 means that the outcome was 0 in the last iteration and was 1 for each of the earlier two iterations. You will need the following arrays: `strategies[i][j][k]`, which gives the action for the ith player using its jth strategy when the kth past occurred; `performance[i][j]`, which gives the performance for the ith player's jth strategy, and `chosen-Strategy[i]`, which gives the strategy chosen by the ith player in the current iteration.

Let N_1 equal the number of players who chose action 1 in one iteration. The outcome is best if at each iteration the value of N_1 is close to $N/2$, because in this way there would be as many players as possible in the minority. The quantity of interest is σ, where σ is defined as

$$\sigma^2 = \sum_k (N_1(k) - \langle N_1 \rangle)^2 / N_{\text{step}}, \tag{14.9}$$

and the sum is over the N_{step} iterations the game, and $N_1(k)$ is the number of players choosing action 1 in the kth iteration. The quantity σ decreases as the efficiency increases. High efficiency means that on the average more players are in the minority. We might think that the efficiency increases as the number of past outcomes increases, because then the players have more information to choose their strategy. However, you might be surprised!

(a) Write a program to simulate the minority game. For simplicity, give each player only two strategies chosen at random. Run your program for a memory m varying from 2 to about 12. A reasonable choice for N is 101, but for testing purposes choose $N = 11$. Each game should be run for at least 1000 iterations, and your results should be averaged over at least 10 independent runs for the same m, with different strategies for the players. Plot the average of σ versus m and describe the behavior for different values of N. Explain why there is a minimum in these plots.

(b) The results of the minority game scale unambiguously. Plot the average of σ^2/N versus $2^m/N$ for different values of N. You should find that your data fall on the same curve. What does 2^m represent? Discuss this scaling behavior and describe the behavior of the efficiency on either side of the minimum. Can you describe your results as a phase transition? Where is the ordered phase and where is the disordered phase?

(c) Plot the spread in the values of σ versus m. The spread can be taken to be the standard deviation of each game's value of σ over many games. Discuss the significance of your results. □

Project 14.25. A cellular automaton for Burger's equation

In Section 14.6 we mentioned that the partial differential equation describing the flow of incompressible fluids, the Navier–Stokes equation, is very difficult to solve numerically. A one-dimensional approximation of the Navier–Stokes equation was given by Burgers, and is given by

$$\frac{\partial n}{\partial t} + c\frac{\partial}{\partial x}\Big(n - \frac{n^2}{2}\Big) = D\frac{\partial^2 n}{\partial x^2} \tag{14.10}$$

where $n(x,t)$ corresponds to the velocity field at position x at time t, c is the linear advection (drift) coefficient, and D is a diffusion coefficient. Equation (14.10) is of general interest because it can be solved analytically and its solutions exhibit discontinuities (shock waves) depending on the values of the parameters and the initial conditions.

Boghosian and Levermore have proposed a cellular automaton that is equivalent to (14.10). The study of this cellular automaton raises many of the same issues as the lattice gas models of the incompressible Navier–Stokes equation considered in Section 14.6. Its study also illustrates the idea that many partial differential equations can be formulated as cellular automata.

We know that if all particles on the lattice move one lattice site to either the right or the left in one time step, then the density of the particles obeys the diffusion equation (see Appendix 7A),

$$\frac{\partial n}{\partial t} = D\frac{\partial^2 n}{\partial x^2} \tag{14.11}$$

where $D = (\Delta x)^2/2\Delta t$, Δx is the lattice spacing, and Δt is the time between successive steps of the random walk. If add a bias so that the probability of a step to the right is $(1+\alpha)/2$ and the probability of a step to the left is $(1-\alpha)/2$, the density of the walkers satisfies

$$\frac{\partial n}{\partial t} + c\frac{\partial n}{\partial x} = D\frac{\partial^2 n}{\partial x^2} \tag{14.12}$$

where $c = \alpha\Delta x/\Delta t$. To incorporate the quadratic term, we add the rule that no two particles occupying the same site may be moving in the same direction. In this way the state of each site is specified by two bits. The right bit is 1 if a particle moving to the right is present and is 0 otherwise. Similarly, the left bit stores information about the presence of a particle moving to the left. Thus each site has four possible states labeled by the binary numbers 00, 01, 10, and 11.

In the first part of the step, the collision substep, the particles change their direction at random at their present lattice sites subject to the exclusion rule. In the second substep, the

$\mathbf{b_1(i,t)}$	$\mathbf{b_0(i,t)}$	$\mathbf{\tilde{b}_1(i,t)}$	$\mathbf{\tilde{b}_0(i,t)}$
0	0	0	0
0	1	$(1-\alpha(i,t)/2$	$(1+\alpha(i,t)/2$
1	0	$(1-\alpha(i,t)/2$	$(1+\alpha(i,t)/2$
1	1	1	1

Table 14.2: Rules for the collision substep.

particles move to the neighboring lattice site in their new direction. We follow Boghosian and Levermore and denote the right (left) bit at lattice site i and time step t by $b_0(x,t)$ ($b_1(x,t)$). After the collision substep, the new states are $\tilde{b}_{0,1}(x,t)$ and are given in Table 14.2, where $\alpha(x,t) = \pm 1$ with mean α. The rules in Table 14.2 may be written in the form

$$\tilde{b}_0(x,t) = \frac{1+\alpha(x,t)}{2} b_0(x,t)|b_1(x,t) + \frac{1-\alpha(x,t)}{2} b_0(x,t)\,\&\, b_1(x,t) \tag{14.13a}$$

$$\tilde{b}_1(x,t) = \frac{1-\alpha(x,t)}{2} b_0(x,t)|b_1(x,t) + \frac{1+\alpha(x,t)}{2} b_0(x,t)\,\&\, b_1(x,t). \tag{14.13b}$$

where | is the inclusive or operator and & denotes the and operator on a pair of bits. In the advection substep, the particles move to the neighboring lattice site in their new direction. The rules for these moves are

$$\tilde{b}_0(x+1,t+1) = \tilde{b}_0(x,t) \tag{14.14a}$$

$$\tilde{b}_1(x-1,t+1) = \tilde{b}_1(x,t). \tag{14.14b}$$

We can combine these two substeps to arrive at the rule for one full time step of the cellular automaton:

$$b_0(x+1,t+1) = \frac{1+\alpha(x,t)}{2} b_0(x,t)|b_1(x,t) + \frac{1-\alpha(x,t)}{2} b_0(x,t)\,\&\, b_1(x,t) \tag{14.15a}$$

$$b_1(x-1,t+1) = \frac{1-\alpha(x,t)}{2} b_0(x,t)|b_1(x,t) + \frac{1+\alpha(x,t)}{2} b_0(x,t)\,\&\, b_1(x,t). \tag{14.15b}$$

Write a program to implement (14.15) using periodic boundary conditions. Choose $c = 1$, $D = 2^{-15}$, and the initial condition

$$n(x, t=0) = 1.0 + 0.4\cos(2\pi x) \tag{14.16}$$

where x denotes the position of a lattice site. Boghosian and Levermore used $2^{16} = 65,536$ lattice sites so that $\Delta x = 2^{-16}$. The bias is given by $\alpha = c\Delta x/2D = 0.25$ and the time step is $\Delta t = (\Delta x)^2/2D = 2^{-18}$. Average your results for 128 lattice sites and plot the average density as a function of x for different values of t up to $t = 1$. Do you see any evidence of a shock wave [a sharp discontinuity in $n(x)$]? □

Project 14.26. Spring-block model of earthquakes

The first simulations of earthquakes were done by Burridge and Knopoff in 1967. Their model represents the motion of one side of a lateral fault that is driven by a slow shear deformation and subject to a nonlinear, velocity-dependent friction force. The model consists of a one-dimensional array of blocks on a substrate (see Figure 14.9). Each block is connected to its nearest neighbors by springs with spring constant k_c, which represent the linear elastic response

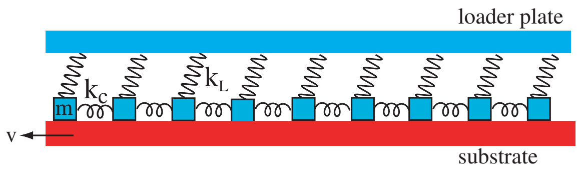


Figure 14.9: Schematic of the Burridge–Knopoff model. Blocks with mass m are attached to their nearest neighbors by springs with spring constant k_c. They are also attached to a fixed loader plate with spring constant k_L. The substrate moves with speed v to the left.

of the system to compressional deformations. Each block is also connected by a spring with spring constant k_L to a fixed loader plate.

The system is loaded by moving the substrate at a constant speed v to the left. Eventually, the force on a block exceeds the static friction threshold F_0 and the block slips. As the block moves, the springs connecting it to its neighbors change length, thus changing the forces acting on them. The neighboring blocks begin to accelerate if the force is sufficient; that is, if the force due to the springs is greater than the static friction force.

The equation of motion of the Burridge–Knopoff model can be written as

$$m\ddot{x}_j = k_c(x_{j+1} - 2x_j + x_{j-1}) - k_L x_j - F(v + \dot{x}_j) \tag{14.17}$$

where x_j is the displacement of the jth block. The force between the blocks is $k_c(x_{j+1} - 2x_j + x_{j-1})$, the force from the loader plate is $-k_L x_j$, and F represents the friction force due to the substrate. Periodic boundary conditions are not used.

As usual, it is convenient to introduce dimensionless variables, which we take to be $u_j = (k_L/F_0)x_j$, $\omega_L^2 = k_L/m$, and $\tau = \omega_L t$. We rewrite (14.17) as

$$\ddot{u}_j = \ell^2(u_{j+1} - 2u_j + u_{j-1}) - u_j - \phi(2\alpha\nu + 2\alpha\dot{u}_j) \tag{14.18}$$

where $\phi(w) = F(w)/F_0$, the stiffness parameter $\ell = \sqrt{k_c/k_L}$, $\nu = vk_L/\omega_L F_0$, and $2\alpha = \omega_L F_0/k_L v$; the dot now denotes differentiation with respect to τ. The equation of motion (14.18) can be solved using the Euler-Richardson algorithm with $\Delta\tau = 10^{-3}$.

The velocity of a block is set to zero if at any time the speed of the block relative to the substrate is less than a parameter v_0, its speed is decreasing, and the force due to the springs is less than F_0. Otherwise, the friction force is given by

$$\phi(w) = \frac{1-\sigma}{1 + \frac{w}{1-\sigma}} \qquad (w > 0), \tag{14.19}$$

where the parameter σ represents the drop of the friction force at the onset of the slip. If a block is stuck, the calculation of the static friction force is a bit more involved. If the total force on a block due to the springs is to the right, then the static friction force is set equal and opposite to the total spring force up to a maximum value of F_0. However, if the total spring force is to the left, the static friction is chosen so that the acceleration of the block is zero. Typical values of the parameters are $F_0 = 1$, $\ell = 10$, $\sigma = 0.01$, $\alpha = 2.5$, and $v_0 = 10^{-5}$.

Initially we set $\dot{u}_j = 0$ for all j and assign small random displacements to all the blocks. The blocks will then move according to (14.18). For simplicity we set the substrate velocity $v = 0$, and when all the blocks become stuck, we move all the blocks to the left by an equal amount such that the total force due to the springs on one block equals unity (F_0). This procedure will then cause one block to move or slip. As this block moves, other neighboring blocks may move leading to an earthquake. Eventually, all the blocks will again become stuck. The main quantities of interest are $P(s)$, the distribution of the number of blocks that have moved during an earthquake, and $P(M)$, the distribution of the net displacement of the blocks during an earthquake, where

$$M = \sum_i \Delta u_i. \tag{14.20}$$

The sum over i in (14.20) is over the blocks involved in an earthquake, and Δu_i is the net displacement of the blocks during the earthquake. Do $P(s)$ and $P(M)$ exhibit scaling consistent with Gutenberg–Richter?

The movement of the blocks represents the slip of the two surfaces of a fault past one another during an earthquake. The stick-slip behavior of this model is similar to that of a real earthquake fault. Other interesting questions are posed in the references (see Klein et al., Ferguson et al., and Mori and Kawamura). □

References and Suggestions for Further Reading

Réka Albert and Albert–László Barabási, "Statistical mechanics of complex networks," Rev. Mod. Phys. **74**, 47–97 (2002).

Per Bak, *How Nature Works* (Copernicus Books, 1999). A good read about self-organized critical phenomena from earthquakes to stock markets. Nature is not as simple as Bak believed, but his interest in complex systems spurred many others to become interested.

P. Bak, "Catastrophes and self-organized criticality," Computers in Physics **5** (4), 430 (1991). A good introduction to self-organized critical phenomena.

Per Bak and Michael Creutz, "Fractals and self-organized criticality," in *Fractals in Science*, Armin Bunde and Shlomo Havlin, editors (Springer–Verlag, 1994).

Per Bak and Kim Sneppen, "Punctuated equilibrium and criticality in a simple model of evolution," Phys. Rev. Lett. **71**, 4083 (1993); Henrik Flyvbjerg, Kim Sneppen, and Per Bak, "Mean field theory for a simple model of evolution," Phys. Rev. Lett. **71**, 4087 (1993).

P. Bak, C. Tang, and K. Wiesenfeld, "Self-organized criticality," Phys. Rev. A **38**, 364–374 (1988).

E. R. Berlekamp, J. H. Conway, and R. K. Guy, *Winning Ways for your Mathematical Plays*, Vol. 2 (Academic Press, 1982). A discussion of how the Game of Life simulates a universal computer.

Bruce M. Boghosian and C. David Levermore, "A cellular automaton for Burger's equation," Complex Systems **1**, 17–30 (1987). Reprinted in Doolen et al.

D. Challet and Y.-C. Zhang, "Emergence of cooperation and organization in an evolutionary game," Physica A **246**, 407–418 (1997), or adap-org/9708006. The authors give the first description of the minority game..

Debashish Chowdhury, Ludger Santen, and Andreas Schadschneider, "Simulation of vehicular traffic: A statistical physics perspective," Computing in Science and Engineering **2** (5), 80–87 (2000).

John W. Clark, Johann Rafelski, and Jeffrey V. Winston, "Brain without mind: Computer simulation of neural networks with modifiable neuronal interactions," Physics Reports **123**, 215–273 (1985).

Aaron Clauset, M. E. J. Newman, and Cristopher Moore, "Finding community structure in very large networks," Phys. Rev. E **70**, 066111-1–6 (2004). This paper describes a faster algorithm than that discussed in Newman and Girvan.

J. P. Crutchfield and M. Mitchell, "The evolution of emergent computation," Proc. Natl. Acad. Sci. **92**, 10742–10746 (1995). The authors use genetic algorithms to evolve a cellular automata model.

Guillaume Deffuant, Fréd'eric Amblard, Gérard Weisbuch, and Thierry Faure, "How can extremism prevail? A study based on the relative agreement interaction model," J. Artificial Societies and Social Simulation **5**(4) paper #1 (2002). This paper and others can be found at <jasss.soc.surrey.ac.uk>.

Gary D. Doolen, Uriel Frisch, Brosl Hasslacher, Steven Orszag, and Stephen Wolfram, editors, *Lattice Gas Methods for Partial Differential Equations* (Addison–Wesley, 1990). A collection of reprints and original articles by many of the leading workers in lattice gas methods.

Stephanie Forrest, editor, *Emergent Computation: Self-Organizing, Collective, and Cooperative Phenomena in Natural and Artificial Computing Networks* (MIT Press, 1991).

Stephen I. Gallant, *Neural Network Learning and Expert Systems* (MIT Press, 1993).

M. Gardner, *Wheels, Life and Other Mathematical Amusements* (W. H. Freeman, 1983).

Peter Grassberger, "Efficient large-scale simulations of a uniformly driven system," Phys. Rev. E **49**, 2436–2444 (1994). Grassberger considered substantially larger lattices and longer simulation times than that used by Olami et al. and found that the Olami, Feder, Christensen model does not exhibit power law scaling.

G. Grinstein and C. Jayaprakash, "Simple models of self-organized criticality," Computers in Physics **9**, 164 (1995).

G. Grinstein, Terence Hwa, and Henrik Jeldtoft Jensen, "$1/f^\alpha$ noise in dissipative transport," Phys. Rev. A **45**, R559–R562 (1992).

B. Hayes, "Computer recreations," Sci. Am. **250** (3), 12–21 (1984). An introduction to cellular automata.

J. E. Hanson and J. P. Crutchfield, "Computational mechanics of cellular automata: An example," Physica D **103**, 169–189 (1997). The authors discuss energence in cellular automata.

Robert Herman, editor, *The Theory of Traffic Flow* (Elsevier, 1961).

John Hertz, Anders Krogh, and Richard G. Palmer, *Introduction to the Theory of Neural Computation* (Addison–Wesley, 1991).

J. J. Hopfield, "Neural networks and physical systems with emergent collective computational abilities," Proc. Natl. Acad. Sci. USA **79**, 2554–2558 (1982).

Navot Israeli and Nigel Goldenfeld, "Computational irreducibility and the predictability of complex physical systems," Phys. Rev. Lett. **92**, 074105 (2004).

H. M. Jaeger, Chu–heng Liu, and Sidney R. Nagel, "Relaxation at the angle of repose," Phys. Rev. Lett. **62**, 40 (1989). These authors discuss experiments on real sandpiles.

W. Klein, C. Ferguson, and J. B. Rundle, "Spinodals and scaling in slider block models," in *Reduction and Predictability of Natural Disasters*, J. B. Rundle, D. L. Turcotte, and W. Klein, editors (Addison–Wesley, 1995). Also see C. D. Ferguson, W. Klein, and John B. Rundle, "Spinodals, scaling, and ergodicity in a threshold model with long-range stress transfer," Phys. Rev. E **60**, 1359–1373 (1999).

J. A. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection* (MIT Press, 1992).

Chris Langton, "Studying artificial life with cellular automata," Physica D **22**, 120–149 (1986). See also Christopher G. Langton, editor, *Artificial Life* (Addison–Wesley, 1989); Christopher G. Langton, Charles Taylor, J. Doyne Farmer, and Steen Rasmussen, editors, *Artificial Life II*, Addison–Wesley (1989); Christopher G. Langton, editor, *Artificial Life III* (Addison–Wesley, 1994).

Roger Lewin, *Complexity: Life at the Edge of Chaos* (University of Chicago Press, 2000). A popular exposition of complexity theory.

Sergei Maslov, Maya Paczuski, and Per Bak, "Avalanches and 1/f noise in evolution and growth models," Phys. Rev. Lett. **73**, 2162 (1994).

Stephan Mertens, "Computational complexity for physicists," Computing in Science and Engineering **4** (3), 31–47 (2002).

Takahiro Mori and Hikaru Kawamura, "Simulation study of the one-dimensional Burridge–Knopoff model of earthquakes," J. Geophysical Res. **111**, B07302 (2006).

K. Nagel and M. Schreckenberg, "A cellular automaton model for freeway traffic," J. Phys. I France **2**, 2221–2229 (1992). Also see <www.traffic.uni-duisburg.de/>.

Kai Nagel, Dietrich E. Wolf, Peter Wagner, and Patrice Simon, "Two-lane traffic rules for cellular automata: A systematic approach," Phys. Rev. E **58**, 1425–1437 (1998).

M. E. J. Newman, "The structure and function of complex networks," SIAM Rev. **45**, 167–256 (2003).

M. E. J. Newman, "Detecting community structure in networks," Eur. Phys. J. B **38**, 321–330 (2004); M. E. J. Newman and M. Girvan, "Finding and evaluating community structure in networks," Phys. Rev. E **69**, 026113-1–15 (2004). These papers describe an algorithm for detecting the heirarchical structure of networks.

J. A. Niesse, R. P. White, and H. R. Mayne, "Genetic algorithm approaches to minimum energy geometry of aromatic hydrocarbon clusters," J. Chem. Phys. **108**, 2208–2218 (1998).

Z. Olami, H. J. S. Feder, and K. Christensen, "Self-organized criticality in a continuous, nonconservative cellular automaton modeling earthquakes," Phys. Rev. Lett. **68**, 1244 (1992).

Suzana Moss de Oliveira, Jorge S. Sá Martins, Paulo Murilo C. de Oliveira, Karen Luz-Burgoa, Armando Ticona, and Thadeu J. P. Penna, "The Penna model for biological aging and speciation," Computing in Science and Engineering **6** (3), 74–81 (2004). Also see Dietrich Stauffer, "The complexity of biological ageing," cond-mat/0310038.

Elaine S. Oran and Jay P. Boris, *Numerical Simulation of Reactive Flow*, 2nd ed. (Cambridge University Press, 2002). Although much of this book assumes an understanding of fluid dynamics, the discussion of simulation methods and the numerical solution of the differential equations of fluid flow does not require much background.

Michel Peyrard, "Nonlinear dynamics and statistical physics of DNA," Nonlinearity **17**, R1–R40 (2004). The author describes a simple mechanical model of DNA [see Figure 10 and Eq. (1)] that is in the same spirit as the Burridge–Knopoff model of earthquakes.

William Poundstone, *The Recursive Universe* (Contemporary Books, 1985). A book on the Game of Life that attempts to draw analogies between the patterns of Life and ideas of information theory and cosmology.

Drek de Solla Price, "Networks of scientific papers," Science **149**, 510–515 (1965); "A genral theory of bibliometric and other cummulative advantage processes," J. Amer. Soc. Inform. Sci. **27**, 292–306 (1976). Possibly the first description of a scale-free network and the explanation for power law distributions.

Daniel H. Rothman and Stéphane Zalesk, *Lattice-Gas Cellular Automata* (Cambridge University Press, 1997). This text includes a discussion of fluid flow through porous media as well as the lattice Boltzmann method for simulating fluids. Also see Daniel H. Rothman and Stéphane Zaleski, "Lattice-gas models of phase separation: interfaces, phase transitions, and multiphase flow," Rev. Mod. Phys. **66**, 1417–1479 (1994).

David E. Rumelhart and James L. McClelland, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, Vol. 1: *Foundations* (MIT Press, 1986). See also Vol. 2 on applications.

Robert Savit, Radu Manuca, and Rick Riolo, "Adaptive competition, market efficiency, and phase transitions," Phys. Rev. Lett. **82**, 2203 (1999). Analysis of the scaling behavior of the minority game.

Herbert A Simon, "On a class of skew distribution functions," Biometrika, **42**, 425–440 (1955). An early paper that shows power laws coming from preferential attachment.

V. Sood and S. Redner, "Voter model on heterogeneous graphs," Phys. Rev. Lett. **94**, 178701 (2005).

Dietrich Stauffer, "Monte Carlo simulations of Sznajd models," J. Artificial Societies and Social Simulation **5**(1) paper #4 (2002). This paper and other relevant papers can be found at `<jasss.soc.surrey.ac.uk>`.

Dietrich Stauffer, "Cellular automata," Chapter 9 in *Fractals and Disordered Systems*, Armin Bunde and Shlomo Havlin, editors (Springer–Verlag, 1991). Also see Dietrich Stauffer, "Programming cellular automata," Computers in Physics **5** (1), 62 (1991).

Daniel L. Stein, editor, *Lectures in the Sciences of Complexity*, Vol. 1 (Addison–Wesley, 1989); Erica Jen, editor, *Lectures in Complex Systems*, Vol. 2 (Addison–Wesley, 1990); Daniel L. Stein and Lynn Nadel, editors, *Lectures in Complex Systems*, Vol. 3 (Addison–Wesley, 1991).

Patrick Sutton and Sheri Boyden, "Genetic algorithms: A general search procedure," Am. J. Phys. **62**, 549–552 (1994). This readable paper discusses the application of genetic algorithms to Ising models and function optimization.

K. Sznajd-Weron and J. Sznajd, "Opinion evolution in closed community," Int. J. Mod. Phys. C **11** (6), 1157–1165 (2000).

Tommaso Toffoli and Norman Margolus, *Cellular Automata Machines—A New Environment for Modeling* (MIT Press, 1987). See also Norman Margolus and Tommaso Toffoli, "Cellular automata machines," in the volume edited by Doolen et al.

D. J. Tritton, *Physical Fluid Dynamics*, 2nd ed. (Oxford Science Publications, 1988). An excellent introductory text that integrates theory and experiment. Although there is only a brief discussion of numerical work, the text provides the background useful for simulating fluids.

M. Mitchell Waldrop, *Complexity: The Emerging Science at the Edge of Order and Chaos* (Simon and Schuster, 1992). A popular exposition of complexity theory.

Stephen Wolfram, editor, *Theory and Applications of Cellular Automata* (World Scientific, 1986). A collection of research papers on cellular automata that range in difficulty from straightforward to specialists only. An extensive annotated bibliography also is given. Two papers in this collection that discuss the classification of one-dimensional cellular automata are S. Wolfram, "Statistical mechanics of cellular automata," Rev. Mod. Phys. **55**, 601–644 (1983), and S. Wolfram, "Universality and complexity in cellular automata," Physica B **10**, 1–35 (1984).

Stephen Wolfram, *A New Kind of Science* (Wolfram Media, 2002). This book discusses many important ideas and computer experiments on cellular automata. More information can be found at <www.stephenwolfram.com/>. An interesting review of this book is given by L. Kadanoff, "Wolfram on cellular automata," Phys. Today **55** (7), 55–56 (2002).

László Zalányi, Gábor Csárdi, Tamás Kiss, Máté Lengyel, Rebecca Warner, Jan Tobochnik, and Péter Érdi, "Properties of a random attachment growing network," Phys. Rev. E. **68**, 066104-1–9 (2003).

Chapter 15

Monte Carlo Simulations of Thermal Systems

We discuss how to simulate thermal systems using a variety of Monte Carlo methods including the traditional Metropolis algorithm. Applications to the Ising model and various particle systems are discussed and more efficient Monte Carlo algorithms are introduced.

15.1 Introduction

The Monte Carlo simulation of the particles in the box problem discussed in Chapter 7 and the molecular dynamics simulations discussed in Chapter 8 have exhibited some of the important qualitative features of macroscopic systems such as the irreversible approach to equilibrium and the existence of equilibrium fluctuations in macroscopic quantities. In this chapter we apply various Monte Carlo methods to simulate the equilibrium properties of thermal systems. These applications will allow us to explore some of the important concepts of statistical mechanics.

Due in part to the impact of computer simulations, the applications of statistical mechanics have expanded from the traditional areas of dense gases, liquids, crystals, and simple models of magnetism to the study of complex materials, particle physics, and theories of the early universe. For example, the demon algorithm introduced in Section 15.3 was developed by a physicist interested in lattice gauge theories which are used to describe the interactions of fundamental particles.

15.2 The Microcanonical Ensemble

We first discuss an isolated system for which the number of particles N, the volume V, and the total energy E are fixed and external influences such as gravitational and magnetic fields can be ignored. The *macrostate* of the system is specified by the values of E, V, and N. At the microscopic level, there are many different ways or *configurations* in which the macrostate (E,V,N) can be realized. A particular configuration or *microstate* is accessible if its properties are consistent with the specified macrostate.

All we know about the accessible microstates is that their properties are consistent with the known physical quantities of the system. Because we have no reason to prefer one microstate

↓↓↓↓	↓↓↓↑	↓↓↑↑	↓↑↑↑	↑↑↑↑
	↓↓↑↓	↓↑↓↑	↑↓↑↑	
	↓↑↓↓	↓↑↑↓	↑↑↓↑	
	↑↓↓↓	↑↓↓↑	↑↑↑↓	
		↑↓↑↓		
		↑↑↓↓		
$4\mu B$	$2\mu B$	0	$-2\mu B$	$-4\mu B$

Table 15.1: The sixteen microstates for a one-dimensional system of $N = 4$ noninteracting spins. The total energy E of each microstate is also shown. If the total energy of the system is $E = -2\mu B$, then there are four accessible microstates (see the fourth column). Hence, in this case the ensemble consists of four systems, each in a different microstate with equal probability.

over another when the system is in equilibrium, it is reasonable to postulate that the system is *equally* likely to be in any one of its accessible microstates. To make this postulate of *equal a priori probabilities* more precise, imagine an isolated system with Ω accessible states. The probability P_s of finding the system in microstate s is

$$P_s = \begin{cases} 1/\Omega, & \text{if } s \text{ is accessible} \\ 0, & \text{otherwise.} \end{cases} \tag{15.1}$$

The sum of P_s over all Ω states is equal to unity. Equation (15.1) is applicable only when the system is in equilibrium.

The averages of physical quantities can be determined in two ways. In the usual laboratory experiment, the physical quantities of interest are measured over a time interval sufficiently long to allow the system to sample a large number of its accessible microstates. We computed such time averages in Chapter 8, where we used the method of molecular dynamics to compute the time-averaged values of quantities such as the temperature and pressure. An interpretation of the probabilities in (15.1) that is consistent with such a time average is that during a sequence of observations, P_s yields the fraction of times that a single system is found in a given microstate.

Although time averages are conceptually simple, it is convenient to imagine a collection or *ensemble* of systems that are identical mental copies characterized by the same macrostate but, in general, by different microstates. In this interpretation, the probabilities in (15.1) describe an ensemble of identical systems, and P_s is the probability that a system in the ensemble is in microstate s. An ensemble of systems specified by E, N, V is called a *microcanonical* ensemble. An advantage of ensembles is that statistical averages can be determined by sampling the states according to the desired probability distribution. Much of the power of Monte Carlo methods is that we can devise sampling methods based on a fictitious dynamics that is more efficient than the real dynamics.

Suppose that a physical quantity A has the value A_s when the system is in microstate s. Then the ensemble average of A is given by

$$\langle A \rangle = \sum_{s=1}^{\Omega} A_s P_s \tag{15.2}$$

where P_s is given by (15.1).

To illustrate these ideas, consider a one-dimensional system of N noninteracting spins on a lattice. The spins can be in one of two possible directions which we take to be up or down.

The total energy of the system is $E = -\mu B \sum_i s_i$, where each lattice site has associated with it a number $s_i = \pm 1$, where $s_i = +1$ for an up spin and $s_i = -1$ for a down spin; B is the magnetic field, and μ is the magnetic moment of a spin. A particular microstate of the system of spins is specified by the set of variables $\{s_1, s_2, \ldots, s_N\}$. In this case the macrostate of the system is specified by E and N.

In Table 15.1 we show the 16 microstates with $N = 4$. If the total energy $E = -2\mu B$, we see that there are four accessible microstates. Hence, in this case there are four systems in the ensemble each with an equal probability. The enumeration of the systems in the ensemble and their probability allows us to calculate ensemble averages for the physical quantities of interest.

Problem 15.1. A simple ensemble average

Consider a one-dimensional system of $N = 4$ noninteracting spins with total energy $E = -2\mu B$. What is the probability P_i that the ith spin is up? Does your answer depend on which spin you choose? □

15.3 The Demon Algorithm

We found in Chapter 8 that we can do a time average of a system of many particles with E, V, and N fixed by integrating Newton's equations of motion for each particle and computing the time-averaged value of the physical quantities of interest. How can we do an ensemble average at fixed E, V, and N? And what can we do if there is no equation of motion available? One way would be to enumerate all the accessible microstates and calculate the ensemble average of the desired physical quantities as we did in Table 15.1. This approach is usually not practical because the number of microstates for even a small system is much too many to enumerate. In the spirit of Monte Carlo, we wish to develop a practical method of obtaining a representative sample of the total number of microstates. One possible procedure is to fix N, choose each spin to be up or down at random, and retain the configuration if it has the desired total energy. However, this procedure is very inefficient because most configurations would not have the desired total energy and would have to be discarded.

An efficient Monte Carlo procedure for simulating systems at a given energy was developed by Creutz in the context of lattice gauge theory. Suppose that we add an extra degree of freedom to the original macroscopic system of interest. For historical reasons, this extra degree of freedom is called a *demon*. The demon transfers energy as it attempts to change the dynamical variables of the system. If the desired change lowers the energy of the system, the excess energy is given to the demon. If the desired change raises the energy of the system, the demon gives the required energy to the system if the demon has sufficient energy. The only constraint is that the demon cannot have negative energy.

We first apply the demon algorithm to a one-dimensional classical system of N noninteracting particles of mass m (an ideal gas). The total energy of the system is $E = \sum_i mv_i^2/2$, where v_i is the velocity of particle i. In general, the demon algorithm is summarized by the following steps:

1. Choose a particle at random and make a trial change in its coordinates.
2. Compute ΔE, the change in the energy of the system due to the change.
3. If $\Delta E \le 0$, the system gives the amount $|\Delta E|$ to the demon, that is, $E_d = E_d - \Delta E$, and the trial configuration is accepted.

4. If $\Delta E > 0$ and the demon has sufficient energy for this change ($E_d \geq \Delta E$), then the demon gives the necessary energy to the system, that is, $E_d = E_d - \Delta E$, and the trial configuration is accepted. Otherwise, the trial configuration is rejected and the configuration is not changed.

The above steps are repeated until a representative sample of states is obtained. After a sufficient number of steps, the demon and the system will agree on an average energy for each. The total energy of the system plus the demon remains constant, and because the demon is only one degree of freedom in comparison to the many degrees of freedom of the system, the energy fluctuations of the system will be of order $1/N$, which is very small for $N \gg 1$.

The ideal gas has a trivial dynamics. That is, because the particles do not interact, their velocities do not change. (The positions of the particles change, but the positions are irrelevant because the energy depends only on the velocity of the particles.) So the use of the demon algorithm is equivalent to a fictitious dynamics that lets us sample the microstates of the system. Of course, we do not need to apply the demon algorithm to an ideal gas because all its properties can be calculated analytically. However, it is a good idea to consider a simple example first.

How do we know that the Monte Carlo simulation of the microcanonical ensemble will yield results equivalent to the time-averaged results of molecular dynamics? The assumption that these two types of averages yield equivalent results is called the *quasi-ergodic* hypothesis. Although these two averages have not been proven to be identical in general, they have been found to yield equivalent results in all cases of interest.

`IdealDemon` and `IdealDemonApp` implement the microcanonical Monte Carlo simulation of the ideal classical gas in one dimension. To change a configuration, we choose a particle at random and change its velocity by a random amount. The parameter `mcs`, the number of Monte Carlo steps per particle, plays an important role in Monte Carlo simulations. On the average, the demon attempts to change the velocity of each particle once per Monte Carlo step per particle. We frequently will refer to the number of Monte Carlo steps per particle as the "time," even though this time has no obvious direct relation to a physical time.

Listing 15.1: The demon algorithm for the one-dimensional ideal gas.

```java
package org.opensourcephysics.sip.ch15;
public class IdealDemon {
   public double v[];
   public int N;
   public double systemEnergy;
   public double demonEnergy;
   public int mcs = 0; // number of MC moves per particle
   public double systemEnergyAccumulator = 0;
   public double demonEnergyAccumulator = 0;
   public int acceptedMoves = 0;
   public double delta;

   public void initialize() {
      v = new double[N]; // array to hold particle velocities
      double v0 = Math.sqrt(2.0*systemEnergy/N);
      for(int i = 0;i<N;++i) {
         v[i] = v0; // give all particles the same initial velocity
      }
      demonEnergy = 0;
      resetData();
   }

   public void resetData() {
      mcs = 0;
      systemEnergyAccumulator = 0;
      demonEnergyAccumulator = 0;
      acceptedMoves = 0;
   }

   public void doOneMCStep() {
      for(int j = 0;j<N;++j) {
         // choose particle at random
         int particleIndex = (int) (Math.random()*N);
         // random change in velocity
         double dv = (2.0*Math.random()-1.0)*delta;
         double trialVelocity = v[particleIndex]+dv;
         double dE = 0.5*(trialVelocity*trialVelocity-
            v[particleIndex]*v[particleIndex]);
         if(dE<=demonEnergy) {
            v[particleIndex] = trialVelocity;
            acceptedMoves++;
            systemEnergy += dE;
            demonEnergy -= dE;
         }
         systemEnergyAccumulator += systemEnergy;
         demonEnergyAccumulator += demonEnergy;
      }
      mcs++;
   }
}
```

Listing 15.2: The target application for the simulation of an ideal gas using the demon algorithm.

```
   package org.opensourcephysics.sip.ch15;
   import org.opensourcephysics.controls.*;

   public class IdealDemonApp extends AbstractSimulation {
      IdealDemon idealGas = new IdealDemon();

      public void initialize() {
         idealGas.N = control.getInt("number of particles N");
         idealGas.systemEnergy = control.getDouble("desired total energy");
         idealGas.delta = control.getDouble("maximum velocity change");
         idealGas.initialize();
      }

      public void doStep() {
         idealGas.doOneMCStep();
      }

      public void stop() {
         double norm = 1.0/(idealGas.mcs*idealGas.N);
         control.println("mcs = "+idealGas.mcs);
         control.println("<Ed> = "+idealGas.demonEnergyAccumulator*norm);
         control.println("<E> = "+idealGas.systemEnergyAccumulator*norm);
         control.println("acceptance ratio = "+idealGas.acceptedMoves*norm);
      }

      public void reset() {
         control.setValue("Number of particles N", 40);
         control.setValue("desired total energy", 40);
         control.setValue("maximum velocity change", 2.0);
      }

      public void resetData() {
         idealGas.resetData();
         idealGas.delta = control.getDouble("delta");
         control.clearMessages();
      }

      public static void main(String[] args) {
         SimulationControl control = SimulationControl.createApp(new IdealDemonApp());
         control.addButton("resetData", "Reset Data"); //
      }
   }
```

Problem 15.2. Monte Carlo simulation of an ideal gas

(a) Use the classes `IdealDemon` and `IdealDemonApp` to investigate the equilibrium properties of an ideal gas. Note that the mass of the particles has been set equal to unity and the initial demon energy is zero. For simplicity, the same initial velocity has been assigned to all the particles. Begin by using the default values given in the listing of `IdealDemonApp`. What is the mean value of the particle velocities after equilibrium has been reached?

(b) The configuration corresponding to all particles having the same velocity is not very likely, and it would be better to choose an initial configuration that is more likely to occur when the system is in equilibrium. In any case, we should let the system evolve until it has reached equilibrium before we accumulate data for the various averages. We call this time the equilibration or relaxation time. We can estimate the equilibration time from a plot of the demon energy versus the time. Alternatively, we can reset the data until the computed averages stop changing systematically. Clicking the Reset Data button sets the accumulated sums to zero without changing the configuration. Determine the mean demon energy $\langle E_d \rangle$ and the mean system energy per particle using the default values for the parameters.

(c) Compute the mean energy of the demon and the mean system energy per particle for $N = 100$ and $E = 10$ and $E = 20$, where E is the total energy of the system. Use your result from part (b) and obtain an approximate relation between the mean demon energy and the mean system energy per particle.

(d) In the microcanonical ensemble the total energy is fixed with no reference to the temperature. Define the kinetic temperature by the relation $\frac{1}{2}m\langle v^2 \rangle = \frac{1}{2}kT_{\text{kinetic}}$, where $\frac{1}{2}m\langle v^2 \rangle$ is the mean kinetic energy per particle of the system. Use this relation to obtain T_{kinetic}. Choose units such that m and Boltzmann's constant k are unity. How is T_{kinetic} related to the mean demon energy? How do your results compare to the relation given in introductory physics textbooks that the total energy of an ideal gas of N particles in three dimensions is $E = \frac{3}{2}NkT$? (In one dimension the analogous relation is $E = \frac{1}{2}NkT$.)

(e) A limitation of most simulations is the finite number of particles. Is the relation between the mean demon energy and mean kinetic energy per particle the same for $N = 2$ and $N = 10$ as it is for $N = 40$? If there is no statistically significant difference between your results for the three values of N, explain why finite N might not be an important limitation for the ideal gas in this simulation. □

Problem 15.3. Demon energy distribution

(a) Add a method to class `IdealDemon` to compute the probability $P(E_d)\Delta E_d$ that the demon has energy between E_d and $E_d + \Delta E_d$. Choose the same parameters as in Problem 15.2 and be sure to determine $P(E_d)$ only after equilibrium has been obtained.

(b) Plot the natural logarithm of $P(E_d)$ and verify that $\ln P(E_d)$ depends linearly on E_d with a negative slope. What is the absolute value of the slope? How does the inverse of this value correspond to the mean energy of the demon and T_{kinetic} as determined in Problem 15.2?

(c) Generalize the `IdealDemon` class and determine the relation between the mean demon energy, the mean energy per particle of the system, and the inverse of the slope of $\ln P(E_d)$ for an ideal gas in two and three dimensions. It is straightforward to write the class so that it is valid for any spatial dimension. □

15.4 The Demon as a Thermometer

We found in Problem 15.3 that the form of $P(E_d)$ is given by

$$P(E_d) \propto e^{-E_d/kT}. \tag{15.3}$$

We also found that the parameter T in (15.3) is related to the kinetic temperature of an ideal gas.

In Problem 15.4 we will do some further simulations to determine the generality of the form (15.3).

Problem 15.4. The Boltzmann probability distribution

Modify your simulation of an ideal gas so that the kinetic energy of a particle is proportional to the absolute value of its momentum instead of the square of its momentum. Such a dependence would hold for a relativistic gas where the particles are moving at velocities close to the speed of light. Choose various values of the total energy E and number of particles N. Is the form of $P(E_d)$ the same as in (15.3)? How does the inverse slope of $\ln P(E_d)$ versus E_d compare to the mean energy per particle of the system in this case? □

According to the equipartition theorem of statistical mechanics, each quadratic degree of freedom contributes $\frac{1}{2}kT$ to the energy per particle. Problem 15.4 shows that the equipartition theorem is not applicable for other dependencies of the particle energy.

Although the microcanonical ensemble is conceptually simple, it does not represent the situation usually found in nature. Most systems are not isolated but are in thermal contact with their environment. This thermal contact allows energy to be exchanged between the laboratory system and its environment. The laboratory system is usually small relative to its environment. The larger system with many more degrees of freedom is commonly referred to as the *heat* *reservoir* or *heat bath*. The term heat refers to energy transferred from one body to another due to a difference in temperature. A heat bath is a system for which such energy transfer causes a negligible change in its temperature.

A system that is in equilibrium with a heat bath is characterized by the temperature of the latter. If we are interested in the equilibrium properties of such a system, we need to know the probability P_s of finding the system in microstate s with energy E_s. The ensemble that describes the probability distribution of a system in thermal equilibrium with a heat bath is known as the *canonical* ensemble. In general, the canonical ensemble is characterized by the temperature T, the number of particles N, and the volume V, in contrast to the microcanonical ensemble which is characterized by the energy E, N, and V.

We have already discussed an example of a system in equilibrium with a heat bath, the demon! In Problems 15.2–15.4, the system of interest was an ideal gas and the demon was an auxiliary (special) particle that facilitated the exchange of energy between the particles of the system. If we take the demon to be the system of interest, we see that the demon exchanges energy with a much bigger system (the ideal gas), which we can take to be the heat bath. We conclude that the probability distribution of the microstates of a system in equilibrium with a heat bath has the same form as the probability distribution of the energy of the demon. (Note that the microstate of the demon is characterized by its energy.) Hence, the probability that a system in equilibrium with a heat bath at temperature T is in microstate s with energy E_s has the form given by (15.3):

$$P_s = \frac{1}{Z} e^{-\beta E_s} \qquad \text{(canonical distribution)}, \tag{15.4}$$

where $\beta = 1/kT$ and Z is a normalization constant. Because $\sum P_s = 1$, Z is given by

$$Z = \sum_s e^{-E_s/kT}. \tag{15.5}$$

The sum in (15.5) is over the microstates of the system for a given N and V. The quantity Z is the *partition function* of the system. The ensemble defined by (15.4) is known as the *canonical* ensemble, and the probability distribution (15.4) is the *Boltzmann* or the *canonical distribution*. The derivation of the Boltzmann distribution is given in textbooks on statistical mechanics. We will simulate systems in equilibrium with a heat bath in Section 15.6.

The partition function plays a key role in statistical mechanics, because the (Helmholtz) free energy F of a system is defined as

$$F = -kT \ln Z. \tag{15.6}$$

All thermodynamic quantities can be found from various derivatives of F. In equilibrium the system will be in the state of minimum F for given values of T, V, and N. (This result follows from the second law of thermodynamics which says that a system with fixed E, V, and N will be in the state of maximum entropy.) We will use the free energy concept in a number of the following sections.

The form (15.4) of $P(E_d)$ provides a simple way of computing the temperature T from the mean demon energy $\langle E_d \rangle$. The latter is given by

$$\langle E_d \rangle = \frac{\int_0^\infty E_d \, e^{-E_d/kT} \, dE_d}{\int_0^\infty e^{-E_d/kT} \, dE_d} = kT. \tag{15.7}$$

We see that T is proportional to the mean demon energy. Note that the result $\langle E_d \rangle = kT$ in (15.7) holds only if the energy of the demon can take on a continuum of values and if the upper limit of integration can be taken to be ∞.

Figure 15.1: The interaction energy between nearest neighbor spins in the absence of an external magnetic field.

The demon is an excellent example of a thermometer. It has a measurable property, namely, its energy, which is proportional to the temperature. Because the demon is only one degree of freedom in comparison to the many degrees of freedom of the system with which it exchanges energy, it disturbs the system as little as possible. For example, the demon could be added to a molecular dynamics simulation and provide an independent measure of the temperature.

15.5 The Ising Model

A popular model of a system of interacting variables is the *Ising* model. The model was proposed by Lenz and investigated by Ising, his graduate student, to study the phase transition from a paramagnet to a ferromagnet (cf. Brush). Ising calculated the thermodynamic properties of the model in one dimension and found that the model does not have a phase transition. However, for two and three dimensions the Ising model does exhibit a transition. The nature of the phase transition in two dimensions and some of the diverse applications of the Ising model are discussed in Section 15.7.

To introduce the Ising model, consider a lattice containing N sites and assume that each lattice site i has associated with it a number s_i, where $s_i = \pm 1$. The s_i are usually referred to as spins. The macroscopic properties of a system are determined by the nature of the accessible microstates. Hence, it is necessary to know the dependence of the energy on the configuration of spins. The total energy E of the Ising model is given by

$$E = -J \sum_{i,j=\mathrm{nn}(i)}^{N} s_i s_j - B \sum_{i=1}^{N} s_i \tag{15.8}$$

where B is proportional to the uniform external magnetic field. We will refer to B as the magnetic field, even though it includes a factor of μ. The first sum in (15.8) represents the energy of interaction of the spins and is over all nearest neighbor pairs. The *exchange constant* J is a measure of the strength of the interaction between nearest neighbor spins (see Figure 15.1). The second sum in (15.8) represents the energy of interaction between the magnetic moments of the spins and the external magnetic field.

If $J > 0$, then the states $\uparrow\uparrow$ and $\downarrow\downarrow$ are energetically favored in comparison to the states $\uparrow\downarrow$ and $\downarrow\uparrow$. Hence, for $J > 0$, we expect that the state of lowest total energy is *ferromagnetic*; that is, the spins all point in the same direction. If $J < 0$, the states $\uparrow\downarrow$ and $\downarrow\uparrow$ are favored and the state of lowest energy is expected to be *antiferromagnetic*, that is, alternate spins are aligned. If we subject the spins to an external magnetic field directed upward, the spins $\uparrow$ and $\downarrow$ possess an additional energy given by $-B$ and $+B$, respectively.

An important virtue of the Ising model is its simplicity. Some of its simplifying features are that the kinetic energy of the atoms associated with the lattice sites has been neglected, only nearest neighbor contributions to the interaction energy are included, and the spins are allowed to have only two discrete values. In spite of the simplicity of the model, we will find that the Ising model exhibits very interesting behavior.

Because we are interested in the properties of an infinite system, we have to choose appropriate boundary conditions. The simplest boundary condition in one dimension is to choose a free surface so that the spins at sites 1 and N each have one nearest neighbor interaction only. Usually a better choice is periodic boundary conditions. For this choice a one-dimensional lattice becomes a ring, and the spins at sites 1 and N interact with one another and, hence, have the same number of interactions as do the other spins.

What are some of the physical quantities whose averages we wish to compute? An obvious physical quantity is the *magnetization* M given by

$$M = \sum_{i=1}^{N} s_i, \tag{15.9}$$

and the magnetization per spin $m = M/N$. Usually we are interested in the average values $\langle M \rangle$ and the fluctuations $\langle M^2 \rangle - \langle M \rangle^2$.

For the familiar case of classical particles with continuously varying position and velocity coordinates, the dynamics is given by Newton's laws. For the Ising model the dependence (15.8) of the energy on the spin configuration is not sufficient to determine the time-dependent properties of the system. That is, the relation (15.8) does not tell us how the system changes from one configuration to another, and we have to introduce the dynamics separately. This dynamics will take the form of various Monte Carlo algorithms.

We first use the demon algorithm to sample configurations of the Ising model. The implementation of the demon algorithm is straightforward. We first choose a spin at random. The trial change corresponds to a flip of the spin from $\uparrow$ to $\downarrow$ or $\downarrow$ to $\uparrow$. We then compute the change in energy of the system and decide whether to accept or reject the trial change. We can determine the temperature T as a function of the energy of the system in two ways. One way is to measure the probability that the demon has energy E_d. Because we know that this probability is proportional to $\exp(-E_d/kT)$, we can determine T from a plot of the logarithm of the probability as a function of E_d. Another way to determine T is to measure the mean demon energy. However, because the possible values of E_d are not continuous for the Ising model, T is not simply proportional to $\langle E_d \rangle$ as it is for the ideal gas. We show in Appendix 15A that for $B = 0$ and the limit of an infinite system, the temperature is related to $\langle E_d \rangle$ by

$$kT/J = \frac{4}{\ln\left(1 + 4J/\langle E_d \rangle\right)}. \tag{15.10}$$

The result (15.10) comes from replacing the integrals in (15.7) by sums over the possible demon energies. Note that in the limit $|J/E_d| \ll 1$, (15.10) reduces to $kT = E_d$ as expected.

The `IsingDemon` class implements the Ising model in one dimension using periodic boundary conditions and the demon algorithm. Once the initial configuration is chosen, the demon algorithm is similar to that described in Section 15.3. However, the spins in the one-dimensional Ising model must be chosen at random. As usual, we will choose units such that $J = 1$.

Listing 15.3: The implementation of the demon algorithm for the one-dimensional Ising model.

```
package org.opensourcephysics.sip.ch15;
import java.awt.*;
import org.opensourcephysics.frames.*;

public class IsingDemon {
   public int[] demonEnergyDistribution;
   int N;               // number of spins
   public int systemEnergy;
   public int demonEnergy = 0;
   public int mcs = 0; // number of MC steps per spin
   public double systemEnergyAccumulator = 0;
   public double demonEnergyAccumulator = 0;
   public int magnetization = 0;
   public double
      mAccumulator = 0, m2Accumulator = 0;
   public int acceptedMoves = 0;
   private LatticeFrame lattice;

   public IsingDemon(LatticeFrame displayFrame) {
      lattice = displayFrame;
   }

   public void initialize(int N) {
      this.N = N;
      lattice.resizeLattice(N, 1); // set lattice size
      lattice.setIndexedColor(1, Color.red);
      lattice.setIndexedColor(-1, Color.green);
      demonEnergyDistribution = new int[N];
      for(int i = 0;i<N;++i) {
      // all spins up, second argument is always 0 for 1D lattice
         lattice.setValue(i, 0, 1);
      }
      int tries = 0;
      int E = -N; // start system in ground state
      magnetization = N; // all spins up
      // try up to 10*N times to flip spins so that system has desired energy
      while((E<systemEnergy)&&(tries<10*N)) {
         int k = (int) (N*Math.random());
         int dE = 2*lattice.getValue(k, 0)
                 *(lattice.getValue((k+1)%N, 0)+lattice.getValue((k-1+N)%N, 0));
         if(dE>0) {
            E += dE;
            int newSpin = -lattice.getValue(k, 0);
            lattice.setValue(k, 0, newSpin);
            magnetization += 2*newSpin;
         }
         tries++;
      }
      systemEnergy = E;
      resetData();
   }

   public double temperature() {
      return 4.0/Math.log(1.0+4.0/(demonEnergyAccumulator/(mcs*N)));
   }
   public void resetData() {
      mcs = 0;
      systemEnergyAccumulator = 0;
      demonEnergyAccumulator = 0;
      mAccumulator = 0;
      m2Accumulator = 0;
      acceptedMoves = 0;
   }

   public void doOneMCStep() {
      for(int j = 0;j<N;++j) {
         int i = (int) (N*Math.random());
         int dE = 2*lattice.getValue(i, 0)
                    *(lattice.getValue((i+1)%N, 0)
                   +lattice.getValue((i-1+N)%N, 0));;
         if(dE<=demonEnergy) {
            int newSpin = -lattice.getValue(i, 0);
            lattice.setValue(i, 0, newSpin);
            acceptedMoves++;
            systemEnergy += dE;
            demonEnergy -= dE;
            magnetization += 2*newSpin;
         }
         systemEnergyAccumulator += systemEnergy;
         demonEnergyAccumulator += demonEnergy;
         mAccumulator += magnetization;
         m2Accumulator += magnetization*magnetization;
         demonEnergyDistribution[demonEnergy]++;
      }
      mcs++;
   }
}
```

Note that for $B = 0$, the change in energy due to a spin flip is either 0 or $\pm 4J$. Hence, it is convenient to choose the initial energy of the system plus the demon to be an integer multiple of $4J$. Because the spins interact, it is difficult to choose an initial configuration of spins with precisely the desired energy. The procedure followed in method `initialize` is to begin with an initial configuration where all spins are up (a configuration of minimum energy) and then randomly flip spins while the energy is less than the desired initial energy.

Problem 15.5. The demon algorithm and the one-dimensional Ising model

(a) Write a target class to use with `IsingDemon` and simulate the one-dimensional Ising model. Choose $N = 100$ and the desired total energy, $E = -20$. Describe qualitatively how the configurations change with time. Then let $E = -100$ and describe any qualitative changes in the configurations.

(b) Compute the demon energy and the magnetization M as a function of the time. As usual, we interpret the time as the number of Monte Carlo steps per spin. What is the approximate time for these quantities to approach their equilibrium values?

(c) Compute the equilibrium values of $\langle E_d \rangle$ and $\langle M^2 \rangle$. About 100 mcs is sufficient for testing

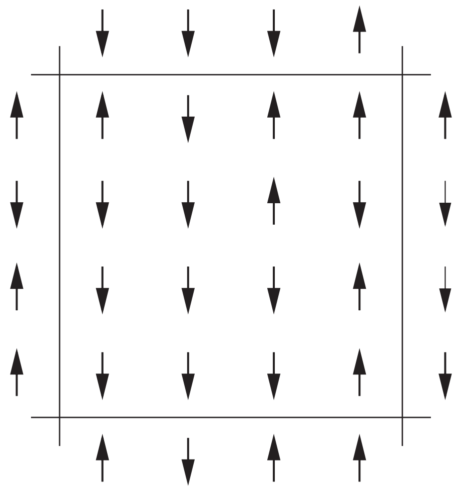


Figure 15.2: One of the 2^N possible configurations of a system of $N = 16$ Ising spins on a square lattice. Also shown are the spins in the four nearest periodic images of the central cell that are used to calculate the energy. An up spin is denoted by ↑ and a down spin is denoted by ↓. Note that the number of nearest neighbors on a square lattice is four. The energy of this configuration is $E = -8J + 4H$ with periodic boundary conditions.

the program and yields results of approximately 20% accuracy. To obtain better than 5% results, choose mcs ≥ 1000.

(d) Compute T for $N = 100$ and $E = -20, -40, -60$, and -80 from the inverse slope of $P(E_d)$ and the relation (15.10). Compare your results to the exact result for an infinite one-dimensional lattice, $E/N = -\tanh(J/kT)$. How do your computed results for E/N depend on N and on the number of Monte Carlo steps per spin? Does $\langle M^2\rangle$ increase or decrease with T?

(e)* Modify IsingDemon to include a nonzero magnetic field and compute $\langle E_d\rangle$, $\langle M\rangle$, and $\langle M^2\rangle$ as a function of B for fixed E. Read the discussion in Appendix 15A and determine the relation of $\langle E_d\rangle$ to T for your choices of B. Or determine T from the inverse slope of $P(E_d)$. Is the equilibrium temperature higher or lower than the $B = 0$ case for the same total energy? □

Problem 15.6. Antiferromagnetic case

Modify IsingDemon so that the antiferromagnetic case, $J = -1$, is treated. Before doing the simulation, describe how you expect the configurations to differ from the ferromagnetic case. What is the lowest energy or ground state configuration? Run the simulation with the spins initially in their ground state and compare your results with your expectations. Compute the mean energy per spin versus temperature and compare your results with the ferromagnetic case. □

***Problem 15.7.** The demon algorithm and the two-dimensional Ising model

(a) Simulate the Ising model on a square lattice using the demon algorithm. The total number of spins $N = L^2$, where L is the length of one side of the lattice. Use periodic boundary conditions as shown in Figure 15.2 so that spins in the left-hand column interact with spins in the right-hand column, etc. Do not include nonequilibrium configurations in your averages.

(b) Compute $\langle E_d \rangle$ and $\langle M^2 \rangle$ as a function of E for $B = 0$. Choose $L = 20$ and run for at least 500 mcs. Use (15.10) to determine the dependence of T on E and plot E versus T.

(c) Repeat the simulations in part (b) for $L = 20$. Run until your averages are accurate to within a few percent. Describe how the energy versus temperature changes with lattice size.

(d) Modify your program to make "snapshots" of the spin configurations. Describe the nature of the configurations at different energies or temperatures. Are they ordered or disordered? Are there domains of up or down spins?

(e) Instead of choosing a spin at random to make a trial change, choose the spins sequentially; that is, choose all the x values in ascending order for $y = 0$, then all the x values for $y = 1$, etc. This procedure updates a site and then immediately uses the new spin value when updating the neighbor. Because this process introduces a directional bias, vary the direction of the updates after each sweep. Do you obtain the same results as part (b)? □

One advantage of the demon algorithm is that it makes fewer demands on the random number generator than the Metropolis algorithm which we will discuss in Section 15.6. The demon algorithm also does not require computationally expensive calculations of the exponential function. Thus, for some systems the demon algorithm can be much faster than the Metropolis algorithm. In the one-dimensional Ising model we must choose the trial spins at random, but in higher dimensions, the spins can be chosen sequentially (see Problem 15.7e). In this case we can do a Monte Carlo simulation without random numbers! Very fast algorithms have been developed using one computer bit per spin and multiple demons (see Appendix 15B).

There are several disadvantages associated with the microcanonical ensemble. One disadvantage is the difficulty of establishing a system at the desired value of the energy. Another disadvantage is conceptual; that is, it is more natural to think of the behavior of macroscopic physical quantities as functions of the temperature rather than the total energy.

15.6 The Metropolis Algorithm

As we have mentioned, most physical systems of interest are not isolated, but exchange energy with their environment. If a system is placed in thermal contact with a heat bath at temperature T, the system reaches thermal equilibrium by exchanging energy with the heat bath until the system reaches the temperature of the heat bath. If we imagine a large number of copies of a system at fixed volume V and number of particles N in equilibrium at temperature T, then the probability P_s that the system is in microstate s with energy E_s is given by (15.4)

We can use the Boltzmann distribution (15.4) to obtain the ensemble average of the physical quantities of interest. For example, the mean energy is given by

$$\langle E \rangle = \sum_s E_s P_s = \frac{1}{Z} \sum_s E_s e^{-\beta E_s}. \tag{15.11}$$

Note that the energy fluctuates in the canonical ensemble.

How can we simulate a system of N particles confined in a volume V at a fixed temperature T? Because we can generate only a finite number m of the total number of M microstates, an estimate for the mean value of a physical quantity A would be given by

$$\langle A\rangle \approx A_m = \frac{\sum\limits_{s=1}^{m} A_s\, e^{-\beta E_s}}{\sum\limits_{s=1}^{m} e^{-\beta E_s}} \tag{15.12}$$

where A_s is the value of the physical quantity A in microstate s. A crude Monte Carlo procedure is to generate a microstate s at random, calculate E_s, A_s, and $e^{-\beta E_s}$, and evaluate the corresponding contribution of the microstate to the sums in (15.12). However, a microstate generated in this way would be very improbable and hence, contribute little to the sums. Instead, we use an *importance sampling* method and generate microstates according to the probability distribution function π_s, which we will choose in the following.

To introduce importance sampling, we rewrite (15.12) by multiplying and dividing by π_s:

$$A_m = \frac{\sum\limits_{s=1}^{m} (A_s/\pi_s)\, e^{-\beta E_s}\, \pi_s}{\sum\limits_{s=1}^{m} (1/\pi_s)\, e^{-\beta E_s}\, \pi_s} \qquad \text{(no importance sampling).} \tag{15.13}$$

If we generate the microstates (configurations) with probability π_s, then (15.13) becomes

$$A_m = \frac{\sum\limits_{s=1}^{m} (A_s/\pi_s)\, e^{-\beta E_s}}{\sum\limits_{s=1}^{m} (1/\pi_s)\, e^{-\beta E_s}} \qquad \text{(importance sampling).} \tag{15.14}$$

That is, if we average over a biased sample generated according to π_s, we need to weight each microstate by $1/\pi_s$ to eliminate the bias. Although any form of π_s could be used, the form of (15.14) suggests that a reasonable choice of π_s is the Boltzmann probability itself, that is,

$$\pi_s = \frac{e^{-\beta E_s}}{\sum\limits_{s=1}^{m} e^{-\beta E_s}}. \tag{15.15}$$

This choice of π_s implies that the estimate A_m of the mean value of A can be written as

$$A_m = \frac{1}{m}\sum_{s=1}^{m} A_s \tag{15.16}$$

where each state is sampled according to the Boltzmann distribution. The choice (15.15) for π_s is due to Metropolis et al.

Although we discussed the Metropolis sampling method in Section 11.7 in the context of the numerical evaluation of integrals, it is not necessary to read Section 11.7 to understand the Metropolis algorithm in the present context. The Metropolis algorithm can be summarized in the context of the simulation of a system of spins as follows. The extension to other types of systems is straightforward.

1. Establish an initial microstate. (The energy of the initial microstate is not important.)

2. Choose a spin at random and make a trial flip.
3. Compute $\Delta E \equiv E_{\text{trial}} - E_{\text{old}}$, the change in the energy of the system due to the trial flip.
4. If ΔE is less than or equal to zero, accept the new microstate and go to step 8.
5. If ΔE is positive, compute the quantity $w = e^{-\beta \Delta E}$.
6. Generate a uniform random number r in the unit interval $[0,1]$.
7. If $r \le w$, accept the new microstate; otherwise retain the previous microstate.
8. Determine the value of the desired physical quantities.
9. Repeat steps (2) through (8) to obtain a sufficient number of microstates.
10. Periodically compute averages over the microstates.

Steps (2) to (7) lead to a transition probability that the system moves from microstate $\{s_i\}$ to $\{s_j\}$ proportional to

$$W(i \to j) = \min\left(1, e^{-\beta \Delta E}\right) \qquad \text{(Metropolis algorithm)} \tag{15.17}$$

where $\Delta E = E_j - E_i$. Because it is necessary to evaluate only the ratio $P_j/P_i = e^{-\beta \Delta E}$, it is not necessary to normalize the probability. Note that because the microstates are generated with a probability proportional to the desired probability, all averages become arithmetic averages as in (15.16). However, because the constant of proportionally is not known, it is not possible to estimate the partition function Z in this way.

Although we chose π_s to be the Boltzmann distribution, other choices of π_s are possible and are useful in some contexts. In addition, the choice (15.17) of the transition probability is not the only one that leads to the Boltzmann distribution. It can be shown that if W satisfies the *detailed balance* condition

$$W(i \to j)\, e^{-\beta E_i} = W(j \to i)\, e^{-\beta E_j} \qquad \text{(detailed balance)}, \tag{15.18}$$

then the corresponding Monte Carlo algorithm generates a sequence of states distributed according to the Boltzmann distribution. The proof that the Metropolis algorithm generates states with a probability proportional to the Boltzmann probability distribution after a sufficient number of steps does not add much to our physical understanding of the algorithm. Instead, in Problems 15.8 and 15.9 we apply the algorithm to the ideal classical gas and to a classical magnet in a magnetic field, respectively, and verify that the Metropolis algorithm yields the Boltzmann distribution after a sufficient number of trial changes have been made.

Note that we have implicitly assumed in our discussion of the demon and Metropolis algorithms that the system is ergodic. That is, we have assumed that the important microstates of the system are being sampled with the desired probability. The existence of ergodicity depends on the way the trial moves are made and on the nature of the energy barriers between microstates. For example, consider a one-dimensional lattice of Ising spins with all spins up. If the spins are updated sequentially from right to left, then if one spin is flipped, all remaining flips would be accepted regardless of the temperature, because the change in energy would be zero. The system would not be ergodic for this implementation of the algorithm, and we would not obtain the correct thermodynamic behavior. A measure of the ergodicity of a system was discussed in Project 8.23.

We first consider the application of the Metropolis algorithm to an ideal classical gas in one dimension and verify that the Metropolis algorithm samples states according to the Boltzmann algorithm. The energy of an ideal gas depends only on the velocity of the particles, and hence a microstate is completely described by a specification of the velocity (or momentum) of each particle. Because the velocity is a continuous variable, it is necessary to describe the accessible microstates so that they are countable, and hence we place the velocity into bins. Suppose we have $N = 10$ particles and divide the possible values of the velocity into twenty bins. Then the total number of microstates would be 20^{10}. Not only would it be difficult to label these 20^{10} states, it would take a prohibitively long time to obtain an accurate estimate of their relative probabilities, and it would be difficult to verify directly that the Metropolis algorithm yields the Boltzmann distribution. For this reason we consider a single classical particle in one dimension in equilibrium with a heat bath and adopt the less ambitious goal of verifying that the Metropolis algorithm generates the Boltzmann distribution for this system.

The Metropolis algorithm is implemented in method `doStep` in class `BoltzmannApp`, and the velocity distribution is plotted. One quantity of interest is the probability $P(v)\Delta v$ that the particle has a velocity between v and $v+\Delta v$. We will choose the temperature to be large enough such that $\Delta v = 1$ provides a sufficiently small bin size to compute $P(v)$ accurately. As usual, we choose units such that the mass of the particle is unity.

Listing 15.4: The Metropolis algorithm for a single particle.

```
package org.opensourcephysics.sip.ch15;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.frames.HistogramFrame;

public class BoltzmannApp extends AbstractSimulation {
   double beta; // inverse temperature
   int mcs;
   int accepted;
   double velocity;
   HistogramFrame velocityDistribution =
      new HistogramFrame("v", "P(v)", "Velocity distribution");

   public void initialize() {
      velocityDistribution.clearData();
      beta = 1.0/control.getDouble("Temperature");
      velocity = control.getDouble("Initial velocity");
      accepted = 0;
      mcs = 0;
   }

   public void doStep() {
      double delta = control.getDouble("Maximum velocity change");
      mcs++;
      double ke = 0.5*velocity*velocity;
      double vTrial = velocity+delta*(2.0*Math.random()-1.0);
      double keTrial = 0.5*vTrial*vTrial;
      double dE = keTrial-ke;
      if((dE<0)||(Math.exp(-beta*dE)>Math.random())) {
         accepted++;
         ke = keTrial;
         velocity = vTrial;
      }
      velocityDistribution.append(velocity);
      control.clearMessages();
      control.println("mcs = "+mcs);
      control.println("acceptance probability = "+(double) (accepted)/mcs);
   }

   public void reset() {
      control.setValue("Maximum velocity change", 10.0);
      control.setValue("Temperature", 10.0);
      control.setValue("Initial velocity", 0.0);
      enableStepsPerDisplay(true);
   }

   public static void main(String[] args) {
      SimulationControl.createApp(new BoltzmannApp());
   }
}
```

Problem 15.8. Simulation of a particle in equilibrium with a heat bath

(a) Choose the temperature $T = 10$, the initial velocity equal to zero, and the maximum change in the particle's velocity to be $\delta = 10.0$. Run for a number of Monte Carlo steps until a plot of $\ln P(\mathbf{v})$ versus $\mathbf{v}$ is reasonably smooth. Describe the qualitative form of $P(\mathbf{v})$. (Remember that the velocity $\mathbf{v}$ can be either positive or negative.)

(b) Because the velocity of the particle characterizes the microstate of this single particle system, we need to plot $\ln P(E_s)$ versus $E_s = mv_s^2/2$ to test if the Metropolis algorithm yields the Boltzmann distribution in this case. (The two values of v, one positive and one negative, for each value of E, correspond to different microstates.) Add code to `BoltzmannApp` to compute $P(E_s)$ and determine the slope of $\ln P(E_s)$ versus E_s. The code for extracting information from the `HistogramFrame` class is given on page 206. Is this slope equal to $-\beta = -1/T$, where T is the temperature of the heat bath?.

(c) Add code to compute the mean energy and velocity. How do your results for the mean energy compare to the exact value? Explain why the computed mean particle velocity is approximately zero even though the initial particle velocity was not zero. To insure that your results do not depend on the initial conditions, let the initial velocity equal zero and recompute the mean energy and velocity. Do your equilibrium results differ from what you found previously?

(d) Add another `HistogramFrame` object to compute the probability $P(E)\Delta E$ where E is the energy of the configuration. Does $P(E)$ have the form of a Boltzmann distribution? If not, what is the functional form of $P(E)$?

(e) The *acceptance probability* is the fraction of trial moves that are accepted. What is the effect of changing the value of δ on the acceptance probability? □

Problem 15.9. Planar spin in an external magnetic field

(a) Consider a classical planar magnet with magnetic moment μ_0. The magnet can be oriented in any direction in the x-y plane, and the energy of interaction of the magnet with an external magnetic field $\mathbf{B}$ is $-\mu_0 B \cos\phi$, where ϕ is the angle between the moment and $\mathbf{B}$. Write a

Monte Carlo program to sample the microstates of this system in thermal equilibrium with a heat bath at temperature T. Compute the mean energy as a function of the ratio $\beta\mu_0 B$.

(b) Compute the probability density $P(\phi)$ and analyze its dependence on the energy. □

In Problem 15.10 we consider the Monte Carlo simulation of a classical ideal gas of N particles in equilibrium with a heat bath. It is convenient to say that one time unit or one Monte Carlo step per particle (mcs) has elapsed after N particles have had a chance to change their coordinates. If the particles are chosen at random, then during one Monte Carlo step per particle, some particles might not be chosen, but all particles will be chosen equally on the average. The advantage of this definition is that the time is independent of the number of particles. However, this definition of time has no obvious relation to a physical time.

Problem 15.10. Simulation of an ideal gas in one dimension

(a) Modify class `BoltzmannApp` to simulate an ideal gas of N particles in one dimension. For simplicity, assume that all particles have the same initial velocity of 10. Let $N = 20$ and $T = 10$ and consider at least 2000 Monte Carlo steps per particle. Choose the value of δ so that the acceptance probability is approximately 40%. What are the mean kinetic energy and mean velocity of the particles?

(b) We might expect the total energy of an ideal gas to remain constant because the particles do not interact with one another and, hence, cannot exchange energy directly. What is the value of the initial total energy of the system in part (a)? Does the total energy remain constant? If not, explain how the energy changes.

(c) What is the nature of the time dependence of the total energy starting from the initial condition in (a)? Estimate the number of Monte Carlo steps per particle necessary for the system to reach thermal equilibrium by computing a moving average of the total energy over a fixed time interval. Does this average change with time after a sufficient time has elapsed? What choice of the initial velocities allows the system to reach thermal equilibrium at temperature T as quickly as possible?

(d) Compute the probability $P(E)\Delta E$ for the system of N particles to have a total energy between E and $E+\Delta E$. Plot $P(E)$ as a function of E and describe the qualitative behavior of $P(E)$. Does $P(E)$ have the form of the Boltzmann distribution? If not, describe the qualitative features of $P(E)$ and determine its functional form.

(e) Compute the mean energy for $T = 10$, 20, 40, 80, and 120 and estimate the heat capacity from its definition $C = \partial E/\partial T$.

(f) Compute the mean square energy fluctuations $\langle(\Delta E)^2\rangle = \langle E^2\rangle - \langle E\rangle^2$ for $T = 10$ and $T = 40$. Compare the magnitude of the ratio $\langle(\Delta E)^2\rangle/T^2$ with the heat capacity determined in part (e). □

You might have been surprised to find in Problem 15.10d that the form of $P(E)$ is a Gaussian centered about the mean energy of the system. What is the relation of this form of $P(E)$ to the central limit theorem (see Problem 7.15)? If the microstates are distributed according to the Boltzmann probability, why is the total energy distributed according to the Gaussian distribution?

15.7 Simulation of the Ising Model

You are probably familiar with ferromagnetic materials, such as iron and nickel, which exhibit a spontaneous magnetization in the absence of an applied magnetic field. This nonzero magnetization occurs only if the temperature is less than a well-defined temperature known as the Curie or critical temperature T_c. For temperatures $T > T_c$, the magnetization vanishes. Hence, T_c separates the disordered phase for $T > T_c$ from the ferromagnetic phase for $T < T_c$.

The origin of magnetism is quantum mechanical in nature and its study is of much experimental and theoretical interest. The study of simple classical models of magnetism has provided much insight. The two- and three-dimensional Ising model is the most commonly studied classical model and is particularly useful in the neighborhood of the magnetic phase transition.

The thermal quantities of interest for the Ising model include the mean energy $\langle E\rangle$ and the heat capacity C. One way to determine C at constant external magnetic field is from its definition $C = \partial\langle E\rangle/\partial T$. An alternative way is to relate C to the statistical fluctuations of the total energy in the canonical ensemble (see Appendix 15B):

$$C = \frac{1}{kT^2}\Big(\langle E^2\rangle - \langle E\rangle^2\Big) \qquad \text{(canonical ensemble).} \tag{15.19}$$

Another quantity of interest is the mean magnetization $\langle M\rangle$ and the corresponding zero field magnetic susceptibility:

$$\chi = \frac{\partial\langle M\rangle}{\partial B}\Big|_{B=0}. \tag{15.20}$$

The zero field magnetic susceptibility χ is an example of a linear response function, because it measures the ability of a spin to respond to a change in the external magnetic field. In analogy to the heat capacity, χ is related to the fluctuations of the magnetization (see Appendix 15C):

$$\chi = \frac{1}{kT}\Big(\langle M^2\rangle - \langle M\rangle^2\Big) \tag{15.21}$$

where $\langle M\rangle$ and $\langle M^2\rangle$ are evaluated in zero external magnetic field. The relations (15.19) and (15.21) are examples of the general relation between linear response functions and equilibrium fluctuations.

The Metropolis algorithm was stated in Section 15.6 as a method for generating states with the desired Boltzmann probability, but the flipping of single spins can also be interpreted as a reasonable approximation to the real dynamics of an anisotropic magnet whose spins are coupled to the vibrations of the lattice. The coupling leads to random spin flips, and we expect that one Monte Carlo step per spin is proportional to the average time between single spin flips observed in the laboratory. Hence, we can regard single spin flips as a time dependent process and observe the relaxation to equilibrium. In the following, we will frequently refer to the application of the Metropolis algorithm to the Ising model as *single spin flip* dynamics.

In Problem 15.11 we use the Metropolis algorithm to simulate the one-dimensional Ising model. Note that the parameters J and kT do not appear separately, but appear together in the dimensionless ratio J/kT. Unless otherwise stated, we measure temperature in units of J/k and set $B = 0$.

Problem 15.11. One-dimensional Ising model

(a) Write a program to simulate the one-dimensional Ising model in equilibrium with a heat bath. Modify method `doOneMCStep` in `IsingDemon` (see class `IsingDemon` on page 591 or class `Ising` on page 602). Use periodic boundary conditions. Assume that the external magnetic field is zero. Draw the microscopic state (configuration) of the system after each Monte Carlo step per spin.

(b) Choose $N = 20$ and $T = 1$ and start with all spins up. What is the initial effective temperature of the system? Run for at least 1000 mcs, where mcs is the number of Monte Carlo steps per spin. Visually inspect the configuration of the system after each Monte Carlo step per spin and estimate the time it takes for the system to reach equilibrium. Does the sign of the magnetization change during the simulation? Increase N and estimate the time for the system to reach equilibrium and for the magnetization to change sign.

(c) Change the initial condition so that the orientation of each spin is chosen at random. What is the initial effective temperature of the system in this case? Estimate the time it takes for the system to reach equilibrium.

(d) Choose $N = 50$ and determine $\langle E\rangle$, $\langle E^2\rangle$, and $\langle M^2\rangle$ as a function of T in the range $0.1 \le T \le 5$. Plot $\langle E\rangle$ as a function of T and discuss its qualitative features. Compare your computed results for $\langle E\rangle$ to the exact result (for $B = 0$):

$$E(T) = -N \tanh \beta J. \tag{15.22}$$

Use the relation (15.19) to determine the T- dependence of C.

(e) As you probably noticed in part (b), the system can overturn completely during a long run and thus the value of $\langle M\rangle$ can vary widely from run to run. Because $\langle M\rangle = 0$ for $T > 0$ for the one-dimensional Ising model, it is better to assume $\langle M\rangle = 0$ and compute χ from the relation $\chi = \langle M^2\rangle/kT$. Use this relation (15.21) to estimate the T-dependence of χ.

(f) One of the best laboratory realizations of a one-dimensional Ising ferromagnet is a chain of bichloride-bridged Fe^{2+} ions known as FeTAC (see Greeney et al.). Measurements of χ yield a value of the exchange interaction J given by $J/k = 17.4$ K. Note that experimental values of J are typically given in temperature units. Use this value of J to plot your Monte Carlo results for χ versus T with T given in Kelvin. At what temperature is χ a maximum for FeTAC?

(g) Is the acceptance probability an increasing or decreasing function of T? Does the Metropolis algorithm become more or less efficient as the temperature is lowered?

(h) Compute the probability $P(E)$ for a system of $N = 50$ spins at $T = 1$. Run for at least 1000 mcs. Plot $\ln P(E)$ versus $(E - \langle E\rangle)^2$ and discuss its qualitative features. □

We next apply the Metropolis algorithm to the Ising model on the square lattice. The `Ising` class is listed in the following.

Listing 15.5: The `Ising` class.

```
package org.opensourcephysics.sip.ch15;
import java.awt.*;
import org.opensourcephysics.frames.*;
```

```java
public class Ising {
   public static final double criticalTemperature =
       2.0/Math.log(1.0+Math.sqrt(2.0));
   public int L = 32;
   public int N = L*L;                  // number of spins
   public double temperature = criticalTemperature;
   public int mcs = 0;                  // number of MC moves per spin
   public int energy;
   public double energyAccumulator = 0;
   public double energySquaredAccumulator = 0;
   public int magnetization = 0;
   public double magnetizationAccumulator = 0;
   public double magnetizationSquaredAccumulator = 0;
   public int acceptedMoves = 0;
   private double[] w = new double[9]; // array to hold Boltzmann factors
   public LatticeFrame lattice;

   public void initialize(int L, LatticeFrame displayFrame) {
      lattice = displayFrame;
      this.L = L;
      N = L*L;
      lattice.resizeLattice(L, L); // set lattice size
      lattice.setIndexedColor(1, Color.red);
      lattice.setIndexedColor(-1, Color.green);
      for(int i = 0;i<L;++i) {
         for(int j = 0;j<L;++j) {
            lattice.setValue(i, j, 1); // all spins up
         }
      }
      magnetization = N;
      energy = -2*N; // minimum energy
      resetData();
      // other array elements never occur for H = 0
      w[8] = Math.exp(-8.0/temperature);
      w[4] = Math.exp(-4.0/temperature);
   }

// allow temperature to be changed in the middle of a simulation
public void changeTemperature(double newTemperature) {
   temperature = newTemperature;
   w[8] = Math.exp(-8.0/temperature);
   w[4] = Math.exp(-4.0/temperature);
}

   public double specificHeat() {
      double energySquaredAverage = energySquaredAccumulator/mcs;
      double energyAverage = energyAccumulator/mcs;
      double heatCapacity = energySquaredAverage-energyAverage*energyAverage;
      heatCapacity = heatCapacity/(temperature*temperature);
      return(heatCapacity/N);
   }
   public double susceptibility() {
      double magnetizationSquaredAverage = magnetizationSquaredAccumulator/mcs;
      double magnetizationAverage = magnetizationAccumulator/mcs;
      return(magnetizationSquaredAverage-
          Math.pow(magnetizationAverage, 2))/(temperature*N);
   }

   public void resetData() {
      mcs = 0;
      energyAccumulator = 0;
      energySquaredAccumulator = 0;
      magnetizationAccumulator = 0;
      magnetizationSquaredAccumulator = 0;
      acceptedMoves = 0;
   }

   public void doOneMCStep() {
      for(int k = 0;k<N;++k) {
         int i = (int) (Math.random()*L);
         int j = (int) (Math.random()*L);
         int dE = 2*lattice.getValue(i, j)
                  *(lattice.getValue((i+1)%L, j)
                  +lattice.getValue((i-1+L)%L, j)
                  +lattice.getValue(i, (j+1)%L)
                  +lattice.getValue(i, (j-1+L)%L));
         if((dE<=0)||(w[dE]>Math.random())) {
            int newSpin = -lattice.getValue(i, j);
            lattice.setValue(i, j, newSpin);
            acceptedMoves++;
            energy += dE;
            magnetization += 2*newSpin;
         }
      }
      energyAccumulator += energy;
      energySquaredAccumulator += energy*energy;
      magnetizationAccumulator += magnetization;
      magnetizationSquaredAccumulator += magnetization*magnetization;
      mcs++;
   }
}
```

One of the most time consuming parts of the Metropolis algorithm is the calculation of the exponential function $e^{-\beta\Delta E}$. Because there are only a small number of possible values of $\beta\Delta E$ for the Ising model (see Figure 15.11), we store the small number of different probabilities for the spin flips in the array `w`. The values of this array are computed in method `initialize`.

To implement the Metropolis algorithm, we determine the change in the energy ΔE and then accept the trial flip if $\Delta E \leq 0$. If this condition is not satisfied, we generate a random number in the unit interval and compare it to $e^{-\beta\Delta E}$. We can use a single if statement for these two conditions, because in Java (and C/C++) the second condition of an `||` (or) statement is evaluated only if the first is false. This feature is very useful because we do not want to generate random numbers when they are not needed, as is the case for $\Delta E \leq 0$. (The same feature holds for the compound `& &` (and) statement for which the second condition is only evaluated if the first is true.)

A typical laboratory system has at least 10^{18} spins. In contrast, the number of spins that can be simulated typically ranges from 10^3 to 10^9. As we have discussed in other contexts, the use of periodic boundary conditions minimizes finite size effects. However, more sophisticated boundary conditions are sometimes convenient. For example, we can give the surface spins extra neighbors, whose direction is related to the mean magnetization of the microstate (see Saslow).

In class `Ising` data for the values of the physical observables are accumulated after each Monte Carlo step per spin. The optimum time for sampling various physical quantities is explored in Problem 15.13. Note that if a flip is rejected, the old configuration is retained. Thermal equilibrium is not described properly unless the old configuration is again included in computing the averages.

Achieving thermal equilibrium can account for a substantial fraction of the total run time for very large systems. The most practical choice of initial conditions in these cases is a configuration from a previous run that is at a temperature close to the desired temperature. The code for reading and saving configurations can be found in Appendix 8A.

Problem 15.12. Equilibration of the two-dimensional Ising model

(a) Write a target class that uses class `Ising` and plots the magnetization and energy as a function of the number of Monte Carlo steps. Your program should also display the mean magnetization, the energy, the specific heat, the susceptibility, and the acceptance probability when the simulation is stopped. Averages such as the mean energy and the susceptibility should be normalized by the number of spins so that it is easy to compare systems with different values of N. Choose the linear dimension $L = 32$ and the heat bath temperature $T = 2$. Estimate the time needed to equilibrate the system given that all the spins are initially up.

(b) Visually determine if the spin configurations are "ordered" or "disordered" at $T = 2$ after equilibrium has been established.

(c) Repeat part (a) with the initial direction of each spin chosen at random. Make sure you explicitly compute the initial energy and magnetization in `initialize`. Does the equilibration time increase or decrease?

(d) Repeat parts (a)–(c) for $T = 2.5$. □

Problem 15.13. Comparison with exact results

In general, a Monte Carlo simulation yields exact answers only after an infinite number of configurations have been sampled. How then can we be sure that our program works correctly, and our results are statistically meaningful? One way is to reproduce exact results in known limits. In the following, we test class `Ising` by considering a small system for which the mean energy and magnetization can be calculated analytically.

(a) Calculate analytically the T-dependence of E, M, C, and χ for the Ising model on the square lattice with $L = 2$. (A summary of the calculation is given in Appendix 15C.) For simplicity, we have omitted the brackets denoting the thermal averages.)

(b) Simulate the Ising model with $L = 2$ and estimate E, M, C, and χ for $T = 0.5$ and 0.25. Use the relations (15.19) to compute C. Compare your estimated values to the exact results found in part (a). Approximately how many Monte Carlo steps per spin are necessary to obtain E and M to within 1%? How many Monte Carlo steps per spin are necessary to obtain C to within 1%?

(c) Choose $L = 4$ and the direction of each spin at random and equilibrate the system at $T = 3$. Look at the time series of M and E after every Monte Carlo step per spin and estimate how often M changes sign. Does E change sign when M changes sign? How often does M change sign for $L = 8$ and $L = 32$ (and $T = 3$)? Although the direction of the spins is initially chosen at random, it is likely that the number of up spins will not exactly cancel the number of down spins. Is that statement consistent with your observations? If the net number of spins is up, how long does the net magnetization remain positive for a given value of L?

(d) The calculation of χ is more complicated because the sign of M can change during the simulation for smaller values of L. Compare your results for χ from using (15.21) and from using (15.21) with $\langle M\rangle$ replaced by $\langle|M|\rangle$. Which way of computing χ gives more accurate results? □

Now that you have checked your program and obtained typical equilibrium configurations, we consider in more detail the calculation of the mean values of the physical quantities of interest. Suppose we wish to compute the mean value of the physical quantity A. In some cases, the calculation of A for a given configuration is time consuming, and we do not want to compute its value more often than necessary. For example, we would not compute A after the flip of only one spin because the values of A in the two configurations would almost be the same. Ideally, we wish to compute A for configurations that are statistically independent. Because we do not know *a priori* the mean number of spin flips needed to obtain configurations that are statistically independent, it is a good idea to estimate this time in your preliminary calculations.

One way to estimate the time interval over which configurations are correlated is to compute the time displaced *autocorrelation* function $C_A(t)$ which is defined as

$$C_A(t) = \frac{\langle A(t+t_0)A(t_0)\rangle - \langle A\rangle^2}{\langle A^2\rangle - \langle A\rangle^2} \tag{15.23}$$

where $A(t)$ is the value of the quantity A at time t. The averages in (15.23) are over all possible time origins t_0. Because the choice of the time origin is arbitrary for an equilibrium system, C_A depends only on the time difference t rather than t and t_0 separately. For sufficiently large t, $A(t)$ and $A(0)$ will become uncorrelated, and hence $\langle A(t+t_0)A(t_0)\rangle \to \langle A(t+t_0)\rangle\langle A(t_0)\rangle = \langle A\rangle^2$. Hence $C_A(t) \to 0$ as $t \to \infty$. Also, $C_A(t=0)$ is normalized to unity. In general, $C_A(t)$ will decay exponentially with t with a decay or correlation time τ_A whose magnitude depends on the choice of the physical quantity A as well as the physical parameters of the system, for example, the temperature.

The time dependence of the two most common correlation functions $C_M(t)$ and $C_E(t)$ is investigated in Problem 15.14. As an example of the calculation of $C_E(t)$, consider the equilibrium time series for E for the $L = 4$ Ising model on the square lattice at $T = 4$: −4, −8, 0, −8, −20, −4, 0, 0, −24, −32, −24, −24, −8, −8, −16, −12. The averages of E and E^2 over these sixteen values are $\langle E\rangle = -12$, $\langle E^2\rangle = 240$, and $\langle E^2\rangle - \langle E\rangle^2 = 96$. We wish to compute $E(t)E(0)$ for all possible choices of the time origin. For example, $E(4)E(0)$ is given by

$$\begin{aligned}\langle E(4)E(0)\rangle = {} & \frac{1}{12}\Big[(-20\times -4) + (-4\times -8) + (0\times 0) \\ & + (0\times -8) + (-24\times -20) + (-32\times -4) \\ & + (-24\times 0) + (-24\times 0) + (-8\times -24) \\ & + (-8\times -32) + (-16\times -24) + (-12\times -24)\Big].\end{aligned} \tag{15.24}$$

We averaged over the twelve possible choices of the origin for the time difference $t = 4$. Verify that $\langle E(4)E(0)\rangle = 460/3$ and $C_E(4) = 7/72$.

To implement this procedure on a computer, we could store the time series in memory, if it is not too long, or save it in a data file. You can save the data for $M(t)$ and $E(t)$ by pressing the Save XML menu item under the File menu on the frame containing the plots for $M(t)$ and $E(t)$. The class IsingAutoCorrelatorApp in Listing 15.6 reads in data created by the IsingApp class. Method computeCorrelation computes the mean and mean square of the magnetization and the energy, which are needed to compute C_M and C_E as defined in (15.23). Then it computes the time displaced autocorrelation for all possible choices of t_0.

Listing 15.6: Listing of class for computing autocorrelation function of M and E.

```
package org.opensourcephysics.sip.ch15;
import java.util.*;
import javax.swing.*;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.display.*;
import org.opensourcephysics.frames.*;

public class IsingAutoCorrelatorApp extends AbstractCalculation {
   PlotFrame plotFrame =
   new PlotFrame("tau", "<E(t+tau)E(t)> and <M(t+tau)M(t)>",
                                  "Time correlations");
   double[] energy = new double[0], magnetization = new double[0];
   int numberOfPoints;

   public void calculate() {
      computeCorrelation(control.getInt("Maximum time interval, tau"));
   }

   public void readXMLData() {
      energy = new double[0];
      magnetization = new double[0];
      numberOfPoints = 0;
      String filename = "ising_data.xml";
      JFileChooser chooser = OSPFrame.getChooser();
      int result = chooser.showOpenDialog(null);
      if(result==JFileChooser.APPROVE_OPTION) {
         filename = chooser.getSelectedFile().getAbsolutePath();
      } else {
         return;
      }
      XMLControlElement xmlControl = new XMLControlElement(filename);
      if(xmlControl.failedToRead()) {
         control.println("failed to read: "+filename);
      } else {
         // gets the datasets in the xml file
         Iterator it =
            xmlControl.getObjects(Dataset.class, false).iterator();
         while(it.hasNext()) {
            Dataset dataset = (Dataset) it.next();
            if(dataset.getName().equals("magnetization")) {
               magnetization = dataset.getYPoints();
            }
            if(dataset.getName().equals("energy")) {
               energy = dataset.getYPoints();
            }
         }
         numberOfPoints = magnetization.length;
         control.println("Reading: "+filename);
         control.println("Number of points = "+numberOfPoints);
      }
      calculate();
      plotFrame.repaint();
   }

   public void computeCorrelation(int tauMax) {
      plotFrame.clearData();
      double
         energyAccumulator = 0, magnetizationAccumulator = 0;
      double
         energySquaredAccumulator = 0, magnetizationSquaredAccumulator = 0;
      for(int t = 0;t<numberOfPoints;t++) {
         energyAccumulator += energy[t];
         magnetizationAccumulator += magnetization[t];
         energySquaredAccumulator += energy[t]*energy[t];
         magnetizationSquaredAccumulator += magnetization[t]*magnetization[t];
      }
      double averageEnergySquared =
         Math.pow(energyAccumulator/numberOfPoints, 2);
      double averageMagnetizationSquared =
         Math.pow(magnetizationAccumulator/numberOfPoints, 2);
      // compute normalization factors
      double normE =
         (energySquaredAccumulator/numberOfPoints)-averageEnergySquared;
      double normM =
         (magnetizationSquaredAccumulator/numberOfPoints)-averageMagnetizationSquared;
      for(int tau = 1;tau<=tauMax;tau++) {
         double c_MAccumulator = 0;
         double c_EAccumulator = 0;
         int counter = 0;
         for(int t = 0;t<numberOfPoints-tau;t++) {
            c_MAccumulator += magnetization[t]*magnetization[t+tau];
            c_EAccumulator += energy[t]*energy[t+tau];
            counter++;
         }
         // correlation function defined so that c(0) = 1 and c(infinity) -> 0
         plotFrame.append(0, tau, ((c_MAccumulator/counter)-
            averageMagnetizationSquared)/normM);
         plotFrame.append(1, tau, ((c_EAccumulator/counter)-
            averageEnergySquared)/normE);
      }
      plotFrame.setVisible(true);
   }

   public void reset() {
      control.setValue("Maximum time interval, tau", 20);
      readXMLData();
   }

   public static void main(String args[]) {
      CalculationControl.createApp(new IsingAutoCorrelatorApp());
   }
}
```

Problem 15.14. Correlation times

(a) As a check on IsingAutoCorrelatorApp, use the time series for E given in the text to do a hand calculation of $C_E(t)$ in the way that it is computed in the computeCorrelation method.

(b) Use class IsingAutoCorrelatorApp to compute the equilibrium values of $C_M(t)$ and $C_E(t)$. Save the values of the magnetization and energy only after the system has reached equilibrium. Estimate the correlation times from the energy and the magnetization correlation functions for $L = 8$, and $T = 3$, $T = 2.3$, and $T = 2$. One way to determine τ is to fit $C(t)$ to the exponential form $C(t) \sim e^{-t/\tau}$. Another way is to define the integrated correlation time as

$$\tau = \sum_{t=1} C(t). \tag{15.25}$$

The sum is cut off at the first negative value of $C(t)$. Are the negative values of $C(t)$ physically meaningful? How does the behavior of $C(t)$ change if you average your results over longer runs? How do your estimates for the correlation times compare with your estimates of the relaxation time found in Problem 15.12? Why would the term "decorrelation time" be more appropriate than "correlation time?" Are the correlation times τ_M and τ_E comparable?

(c) To simulate the relaxation to equilibrium as realistically as possible, we have randomly selected the spins to be flipped. However, if we are interested only in equilibrium properties, it might be possible to save computer time by selecting the spins sequentially. Determine if the correlation time is greater, smaller, or approximately the same if the spins are chosen sequentially rather than randomly. If the correlation time is greater, does it still save CPU time to choose spins sequentially? Why is it not desirable to choose spins sequentially in the one-dimensional Ising model? □

How can we quantify the accuracy of our measurements, for example, the accuracy of the estimated mean energy? As discussed in Chapter 11, the usual measure of the accuracy is the standard deviation of the mean. If we make n measurements of E, then the most probable error in $\langle E\rangle$ is given by

$$\sigma_m = \frac{\sigma}{\sqrt{n}} \tag{15.26}$$

where the standard deviation σ is defined as

$$\sigma^2 = \langle E^2\rangle - \langle E\rangle^2. \tag{15.27}$$

The difficulty is that, in general, our measurements of the time series E_i are not independent, but are correlated. Hence, σ_m as given by (15.26) is an underestimate of the actual error.

*__Problem 15.15.__ Estimate of errors

One way to determine whether the measurements are independent is to compute the correlation time. Another way is based on the idea that the magnitude of the error should not depend on how we group the data (see Section 11.4). For example, suppose that we group every two data points to form $n/2$ new data points $E_i^{(2)}$ given by $E_i^{(g=2)} = (1/2)[E_{2i-1} + E_{2i}]$. If we replace n by $n/2$ and E by $E^{(2)}$ in (15.26) and (15.27), we would find the same value of σ_m as before, provided that the original E_i are independent. If the computed σ_m is not the same, we continue this averaging process until σ_m calculated from

$$E_i^{(g)} = \frac{1}{2}\Big[E_{2i-1}^{(g/2)} + E_{2i}^{(g/2)}\Big] \qquad (g = 2, 4, 8, \ldots), \tag{15.28}$$

is approximately the same as that calculated from $E^{(g/2)}$.

(a) Use this averaging method to estimate the errors in your measurements of $\langle E\rangle$ and $\langle M\rangle$. Choose $L = 8$, $T = T_c = 2/\ln(1+\sqrt{2}) \approx 2.269$, and mcs ≥ 16384 and calculate averages after every Monte Carlo step per spin after the system has equilibrated. (The significance of T_c will be explored in Section 15.8.) A rough measure of the correlation time is the number of terms in the time series that need to be averaged for σ_m to be approximately unchanged. What is the qualitative dependence of the correlation time on $T - T_c$?

(b) Repeat for $L = 16$. Do you need more Monte Carlo steps than in part (a) to obtain statistically independent data? If so, why?

(c) The exact value of E/N for the Ising model on a square lattice with $L = 16$ and $T = T_c = 2/\ln(1+\sqrt{2})$ is given by $E/N = -1.45306$ (to five decimal places). The exact result for E/N allows us to determine the actual error in this case. Compute $\langle E\rangle$ by averaging E after each Monte Carlo step per spin for mcs $\geq 10^6$. Compare your actual error to the estimated error given by (15.26) and (15.27) and discuss their relative values. □

15.8 The Ising Phase Transition

Now that we have tested our program for the two-dimensional Ising model, we explore some of its properties.

Problem 15.16. Qualitative behavior of the two-dimensional Ising model

(a) Use class `Ising` and your version of `IsingApp` to compute the mean magnetization, the mean energy, the heat capacity, and the susceptibility. Because we will consider the Ising model for different values of L, it will be convenient to convert these quantities to intensive quantities such as the mean energy per spin, the specific heat (per spin), and the susceptibility per spin. For simplicity, we will use the same notation for both the extensive and the corresponding intensive quantities. Choose $L = 4$ and consider T in the range $1.5 \leq T \leq 3.5$ in steps of $\Delta T = 0.2$. Choose the initial condition at $T = 3.5$ such that the orientation of the spins is chosen at random. Because all the spins might overturn and the magnetization would change sign during the course of your observation, estimate the mean value of $|M|$ in addition to that of M. The susceptibility should be calculated as

$$\chi = \frac{1}{kT}[\langle M^2\rangle - \langle |M|\rangle^2]. \tag{15.29}$$

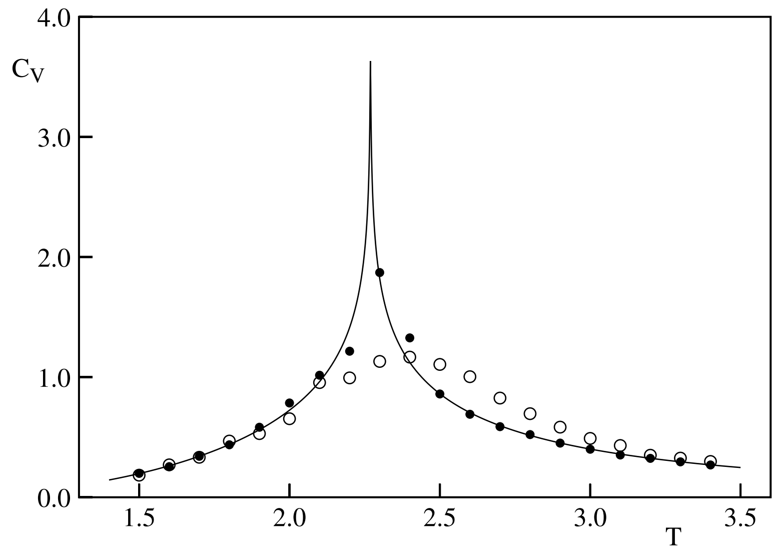


Figure 15.3: The temperature dependence of the specific heat C (per spin) of the Ising model on a square lattice with periodic boundary conditions for $L = 8$ and $L = 16$. One thousand Monte Carlo steps per spin were used for each value of the temperature. The continuous line represents the temperature dependence of C in the limit of an infinite lattice. (Note that C is infinite at $T = T_c$ for an infinite lattice.)

Use at least 1000 Monte Carlo steps per spin and estimate the number of equilibrium configurations needed to obtain $\langle M\rangle$ and $\langle E\rangle$ to 5% accuracy. Plot $\langle E\rangle$, m, $|m|$, C, and χ as a function of T and describe their qualitative behavior. Do you see any evidence of a phase transition?

(b) Repeat the calculations of part (a) for $L = 8$ and $L = 16$. Plot $\langle E\rangle$, m, $|m|$, C, and χ as a function of T and describe their qualitative behavior. Is the evidence of a phase transition more obvious?

(c) The correlation length ξ can be obtained from the r-dependence of the spin correlation function $c(r)$. The latter is defined as

$$c(r) = \langle s_i s_j\rangle - m^2 \tag{15.30}$$

where r is the distance between sites i and j. The system is translationally invariant so we write $\langle s_i\rangle = \langle s_j\rangle = m$. The average is over all sites for a given configuration and over many configurations. Because the spins are not correlated for large r, $c(r) \to 0$ in this limit. Assume that $c(r) \sim e^{-r/\xi}$ for r sufficiently large and estimate ξ as a function of T. How does your estimate of ξ compare with the size of the domains of spins with the same orientation? □

Our studies of phase transitions are limited by the relatively small system sizes we can simulate. Nevertheless, we observed in Problem 15.16 that even systems as small as $L = 4$ exhibit behavior that is reminiscent of a phase transition. In Figure 15.3 we show our Monte Carlo data for the T-dependence of the specific heat of the two-dimensional Ising model for

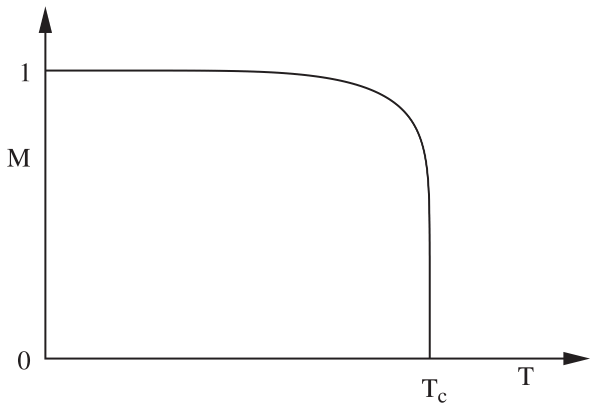


Figure 15.4: The temperature dependence of $m(T)$, the mean magnetization per spin, for the Ising model in two dimensions in the thermodynamic limit.

$L = 8$ and $L = 16$. We see that C exhibits a broad maximum which becomes sharper for larger L. Does your data for C exhibit similar behavior?

We next summarize some of the qualitative properties of ferromagnetic systems in zero magnetic field in the thermodynamic limit ($N \to \infty$). At $T = 0$, the spins are perfectly aligned in either direction; that is, the mean magnetization per spin $m(T) = \langle M(T)\rangle/N$ is given by $m(T = 0) = \pm 1$. As T is increased, the magnitude of $m(T)$ decreases continuously until $T = T_c$ at which $m(T)$ vanishes (see Figure 15.4). Because $m(T)$ vanishes continuously rather than abruptly, the transition is termed *continuous* rather than discontinuous. (The term *first order* describes a discontinuous transition.)

How can we characterize a continuous magnetic phase transition? Because a nonzero m implies that a net number of spins are spontaneously aligned, we designate m as the *order parameter* of the system. Near T_c, we can characterize the behavior of many physical quantities by power law behavior just as we characterized the percolation threshold (see Table 12.1). For example, we can write m near T_c as

$$m(T) \sim (T_c - T)^{\beta} \tag{15.31}$$

where β is a *critical* exponent (not to be confused with the inverse temperature). Various thermodynamic derivatives such as the susceptibility and specific heat diverge at T_c and are characterized by critical exponents. We write

$$\chi \sim |T - T_c|^{-\gamma}, \tag{15.32}$$

and

$$C \sim |T - T_c|^{-\alpha} \tag{15.33}$$

where we have introduced the critical exponents γ and α. We have assumed that χ and C are characterized by the same critical exponents above and below T_c.

Another measure of the magnetic fluctuations is the linear dimension $\xi(T)$ of a typical magnetic domain. We expect the *correlation length* $\xi(T)$ to be the order of a lattice spacing for $T \gg T_c$. Because the alignment of the spins becomes more correlated as T approaches T_c from above, $\xi(T)$ increases as T approaches T_c. We can characterize the divergent behavior of $\xi(T)$ near T_c by the critical exponent ν:

$$\xi(T) \sim |T - T_c|^{-\nu}. \tag{15.34}$$

As we found in our discussion of percolation in Chapter 12, a finite system cannot exhibit a true phase transition. We expect that if $\xi(T)$ is less than the linear dimension L of the system, our simulations will yield results comparable to an infinite system. In contrast, if T is close to T_c, our simulations will be limited by finite-size effects. Because we can simulate only finite lattices, it is difficult to obtain estimates for the critical exponents α, β, and γ by using the definitions (15.31)–(15.33) directly. We learned in Section 12.4 that we can use *finite-size scaling* to extrapolate finite L results to $L \to \infty$. For example, from Figure 15.3 we see that the temperature at which C exhibits a maximum becomes better defined for larger lattices. This behavior provides a simple definition of the transition temperature $T_c(L)$ for a finite system. According to finite size scaling theory, $T_c(L)$ scales as

$$T_c(L) - T_c(L = \infty) \sim aL^{-1/\nu} \qquad (15.35)$$

where a is a constant and ν is defined in (15.34). The finite size of the lattice is important when the correlation length is comparable to the linear dimension of the system:

$$\xi(T) \sim L \sim |T - T_c|^{-\nu}. \qquad (15.36)$$

As in Section 12.4, we can set $T = T_c$ and consider the L-dependence of M, C, and χ:

$$m(T) \sim (T_c - T)^{\beta} \to L^{-\beta/\nu} \qquad (15.37)$$

$$C(T) \sim |T - T_c|^{-\alpha} \to L^{\alpha/\nu} \qquad (15.38)$$

$$\chi(T) \sim |T - T_c|^{-\gamma} \to L^{\gamma/\nu}. \qquad (15.39)$$

In Problem 15.17 we use the relations (15.37)–(15.39) to estimate the critical exponents β, γ, and α.

Problem 15.17. Finite-size scaling for the two-dimensional Ising model

(a) Use the relation (15.35) together with the exact result $\nu = 1$ to estimate the value of T_c for an infinite square lattice. Because it is difficult to obtain a precise value for T_c with small lattices, we will use the exact result $kT_c/J = 2/\ln(1+\sqrt{2}) \approx 2.269$ for the infinite lattice in the remaining parts of this problem.

(b) Determine the mean value of the absolute value of the magnetization per spin $|m|$, the specific heat C, and the susceptibility χ at $T = T_c$ for $L = 4$, 8, 16, and 32. Compute χ using (15.21) with $\langle|M|\rangle$ instead of $\langle M\rangle$. Use as many Monte Carlo steps per spin as possible. Plot the logarithm of $|m|$ and χ versus L and use the scaling relations (15.37)–(15.39) to determine the critical exponents β and γ. Use the exact result $\nu = 1$. Do your log-log plots of $|m|$ and χ yield reasonably straight lines? Compare your estimates for β and γ with the exact values given in Table 12.1.

(c) Make a log-log plot of C versus L. If your data for C is sufficiently accurate, you will find that the log-log plot of C versus L is not a straight line but shows curvature. The reason is that the exponent α in (15.33) equals zero for the two-dimensional Ising model, and hence (15.38) needs to be interpreted as

$$C \sim C_0 \ln L. \qquad (15.40)$$

Is your data for C consistent with (15.40)? The constant C_0 in (15.40) is approximately 0.4995. □

So far we have performed our Ising model simulations on a square lattice. How do the critical temperature and the critical exponents depend on the symmetry and the dimension of the lattice? Based on your experience with the percolation transition in Chapter 12, you probably know the answer.

Problem 15.18. The effects of symmetry and dimension on the critical properties

(a) The simulation of the Ising model on the triangular lattice is relevant to the understanding of the experimentally observed phases of materials that can be absorbed on the surface of graphite. The nature of the triangular lattice is discussed in Chapter 8 (see Figure 8.5). The main difference between the triangular lattice and the square lattice is the number of nearest neighbors. Make the necessary modifications in your program' for example, determine the energy changes due to a flip of a single spin and the corresponding values of the transition probabilities. Compute C and χ for different values of T in the interval $[2,5]$. Assume that $\nu = 1$ and use finite-size scaling to estimate T_c in the limit of an infinite triangular lattice. Compare your estimate of T_c to the known value $kT_c/J = 3.641$ (to three decimal places).

(b) No exact analytic results are available for the Ising model in three dimensions. (It has been shown by Istrail that this model cannot be solved analytically.) Write a Monte Carlo program to simulate the Ising model on the simple cubic lattice. Compute C and χ for T in the range $3.2 \leq T \leq 5$ in steps of 0.2 for different values of L. Estimate $T_c(L)$ from the maximum of C and χ. How do these estimates of $T_c(L)$ compare? Use the values of $T_c(L)$ that exhibit a stronger L-dependence and plot $T_c(L)$ versus $L^{-1/\nu}$ for different values of ν in the range 0.5 to 1 (see [15.35)]. Show that the extrapolated value of $T_c(L = \infty)$ does not depend sensitively on the value of ν. Compare your estimate for $T_c(L = \infty)$ to the known value $kT_c/J = 4.5108$ (to four decimal places).

(c) Compute $|m|$, C, and χ at $T = T_c \approx 4.5108$ for different values of L on the simple cubic lattice. Do a finite-size scaling analysis to estimate β/ν, α/ν, and γ/ν. The best known values of the critical exponents for the three-dimensional Ising model are given in Table 12.1. For comparison, published Monte Carlo results in 1976 for the finite-size behavior of the Ising model on the simple cubic Ising lattice are for $L = 6$ to $L = 20$; 2000–5000 Monte Carlo steps per spin were used for calculating the averages after equilibrium had been reached. Can you obtain more accurate results? □

Problem 15.19. Critical slowing down

(a) Consider the Ising model on a square lattice with $L = 16$. Compute the autocorrelation functions $C_M(t)$ and $C_E(t)$ and determine the correlation times τ_M and τ_E for $T = 2.5$, 2.4, and 2.3. Determine the correlation times as discussed in Problem 15.14b. How do these correlation times compare with one another? Show that τ increases as the critical temperature is approached, an effect known as *critical slowing down*.

(b) We can characterize critical slowing down by the dynamical critical exponent z defined by

$$\tau \sim \xi^z. \tag{15.41}$$

On a finite lattice we have $\tau \sim L^z$ at $T = T_c$. Compute τ for different values of L at $T = T_c$ and make a very rough estimate of z. (The value of z for the two-dimensional Ising model with spin flip dynamics is ≈ 2.167.) □

The values of τ and z found in Problem 15.19 depend on our choice of dynamics (algorithm). The reason for the large value of z is the existence of large domains of parallel spins near the critical point. It is difficult for the Metropolis algorithm to decorrelate a domain because it has to do so one spin at a time. What is the probability of flipping a single spin in the middle of a domain at $T = T_c$? Which spins in a domain are more likely to flip? What is the dominant mechanism for decorrelating a domain of spins? In one dimension Cordery et al. showed how z can be calculated exactly by considering the motion of a domain wall as a random walk.

Although we have generated a trial change by flipping a single spin, it is possible that other types of trial changes would be more efficient. A problem of much current interest is the development of more efficient algorithms near phase transitions (see Project 15.32).

15.9 Other Applications of the Ising Model

Because the applications of the Ising model range from flocking birds to beating hearts, we can mention only a few of the applications here. In the following, we briefly describe applications of the Ising model to first-order phase transitions, lattice gases, antiferromagnetism, and the order-disorder transition in binary alloys.

So far we have discussed the continuous phase transition in the Ising model and have found that the energy and magnetization vary continuously with the temperature, and thermodynamic derivatives such as the specific heat and the susceptibility diverge near T_c (in the limit of an infinite lattice). In Problem 15.20 we discuss a simple example of a *first-order* phase transition. Such transitions are accompanied by *discontinuous* (finite) changes in thermodynamic quantities such as the energy and the magnetization.

Problem 15.20. The Ising model in an external magnetic field

(a) Modify your two-dimensional Ising program so that the energy of interaction with an external magnetic field B is included. It is convenient to measure B in terms of the dimensionless ratio $h = \beta B$. (Remember that B has already absorbed a factor of μ.) Compute m, the mean magnetization per spin, as a function of h for $T < T_c$. Consider a square lattice with $L = 32$ and equilibrate the system at $T = 1.8$ and $h = 0$. Adopt the following procedure to obtain $m(h)$.

   (i) Use an equilibrium configuration at $h = 0$ as the initial configuration for $h_1 = \Delta h = 0.2$.

   (ii) Run the system for 100 Monte Carlo steps per spin before computing averages.

   (iii) Average m over 100 Monte Carlo steps per spin.

   (iv) Use the last configuration for h_n as the initial configuration for $h_{n+1} = h_n + \Delta h$.

   (v) Repeat steps (ii)–(iv) until $m \approx 0.95$. Plot m versus h. Do the measured values of m correspond to equilibrium averages?

(b) Start from the last configuration in part (a) and decrease h by $\Delta h = -0.2$ in the same way as in part (a) until h passes through zero and $m \approx -0.95$. Extend your plot of m versus h to include negative h values. Does m remain positive for small negative h? Do the measured values of m for negative h correspond to equilibrium averages? Draw the spin configurations for several values of h. Do you see evidence of domains?

(c) Now increase h by $\Delta h = 0.2$ until the m versus h curve forms an approximately closed loop. What is the value of m at $h = 0$? This value of m is the spontaneous magnetization.

(d) A first-order phase transition is characterized by a discontinuity (for an infinite lattice) in the order parameter. In the present case the transition is characterized by the behavior of m as a function of h. What is your measured value of m for $h = 0.2$? If $m(h)$ is double valued, which value of m corresponds to the equilibrium state, an absolute minima in the free energy? Which value of m corresponds to a *metastable* state, a relative minima in the free energy? What are the equilibrium and metastable values of m for $h = -0.2$? First-order transitions exhibit *hysteresis*, and the properties of the system depend on the history of the system, for example, whether h is increasing or decreasing. Because of the long lifetime of metastable states near a first–order phase transition, a system can mistakenly be interpreted as being in the state of minimum free energy. We also know that near a continuous phase transition, the relaxation to equilibrium becomes very long (see Problem 15.19), and hence a system with a continuous phase transition can also behave as if it were in a metastable state. For these reasons it is difficult to distinguish the nature of a phase transition using computer simulations. This problem is discussed further in Section 15.11.

(e) Repeat the above simulations for $T = 3$, a temperature above T_c. Why do your results differ from the simulations in parts (a)–(c) done for $T < T_c$? □

The Ising model also describes systems that might appear to have little in common with ferromagnetism. For example, we can interpret the Ising model as a lattice gas, where a down spin represents a lattice site occupied by a molecule and an up site represents an empty site. Each lattice site can be occupied by at most one molecule, and the molecules interact with their nearest neighbors. The lattice gas is a crude model of the behavior of a real gas of molecules and is a simple model of the liquid-gas transition and the critical point. What properties does the lattice gas have in common with a real gas? What properties of real gases does the lattice gas neglect?

If we wish to simulate a lattice gas, we have to decide whether to do the simulation at fixed density or at fixed chemical potential μ and a variable number of particles. The implementation of the latter is straightforward because the grand canonical ensemble for a lattice gas is equivalent to the canonical ensemble for Ising spins in an external magnetic field; that is, the effect of the magnetic field is to fix the mean number of up spins. Hence, we can simulate a lattice gas in the grand canonical ensemble by doing spin flip dynamics. (The volume of the lattice is an irrelevant parameter.)

Another application of a lattice gas model is to phase separation in a binary or A-B alloy. In this case spin up and spin down correspond to a site occupied by an A atom and B atom, respectively. As an example, the alloy β-brass has a low temperature ordered phase in which the two components (copper and zinc) have equal concentrations and form a cesium chloride structure. As the temperature is increased, some zinc atoms exchange positions with copper atoms, but the system is still ordered. However, above the critical temperature $T_c = 742\,\mathrm{K}$, the zinc and copper atoms become mixed and the system is disordered. This transition is an example of an *order-disorder* transition.

If we wish to approximate the actual dynamics of an alloy, then the number of A atoms and the number of B atoms is fixed, and we cannot use spin flip dynamics to simulate a binary alloy. A dynamics that does conserve the number of down and up spins is known as *spin exchange* or Kawasaki dynamics. In this dynamics a trial *interchange* of two nearest neighbor spins is made and the change in energy ΔE is calculated. The criterion for the acceptance or rejection of the trial change is the same as before.

Problem 15.21. Simulation of a lattice gas

(a) Modify your Ising program so that spin exchange dynamics rather than spin flip dynamics is implemented. Determine the possible values of ΔE on the square lattice and the possible values of the transition probability and change the way a trial change is made. If we are interested only in the mean value of quantities such as the total energy, we can reduce the computation time by not considering the interchange of parallel spins (which has no effect). For example, we can keep a list of bonds between occupied and empty sites and make trial moves by choosing bonds at random from this list. For small lattices such a list is unnecessary, and a trial move can be generated by simply choosing a spin and one of its nearest neighbors at random.

(b) Consider a square lattice with $L = 32$ and 512 sites initially occupied. (The number of occupied sites is a conserved variable and must be specified initially.) Determine the mean energy for T in the range $1 \leq T \leq 4$. Plot the mean energy as a function of T. Does the energy appear to vary continuously?

(c) Repeat the calculations of part (b) with 612 sites initially occupied and plot the mean energy as a function of T. Does the energy vary continuously? Do you see any evidence of a first-order phase transition?

(d) Because down spins correspond to particles, we can compute their single particle diffusion coefficient. Use an array to record the position of each particle as a function of time. After equilibrium has been reached, compute $\langle R(t)^2 \rangle$, the mean square displacement of a particle. Is it necessary to "interchange" two like spins? If the particles undergo a random walk, the self-diffusion constant D is defined as

$$D = \lim_{t\to\infty} \frac{1}{2dt} \langle R(t)^2 \rangle. \tag{15.42}$$

Estimate D for different temperatures and numbers of occupied sites. Note that if a particle starts at x_0 and returns to x_0 by moving in one direction on the average using periodic boundary conditions, the net displacement in the x direction is L not 0 (see Section 8.10 for a discussion of how to compute the diffusion constant for systems with periodic boundary conditions). □

Although you are probably familiar with ferromagnetism, for example, a magnet on a refrigerator door, nature provides more examples of antiferromagnetism. In the language of the Ising model, antiferromagnetism means that the exchange parameter J is negative and nearest neighbor spins prefer to be aligned in opposite directions. As we will see in Problem 15.22, the properties of the antiferromagnetic Ising model on a square lattice are similar to the ferromagnetic Ising model. For example, the energy and specific heat of the ferromagnetic and antiferromagnetic Ising models are identical at all temperatures in zero magnetic field, and the system exhibits a phase transition at the Néel temperature T_N. On the other hand, the total magnetization and susceptibility do not exhibit critical behavior near T_N. Instead, we need to define two sublattices for the square lattice corresponding to the red and black squares of a checkerboard and introduce the staggered magnetization M_s, which is equal to the difference of the magnetization of the two sublattices. We will find in Problem 15.22 that the temperature dependence of M_s and the staggered susceptibility χ_s are identical to the analogous quantities in the ferromagnetic Ising model.

Problem 15.22. Antiferromagnetic Ising model

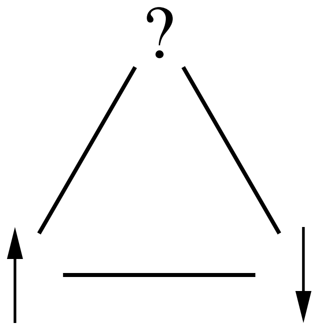


Figure 15.5: An example of frustration on a triangular lattice. The interaction in antiferromagnetic.

(a) Modify the `Ising` class to simulate the antiferromagnetic Ising model on the square lattice in zero magnetic field. Because J does not appear explicitly in class `Ising`, change the sign of the energy calculations in the appropriate places in the program. To compute the staggered magnetization on a square lattice, define one sublattice to be the sites (x, y) for which the product $\mathrm{mod}(x, 2) \times \mathrm{mod}(y, 2) = 1$; the other sublattice corresponds to the remaining sites.

(b) Choose $L = 32$ and all spins up initially. What configuration of spins corresponds to the state of lowest energy? Compute the temperature dependence of the mean energy, the magnetization, the specific heat, and the susceptibility. Does the temperature dependence of any of these quantities show evidence of a phase transition?

(c) In part (b) you might have noticed that χ shows a cusp. Compute χ for different values of L at $T = T_N \approx 2.269$. Do a finite-size scaling analysis and verify that χ does not diverge at $T = T_N$.

(d) Compute the temperature dependence of M_s and the staggered susceptibility χ_s defined as [see (15.21)]

$$\chi_s = \frac{1}{kT}\Big[\langle {M_s}^2\rangle - \langle M_s\rangle^2\Big]. \tag{15.43}$$

(Below T_c it is better to compute $\langle |M_s|\rangle$ instead of $\langle M_s\rangle$ for small lattices.) Verify that the temperature dependence of M_s for the antiferromagnetic Ising model is the same as the temperature dependence of M for the Ising ferromagnet. Could you have predicted this similarity without doing the simulation? Does χ_s show evidence of a phase transition?

(e) Consider the behavior of the antiferromagnetic Ising model on a triangular lattice. Choose $L \geq 32$ and compute the same quantities as before. Do you see any evidence of a phase transition? Draw several configurations of the system at different temperatures. Do you see evidence of many small domains at low temperatures? Is there a unique ground state? If you cannot find a unique ground state, you share the same frustration as do the individual spins in the antiferromagnetic Ising model on the triangular lattice. We say that this model exhibits *frustration* because there is no spin configuration on the triangular lattice such that all spins are able to minimize their energy (see Figure 15.5). □

The Ising model is one of many models of magnetism. The Heisenberg, Potts, and x-y models are other examples of models of magnetic materials. Monte Carlo simulations of these

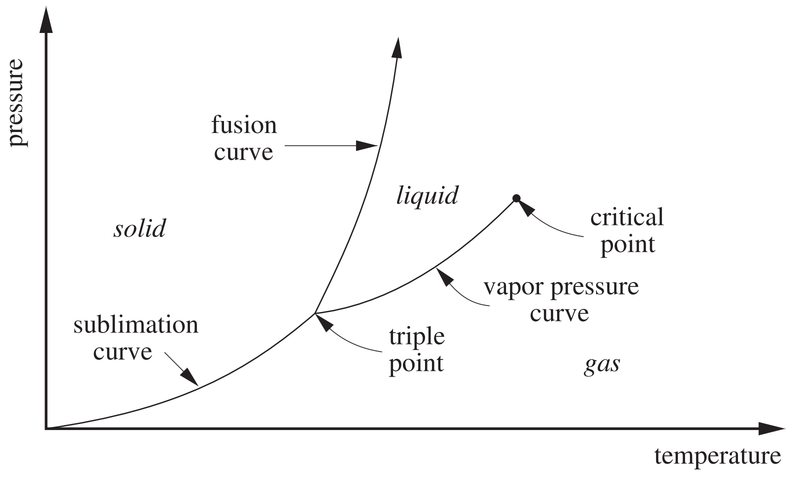


Figure 15.6: A sketch of the phase diagram for a simple material.

models and others have been important in the development of our understanding of phase transitions in both magnetic and nonmagnetic materials. Some of these models are discussed in Section 15.14.

15.10 Simulation of Classical Fluids

The existence of matter as a solid, liquid, and gas is well known (see Figure 15.6). Our goal in this section is to use Monte Carlo methods to gain additional insight into the qualitative differences between these phases.

The Monte Carlo simulation of classical systems is simplified considerably by the fact that the velocity (momentum) variables are decoupled from the position variables. The total energy of the system can be written as $E = K(\{\mathbf{v}_i\}) + U(\{\mathbf{r}_i\})$, where the kinetic energy K is a function of only the particle velocities $\{\mathbf{v}_i\}$, and the potential energy U is a function of only the particle positions $\{\mathbf{r}_i\}$. This separation implies we need to sample only the positions of the molecules, that is, the "configurational" degrees of freedom. Because the velocity appears quadratically in the kinetic energy, it can be shown using classical statistical mechanics that the contribution of the velocity coordinates to the mean energy is $\frac{1}{2}kT$ per degree of freedom. Is this simplification possible for quantum systems?

The physically relevant quantities of a fluid include its mean energy, specific heat, and equation of state. Another interesting quantity is the *radial distribution function* $g(r)$ which we introduced in Chapter 8. We will find in Problems 15.23–15.25 that $g(r)$ is a probe of the density fluctuations and, hence, a probe of the local order in the system. If only two-body forces are present, the mean potential energy per particle can be expressed as [see (8.16)]

$$\frac{U}{N} = \frac{\rho}{2}\int g(r)u(r)\,d\mathbf{r}, \tag{15.44}$$

and the (virial) equation of state can be written as [see (8.17)]

$$\frac{\beta P}{\rho} = 1 - \frac{\beta\rho}{2d}\int g(r)\,r\frac{du(r)}{dr}\,d\mathbf{r}. \tag{15.45}$$

Hard core interactions. To separate the effects of the short range repulsive interaction from the longer range attractive interaction, we first investigate a model of *hard disks* with the interparticle interaction

$$u(r) = \begin{cases} +\infty & (r < \sigma) \\ 0 & (r \geq \sigma). \end{cases} \tag{15.46}$$

Such an interaction has been extensively studied in one dimension (hard rods), two dimensions (hard disks), and three dimensions (hard spheres). Hard sphere systems were the first systems studied by Metropolis and coworkers.

Because there is no attractive interaction present in (15.46), there is no transition from a gas to a liquid. Is there a phase transition between a fluid phase at low densities and a solid at high densities? Can a solid form in the absence of an attractive interaction?

What are the physically relevant quantities for a system with a hard core interaction? The mean potential energy is of no interest because the potential energy is always zero. The major quantity of interest is $g(r)$ which yields information about the correlations of the particles and the equation of state. If the interaction is given by (15.46), it can be shown that (15.45) reduces to

$$\frac{\beta P}{\rho} = 1 + \frac{2\pi}{3}\rho\sigma^3 g(\sigma) \qquad (d = 3) \tag{15.47a}$$

$$= 1 + \frac{\pi}{2}\rho\sigma^2 g(\sigma) \qquad (d = 2) \tag{15.47b}$$

$$= 1 + \rho\sigma g(\sigma). \qquad (d = 1) \tag{15.47c}$$

We will calculate $g(r)$ for different values of r and then extrapolate our results to $r = \sigma$ (see Problem 15.23b).

Because the application of molecular dynamics and Monte Carlo methods to hard disks is similar, we discuss the latter method only briefly and do not include a program. The idea is to choose a disk at random and move it to a trial position as implemented in the following:

```
int i = (int)(N*Math.random()); // choose a particle at random
xtrial += (2.0*Math.random() - 1.0)*delta; // delta is maximum displacement
ytrial += (2.0*Math.random() - 1.0)*delta;
```

If the new position overlaps another disk, the move is rejected and the old configuration is retained; otherwise, the move is accepted. A reasonable, although not necessarily optimum, choice for the maximum displacement δ is to choose δ such that approximately 20% of the trial moves are accepted.

The major difficulty in implementing this algorithm is determining the overlap of two particles. If the number of particles is not too large, it is sufficient to compute the distances between the trial particle and all the other particles, instead of just considering the smaller number of particles that are in the immediate vicinity of the trial particle. For larger systems this procedure is too time consuming, and it is better to divide the system into cells and to only compute the distances between the trial particle and particles in the same and neighboring cells.

The choice of initial positions for the disks is more complicated than it might first appear. One strategy is to place each successive disk at random in the box. If a disk overlaps one that is already present, generate another pair of random numbers and attempt to place the disk again. If the desired density is low, an acceptable initial configuration can be computed fairly quickly in this way, but if the desired density is high, the probability of adding a disk will be very small (see Problem 15.24a). To reach higher densities, we might imagine beginning with the desired number of particles in a low density configuration and moving the boundaries of the central cell inward until a boundary just touches one of the disks. Then the disks are moved a number of Monte Carlo steps and the boundaries are moved inward again. This procedure also becomes more difficult as the density increases. The most efficient procedure is to start the disks on a lattice at the highest density of interest such that no overlap of disks occurs.

We first consider a one-dimensional system of hard rods for which the equation of state and $g(r)$ can be calculated exactly. The equation of state is given by

$$\frac{P}{NkT} = \frac{1}{L - N\sigma}. \tag{15.48}$$

Because hard rods cannot pass through one another, the excluded volume is $N\sigma$ and the available volume is $L - N\sigma$. Note that the form of (15.48) is the same as the van der Waals equation of state (cf. Reif) with the contribution from the attractive part of the interaction equal to zero.

Problem 15.23. Monte Carlo simulation of hard rods

(a) Write a program to do a Monte Carlo simulation of a system of hard rods. Adopt periodic boundary conditions and refer to class `HardDisks` in Chapter 8 for the structure of the program. The major difference is the nature of the trial moves. Measure all lengths in terms of the hard rod diameter σ. Choose $L = 36$ and $N = 30$. How does the number density $\rho = N/L$ compare to the maximum possible density? Choose the initial positions to be on a one-dimensional grid and let the maximum displacement be $\delta = 0.1$. Approximately how many Monte Carlo steps per particle are necessary to reach equilibrium? What is the equilibrium acceptance probability? Compute the pair correlation function $g(x)$.

(b) Compute $g(x)$ as a function of the distance x for $x \le L/2$. Why does $g(x) = 0$ for $x < 1$? What is the physical interpretation of the peaks in $g(x)$? Because the mean pressure can be determined from $g(x)$ at $x = 1^+$ [see (15.47)], determine $g(x)$ at contact. An easy way to extrapolate your results for $g(x)$ to $x = 1$ is to fit the three values of $g(x)$ closest to $x = 1$ to a parabola. Use your result for $g(x = 1^+)$ to determine the mean pressure.

(c) Compute $g(x)$ at several lower densities by using an equilibrium configuration from a previous run and increasing L. How do the size and the location of the peaks in $g(x)$ change? □

Problem 15.24. Monte Carlo simulation of hard disks

(a) The maximum packing density can be found by placing the disks on a triangular lattice with the nearest neighbor distance equal to the disk diameter σ. What is the maximum packing density of hard disks; that is, how many disks can be packed together in a cell of area A?

(b) Write a simple program that adds disks at random into a rectangular box of area $A = L_x \times L_y$ with the constraint that no two disks overlap. If a disk overlaps a disk already present, generate another pair of random numbers and try to place the disk again. If the density is low, the probability of adding a disk is high, but if the desired density is high most of the disks will be rejected. For simplicity, do not worry about periodic boundary conditions and accept a disk if its center lies within the box. Choose $L_x = 6$ and $L_y = \sqrt{3}L_x/2$ and determine the maximum density $\rho = N/A$ that you can attain in a reasonable amount of CPU time. How does this density compare to the maximum packing density? What is the qualitative nature of the density dependence of the acceptance probability?

(c) Modify your Monte Carlo program for hard rods to a system of hard disks. Begin at a density ρ slightly lower than the maximum packing density ρ_0. Choose $N = 64$ with $L_x = 8.81$ and $L_y = \sqrt{3}L_x/2$. Compare the density $\rho = N/(L_xL_y)$ to the maximum packing density. Choose the initial positions of the particles to be on a triangular lattice. A reasonable first choice for the maximum displacement δ is $\delta = 0.1$. Compute $g(r)$ for $\rho/\rho_0 = 0.95$, 0.92, 0.88, 0.85, 0.80, 0.70, 0.60, and 0.30. Keep the ratio of L_x/L_y fixed and save a configuration from the previous run to be the initial configuration of the new run at lower ρ. (See page 266 for how to save and read configurations.) Allow at least 400 Monte Carlo steps per particle for the system to equilibrate and average $g(r)$ for `mcs` ≥ 400.

(d) What is the qualitative behavior of $g(r)$ at high and low densities? For example, describe the number and height of the peaks of $g(r)$. If the system is crystalline, then $g(\mathbf{r})$ is not spherically symmetric. What would you compute in this case?

(e) Use your results for $g(r = 1^+)$ to compute the mean pressure P as a function of ρ [see (15.47b)]. Plot the ratio PV/NkT as a function of ρ, where the volume V is the area of the system. How does the temperature T enter into the Monte Carlo simulation? Is the ratio PV/NkT an increasing or decreasing function of ρ? At low densities we might expect the system to act like an ideal gas with the volume replaced by $(V - N\sigma)$. Compare your low density results with this prediction.

(f) Take snapshots of the disks at intervals of ten to twenty Monte Carlo steps per particle. Do you see any evidence of the solid becoming a fluid at lower densities?

(g) Compute an effective diffusion coefficient D by determining the mean square displacement $\langle R^2(t)\rangle$ of the particles after equilibrium is reached. Use the relation (15.42) and identify the time t with the number of Monte Carlo steps per particle. Estimate D for the densities considered in part (b) and plot the product ρD as a function of ρ. What is the dependence of D on ρ for a dilute gas? Can you identify a range of ρ where D drops abruptly? Do you observe any evidence of a phase transition?

(h) The magnitude of the maximum displacement parameter δ is arbitrary. If the density is high and δ is large, then a high proportion of the trial moves will be rejected. On the other hand, if δ is small, the acceptance probability will be close to unity, but the successive configurations will be strongly correlated. Hence, if δ is too large or is too small, the simulation would be inefficient. One way to choose δ is to find the value of δ that maximizes the mean square displacement over a fixed time interval. The idea is that the mean square displacement is a measure of the exploration of phase space. Fix the density and determine the value of δ that maximizes $\langle R^2(t)\rangle$. What is the corresponding acceptance probability? □

Continuous potentials. Our simulations of hard disks suggest that there is a phase transition from a fluid at low densities to a solid at higher densities. This conclusion is consistent with molecular dynamics and Monte Carlo studies of larger systems. Although the existence of a fluid-solid transition for hard sphere and hard disk systems is now well accepted, the relatively small numbers of particles used in any simulation should remind us that results of this type cannot be taken as evidence independently of any theoretical justification.

The existence of a fluid-solid transition for hard spheres implies that the transition is determined by the repulsive part of the potential. We now consider a system with both a repulsive and an attractive contribution. Our primary goal will be to determine the influence of the attractive part of the potential on the structure of a liquid.

We adopt as our model interaction the Lennard–Jones potential:

$$u(r) = 4\epsilon\Big[\Big(\frac{\sigma}{r}\Big)^{12} - \Big(\frac{\sigma}{r}\Big)^{6}\Big]. \tag{15.49}$$

The nature of the Lennard–Jones potential and the appropriate choice of units for simulations was discussed in Chapter 8 (see Table 8.1). We consider in Problem 15.25 the application of the Metropolis algorithm to a system of N particles in a cell of fixed volume V (area) interacting via the Lennard–Jones potential. Because the simulation is at fixed T, V, and N, the simulation samples configurations of the system according to the Boltzmann distribution (15.4).

Problem 15.25. Monte Carlo simulation of a Lennard–Jones system

(a) The properties of a two-dimensional Lennard–Jones system have been studied by many workers under a variety of conditions. Write a program to compute the total energy of a system of N particles on a triangular lattice of area $L_x \times L_y$ with periodic boundary conditions. Choose $N = 64, L_x = 9.2$, and $L_y = \sqrt{3}L_x/2$. Why does this energy correspond to the energy at temperature $T = 0$? Does the energy per particle change if you consider bigger systems at the same density?

(b) Write a program to compute the mean energy, pressure, and the radial distribution function using the Metropolis algorithm. One way of computing the change in the potential energy of the system due to a trial move of one of the particles is to use an array pe for the potential energy of interaction of each particle. For simplicity, compute the potential energy of particle i by considering its interaction with the other $N - 1$ particles. The total potential energy of the system is the sum of the array elements pe(i) over all N particles divided by two to account for double counting. For simplicity, accumulate data after each Monte Carlo step per particle.

(c) Choose the same values of N, L_x, and L_y as in part (a) but give each particle an initial random displacement from its triangular lattice site of magnitude 0.2. Do the Monte Carlo simulation at a very low temperature such as $T = 0.1$. Choose the maximum trial displacement $\delta = 0.15$ and consider mcs ≥ 400. Does the system retain its symmetry? Does the value of δ affect your results?

(d) Use the same initial conditions as in part (a), but take $T = 0.5$. Choose $\delta = 0.15$ and run for a number of Monte Carlo steps per particle that is sufficient to yield a reasonable result for the mean energy. Do a similar simulation at $T = 1$ and $T = 2$. What is the best choice of the initial configuration in each case? The harmonic theory of solids predicts that the total energy of a system is due to a $T = 0$ contribution plus a term due to the harmonic oscillation of the atoms. The contribution of the latter part should be proportional to the temperature. Compare your results for $E(T) - E(0)$ with this prediction. Use the values of σ and ϵ given in Table 8.1 to determine the temperature and energy in SI units for your simulations of solid argon.

(e) Decrease the density by multiplying L_x, L_y, and all the particle coordinates by 1.07. What is the new value of ρ? Estimate the number of Monte Carlo steps per particle needed to compute E and P at $T = 0.5$ to approximately 10% accuracy. Is the total energy positive or negative? How do E and P compare to their ideal gas values? Follow the method discussed in Problem 15.24 and compute an effective diffusion constant. Is the system a liquid or a solid? Plot $g(r)$ versus r and compare $g(r)$ to your results for hard disks at the same density.

What is the qualitative behavior of $g(r)$? What is the interpretation of the peaks in $g(r)$ in terms of the structure of the liquid? If time permits, consider a larger system at the same density and temperature and compute $g(r)$ for larger r.

(f) Consider the same density as in part (e) at $T = 0.6$ and $T = 1$. Look at some typical configurations of the particles. Use your results for $E(T)$, $P(T)$, $g(r)$ and the other data you have collected and discuss whether the system is a gas, liquid, or solid at these temperatures. What criteria can you use to distinguish a gas from a liquid? If time permits, repeat these calculations for $\rho = 0.7$.

(g) Compute E, P, and $g(r)$ for $N = 64$, $L_x = L_y = 20$, and $T = 3$. These conditions correspond to a dilute gas. How do your results for P compare with the ideal gas equation of state? How does $g(r)$ compare with the results you obtained for the liquid?

(h) The chemical potential can be measured using the Widom insertion method. From thermodynamics we know that

$$\mu = \left(\frac{\partial F}{\partial N}\right)_{V,T} = -kT \ln \frac{Z_{N+1}}{Z_N} \tag{15.50}$$

in the limit $N \to \infty$, where F is the Helmholtz free energy and Z_N is the partition function for N particles. The ratio Z_{N+1}/Z_N is the average of $e^{-\beta\Delta E}$ over all possible states of the added particle with added energy ΔE. The idea is to compute the change in the energy ΔE that would occur if an imaginary particle were added to the N particle system at random. Average the value of $e^{-\beta\Delta E}$ over many configurations generated by the Metropolis algorithm. The chemical potential is then given by

$$\mu = -kT \ln\langle e^{-\beta\Delta E}\rangle. \tag{15.51}$$

Note that in the Widom insertion method, no particle is actually added to the system during the simulation. The chemical potential computed in (15.51) is the excess chemical potential and does not include the part of the chemical potential due to the momentum degrees of freedom, which is equal to the chemical potential of an ideal gas. Compute the chemical potential of a dense gas, liquid, and solid. In what sense is the chemical potential a measure of how easy it is to add a particle to the system? □

15.11 Optimized Monte Carlo Data Analysis

As we have seen, the important physics near a phase transition occurs on long length scales. For this reason, we might expect that simulations, which for practical reasons are restricted to relatively small systems, might not be useful for simulations near a phase transition. Nevertheless, we have found that methods such as finite-size scaling can yield information about how systems behave in the thermodynamic limit. We now explore some additional Monte Carlo techniques that are useful near a phase transition.

The Metropolis algorithm yields mean values of various thermodynamic quantities, for example, the energy at particular values of the temperature T. Near a phase transition many thermodynamic quantities change rapidly, and we need to determine these quantities at many closely spaced values of T. If we were to use standard Monte Carlo methods, we would have to do many simulations to cover the desired range of values of T. To overcome this problem, we introduce the use of *histograms* which allow us to extract more information from a single Monte

Carlo simulation. The idea is to use our knowledge of the equilibrium probability distribution at one value of T (and other external parameters) to estimate the desired thermodynamic averages at neighboring values of the external parameters.

The first step of the single histogram method is to simulate the system at an inverse temperature β_0 which is near the values of β of interest and measure the energy of the system after every Monte Carlo step per spin (or other fixed interval). The measured probability that the system has energy E can be expressed as

$$P(E,\beta_0) = \frac{H_0(E)}{\sum_E H_0(E)}. \tag{15.52}$$

The histogram $H_0(E)$ is the number of configurations with energy E, and the denominator is the total number of measurements of E. Because the probability of a given configuration is given by the Boltzmann distribution, we have

$$P(E,\beta) = \frac{g(E)\,e^{-\beta E}}{\sum_E g(E)\,e^{-\beta E}} \tag{15.53}$$

where $g(E)$ is the number of microstates with energy E. (The density of states $g(E)$ should not be confused with the radial distribution function $g(r)$. If the energy is a continuous function, $g(E)$ becomes the number of states per unit energy interval. However, $g(E)$ is usually referred to as the density of states regardless of whether E is a continuous or discrete variable.) If we compare (15.52) and (15.53) and note that $g(E)$ is independent of T, we can write

$$g(E) = a_0 H_0(E) e^{\beta_0 E} \tag{15.54}$$

where a_0 is a proportionality constant that depends on β_0. If we eliminate $g(E)$ from (15.53) by using (15.54), we obtain the desired relation

$$P(E,\beta) = \frac{H_0(E)\,e^{-(\beta-\beta_0)E}}{\sum_E H_0(E)\,e^{-(\beta-\beta_0)E}}. \tag{15.55}$$

Note that we have expressed the probability at the inverse temperature β in terms of $H_0(E)$, the histogram at the inverse temperature β_0.

Because β is a continuous variable, we can estimate the β dependence of the mean value of any function A that depends on E, for example, the mean energy and the specific heat. We write the mean of $A(E)$ as

$$\langle A(\beta)\rangle = \sum_E A(E)P(E,\beta). \tag{15.56}$$

If the quantity A depends on another quantity M, for example, the magnetization, then we can generalize (15.56) to

$$\langle A(\beta)\rangle = \sum_{E,M} A(E,M)P(E,M,\beta) \tag{15.57a}$$

$$= \frac{\sum_{E,M} A(E,M)H_0(E,M)e^{-(\beta-\beta_0)E}}{\sum_{E,M} H_0(E,M)e^{-(\beta-\beta_0)E}}. \tag{15.57b}$$

The histogram method is useful only when the configurations relevant to the range of temperatures of interest occur with reasonable probability during the simulation at temperature

T_0. For example, if we simulate an Ising model at low temperatures at which only ordered configurations occur (most spins aligned in the same direction), we cannot use the histogram method to obtain meaningful thermodynamic averages at high temperatures for which most configurations are disordered.

Problem 15.26. Application of the histogram method

(a) Consider a 4×4 Ising lattice in zero magnetic field and use the Metropolis algorithm to compute the mean energy per spin, the mean magnetization per spin, the specific heat, and the susceptibility per spin for $T = 1$ to $T = 3$ in steps of $\Delta T = 0.05$. Average over at least 5000 Monte Carlo steps for each value of T per spin after equilibrium has been reached.

(b) What are the minimum and maximum values of the total energy E that might be observed in a simulation of a Ising model on a 4×4 lattice? Use these values to set the size of the array needed to accumulate data for the histogram $H(E)$. Accumulate data for $H(E)$ at $T = 2.27$, a value of T close to T_c, for at least 5000 Monte Carlo steps per spin after equilibration. Compute the energy and specific heat using (15.56). Compare your computed results with the data obtained by simulating the system directly, that is, without using the histogram method, at the same temperatures. At what temperatures does the histogram method break down?

(c) What are the minimum and maximum values of the magnetization M that might be observed in a simulation of a Ising model on a 4×4 lattice? Use these values to set the size of the two-dimensional array needed to accumulate data for the histogram $H(E,M)$. Accumulate data for $H(E,M)$ at $T = 2.27$, a value of T close to T_c, for at least 5000 Monte Carlo steps per spin after equilibration. Compute the same thermodynamic quantities as in part (a) using (15.57b). Compare your computed results with the data obtained by simulating the system directly, that is, without using the histogram method, at the same temperatures. At what temperatures does the histogram method break down?

(d) Repeat part (c) for a simulation centered about $T = 1.5$ and $T = 2.5$.

(e) Repeat part (c) for an 8×8 and a 16×16 lattice at $T = 2.27$. □

The histogram method can be used to do a more sophisticated finite-size scaling analysis to determine the nature of a transition. Suppose that we perform a Monte Carlo simulation and observe a peak in the specific heat as a function of the temperature. What can this observation tell us about a possible phase transition? In general, we can conclude very little without doing a careful analysis of the behavior of the system at different sizes. For example, a discontinuity in the energy in an infinite system might be manifested in small systems by a broad peak in the specific heat. However, we have seen that the specific heat of a system with a continuous phase transition in the thermodynamic limit may manifest itself in the same way in a small system. Another difficulty is that the peak in the specific heat of a small system occurs at a temperature that differs from the transition temperature in the infinite system (see Project 15.37). Finally, there might be no transition at all, and the peak might simply represent a broad crossover from high to low temperature behavior (see Project 15.38).

We now discuss a method due to Lee and Kosterlitz that uses the histogram data to determine the nature of a phase transition (if it exists). To understand this method, we use the Helmholtz free energy F of a system. At low T, the low energy configurations dominate the contributions to the partition function Z, even though there are relatively few such configurations. At high T, the number of disordered configurations with high E is large, and hence high energy configurations dominate the contribution to Z. These considerations suggest that it is useful to define a restricted free energy $F(E)$ that includes only the configurations at a particular energy E. We define

$$F(E) = -kT \ln\Big[g(E)\, e^{-\beta E}\Big]. \tag{15.58}$$

For systems with a first-order phase transition, a plot of $F(E)$ versus E will show two local minima corresponding to configurations that are characteristic of the high and low temperature phases. At low T the minimum at the lower energy will be the absolute minimum, and at high T the higher energy minimum will be the absolute minimum of F. At the transition, the two minima will have the same value of $F(E)$. For systems with no transition in the thermodynamic limit, there will only be one minimum for all T.

How will $F(E)$ behave for the relatively small lattices that we can simulate? In systems with first-order transitions, the distinction between low and high temperature phases will become more pronounced as the system size is increased. If the transition is continuous, there are domains at all sizes, and we expect that the behavior of $F(E)$ will not change significantly as the system size increases. If there is no transition, there might be a spurious double minima for small systems, but this spurious behavior should disappear for larger systems. Lee and Kosterlitz proposed the following method for categorizing phase transitions.

1. Do a simulation at a temperature close to the suspected transition temperature and compute $H(E)$. Usually the temperature at which the peak in the specific heat occurs is chosen as the simulation temperature.

2. Use the histogram method to compute $F(E) \propto -\ln H_0(E) + (\beta - \beta_0)E$ at neighboring values of T. If there are two minima in $F(E)$, vary β until the values of $F(E)$ at the two minima are equal. This temperature is an estimate of the possible transition temperature T_c.

3. Measure the difference ΔF at T_c between $F(E)$ at the minima and $F(E)$ at the maximum between the two minima.

4. Repeat steps (1)–(3) for larger systems. If ΔF increases with size, the transition is first order. If ΔF remains the same, the transition is continuous. If ΔF decreases with size, there is no thermodynamic transition.

The above procedure is applicable when the phase transition occurs by varying the temperature. Transitions can also occur by varying the pressure or the magnetic field. These *field-driven transitions* can be tested by a similar method. For example, consider the Ising model in a magnetic field at low temperatures below T_c. As we vary the magnetic field from positive to negative, there is a transition from a phase with magnetization $M > 0$ to a phase with $M < 0$. Is this transition first order or continuous? To answer this question, we can use the Lee–Kosterlitz method with a histogram $H(E,M)$ generated at zero magnetic field and calculate $F(M)$ instead of $F(E)$. The quantity $F(M)$ is proportional to $-\ln \sum_E H(E,M)\, e^{-(\beta-\beta_0)E}$. Because the states with positive and negative magnetization are equally likely to occur for zero magnetic field, we should see a double minima structure for $F(M)$ with equal minima. As we increase the size of the system, ΔF should increase for a first-order transition and remain the same for a continuous transition.

Problem 15.27. Characterization of a phase transition

(a) Use your modified version of class `Ising` from Problem 15.26 to determine $H(E,M)$. Read the $H(E,M)$ data from a file and compute and plot $F(E)$ for the range of temperatures of interest. First generate data at $T = 2.27$ and use the Lee–Kosterlitz method to verify that the Ising model in two dimensions has a continuous phase transition in zero magnetic field. Consider lattices of sizes $L = 4$, 8, and 16.

(b) Do a Lee–Kosterlitz analysis of the Ising model at $T = 2$ and zero magnetic field by plotting $F(M)$. Determine if the transition from $M > 0$ to $M < 0$ is first order or continuous. This transition is called field driven because the transition occurs if we change the magnetic field. Make sure your simulations sample configurations with both positive and negative magnetization by using small values of L such as $L = 4$, 6, and 8.

(c) Repeat part (b) at $T = 2.5$ and determine if there is a field-driven transition at $T = 2.5$. □

*__Problem 15.28.__ The Potts model
In the q-state Potts model, the total energy or Hamiltonian of the lattice is given by

$$E = -J \sum_{i,j=\mathrm{nn}(i)} \delta_{s_i,s_j} \tag{15.59}$$

where s_i at site i can have the values $1, 2, \ldots, q$; the Kronecker delta function $\delta_{a,b}$ equals unity if $a = b$, and is zero otherwise. As before, we will measure the temperature in energy units. Convince yourself that the $q = 2$ Potts model is equivalent to the Ising model (except for a trivial difference in the energy minimum). One of the many applications of the Potts model is to helium absorbed on the surface of graphite. The graphite-helium interaction gives rise to preferred adsorption sites directly above the centers of the honeycomb graphite surface. As discussed by Plischke and Bergersen, the helium atoms can be described by a three-state Potts model.

(a) The transition in the Potts model is continuous for small q and first order for larger q. Write a Monte Carlo program to simulate the Potts model for a given value of q and store the histogram $H(E)$. Test your program by comparing the output for $q = 2$ with your Ising model program.

(b) Use the Lee–Kosterlitz method to analyze the nature of the phase transition in the Potts model for $q = 3$, 4, 5, 6, and 10. First find the location of the specific heat maximum, and then collect data for $H(E)$ at the specific heat maximum. Lattice sizes of order $L \geq 50$ are required to obtain convincing results for some values of q. □

Another way to determine the nature of a phase transition is to use the Binder cumulant method. The cumulant is defined by

$$U_L \equiv 1 - \frac{\langle E^4 \rangle}{3\langle E^2 \rangle^2}. \tag{15.60}$$

It can be shown that the minimum value of U_L is

$$U_{L,\min} = \frac{2}{3} - \frac{1}{3}\left(\frac{E_+^2 - E_-^2}{2E_+E_-}\right)^2 + O(L^{-d}) \tag{15.61}$$

where E_+ and E_- are the energies of the two phases in a first-order transition. These results are derived by considering the distribution of energy values to be a sum of Gaussians about each phase at the transition, which become sharper and sharper as $L \to \infty$. If $U_{L,\min} = 2/3$ in the infinite size limit, then the transition is continuous.

Problem 15.29. The Binder cumulant and the nature of the transition

(a) Suppose that the energy in a system is given by a Gaussian distribution with a zero mean. What is the corresponding value of U_L?

(b) Consider the two-dimensional Ising model in the absence of a magnetic field and consider the cumulant

$$V_L \equiv 1 - \frac{\langle M^4 \rangle}{3\langle M^2 \rangle^2}. \tag{15.62}$$

Compute V_L for a temperature much higher than T_c. What is the value of V_L? What is the value of V_L at $T = 0$?

(c) Compute V_L for values of T in the range $2.20 \leq T \leq 2.35$ for $L = 10$, 20, and 40. Plot V_L as a function of T for these three values of L. Note that the three curves for V_L cross at a value of T that is approximately T_c. What is the approximate value of V_L at this crossing? Can you conclude that the transition is continuous?

(d) Repeat Problem 15.28 using the Binder cumulant method and determine the nature of the transition. □

15.12 *Other Ensembles

So far, we have considered the microcanonical ensemble (fixed N, V, and E) and the canonical ensemble (fixed N, V, and T). Monte Carlo methods are very flexible and can be adapted to the calculation of averages in any ensemble. Two other ensembles of particular importance are the constant pressure and the grand canonical ensembles. The main difference in the Monte Carlo method is that there are additional moves corresponding to changing the volume or changing the number of particles. The constant pressure ensemble is particularly important for studying first-order phase transitions because the phase transition occurs at a fixed pressure, unlike a constant volume simulation where the system passes through a two phase coexistence region before changing phase completely as the volume is changed.

In the NPT ensemble, the probability of a microstate is proportional to $e^{-\beta(E+PV)}$. For a classical system, the mean value of a physical quantity A that depends on the positions of the particles can be expressed as

$$\langle A \rangle_{\rm NPT} = \frac{\int_0^\infty dV\, e^{-\beta PV} \int d\mathbf{r}_1\, d\mathbf{r}_2 \cdots d\mathbf{r}_N A(\{\mathbf{r}\})\, e^{-\beta U(\{\mathbf{r}\})}}{\int_0^\infty dV\, e^{-\beta PV} \int d\mathbf{r}_1\, d\mathbf{r}_2 \cdots d\mathbf{r}_N\, e^{-\beta U(\{\mathbf{r}\})}}. \tag{15.63}$$

The potential energy $U(\{\mathbf{r}\})$ depends on the set of particle coordinates ($\{\mathbf{r}\}$). To simulate the NPT ensemble, we need to sample the coordinates $\mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_N$ of the particles and the volume V of the system. For simplicity, we assume that the central cell is a square or a cube so that $V = L^d$. It is convenient to use the set of scaled coordinates $\{\mathbf{s}\}$, where $\mathbf{s}_i$ is defined as

$$\mathbf{s}_i = \frac{\mathbf{r}_i}{L}. \tag{15.64}$$

If we substitute (15.64) into (15.63), we can write $\langle A \rangle_{\rm NPT}$ as

$$\langle A \rangle_{\rm NPT} = \frac{\int_0^\infty dV\, e^{-\beta PV} V^N \int d\mathbf{s}_1\, d\mathbf{s}_2 \cdots d\mathbf{s}_N A(\{\mathbf{s}\})\, e^{-\beta U(\{\mathbf{s}\})}}{\int_0^\infty dV\, e^{-\beta PV} V^N \int d\mathbf{s}_1\, d\mathbf{s}_2 \cdots d\mathbf{s}_N\, e^{-\beta U(\{\mathbf{s}\})}} \tag{15.65}$$

where the integral over $\{\mathbf{s}\}$ is over the unit square (cube). The factor of V^N arises from the change of variables $\mathbf{r} \to \mathbf{s}$. If we let $V^N = e^{\ln V^N} = e^{N \ln V}$, we see that the quantity that is analogous to the Boltzmann factor can be written as

$$e^{-W} = e^{-\beta PV - \beta U(\{\mathbf{s}\}) + N \ln V}. \tag{15.66}$$

Because the pressure is fixed, a trial configuration is generated from the current configuration by either randomly displacing a particle or making a random change in the volume, for example, $V \to V + \delta(2r - 1)$, where r is a uniform random number in the unit interval and δ is the maximum change in volume. The trial configuration is accepted if the change $\Delta W \leq 0$ and with probability $e^{-\Delta W}$ if $\Delta W > 0$. It is not necessary or efficient to change the volume after every Monte Carlo step per particle.

In the grand canonical or μVT ensemble, the chemical potential μ is fixed and the number of particles fluctuates. The average of any function of the particle positions can be written (in three dimensions) as

$$\langle A \rangle_{\mu\mathrm{VT}} = \frac{\sum\limits_{N=0}^{\infty} (1/N!)\, \lambda^{-3N}\, e^{\beta\mu N} \int d\mathbf{r}_1 d\mathbf{r}_2 \cdots d\mathbf{r}_N A(\{\mathbf{r}\})\, e^{-\beta U_N(\{\mathbf{r}\})}}{\sum\limits_{N=0}^{\infty} (1/N!)\, \lambda^{-3N} e^{\beta\mu N} \int d\mathbf{r}_1 d\mathbf{r}_2 \cdots d\mathbf{r}_N\, e^{-\beta U_N(\{\mathbf{r}\})}} \tag{15.67}$$

where $\lambda = (h^2/2\pi mkT)^{1/2}$. We have made the N-dependence of the potential energy U explicit. If we write $1/N! = e^{-\ln N!}$ and $\lambda^{-3N} = e^{-N \ln \lambda^3}$, we can write the quantity that is analogous to the Boltzmann factor as

$$e^{-W} = e^{\beta\mu N - N \ln \lambda^3 - \ln N! + N \ln V - \beta U_N}. \tag{15.68}$$

If we write the chemical potential as

$$\mu = \mu^* + kT \ln(\lambda^3/V), \tag{15.69}$$

then W can be expressed as

$$e^{-W} = e^{-\beta\mu^* N - \ln N! - \beta U_N}. \tag{15.70}$$

There are two possible ways of obtaining a trial configuration. The first involves the displacement of a selected particle; such a move is accepted or rejected according to the usual criteria, that is, by the change in the potential energy U_N. In the second possible way, we choose with equal probability whether to attempt to add a particle at a randomly chosen position in the central cell or to remove a particle that is already present. In either case, the trial configuration is accepted if W in (15.70) is increased. If W is decreased, the change is accepted with a probability equal to

$$\frac{1}{N+1} e^{\beta\left(\mu^* - (U_{N+1} - U_N)\right)} \qquad \text{(insertion)}, \tag{15.71a}$$

or

$$N e^{-\beta\left(\mu^* + (U_{N-1} - U_N)\right)} \qquad \text{(removal)}. \tag{15.71b}$$

In this approach μ^* is an input parameter, and μ is not determined until the end of the calculation when $\langle N \rangle_{\mu\mathrm{VT}}$ is obtained.

As we have discussed, the probability that a system at a temperature T has energy E is given by [see (15.53)]

$$P(E, \beta) = \frac{g(E)\, e^{-\beta E}}{Z}. \tag{15.72}$$

If the density of states $g(E)$ was known, we could calculate the mean energy (and other thermodynamic quantities) at any temperature from the relation

$$\langle E\rangle = \frac{1}{Z}\sum_E E g(E) e^{-\beta E}. \tag{15.73}$$

Hence, the density of states is a quantity of much interest.

Suppose that we were to try to compute $g(E)$ by doing a random walk in energy space by flipping the spins at random and accepting all configurations that are obtained. Then the histogram of the energy $H(E)$ the number of visits to each possible energy E of the system, would converge to $g(E)$ if the walk visited all possible configurations. In practice, it would be impossible to realize such a long random walk given the extremely large number of configurations. For example, the Ising model on a 10×10 square lattice has $2^{100} \approx 1.3 \times 10^{30}$ spin configurations.

The main difficulty of doing a simple random walk to determine $g(E)$ is that the walk would spend most of its time visiting the same energy values over and over again and would not reach the values of E that are less probable. The idea of the *Wang–Landau* algorithm is to do a random walk in energy space by flipping single spins at random and to accept the changes with a probability that is proportional to the reciprocal of the density of states. That is, energy values that would be visited often using a simple random walk would be visited less often because they have a bigger density of states. There is only one problem—we don't know the density of states. We will see that the Wang–Landau algorithm estimates the density of states at the same time that it does a random walk in phase space. For simplicity, we discuss the algorithm in the context of the Ising model for which E is a discrete variable.

1. Start with an initial arbitrary configuration of spins and a guess for the density of states. The simplest guess is to set $g(E) = 1$ for all possible energies E.

2. Choose a spin at random and make a trial flip. Compute the energy before the flip, E_1, and after, E_2, and accept the change with probability

$$p(E_1 \to E_2) = \min\left(\frac{g(E_1)}{g(E_2)}, 1\right), \tag{15.74}$$

   Equation (15.74) implies that if $g(E_2) \leq g(E_1)$, the state with energy E_2 is always accepted; otherwise, it is accepted with probability $g(E_1)/g(E_2)$. That is, the state with energy E_2 is accepted if a random number $r \leq g(E_1)/g(E_2)$.

3. Suppose that after step (2) the energy of the system is E. (E is E_2 if the change is accepted or remains at E_1 if the change is not accepted.) Then

$$g(E) = f g(E) \tag{15.75}$$

$$H(E) = H(E) + 1. \tag{15.76}$$

   That is, we multiply the current value of $g(E)$ by the modification factor $f > 1$, and we update the existing entry for $H(E)$ in the energy histogram. Because $g(E)$ becomes very large, in practice we must work with the logarithm of the density of states so that $\ln(g(E))$ will fit into double precision numbers. Therefore, each update of the density of states is implemented as $\ln(g(E)) \to \ln(g(E)) + \ln(f)$, and the ratio of the density of states is computed as $\exp[\ln(g(E_1)) - \ln(g(E_2))]$.

4. A reasonable choice of the initial modification factor is $f = f_0 = e^1 \simeq 2.71828\ldots$. If f_0 is too small, the random walk will need a very long time to reach all possible energies; however, too large a choice of f_0 will lead to large statistical errors.

5. Proceed with the random walk in energy space until a "flat" histogram $H(E)$ is obtained, that is, until all the possible energy values are visited an approximately equal number of times. If the histogram was truly flat, all the possible energies would have been visited an equal number of times. Of course, it is impossible to obtain a perfectly flat histogram, and we will say that $H(E)$ is flat when $H(E)$ for all possible E is not less than p of the average histogram $\langle H(E)\rangle$; p is chosen according to the size and the complexity of the system and the desired accuracy of the density of states. For the two-dimensional Ising model on small lattices, p can be chosen to be as high as 0.95, but for large systems the criterion for flatness may never be satisfied if p is too close to unity.

6. Once the flatness criterion has been satisfied, reduce the modification factor f using a function such as $f_1 = \sqrt{f_0}$, reset the histogram to $H(E) = 0$ for all values of E, and begin the next iteration of the random walk during which the density of states is modified by f_1 at each step. The density of states is not reset during the simulation. We continue performing the random walk until the histogram $H(E)$ is again flat.

7. Reduce the modification factor $f_{i+1} = \sqrt{f_i}$, reset the histogram to $H(E) = 0$ for all values of E, and continue the random walk. Stop the simulation when f is smaller than a predefined value (such as $f_{\rm final} = \exp(10^{-8}) \approx 1.00000001$). The modification factor acts as a control parameter for the accuracy of the density of states during the simulation and also determines how many Monte Carlo sweeps are necessary for the entire simulation.

At the end of the simulation, the algorithm provides only a relative density of states. To determine the normalized density of states $g_n(E)$, we can either use the fact that the total number of states for the Ising model is $\sum_E g(E) = 2^N$ or that the number of ground states (for which $E = -2N$) is 2. The latter normalization guarantees the accuracy of the density of states at low energies, which is important in the calculation of thermodynamic quantities at low temperature. If we apply the former condition, we cannot guarantee the accuracy of $g(E)$ for energies at or near the ground state, because the rescaling factor is dominated by the maximum density of states. We can use one of these two normalization conditions to obtain the absolute density of states and use the other normalization condition to check the accuracy of our result.

Problem 15.30. Sampling the density of states

(a) Implement the Wang–Landau algorithm for the two-dimensional Ising model for $L = 4$, 8, and 16. For simplicity, choose $p = 0.8$ as your criterion for flatness. How many Monte Carlo steps per spin are needed for each iteration? Determine the density of states and describe its qualitative dependence on E.

(b) Compute $P(E) = g(E)e^{-\beta E}/Z$ for different temperatures for the $L = 16$ system. If $T = 0.1$, what range of energies will contribute to the specific heat? What is the range of relevant energies for $T = 1.0$, $T = T_c$, and $T = 4.0$?

(c) Use the density of states that you computed in part (a) to compute the mean energy, the specific heat, the free energy, and the entropy as a function of temperature. Compare your results to your results for $\langle E\rangle$ and C that you found using the Metropolis algorithm in Problem 15.16.

(d) Use the Wang–Landau algorithm to determine the density of states for the one-dimensional Ising model. In this case you can compare your computed values of $g(E)$ to the exact answer:

$$g(E) = 2\frac{N!}{i!(N-i)!} \tag{15.77}$$

where $E = 2i - N$, $i = 0, 2, \ldots, N$, and N is even. How does the accuracy of the computed values of $g(E)$ depend on the choice of p for the flatness criterion? (Exact results are available for $g(E)$ for the two-dimensional Ising model as well, but no explicit combinatorial formula exists. See the article by Beale.)

(e)* The results that you have obtained so far have probably convinced you that the Wang–Landau algorithm is ideal for simulating a variety of systems with many degrees of freedom. What about critical slowing down? Does the Wang–Landau algorithm overcome this limitation of other single spin flip algorithms? To gain some insight, we ask, given the exact $g(E)$, how efficiently does the Wang–Landau sample the different values of E? Use either the exact density of states in two dimensions computed by Beale or the approximate one that you computed in part (a) and set $f = 1$. Because the system is doing a random walk in energy space, it is reasonable to compute the diffusion constant of the random walker in energy space:

$$D_E(t) = \langle [E(t) - E(0)]^2 \rangle / t \tag{15.78}$$

where t is the time difference, and the choice of the time origin is arbitrary. The idea is to find the dependence of D on the energy E of the system at a particular time origin. How long does it take the system to return to this energy? Run for a sufficiently long time so that D_E is independent of t. Plot D_E as a function of E. Where is D a maximum? If time permits, determine D_E at the energy E_c corresponding to the critical temperature. How does D_{E_c} depend on L? □

15.13 More Applications

You are probably convinced that Monte Carlo methods are powerful, flexible, and applicable to a wide variety of systems. Extensions to the Monte Carlo methods that we have not discussed include multiparticle moves, biased moves where particles tend to move in the direction of the force on them, bit manipulation for Ising-like models, and the use of multiple processors to update different parts of a large system simultaneously. We also have not described the simulation of systems with long-range potentials such as Coulombic systems and dipole-dipole interactions. For these potentials, it is necessary to include the interactions of the particles in the center cell with the infinite set of periodic images.

We conclude this chapter with a discussion of Monte Carlo methods in a context that might seem to have little in common with the types of problems we have discussed. This context is called *multivariate* or *combinatorial optimization*, a fancy way of saying, "How do you find the global minimum of a function that depends on many parameters?" Problems of this type arise in many areas of scheduling and design as well as in physics, biology, and chemistry. We explain the nature of this type of problem for the *traveling salesman problem*, although we would prefer to call it the traveling peddler or traveling salesperson problem.

Suppose that a salesman wishes to visit N cities and follow a route such that no city is visited more than once and the end of the trip coincides with the beginning. Given these constraints, the problem is to find the optimum route such that the total distance traveled is a minimum. An

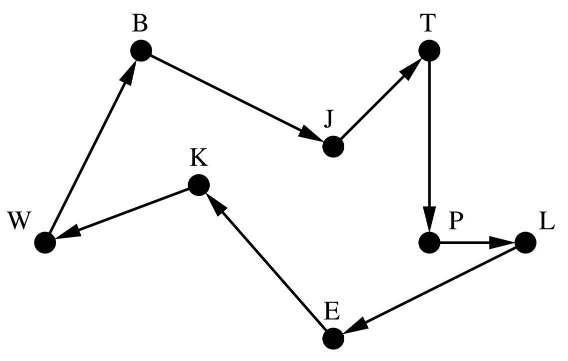


Figure 15.7: What is the optimum route for this random arrangement of $N = 8$ cities? The route begins and ends at city W. A possible route is shown.

example of $N = 8$ cities and a possible route is shown in Figure 15.7. All known exact methods for determining the optimal route require a computing time that increases as e^N, and hence, in practice, an exact solution can be found only for a small number of cities. (The traveling salesman problem belongs to a large class of problems known as NP-complete. The NP refers to nondeterministic polynomial. Such problems cannot be done in a time proportional to a finite polynomial in N on standard computers, though polynomial time algorithms are known for hypothetical nondeterministic (quantum) computers.) What is a reasonable estimate for the maximum number of cities that you can consider without the use of a computer?

To understand the nature of the different approaches to the traveling salesman problem, consider the plot in Figure 15.8 of the "energy" or "cost" function $E(a)$. We can associate $E(a)$ with the length of the route and interpret a as a parameter that represents the order in which the cities are visited. If $E(a)$ has several local minima, what is a good strategy for finding the global (absolute) minimum of $E(a)$? One way is to vary a systematically and find the value of E everywhere. This way corresponds to an exact enumeration method and would mean knowing the length of each possible route, an impossible task if the number of cities is too large. Another way is to use a *heuristic method*, that is, an approximate method for finding a route that is close to the absolute minimum. One strategy is to choose a value of a, generate a small random change δa, and accept this change if $E(a + \delta a)$ is less than or equal to $E(a)$. This iterative improvement strategy corresponds to a search for steps that lead downhill (see Figure 15.8). Because this strategy usually leads to a local and not a global minimum, it is useful to begin from several initial choices of a and to keep the best result. What would be the application of this type of strategy to the salesman problem?

Because we cannot optimize the path exactly when N becomes large, we have to be satisfied with solving the optimization problem approximately and finding a relatively good local minimum. To understand the motivation for the *simulated annealing* algorithm, consider a seemingly unrelated problem. Suppose we wish to make a perfect single crystal. You might know that we should start with the material at a high temperature at which the material is a liquid melt and then gradually lower the temperature. If we lower the temperature too quickly (a rapid quench), the resulting crystal would have many defects or not become a crystal at all. The gradual lowering of the temperature is known as *annealing*.

The method of annealing can be used to estimate the minimum of $E(a)$. We choose a value of a, generate a small random change δa, and calculate $E(a + \delta a)$. If $E(a + \delta a)$ is less than or equal to $E(a)$, we accept the change. However, if $\Delta E = E(a + \delta a) - E(a) > 0$, we accept the change

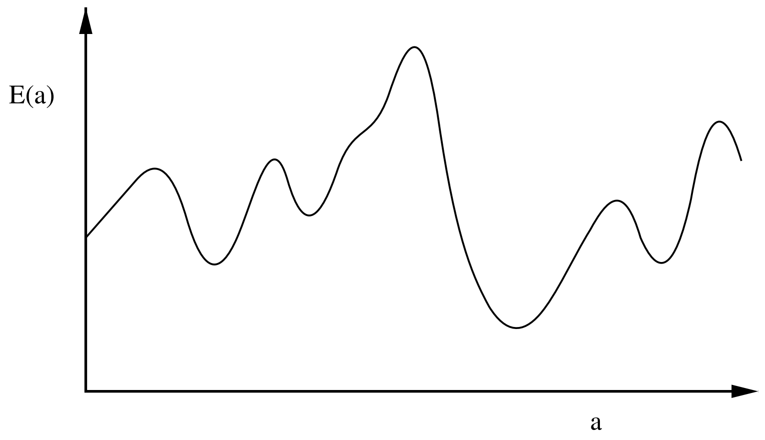


Figure 15.8: Plot of the function $E(a)$ as a function of the parameter a.

with a probability $p = e^{-\Delta E/T}$, where T is an effective temperature. This procedure is the familiar Metropolis algorithm with the temperature playing the role of a control parameter. The *simulated annealing* process consists of first choosing a value for T for which most moves are accepted and then gradually lowering the temperature. At each temperature, the simulation should last long enough for the system to reach quasiequilibrium. The annealing schedule, that is, the rate of temperature decrease, determines the quality of the solution.

The idea is to allow moves that result in solutions of worse quality than the current solution (uphill moves) in order to escape from local minima. The probability of doing such a move is decreased during the search. The slower the temperature is lowered, the higher the chance of finding the optimum solution, but the longer the run time. The effective use of simulated annealing depends on finding a annealing schedule that yields good solutions without taking too much time. It has been proven that if the cooling rate is sufficiently slow, the absolute (global) minimum will eventually be reached. The bounds for "sufficiently slow" depend on the properties of the search landscape (the nature of $E(a)$) and are exceeded for most problems of interest. However, simulated annealing is usually superior to conventional heuristic algorithms.

The moral of the simulated annealing method is that sometimes it is necessary to climb a hill to reach a valley. The first application of the method of simulated annealing was to the optimal design of computers. In Problem 15.31 we apply this method to the traveling salesman problem.

Problem 15.31. Simulated annealing and the traveling salesman problem

(a) Generate a random arrangement of $N = 8$ cities in a square of linear dimension $L = \sqrt{N}$ and calculate the optimum route by hand. Then write a Monte Carlo program and apply the method of simulated annealing to this problem. For example, use two arrays to store the x- and y coordinate of each city and an array to store the distances between them. The state of the system, that is, the route represented by a sequence of cities, can be stored in another array. The length of this route is associated with the energy of an imaginary thermal system. A reasonable choice for the initial temperature is one that is the same order as the initial energy. One way to generate a random rearrangement of the route is to choose two cities at random and to interchange the order of visit. Choose this method or one that you devise and find a reasonable annealing schedule. Compare your annealing results to exact results whenever possible. Extend your results to larger N, for example, $N = 12$, 24, and 48. For a given annealing schedule, determine the probability of finding a route of a given length. More suggestions can be found in the references.

(b) The microcanonical Monte Carlo algorithm (demon) discussed in Section 15.3 can also be used to do simulated annealing. The advantages of the demon algorithm are that it is deterministic and allows large temperature fluctuations. One way to implement the analog of simulated annealing is to impose a maximum value on the energy of the demon, $E_{d,\max}$, which is gradually decreased. Guo et al. choose $E_{d,\max}$ to be initially equal to $\sqrt{N}/4$. Their results are comparable to the usual simulated annealing method but require approximately half the CPU time. Apply this method to the same city positions that you considered in part (a) and compare your results. □

15.14 Projects

Many of the original applications of Monte Carlo methods were done for systems of approximately one hundred particles and lattices of order 32^2 spins. It would be instructive to redo many of these applications with much better statistics and with larger system sizes. In the following, we discuss some additional recent developments, but we have omitted other important topics such as Brownian dynamics and umbrella sampling. More ideas for projects can be found in the references.

Project 15.32. Overcoming critical slowing down

The usual limiting factor of most simulations is the speed of the computer. Of course, one way to overcome this problem is to use a faster computer. Near a continuous phase transition, the most important limiting factor on even the fastest available computers is the existence of critical slowing down (see Problem 15.19). In this project we discuss the nature of critical slowing down and ways of overcoming it in the context of the Ising model.

As we have mentioned, the existence of critical slowing down is related to the fact that the size of the correlated regions of spins becomes very large near the critical point. The large size of the correlated regions and the corresponding divergent behavior of the correlation length ξ near T_c implies that the time τ required for a region to lose its coherence becomes very long if a *local* dynamics is used. At $T = T_c$, $\tau \sim L^z$ for $L \gg 1$. For single spin flip algorithms, $z \approx 2$ and τ becomes very large for $L \gg 1$. On a serial computer, the CPU time needed to obtain n configurations increases as L^2, the time needed to visit L^2 spins. This factor of L^2 is expected and not a problem because a larger system contains proportionally more information. However, the time needed to obtain n approximately *independent* configurations is of order $\tau L^2 \sim L^{2+z} \approx L^4$ for the Metropolis algorithm. We conclude that an increase of L by a factor of 10 requires 10^4 more computing time. Hence, the existence of critical slowing down limits the maximum value of L that can be considered.

If we are interested only in the static properties of the Ising model, the choice of dynamics is irrelevant as long as the transition probability satisfies the detailed balance condition (15.18). It is reasonable to look for a *global* algorithm for which groups or *clusters* of spins are flipped simultaneously. We are already familiar with cluster properties in the context of percolation (see Chapter 12). A naive definition of a cluster of spins might be a domain of parallel nearest neighbor spins. We can make this definition explicit by introducing a bond between any two nearest neighbor spins that are parallel. The introduction of a bond between parallel spins

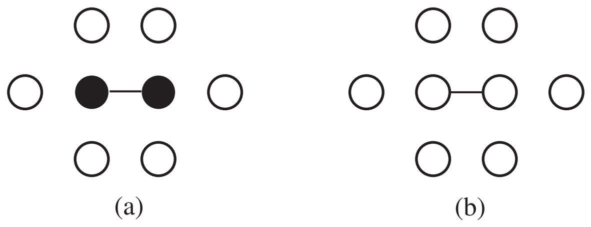


Figure 15.9: (a) A cluster of two up spins. (b) A cluster of two down spins. The filled and open circles represent the up and down spins, respectively. Note the bond between the two spins in the cluster. Adapted from Newman and Barkema.

defines a site-bond percolation problem. More generally, we may assume that such a bond exists with probability p and that this bond probability depends on the temperature T.

The dependence of p on T can be determined by requiring that the percolation transition of the clusters occurs at the Ising critical point and by requiring that the critical exponents associated with the clusters be identical to the analogous thermal exponents. For example, we can define a critical exponent ν_p to characterize the divergence of the connectedness length of the clusters near p_c. The analogous thermal exponent ν quantifies the divergence of the thermal correlation length ξ near T_c. We will argue in the following that these (and other) critical exponents are identical if we define the bond probability as

$$p = 1 - e^{-2J/kT} \qquad \text{(bond probability)}. \tag{15.79}$$

The relation (15.79) holds for any spatial dimension. What is the value of p at $T = T_c$ for the two-dimensional Ising model on the square lattice?

A simple argument for the temperature dependence of p in (15.79) is as follows. Consider the two configurations in Figure 15.9 which differ from one another by the flip of the cluster of two spins. In Figure 15.9(a) the six nearest neighbor spins of the cluster are in the opposite direction and, hence, are not part of the cluster. Thus, the probability of this configuration with a cluster of two spins is $p\,e^{-\beta J} e^{6\beta J}$, where p is the probability of a bond between the two up spins, $e^{-\beta J}$ is proportional to the probability that these two spins are parallel, and $e^{6\beta J}$ is proportional to the probability that the six nearest neighbors are antiparallel. In Figure 15.9(b) the cluster spins have been flipped, and the possible bonds between the cluster spins and its nearest neighbors have to be "broken." The probability of this configuration with a cluster of two (down) spins is $p(1-p)^6 e^{-\beta J} e^{-6\beta J}$, where the factor of $(1-p)^6$ is the probability that the six nearest neighbor spins are not part of the cluster. Because we want the probability that a cluster is flipped to be unity, we need to have the probability of the two configurations and their corresponding clusters be the same. Hence, we must have

$$pe^{\beta J} e^{-6\beta J} = p(1-p)^6 e^{\beta J} e^{6\beta J}, \tag{15.80}$$

or $(1-p)^6 = e^{-12\beta J}$. It is straightforward to solve for p and obtain the relation (15.79).

Now that we know how to generate clusters of spins, we can use these clusters to construct a global dynamics instead of only flipping one spin at a time as in the Metropolis algorithm. The idea is to grow a single (site-bond) percolation cluster in a way that is analogous to the single (site) percolation cluster algorithm discussed in Section 13.1. The algorithm can be implemented by the following steps:

(i) Choose a seed spin at random. Its four nearest neighbor sites (on the square lattice) are the perimeter sites. Form an ordered array corresponding to the perimeter spins that are parallel to the seed spin and define a counter for the total number of perimeter spins.

(ii) Choose the first spin in the ordered perimeter array. Remove it from the array and replace it by the last spin in the array. Generate a random number r. If $r \leq p$, the bond exists between the two spins, and the perimeter spin is added to the cluster.

(iii) If the spin is added to the cluster, inspect its parallel perimeter spins. If any of these spins are not already a part of the cluster, add them to the end of the array of perimeter spins.

(iv) Repeat steps (ii) and (iii) until no perimeter spins remain.

(v) Flip all the spins in the single cluster.

This algorithm is known as single cluster flip or *Wolff* dynamics. Note that bonds, rather than sites, are tested so that a spin might have more than one chance to join a cluster. In the following, we consider both the static and dynamical properties of the two-dimensional Ising model using the Wolff algorithm to generate the configurations.

(a) Modify your program for the Ising model on a square lattice so that single cluster flip dynamics (the Wolff algorithm) is used. Compute the mean energy and magnetization for $L = 16$ as a function of T for $T = 2.0$ to 2.7 in steps of 0.1. Compare your results to those obtained using the Metropolis algorithm. How many cluster flips do you need to obtain comparable accuracy at each temperature? Is the Wolff algorithm more efficient at every temperature near T_c?

(b) Fix T at the critical temperature of the infinite lattice ($T_c = 2/\ln(1 + \sqrt{2})$) and use finite size scaling to estimate the values of the various static critical exponents, for example, γ and α. Compare your results to those obtained using the Metropolis algorithm.

(c) Because we are generating site-bond percolation clusters, we can study their geometrical properties as we did for site percolation. For example, measure the distribution sn_s of cluster sizes at $p = p_c$ (see Problem 13.3). How does n_s depend on s for large s (see Project 13.15)? What is the fractal dimension of the clusters in the Ising model at $T = T_c$?

(d) The natural unit of time for single cluster flip dynamics is the number of cluster flips $t_{\rm cf}$. Measure $C_M(t_{\rm cf})$ and/or $C_E(t_{\rm cf})$ and estimate the corresponding correlation time $\tau_{\rm cf}$ for $T =$ 2.5, 2.4, 2.3, and T_c for $L = 16$. As discussed in Problem 15.19, $\tau_{\rm cf}$ can be found from the relation, $\tau_{\rm cf} = \sum_{t_{\rm cf}=1} C(t_{\rm cf})$. The sum is cut–off at the first negative value of $C(t_{\rm cf})$. Estimate the value of $z_{\rm cf}$ from the relation $\tau_{\rm cf} = L^{z_{\rm cf}}$.

(e) To compare our results for the Wolff algorithm to our results for the Metropolis algorithm, we should use the same unit of time. Because only a fraction of the spins are updated at each cluster flip, the time $t_{\rm cf}$ is not equal to the usual unit of time, which corresponds to an update of the entire lattice or one Monte Carlo step per spin. We have that τ measured in Monte Carlo steps per spin is related to $\tau_{\rm cf}$ by $\tau = \tau_{\rm cf}\langle c\rangle/L^2$, where $\langle c\rangle$ is the mean number of spins in the single clusters, and L^2 is the number of spins in the entire lattice. Verify that the mean cluster size scales as $\langle c\rangle \sim L^{\gamma/\nu}$ with $\gamma = 7/4$ and $\nu = 1$. (The quantity $\langle c\rangle$ is the same quantity as the mean cluster size S defined in Chapter 12. The exponents characterizing the divergence of the various properties of the clusters are identical to the analogous thermal exponents.)

(f) To obtain the value of z that is directly comparable to the value found for the Metropolis algorithm, we need to rescale the time as in part (e). We have that $\tau \sim L^z \propto L^{z_{\rm cf}} L^{\gamma/\nu} L^{-d}$. Hence, z is related to the measured value of $z_{\rm cf}$ by $z = z_{\rm cf} - (d - \gamma/\nu)$. What is your estimated value of z? (It has been estimated that $z_{\rm cf} \approx 0.50$ for the $d = 2$ Ising model, which would imply that $z \approx 0.25$.)

(g) One of the limitations of the usual implementation of the Metropolis algorithm is that only one spin is flipped at a time. However, there is no reason why we could not choose f spins at random, compute the change in energy ΔE for flipping these f spins, and accepting or rejecting the trial move in the usual way according to the Boltzmann probability. Explain why this generalization of the Metropolis algorithm would be very inefficient, especially if $f >> 1$. We conclude that the groups of spins to be flipped must be chosen with the physics of the system in mind and not simply at random. □

Another cluster algorithm is to assign all bonds between parallel spins with probability p. As usual, no bonds are included between sites that have different spin orientations. From this configuration of bonds, we can form clusters of spins using one of the cluster identification algorithms we discussed in Chapter 12. The smallest cluster contains a single spin. After the clusters have been identified, all the spins in each cluster are flipped with probability 1/2. This algorithm is known as the *Swendsen-Wang* algorithm and preceded the Wolff algorithm. Because the Wolff algorithm is easier to program and gives a smaller value of z than the Swendsen-Wang algorithm for the $d = 3$ and $d = 4$ Ising models, the Wolff algorithm is more commonly used.

Project 15.33. Invaded cluster algorithm

In Problem 13.7 we found that invasion percolation is an example of a self-organized critical phenomenon. In this cluster growth algorithm, random numbers are independently assigned to the bonds of a lattice. The growth starts from the seed sites of the left-most column. At each step the cluster grows by the occupation of the perimeter bond with the smallest random number. The growth continues until the cluster satisfies a stopping condition. We found that if we stop adding sites when the cluster is comparable in extent to the linear dimension L, then the fraction of bonds that are occupied approaches the percolation threshold p_c as $L \to \infty$. The invaded percolation algorithm automatically finds the percolation threshold!

Machta and co-workers have used this idea to find the critical temperature of a spin system without knowing its value in advance. For simplicity, we will discuss their algorithm in the context of the Ising model, although it can be easily generalized to the q-state Potts model (see the references). Consider a lattice on which there is a spin configuration $\{s_i\}$. The bonds of the lattice are assigned a random order. Bonds (i, j) are tested in this assigned order to see if s_i is parallel to s_j. If so, the bond is occupied and spins i and j are a part of the same cluster. Otherwise, the bond is not occupied and is not considered for the remainder of the current Monte Carlo step. The set of occupied bonds partitions the lattice into clusters of connected sites. The clusters can be found using the Newman–Ziff algorithm (see Section 12.3). The cluster structure evolves until a stopping condition is satisfied. Then a new spin configuration is obtained by flipping each cluster with probability 1/2, thus completing one Monte Carlo step. The fraction f of bonds that were occupied during the growth process and the energy of the system are measured. The bonds are then randomly reordered and the process begins again. Note that the temperature is not an input parameter.

If open boundary conditions are used, the appropriate stopping rule is that a cluster spans the lattice (see Chapter 12, page 450). For periodic boundary conditions, the spanning rule discussed in Project 12.17 is appropriate.

Write a program to simulate the invaded cluster algorithm for the Ising model on the square lattice. Start with all spins up and determine how many Monte Carlo steps are needed for equilibration. How does this number compare to that required by the Metropolis algorithm at the critical temperature for the same value of L? An estimate for the critical temperature can be found from the relation (15.79) with f corresponding to p.

After you are satisfied that your program is working properly, determine the dependence of the critical temperature on the concentration c of nonmagnetic impurities. That is, randomly place nonmagnetic impurities on a fraction c of the sites. □

Project 15.34. Physical test of random number generators

In Section 7.9 we discussed various statistical tests for the quality of random number generators. In this project we will find that the usual statistical tests might not be sufficient for determining the quality of a random number generator for a particular application. The difficulty is that the quality of a random number generator for a specific application depends in part on how the subtle correlations that are intrinsic to all deterministic random number generators couple to the way that the random number sequences are used. In this project we explore the quality of two random number generators when they are used to implement single spin flip dynamics (the Metropolis algorithm) and single cluster flip dynamics (Wolff algorithm) for the two-dimensional Ising model.

(a) Write methods to generate sequences of random numbers based on the linear congruential algorithm

$$x_n = 16807\, x_{n-1} \bmod (2^{31} - 1), \tag{15.81}$$

and the generalized feedback shift register (GFSR) algorithm

$$x_n = x_{n-103} \oplus x_{n-250}. \tag{15.82}$$

In both cases x_n is the nth random number. Both algorithms require that x_n be divided by the largest possible value of x_n to obtain numbers in the range $0 \le x_n < 1$. The GFSR algorithm requires bit manipulation. Which random number generator does a better job of passing the various statistical tests discussed in Problem 7.35?

(b) Use the Metropolis algorithm and the linear congruential random number generator to determine the mean energy per spin E/N and the specific heat (per spin) C for the $L = 16$ Ising model at $T = T_c = 2/\ln(1 + \sqrt{2})$. Make ten independent runs (that is, ten runs that use different random number seeds) and compute the standard deviation of the means σ_m from the ten values of E/N and C, respectively. Published results by Ferrenberg, Landau, and Wong are for 10^6 Monte Carlo steps per spin for each run. Calculate the differences δ_e and δ_c between the average of E/N and C over the ten runs and the exact values (to five decimal places), $E/N = -1.45306$ and $C = 1.49871$. If the ratio δ/σ_m for the two quantities is order unity, then the random number generator does not appear to be biased. Repeat your runs using the GFSR algorithm to generate the random number sequences. Do you find any evidence of statistical bias?

(c) Repeat part (b) using Wolff dynamics. Do you find any evidence of statistical bias?

(d) Repeat the computations in parts (b) and (c) using the random number generator supplied with your programming language. □

Project 15.35. Nucleation and the Ising model

(a) Equilibrate the two-dimensional Ising model at $T = 4T_c/9$ and $B = 0.3$ for a system with $L \geq 50$. What is the equilibrium value of m? Then flip the magnetic field so that it points down, that is, $B = -0.3$. Use the Metropolis algorithm and plot m as a function of the time t (the number of Monte Carlo steps per spin). What is the qualitative behavior of $m(t)$? Does it fluctuate about a positive value for a time long enough to determine various averages? If so, the system can be considered to have been in a *metastable state*. Watch the spins evolve for a time before m changes sign. Visually determine a place in the lattice where a "droplet" of the stable phase (down spins) first appears and then grows. Change the random number seed and rerun the simulation. Does the droplet appear in the same spot at the same time? Can the magnitude of the field be increased further, or is there an upper bound above which a metastable state is not well defined?

(b) As discussed in Project 15.32, we can define clusters of spins by placing a bond with probability p between parallel spins. In this case there is an external field and the proper definition of the clusters is more difficult. For simplicity, assume that there is a bond between all nearest–neighbor down spins and find all the clusters of down spins. One way to identify the droplet that initiates the decay of the metastable state is to monitor the number of spins in the largest cluster as a function of time after the quench. At what time does the number of spins in the largest cluster begin to grow quickly? This time is an estimate of the *nucleation time*. Another way of estimating the nucleation time is to follow the evolution of the center of mass of the largest cluster. For early times after the quench, the center of mass position has large fluctuations. However, at a certain time these fluctuations decrease considerably, which is another criterion for the nucleation time. What is the order of magnitude of the nucleation time?

(c) While the system is in a metastable state, clusters of down spins grow and shrink randomly until eventually one of the clusters becomes large enough to grow, nucleation occurs, and the system decays to its stable macroscopic state. The cluster that initiates this decay is called the nucleating droplet. This type of nucleation is due to spontaneous thermal fluctuations and is called *homogeneous nucleation*. Although the criteria for the nucleation time that we used in part (b) are plausible, they are not based on fundamental considerations. From theoretical considerations the nucleating droplet can be thought of as a cluster that just makes it to the top of the saddle point of the free energy that separates the metastable and stable states. We can identify the nucleating droplet by using the fact that a saddle point structure should initiate the decay of the metastable state 50% of the time. The idea is to save the spin configurations at regular intervals at about the time that nucleation is thought to have occurred. We then restart the simulation using a saved configuration at a certain time and use a different random number sequence to flip the spins. If we have intervened at a time such that the largest cluster decays in more than 50% of the trials, then the intervention time (the time at which we changed the random number seed) is before nucleation. Similarly, if less than 50% of the clusters decay, the intervention is after the nucleation time. The nucleating droplet is the cluster that decays in approximately half of the trial interventions. Because we need to do a number of interventions (usually in the range 20–100) at different times, the intervention method is much more CPU intensive than the other criteria. However, it has the advantage that it has a sound theoretical basis. Redo some of the simulations that you did in part (b) and compare the different estimates of the nucleation time. What is the nature and size of the nucleating droplet? If time permits, determine the probability that the system nucleates at time t for a given quench depth. (Measure the time t after the flip of the field.)

(d) *Heterogeneous nucleation* occurs in nature because of the presence of impurities, defects, or walls. One way of simulating heterogeneous nucleation in the Ising model is to fix a certain number of spins in the direction of the stable phase (down). For simplicity, choose the impurity to be five spins in the shape of a + sign. What is the effect of the impurity on the lifetime of the metastable state? What is the probability of droplet growth on and off the impurity as a function of quench depth B?

(e) The questions raised in parts (b)–(d) become even more interesting when the interaction between the spins extends beyond nearest neighbors. Assume that a given spin interacts with all spins that are within a distance R with an interaction strength of $4J/q$, where q is the number of spins within the interaction range R. (Note that $q = 4$ for nearest neighbor interactions on the square lattice.) A good choice is $R = 10$, although your preliminary simulations should be for smaller R. How does the value of T_c change as R is increased? □

Project 15.36. The *n*-fold way: Simulations at low temperature

Monte Carlo simulations become very inefficient at low temperatures because almost all trial configurations will be rejected. For example, consider an Ising model for which all spins are up, but a small magnetic field is applied in the negative direction. The equilibrium state will have most spins pointing down. Nevertheless, if the magnetic field is small and the temperature is low enough, equilibrium will take a very long time to occur.

What we need is a more efficient way of sampling configurations if the acceptance probability is low. The *n-fold way* algorithm is one such method. The idea is to accept more low probability configurations but to weight them appropriately. If we use the usual Metropolis rule, then the probability of flipping the ith spin is

$$p_i = \min\left[1, e^{-\Delta E/kT}\right]. \tag{15.83}$$

One limitation of the Metropolis algorithm is that it becomes very inefficeint if the probabilities p_i are very small. If we sum over all the spins, then we can define the total weight

$$Q = \sum_i p_i. \tag{15.84}$$

The idea is to choose a spin to flip (with probability one) by computing a random number r_Q between 0 and Q and finding spin i that satisfies the condition:

$$\sum_{k=0}^{i-1} p_k \le r_Q < \sum_{k=0}^{i} p_k. \tag{15.85}$$

There are two more ingredients we need to make this algorithm practical. We need to determine how long a configuration would remain unchanged if we had used the Metropolis algorithm. Also, the algorithm would be very inefficient because on average the computation of which spin to flip from (15.85) would take $O(N)$ computations. This second problem can be easily overcome by realizing that there are only a few possible values of p_i. For example, for the Ising model on a square lattice in a magnetic field, there are only $n = 10$ possible values of p_i. Thus, instead of (15.85), we have

$$\sum_{\alpha=0}^{i-1} n_\alpha p_\alpha \le r_Q < \sum_{\alpha=0}^{i} n_\alpha p_\alpha \tag{15.86}$$

where α labels one of the n possible values of p_i or classes, and n_α is the number of spins in class α. Hence, instead of $O(N)$ calculations, we need to perform only $O(n)$ calculations. Once we know which class we have chosen, we can randomly flip one of the spins in that class.

Next we need to determine the time spent in a configuration. The probability in one Metropolis Monte Carlo step of choosing a spin at random is $1/N$, and the probability of actually flipping that spin is p_i, which is given by (15.83). Thus, the probability of flipping any spin is

$$\frac{1}{N}\sum_{i=0}^{N-1} p_i = \frac{1}{N}\sum_{\alpha=0}^{n-1} n_\alpha p_\alpha = \frac{Q}{N}. \tag{15.87}$$

The probability of not flipping any spin is $q \equiv 1 - Q/N$, and the probability of not flipping after s steps is q^s. Thus, if we generate a random number r between 0 and 1, the time s in Monte Carlo steps per spin to remain in the current configuration will be determined by solving

$$q^{s-1} \le r < q^s. \tag{15.88}$$

If $Q/N << 1$, then both sides of (15.88) are approximately equal, and we can approximate s by

$$s \approx \frac{\ln r}{\ln q} = \frac{\ln r}{\ln(1 - Q/N)} \approx -\frac{N}{Q}\ln r. \tag{15.89}$$

That is, we would have to wait s Monte Carlo steps per spin on the average before we would flip a spin using the Metropolis algorithm. Note that the random number r in (15.88) and (15.89) should not be confused with the random number r_Q in (15.86).

The n-fold algorithm can be summarized by the following steps:

(i) Start with an initial configuration and determine the class to which each spin belongs. Store all the possible values of p_i in an array. Compute Q. Store in an array the number of spins in class α, n_α.

(ii) Determine s from (15.89). Accumulate any averages, such as the energy and magnetization weighted by s. Also, accumulate the total time `tTotal += s`.

(iii) Choose a class of spin using (15.86) and randomly choose which spin in the chosen class to flip.

(iv) Update the classes of the chosen spin and its four neighbors.

(v) Repeat steps (ii)–(iv).

To conveniently carry out step (iv), set up the following arrays: `spinClass[i]` returns the class of the ith spin, `spinInClass[k][alpha]` returns the kth spin in class α, and `spinIndex[i]` returns the value of k for the ith spin to use in the array `spinInClass[k][alpha]`. If we define the local field of a spin by the sum of the fields of its four neighbors, then this local field can take on the values $\{-4, -2, 0, 2, 4\}$. The ten classes correspond to these five local field values and the center spin equal to -1 plus these five local field values and the center spin equal to $+1$. If we order these ten classes from 0 to 9, then the class of a spin that is flipped changes by $+5$ mod 10, and the class of a neighbor changes by the new spin value equal to ± 1.

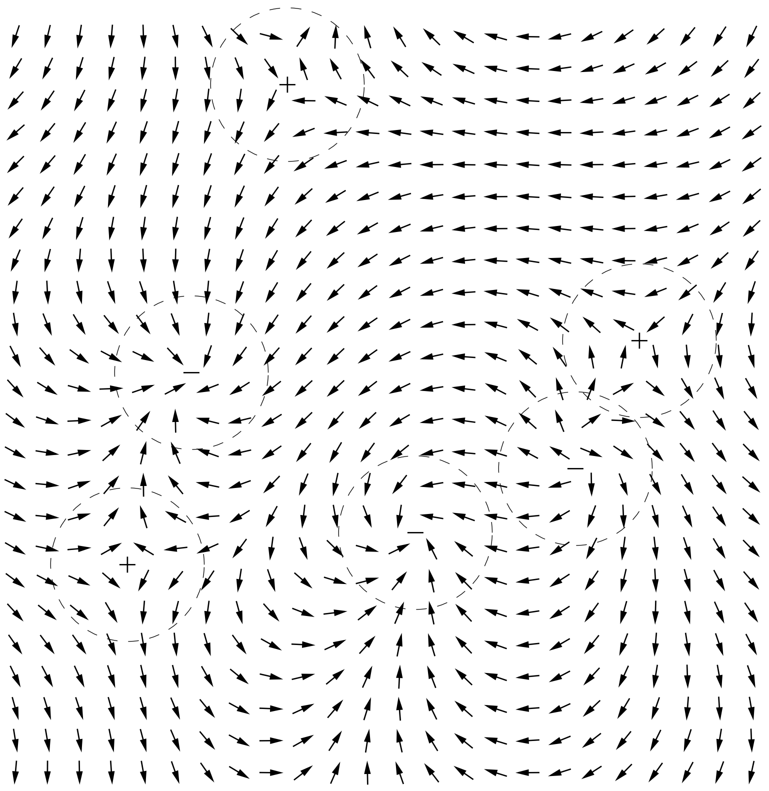


Figure 15.10: A typical configuration of the planar model on a 24×24 square lattice that has been quenched from $T = \infty$ to $T = 0$ and equilibrated for 200 Monte Carlo steps per spin after the quench. Note that there are six vortices. The circle around each vortex is a guide to the eye and is not meant to indicate the size of the vortex.

(a) Write a program to implement the n-fold way algorithm for the Ising model on a square lattice with an applied magnetic field. Check your program by comparing various averages at a few temperatures with the results from your program using the Metropolis algorithm.

(b) Choose the magnetic field $B = -0.5$ at the temperature $T = 1$. Begin with an initial configuration of all spins up and use the n-fold way to estimate how long it takes before the majority of the spins flip. Do the same simulation using the Metropolis algorithm. Which algorithm is more efficient?

(c) Repeat part (b) for other temperature and field values. For what conditions is the n-fold way algorithm more efficient than the standard Metropolis algorithm?

(d) Repeat part (b) for different values of the magnetic field and plot the number of Monte Carlo steps needed to flip the spins as a function of $1/|B|$ for values of B from 0 to ≈ 3. Average over at least 10 starting configurations for each field value. □

Project 15.37. The Kosterlitz–Thouless transition

The planar model (also called the x-y model) consists of spins of unit magnitude that can point in any direction in the x-y plane. The energy or Hamiltonian function of the planar model in zero magnetic field can be written as

$$E = -J \sum_{i,j=nn(i)} [s_{i,x}s_{j,x} + s_{i,y}s_{j,y}] \tag{15.90}$$

where $s_{i,x}$ represents the x-component of the spin at the ith site, J measures the strength of the interaction, and the sum is over all nearest neighbors. We can rewrite (15.90) in a simpler form by substituting $s_{i,x} = \cos\theta_i$ and $s_{i,y} = \sin\theta_i$. The result is

$$E = -J \sum_{i,j=nn(i)} \cos(\theta_i - \theta_j) \tag{15.91}$$

where θ_i is the angle that the ith spin makes with the x- axis. The most studied case is the two-dimensional model on a square lattice. In this case the mean magnetization $\langle \mathbf{M} \rangle = 0$ for all temperatures $T > 0$, but, nevertheless, there is a phase transition at a nonzero temperature T_{KT}, which is known as the Kosterlitz–Thouless (KT) transition. For $T \leq T_{\mathrm{KT}}$, the spin-spin correlation function $C(r)$ decreases as a power law; for $T > T_{\mathrm{KT}}$, $C(r)$ decreases exponentially. The power law decay of $C(r)$ for $T \leq T_{\mathrm{KT}}$ implies that every temperature below T_{KT} acts as if it was a critical point. We say that the planar model has a line of critical points. In the following, we explore some of the properties of the planar model and the mechanism that causes the transition.

(a) Write a program that uses the Metropolis algorithm to simulate the planar model on a square lattice using periodic boundary conditions. Because θ and hence the energy of the system is a continuous variable, it is not possible to store the previously computed values of the Boltzmann factor for each possible value of ΔE. Instead of computing $e^{-\beta\Delta E}$ for each trial change, it is faster to set up an array `w` such that the array element `w(j)` $= e^{-\beta\Delta E}$, where `j` is the integer part of $1000\Delta E$. This procedure leads to an energy resolution of 0.001, which should be sufficient for most purposes.

(b) One way to show that the magnetization $\langle \mathbf{M} \rangle$ vanishes for all T is to compute $\langle \theta^2 \rangle$, where θ is the angle that a spin makes with the magnetization $\mathbf{M}$ for a given configuration. (Although the mean magnetization vanishes, $\mathbf{M} \neq 0$ at any given time.) Compute $\langle \theta^2 \rangle$ as a function of the number of spins N at $T = 0.1$ and show that $\langle \theta^2 \rangle$ diverges as $\ln N$. Begin with a 4×4 lattice and choose the maximum change in θ_i to be $\Delta\theta_{\max} = 1.0$. If necessary, change $\theta_{\max}$ so that the acceptance probability is about 40%. If $\langle \theta^2 \rangle$ diverges, then the fluctuations in the direction of the spins diverges, which implies that there is no preferred direction for the spins, and hence the mean magnetization vanishes.

(c) Modify your program so that an arrow is drawn at each site to show the orientation of each spin. You can use the `Vector2DFrame` to draw a lattice of arrows. Look at a typical configuration and analyze it visually. Begin with a 32×32 lattice with spins pointing in random directions and do a temperature quench to $T = 0.5$. (Simply change the value of β in the Boltzmann probability.) Such a quench should lock in some long lived but metastable vortices. A vortex is a region of the lattice where the spins rotate by at least 2π as your eye moves around a closed path (see Figure 15.10). To determine the center of a vortex, choose a group of four spins that are at the corners of a unit square and determine whether the spins rotate by $\pm 2\pi$ as your eye goes from one spin to the next in a counterclockwise direction around the square. Assume that the difference between the direction of two neighboring spins $\delta\theta$ is in the range $-\pi < \delta\theta < \pi$. A total rotation of $+2\pi$ indicates the existence of a positive vortex, and a change of -2π indicates a negative vortex. Count the number of positive and negative vortices. Repeat these observations for several configurations. What can you say about the number of vortices of each sign?

(d) Write a method to determine the existence of a vortex for each 1×1 square of the lattice. Represent the center of the vortices using a different symbol to distinguish between a positive and a negative vortex. Do a Monte Carlo simulation to compute the mean energy, the specific heat, and number of vortices in the range from $T = 0.5$ to $T = 1.5$ in steps of 0.1. Use the last configuration at the previous temperature as the first configuration for the next temperature. Begin at $T = 0.5$ with all $\theta_i = 0$. Draw the vortex locations for the last configuration at each temperature. Use at least 1000 Monte Carlo steps per spin at each temperature to equilibrate and at least 5000 Monte Carlo steps per spin for computing the averages. Use an 8×8 or 16×16 lattice if your computer resources are limited and larger lattices if you have sufficient resources. Describe the T-dependence of the energy, the specific heat, and the vorticity (equal to the number of vortices per unit area). Plot the logarithm of the vorticity versus T for $T < 1.1$. What can you conclude about the T-dependence of the vorticity? Explain why this form is reasonable. Describe the vortex configurations. At what temperature do you find a vortex that appears to be free, that is, a vortex that is not obviously paired with another vortex of opposite sign?

(e) The Kosterlitz–Thouless theory predicts that the susceptibility χ diverges above the transition as

$$\chi \sim A\, e^{b/\epsilon^\nu} \tag{15.92}$$

where ϵ is the reduced temperature $\epsilon = (T - T_{\rm KT})/T_{\rm KT}$, $\nu = 0.5$, and A and b are nonuniversal constants. Compute χ from the relation (15.21) with $\mathbf{M} = 0$. Assume the exponential form (15.92) for χ in the range $T = 1$ and $T = 1.2$ with $\nu = 0.7$ and find the best values of $T_{\rm KT}$, A, and b. (Although theory predicts $\nu = 0.5$, simulations for small systems indicate that $\nu = 0.7$ gives a better fit.) One way to determine $T_{\rm KT}$, A, and b is to assume a value of $T_{\rm KT}$ and then do a least squares fit of $\ln \chi$ to determine A and b. Choose the set of parameters that minimizes the variance of $\ln \chi$. How does your estimated value of $T_{\rm KT}$ compare with the temperature at which free vortices first appear? At what temperature does the specific heat have a peak? The Kosterlitz–Thouless theory predicts that the specific heat peak does not occur at $T_{\rm KT}$. This prediction has been confirmed by simulations (see Tobochnik and Chester). To obtain quantitative results, you will need lattices larger than 32×32. □

Project 15.38. The classical Heisenberg model in two dimensions

The energy or Hamiltonian of the classical Heisenberg model is similar to the Ising model and the planar model, except that the spins can point in any direction in three dimensions. The energy in zero external magnetic field is

$$E = -J \sum_{i,j=\mathrm{nn}(i)}^{N} \mathbf{s}_i \cdot \mathbf{s}_j = -J \sum_{i,j=\mathrm{nn}(i)}^{N} [s_{i,x}s_{j,x} + s_{i,y}s_{j,y} + s_{i,z}s_{j,z}] \tag{15.93}$$

where $\mathbf{s}$ is a classical vector of unit length. The spins have three components, in contrast to the spins in the Ising model which only have one component and the spins in the planar model which have two components.

We will consider the two-dimensional Heisenberg model for which the spins are located on a two-dimensional lattice. Early simulations and approximate theories led researchers to believe that there was a continuous phase transition, similar to that found in the Ising model. The Heisenberg model received more interest after it was related to quark confinement. Lattice models of the interaction between quarks, called lattice gauge theories, predict that the confinement of quarks could be explained if there are no phase transitions in these models. (The lack of a phase transition in these models implies that the attraction between quarks grows with distance.) The two-dimensional Heisenberg model is an analog of the four-dimensional models used to model quark-quark interactions. Shenker and Tobochnik used a combination of Monte Carlo and renormalization group methods to show that this model does not have a phase transition. Subsequent work on lattice gauge theories showed similar behavior.

(a) Modify your Ising model program to simulate the Heisenberg model in two dimensions. One way to do so is to define three arrays, one for each of the three components of the unit spin vectors. A trial Monte Carlo move consists of randomly changing the direction of a spin, $\mathbf{s}_i$. First compute a small vector $\Delta\mathbf{s} = \Delta s_{\text{max}}(q_1, q_2, q_3)$, where $-1 \leq q_n \leq 1$ is a uniform random number, and Δs_{max} is the maximum change of any spin component. If $|\Delta\mathbf{s}| > \Delta s_{\text{max}}$, compute another $\Delta\mathbf{s}$. This latter step is necessary to insure that the change in a spin direction is symmetrically distributed around the current spin direction. Then let the trial spin equal $\mathbf{s}_i + \Delta\mathbf{s}$ normalized to a unit vector. The standard Metropolis algorithm can now be used to determine if the trial spin is accepted. Compute the mean energy, the specific heat, and the susceptibility as a function of T. Choose lattice sizes of $L = 8$, 16, 32, and larger, if possible, and average over at least 2000 Monte Carlo steps per spin at each temperature. Is there any evidence of a phase transition? Does the susceptibility appear to diverge at a nonzero temperature? Plot the logarithm of the susceptibility versus the inverse temperature and determine the temperature dependence of the susceptibility in the limit of low temperatures.

(b) Use the Lee–Kosterlitz analysis at the specific heat peak to determine if there is a phase transition. □

Project 15.39. Domain growth kinetics

When the Ising model is quenched from a high temperature to very low temperatures, domains of the ordered low temperature phase typically grow with time as a power law $R \sim t^{\alpha}$, where R is a measure of the average linear dimension of the domains. A simple measure of the domain size is the perimeter length of a domain which can be computed from the energy per spin ϵ, and is given by

$$R = \frac{2}{2+\epsilon}. \tag{15.94}$$

Equation (15.94) can be motivated by the following argument. Imagine a region of N spins made up of a domain of up spins with a perimeter size R embedded in a sea of down spins. The total energy of this region is $-2N + 2R$, where for each spin on the perimeter, the energy is increased by 2 because one of the neighbors of a perimeter spin will be of opposite sign. The energy per spin is $\epsilon = -2 + 2R/N$. Because N is of order R^2, we arrive at the result given in (15.94).

(a) Modify your Ising model program so that the initial configuration is random, that is, a typical high temperature configuration. Write a target class to simulate a quench of the system. The input parameters should be the lattice size, the quench temperature (use 0.5 initially), the maximum time (measured in Monte Carlo steps per spin) for each quench, and the number of Monte Carlo steps between drawing the lattice. Plot $\ln\langle R\rangle$ versus $\ln t$ after each quench is finished, where t is measured from the time of the quench.

(b) Choose $L = 64$ and a maximum time of 128 mcs. Averages over 10 quenches will give acceptable results. What value do you obtain for α? Repeat for other temperatures and system sizes. Does the exponent change? Run for a longer maximum time to check your results.

(c) Modify your program to simulate the q-state Potts model. Consider various values of q. Do your results change? Results for large q and large system sizes are given in Grest et al.

(d)* Modify your program to simulate a three-dimensional system. How should you modify (15.94)? Are your results similar? □

Project 15.40. Ground state energy of the Ising spin glass

A spin glass is a magnetic system with frozen-in disorder. An example of such a system is the Ising model with the exchange constant J_{ij} between nearest neighbor spins randomly chosen to be ± 1. The disorder is said to be "frozen-in" because the set of interactions $\{J_{ij}\}$ does not change with time. Because the spins cannot arrange themselves so that every pair of spins is in its lowest energy state, the system exhibits frustration similar to the antiferromagnetic Ising model on a triangular lattice (see Problem 15.22). Is there a phase transition in the spin glass model, and if so, what is its nature? The answers to these questions are very difficult to obtain by doing simulations. One of the difficulties is that we need to do not only an average over the possible configurations of spins for a given set of $\{J_{ij}\}$, but also an average over different realizations of the interactions. Another difficulty is that there are many local minima in the energy (free energy at finite temperature) as a function of the configurations of spins, and it is very difficult to find the global minimum. As a result, Monte Carlo simulations typically become stuck in these local minima or metastable states. Detailed finite-size scaling analyses of simulations indicate that there might be a transition in three dimensions. It is generally accepted that the transition in two dimensions is at zero temperature. In the following, we will look at some of the properties of an Ising spin glass on a square lattice at low temperatures.

(a) Write a program to apply simulated annealing to an Ising spin glass using the Metropolis algorithm with the temperature fixed at each stage of the annealing schedule (see Problem 15.31a). Search for the lowest energy configuration for a fixed set of $\{J_{ij}\}$. Use at least one other annealing schedule for the same $\{J_{ij}\}$ and compare your results. Then find the ground state energy for at least ten other sets of $\{J_{ij}\}$. Use lattice sizes of $L = 5$ and $L = 10$. Discuss the nature of the ground states you are able to find. Is there much variation in the ground state energy E_0 from one set of $\{J_{ij}\}$ to another? Theoretical calculations give an average over realizations of $\overline{E_0}/N \approx -1.4$. If you have sufficient computer resources, repeat your computations for the three-dimensional spin glass.

(b) Modify your program to do simulated annealing using the demon algorithm (see Problem 15.31b). How do your results compare to those that you found in part (a)? □

Project 15.41. Zero temperature dynamics of the Ising model

We have seen that various kinetic growth models (Section 13.3) and reaction-diffusion models (Section 7.8) lead to interesting and nontrivial behavior. Similar behavior can be seen in the zero temperature dynamics of the Ising model. Consider the one-dimensional Ising model with $J > 0$ and periodic boundary conditions. The initial orientation of the spins is chosen at random. We update the configurations by choosing a spin at random and computing the change in energy ΔE. If $\Delta E < 0$, then flip the spin; else if $\Delta E = 0$, flip the spin with 50% probability. The spin is not flipped if $\Delta E > 0$. This type of Monte Carlo update is known as Glauber dynamics. How does this algorithm differ from the Metropolis algorithm at $T = 0$?

(a) A quantity of interest is $f(t)$, the fraction of spins that have not yet flipped at time t. As usual, the time is measured in terms of Monte Carlo steps per spin. Published results (Derrida et al.) for $N = 10^5$ indicate that $f(t)$ behaves as

$$f(t) \sim t^{-\theta}, \tag{15.95}$$

for $t \approx 3$ to $t \approx 10,000$. The exact value of θ is 0.375. Verify this result and extend your results to the one-dimensional q-state Potts model. In the latter model each site is initially given a random integer between 1 and q. A site is chosen at random and set equal to either of its two neighbors with equal probability.

(b) Another interesting quantity is the probability distribution $P_n(t)$ that n sites have not yet flipped as a function of the time t (see Das and Sen). Plot P_n versus n for two times on the same graph. Discuss the shape of the curves and their differences. Choose $L \geq 100$ and $t = 50$ and 100. Try to fit the curves to a Gaussian distribution. Because the possible values of n are bounded, fit each side of the maximum of P_n to a Gaussian with different widths. There are a number of scaling properties that can be investigated. Show that $P_{n=0}(t)$ scales approximately as t/L^2. Thus, if you compute $P_{n=0}(t)$ for a number of different times and lengths such that t/L^2 has the same value, you should obtain the same value of $P_{n=0}$. □

Project 15.42. The inverse power law potential

Consider the inverse power law potential

$$V(r) = V_0\left(\frac{\sigma}{r}\right)^n, \tag{15.96}$$

with $V_0 > 0$. One reason for the interest in potentials of this form is that thermodynamic quantities such as the mean energy E do not depend on V_0 and σ separately, but depend on a single dimensionless parameter, which is defined as (see Project 8.25)

$$\Gamma = \frac{V_0}{kT}\frac{\sigma}{a} \tag{15.97}$$

where a is defined in three and two dimensions by $4\pi a^3\rho/3 = 1$ and $\pi a^2\rho = 1$, respectively. The length a is proportional to the mean distance between particles. A Coulomb interaction corresponds to $n = 1$, and a hard sphere system corresponds to $n \to \infty$. What phases do you expect to occur for arbitrary n?

(a) Compare the qualitative features of $g(r)$ for a "soft" potential with $n = 4$ to a system of hard disks at the same density.

(b) Let $n = 12$ and compute the mean energy E as a function of Γ for a three-dimensional system with $N = 16$, 32, 64, and 128. Does E depend on N? Can you extrapolate your results for the N-dependence of E to $N \to \infty$? Do you see any evidence of a fluid-solid phase transition? If so, estimate the value of Γ at which it occurs. What is the nature of the transition if it exists? What is the symmetry of the ground state?

(c) Let $n = 4$ and determine the symmetry of the ground state. For this value of n, there is a solid-to-solid phase transition at which the solid changes symmetry. To determine the value of Γ at which this phase transition exists and the symmetry of the smaller Γ solid phase (see Dubin and Dewitt), it is necessary to use a Monte Carlo method in which the shape of the simulation cell changes to accomodate the different symmetry (the Rahman–Parrinello method), an interesting project. An alternative is to prepare a bcc lattice at $\Gamma =\approx 105$ (for example, $T = 0.06$ and $\rho = 0.95$). Then instantaneously change the potential from $n = 4$ to $n = 12$; the new value of Γ is ≈ 4180, and the new stable phase is fcc. The transition can be observed by watching the evolution of $g(r)$. □

Project 15.43. Rare gas clusters

There has been much recent interest in structures that contain many particles but that are not macroscopic. An example is the unusual structure of sixty carbon atoms known as a "buckeyball." A less unusual structure is a cluster of argon atoms. Questions of interest include the structure of the clusters, the existence of "magic" numbers of particles for which the cluster is particularly stable, the temperature dependence of the quantities, and the possibility of different phases. This latter question has been subject to some controversy because transitions between different kinds of behavior in finite systems are not well defined, as they are for infinite systems.

(a) Write a Monte Carlo program to simulate a three-dimensional system of particles interacting via the Lennard–Jones potential. Use open boundary conditions; that is, do not enclose the system in a box. The number of particles N and the temperature T should be input parameters.

(b) Find the ground state energy E_0 as a function of N. For each value of N begin with a random initial configuration and accept any trial displacement that lowers the energy. Repeat for at least ten different initial configurations. Plot E_0/N versus N for $N = 2$ to 20 and describe the qualitative dependence of E_0/N on N. Is there any evidence of magic numbers, that is, value(s) of N for which E_0/N is a minimum? For each value of N save the final configuration. Plot the positions of the atoms. Does the cluster look like a part of a crystalline solid?

(c) Repeat part (b) using simulated annealing. The initial temperature should be sufficiently low so that the particles do not move far away from each other. Slowly lower the temperature according to some annealing schedule. Are your results for E_0/N lower than those you obtained in part (b)?

(d) To gain more insight into the structure of the clusters, compute the mean number of neighbors per particle for each value of N. What is a reasonable criteria for two particles to be neighbors? Also compute the mean distance between each pair of particles. Plot both quantities as a function of N and compare their dependence on N with your plot of E_0/N.

(e) Do you find any evidence for a "melting" transition? Begin with the configuration that has the minimum value of E_0/N and slowly increase the temperature T. Compute the energy per particle and the mean square displacement of the particles from their initial positions. Plot your results for these quantities versus T. □

Project 15.44. The hard disks fluid-solid transition

Although we have mentioned (see Section 15.10) that there is much evidence for a fluid-solid transition in a hard disk system, the nature of the transition still is a problem of current research. In this project we follow the work of Lee and Strandburg and apply the constant pressure Monte Carlo method (see Section 15.12) and the Lee–Kosterlitz method (see Section 15.11) to investigate the nature of the transition. Consider $N = L^2$ hard disks of diameter $\sigma = 1$ in a two-dimensional box of volume $V = \sqrt{3}L^2 v/2$ with periodic boundary conditions. The quantity

$v \geq 1$ is the reduced volume and is related to the density ρ by $\rho = N/V = 2/(\sqrt{3}v)$; $v = 1$ corresponds to maximum packing. The aspect ratio of $2/\sqrt{3}$ is used to match the perfect triangular lattice. Do a constant pressure (actually constant $p^* = P/kT$) Monte Carlo simulation. The trial displacement of each disk is implemented as discussed in Section 15.10. Lee and Strandburg find that a maximum displacement of 0.09 gives a 45% acceptance probability. The other type of move is a random isotropic change of the volume of the system. If the change of the volume leads to an overlap of the disks, the change is rejected. Otherwise, if the trial volume $\tilde{V}$ is less than the current volume V, the change is accepted. A larger trial volume is accepted with probability

$$e^{-p^*(\tilde{V}-V)+N\ln(\tilde{V}/V)}. \tag{15.98}$$

Volume changes are attempted 40–200 times for each set of individual disk moves. The quantity of interest is $N(v)$, the distribution of the reduced volume v. Because we need to store information about $N(v)$ in an array, it is convenient to discretize the volume in advance and choose the mesh size so that the acceptance probability for changing the volume by one unit is 40–50%. Do a Monte Carlo simulation of the hard disk system for $L = 10$ ($N = 100$) and $p^* = 7.30$. Published results are for 10^7 Monte Carlo steps. To apply the Lee–Kosterlitz method, smooth $\ln N(v)$ by fitting it to an eighth-order polynomial. Then extrapolate $\ln N(v)$ using the histogram method to determine $p_c^*(L = 10)$, the pressure at which the two peaks of $N(v)$ are of equal height. What is the value of the free energy barrier ΔF? If sufficient computer resources are available, compute ΔF for larger L (published results are for $L = 10$, 12, 14, 16, and 20) and determine if ΔF depends on L. Can you reach any conclusions about the nature of the transition? □

Project 15.45. Vacancy mediated dynamics in binary alloys

When a binary alloy is rapidly quenched from a high temperature to a low temperature unstable state, a pattern of domain formation called *spinodal decomposition* takes place as the two metals in the alloy separate. This process is of much interest experimentally. Lifshitz and Slyozov have predicted that at long times, the linear domain size increases with time as $R \sim t^{1/3}$. This result is independent of the dimension for $d \geq 2$, and has been verified experimentally and in computer simulations. The behavior is modified for binary fluids due to hydrodynamic effects.

Most of the computer simulations of this growth process have been based on the Ising model with spin exchange dynamics. In this model there is an A or B atom (spin up or spin down) at each site, where A and B represent different metals. The energy of interaction between atoms on two neighboring sites is $-J$ if the two atoms are the same type and $+J$ if they are different. Monte Carlo moves are made by exchanging unlike atoms. (The number of A and B atoms must be conserved.) A typical simulation begins with an equilibrated system at high temperatures. Then the temperature is changed instantaneously to a low temperature below the critical temperature T_c. If there are equal numbers of A and B atoms on the lattice, then spinodal decomposition occurs. If you watch a visualization of the evolution of the system, you will see wavy-like domains of each type of atom thickening with time.

The growth of the domains is very slow if we use spin exchange dynamics. We will see that if simulations are performed with vacancy mediated dynamics, the scaling behavior begins at much earlier times. Because of the large energy barriers that prevent real metallic atoms from exchanging position, it is likely that spinodal decomposition in real alloys also occurs with vacancy mediated dynamics. We can do a realistic simulation by including just one vacancy because the number of vacancies in a real alloy is also very small. In this case the only possible Monte Carlo move on a square lattice is to exchange the vacancy with one of its four neighboring atoms. To implement this algorithm, you will need an array to keep track of which type of atom is at each lattice site and variables to keep track of the location of the single vacancy. The simulation will run very fast because there is little bookkeeping and all the possible trial moves are potentially good ones. In contrast, in standard spin exchange dynamics, it is necessary to either waste computer time checking for unlike nearest neighbor atoms or keep track of where they are.

The major quantity of interest is the growth of the domain size R. One way to determine R is to measure the pair correlation function $C(r) = \langle s_i s_j \rangle$, where $r = |\mathbf{r}_i - \mathbf{r}_j|$, and $s_i = 1$ for an A atom and $s_i = -1$ for a B atom. The first zero in $C(r)$ is a measure of the domain size. An alternative measure of the domain size is the quantity $R = 2/(\langle E \rangle/N + 2)$, where $\langle E \rangle/N$ is the average energy per site and N is the number of sites (see Project 15.39). The quantity R is a rough measure of the length of the perimeter of a domain and is proportional to the domain size.

(a) Write a program to simulate vacancy mediated dynamics. The initial state consists of the random placement of A and B atoms (half of the sites have A and half B atoms); one vacancy replaces one of the atoms. Explain why this configuration corresponds to infinite temperature. Choose a square lattice with $L \geq 50$.

(b) Instantaneously quench the system by running the Metropolis algorithm at a temperature of $T = T_c/2 \approx 1.13$. You should first look at the lattice after every attempted move of the vacancy to see the effect of vacancy dynamics. After you are satisfied that your program is working correctly and that you understand the algorithm, speed up the simulation by only collecting data and showing the lattice at times equal to $t = 2^n$ where $n = 1, 2, 3 \ldots$. Measure the domain size using either the energy or $C(r)$ as a function of time averaged over many different initial configurations.

(c) At what time does the $\log R$ versus $\log t$ plot become linear? Do both measures of the domain size give the same results? Does the behavior change for different quench temperatures? Try $0.2T_c$ and $0.7T_c$. A log-log plot of the domain size versus time should give the exponent $1/3$.

(d) Repeat the measurements in three dimensions. Do you obtain the same exponent? □

Project 15.46. Heat flow using the demon algorithm

In our applications of the demon algorithm one demon shared its energy equally with all the spins. As a result the spins all attained the same mean energy of interaction. Many interesting questions arise when the system is not spatially uniform and is in a nonequilibrium but time-independent (steady) state.

Let us consider heat flow in a one-dimensional Ising model. Suppose that instead of all the sites sharing energy with one demon, each site has its own demon. We can study the flow of heat by requiring the demons at the boundary spins to satisfy different conditions than the demons at the other spins. The demon at spin 0 adds energy to the system by flipping this spin so that it is in its highest energy state, that is, in the opposite direction of spin 1. The demon at spin $N - 1$ removes energy from the system by flipping spin $N - 1$ so that it is in its lowest energy state, that is, in the same direction as spin $N - 2$. As a result, energy flows from site 0 to site $N - 1$ via the demons associated with the intermediate sites. In order that energy not build up at the "hot" end of the Ising chain, we require that spin 0 can only add energy to the system if spin $N - 1$ simultaneously removes energy from the system. Because the demons at the two ends of the lattice satisfy different conditions than the other demons, we do not use periodic boundary conditions.

The temperature is determined by the generalization of the relation (15.10); that is, the temperature at site i is related to the mean energy of the demon at site i. To control the temperature gradient, we can update the end spins at a different rate than the other spins. The maximum temperature gradient occurs if we update the end spins after every update of an internal spin. A smaller temperature gradient occurs if we update the end spins less frequently. The temperature gradient between any two spins can be determined from the temperature profile, the spatial dependence of the temperature. The energy flow can be determined by computing the magnitude of the energy per unit time that enters the lattice at site 0.

To implement this procedure we modify `IsingDemon` by converting the variables `demonEnergy` and `demonEnergyAccumulator` to arrays. We do the usual updating procedure for spins 1 through $N-2$ and visit spins 0 and $N-1$ at regular intervals denoted by `timeToAddEnergy`. The class `ManyDemons` can be downloaded from the ch15 directory.

(a) Write a target class that inputs the number of spins N and the initial energy of the system, outputs the number of Monte Carlo steps per spin and the energy added to the system at the high temperature boundary, and plots the temperature as a function of position.

(b) As a check on `ManyDemons`, modify the class so that all the demons are equivalent; that is, impose periodic boundary conditions and do not use method `boundarySpins`. Compute the mean energy of the demon at each site and use (15.10) to define a local site temperature. Use $N \geq 52$ and run for about 10,000 mcs. Is the local temperature approximately uniform? How do your results compare with the single demon case?

(c) In `ManyDemons` the energy is added to the system at site 0 and is removed at site $N-1$. Determine the mean demon energy for each site and obtain the corresponding local temperature and the mean energy of the system. Draw the temperature profile by plotting the temperature as a function of site number. The temperature gradient is the difference in temperature from site $N-2$ to site 1 divided by the distance between them. (The distance between neighboring sites is unity.) Because of local temperature fluctuations and edge effects, the temperature gradient should be estimated by fitting the temperature profile in the middle of the lattice to a straight line. Reasonable choices for the parameters are $N = 52$ and `timeToAddEnergy` = 1. Run for at least 10000 mcs.

(d) The heat flux Q is the energy flow per unit length per unit time. The energy flow is the amount of energy that demon 0 adds to the system at site 0. The time is conveniently measured in terms of Monte Carlo steps per spin. Determine Q for the parameters used in part (c).

(e) If the temperature gradient $\partial T/\partial x$ is not too large, the heat flux Q is proportional to $\partial T/\partial x$. We can determine the *thermal conductivity* κ by the relation

$$Q = -\kappa \frac{\partial T}{\partial x}. \tag{15.99}$$

Use your results for $\partial T/\partial x$ and Q to estimate κ.

(f) Determine Q, the temperature profile, and the mean temperature for different values of `timeToAddEnergy`. Is the temperature profile linear for all values of `timeToAddEnergy`? If the temperature profile is linear, estimate $\partial T/\partial x$ and determine κ. Does κ depend on the mean temperature?

Note that by using many demons we were able to compute a temperature profile by using an algorithm that manipulates only integer numbers. The conventional approach is to solve a heat equation similar in form to the diffusion equation. Now we use the same idea to compute the magnetization profile when the end spins of the lattice are fixed.

(g) Modify `ManyDemons` by not calling `method boundarySpins`. Also, constrain spins 0 and $N-1$ to be $+1$ and -1, respectively. Estimate the magnetization profile by plotting the mean value of the spin at each site versus the site number. Choose $N = 22$ and `mcs` ≥ 1000. How do your results vary as you increase N?

(h) Compute the mean demon energy and, hence, the local temperature at each site. Does the system have a uniform temperature even though the magnetization is not uniform? Is the system in thermal equilibrium?

(i) The effect of the constraint on the end spins is easier to observe in two and three dimensions than in one dimension. Write a program for a two-dimensional Ising model on a $L\times L$ square lattice. Constrain the spins at site (i,j) to be $+1$ and -1 for $i = 0$ and $i = L-1$, respectively. Use periodic boundary conditions in the y direction. How do your results compare with the one-dimensional case?

(j) Remove the periodic boundary condition in the y direction and constrain all the boundary spins from $i = 0$ to $(L/2)-1$ to be $+1$ and the other boundary spins to be -1. Choose an initial configuration where all the spins on the left half of the system are $+1$ and the others are -1. Do the simulation and draw a configuration of the spins once the system has reached equilibrium. Draw a line between each pair of spins of opposite sign. Describe the curve separating the $+1$ spins from the -1 spins. Begin with $L = 20$ and determine what happens as L is increased. □

Appendix 15A: Relation of the Mean Demon Energy to the Temperature

We know that the energy of the demon E_d is constrained to be positive and that the probability for the demon to have energy E_d is proportional to $e^{-E_d/kT}$. Hence, in general, $\langle E_d\rangle$ is given by

$$\langle E_d\rangle = \frac{\sum_{E_d} E_d\, e^{-E_d/kT}}{\sum_{E_d} e^{-E_d/kT}} \tag{15.100}$$

where the summations in (15.100) are over the possible values of E_d. If an Ising spin is flipped in zero magnetic field, the minimum nonzero decrease in energy of the system is $4J$ (see Figure 15.11). Hence, the possible energies of the demon are $0, 4J, 8J, 12J, \ldots$. We write $x = 4J/kT$ and perform the summations in (15.100). The result is

$$\langle E_d/kT\rangle = \frac{0 + xe^{-x} + 2xe^{-2x} + \cdots}{1 + e^{-x} + e^{-2x} + \cdots} = \frac{x}{e^x - 1}. \tag{15.101}$$

The form (15.10) can be obtained by solving (15.101) for T in terms of E_d. Convince yourself that the relation (15.101) is independent of dimension for lattices with an even number of nearest neighbors.

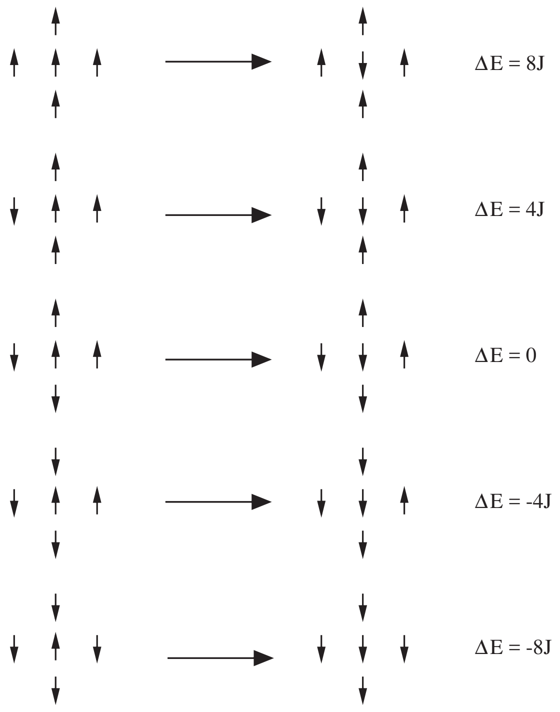


Figure 15.11: The five possible transitions of the Ising model on the square lattice with spin flip dynamics.

If the magnetic field is nonzero, the possible values of the demon energy are 0, $2H$, $4J-2H$, $4J+2H$, ... If J is a multiple of H, then the result is the same as before with $4J$ replaced by $2H$, because the possible energy values for the demon are multiples of $2H$. If the ratio $4J/2H$ is irrational, then the demon can take on a continuum of values, and thus $\langle E_d\rangle = kT$. The other possibility is that $4J/2H = m/n$, where m and n are prime positive integers that have no common factors (other than 1). In this case it can be shown that (see Mak)

$$kT/J = \frac{4/m}{\ln(1+4J/m\langle E_d\rangle)}. \tag{15.102}$$

Surprisingly, (15.102) does not depend on n. Test these relations for $H \neq 0$ by choosing values of J and H and computing the sums in (15.100) directly.

Appendix 15B: Fluctuations in the Canonical Ensemble

We first obtain the relation of the constant volume heat capacity C_V to the energy fluctuations in the canonical ensemble. We write C_V as

$$C_V = \frac{\partial\langle E\rangle}{\partial T} = -\frac{1}{kT^2}\frac{\partial\langle E\rangle}{\partial\beta}. \tag{15.103}$$

From (15.11) we have

$$\langle E\rangle = -\frac{\partial}{\partial\beta}\ln Z, \tag{15.104}$$

and

$$\frac{\partial\langle E\rangle}{\partial\beta} = -\frac{1}{Z^2}\frac{\partial Z}{\partial\beta}\sum_s E_s\,e^{-\beta E_s} - \frac{1}{Z}\sum_s E_s^2\,e^{-\beta E_s} \tag{15.105}$$

$$= \langle E\rangle^2 - \langle E^2\rangle. \tag{15.106}$$

The relation (15.19) follows from (15.103) and (15.106). Note that the heat capacity is at constant volume because the partial derivatives were performed with the energy levels E_s kept constant. The corresponding quantity for a magnetic system is the heat capacity at constant external magnetic field.

The relation of the magnetic susceptibility χ to the fluctuations of the magnetization M can be obtained in a similar way. We assume that the energy can be written as

$$E_s = E_{0,s} - HM_s \tag{15.107}$$

where $E_{0,s}$ is the energy of interaction of the spins in the absence of a magnetic field, H is the external applied field, and M_s is the magnetization in the s state. The mean magnetization is given by

$$\langle M\rangle = \frac{1}{Z}\sum M_s\,e^{-\beta E_s}. \tag{15.108}$$

Because $\partial E_s/\partial H = -M_s$, we have

$$\frac{\partial Z}{\partial H} = \sum_s \beta M_s\,e^{-\beta E_s}. \tag{15.109}$$

Hence, we obtain

$$\langle M\rangle = \frac{1}{\beta}\frac{\partial}{\partial H}\ln Z. \tag{15.110}$$

If we use (15.108) and (15.110), we find

$$\frac{\partial\langle M\rangle}{\partial H} = -\frac{1}{Z^2}\frac{\partial Z}{\partial H}\sum_s M_s\,e^{-\beta E_s} + \frac{1}{Z}\sum_s \beta M_s^2\,e^{-\beta E_s} \tag{15.111}$$

$$= -\beta\langle M\rangle^2 + \beta\langle M^2\rangle. \tag{15.112}$$

The relation (15.21) for the zero-field susceptibility follows from (15.112) and the definition (15.20).

Appendix 15C: Exact Enumeration of the 2×2 Ising Model

Because the number of possible states or configurations of the Ising model increases as 2^N, we can enumerate the possible configurations only for small N. As an example, we calculate the various quantities of interest for a 2×2 Ising model on the square lattice with periodic boundary conditions. In Table 15.2 we group the sixteen states according to their total energy and magnetization.

We can compute all the quantities of interest using Table 15.2. The partition function is given by

$$Z = 2\,e^{8\beta J} + 12 + 2\,e^{-8\beta J}. \tag{15.113}$$

# Spins Up	**g(E,M)**	**Energy**	**Magnetization**
4	1	−8	4
3	4	0	2
2	4	0	0
2	2	8	0
1	4	0	−2
0	1	−8	−4

Table 15.2: The energy and magnetization of the 2^4 states of the zero field Ising model on the 2×2 square lattice. The quantity $g(E,M)$ is the number of microstates with the same energy.

If we use (15.104) and (15.113), we find

$$\langle E \rangle = -\frac{\partial}{\partial \beta} \ln Z = -\frac{1}{Z}\Big[2(8)e^{8\beta J} + 2(-8)e^{-8\beta J}\Big]. \tag{15.114}$$

Because the other quantities of interest can be found in a similar manner, we only give the results:

$$\langle E^2 \rangle = \frac{1}{Z}\Big[(2\times 64)\,e^{8\beta J} + (2\times 64)\,e^{-8\beta J}\Big] \tag{15.115}$$

$$\langle M \rangle = \frac{1}{Z}(0) = 0 \tag{15.116}$$

$$\langle |M| \rangle = \frac{1}{Z}\Big[(2\times 4)\,e^{8\beta J} + 8\times 2\Big] \tag{15.117}$$

$$\langle M^2 \rangle = \frac{1}{Z}\Big[(2\times 16)\,e^{8\beta J} + 8\times 4\Big]. \tag{15.118}$$

The dependence of C and χ on βJ can be found by using (15.114) and (15.115) and (15.116) and (15.118), respectively.

References and Suggestions for Further Reading

M. P. Allen and D. J. Tildesley, *Computer Simulation of Liquids* (Clarendon Press, 1987). See Chapter 4 for a discussion of Monte Carlo methods.

Paul D. Beale, "Exact distribution of energies in the two-dimensional Ising model," Phys. Rev. Lett. **76**, 78 (1996). The author discusses a Mathematica program that can compute the exact density of states for the two-dimensional Ising model.

K. Binder, editor, *Monte Carlo Methods in Statistical Physics*, 2nd ed. (Springer–Verlag, 1986). Also see K. Binder, editor, *Applications of the Monte Carlo Method in Statistical Physics* (pringer–Verlag, 1984) and K. Binder, editor, *The Monte Carlo Method in Condensed Matter Physics* (Springer–Verlag, 1992). The latter book discusses the Binder cumulant method in the introductory chapter.

Marvin Bishop and C. Bruin, "The pair correlation function: A probe of molecular order," Am. J. Phys. **52**, 1106–1108 (1984). The authors compute the pair correlation function for a two-dimensional Lennard–Jones model.

A. B. Bortz, M. H. Kalos and J. L. Lebowitz, "A new algorithm for Monte Carlo simulation of Ising spin systems," J. Comput. Phys. **17**, 10–18 (1975). This paper first introduced the n-fold way algorithm, which was rediscovered independently by many workers in the 1970s and 80s.

S. G. Brush, "History of the Lenz–Ising model," Rev. Mod. Phys. **39**, 883–893 (1967).

James B. Cole, "The statistical mechanics of image recovery and pattern recognition,"Am. J. Phys. **59**, 839–842 (1991). A discussion of the application of simulated annealing to the recovery of images from noisy data.

R. Cordery, S. Sarker, and J. Tobochnik, "Physics of the dynamical critical exponent in one dimension," Phys. Rev. B **24**, 5402–5403 (1981).

Michael Creutz, "Microcanonical Monte Carlo simulation," Phys. Rev. Lett. **50**, 1411 (1983). See also Gyan Bhanot, Michael Creutz, and Herbert Neuberger, "Microcanonical simulation of Ising systems," Nuc. Phys. B **235**, 417–434 (1984).

Pratap Kumar Das and Parongama Sen, "Probability distributions of persistent spins in an Ising chain," J. Phys. A **37**, 7179–7184 (2004).

B. Derrida, A. J. Bray, and C. Godrèche, "Non-trivial exponents in the zero temperature dynamics of the 1D Ising and Potts models," J. Phys. A **27**, L357–L361 (1994); B. Derrida, V. Hakim, and V. Pasquier, "Exact first passage exponents in 1d domain growth: Relation to a reaction-diffusion model," Phys. Rev. Lett. **75**, 751 (1995).

Daniel H. E. Dubin and Hugh Dewitt, "Polymorphic phase transition for inverse-power-potential crystals keeping the first-order anharmonic correction to the free energy," Phys. Rev. B **49**, 3043–3048 (1994).

Jerome J. Erpenbeck and Marshall Luban, "Equation of state for the classical hard-disk fluid," Phys. Rev. A **32**, 2920–2922 (1985). These workers use a combined molecular dynamics/-Monte Carlo method and consider 1512 and 5822 disks.

Alan M. Ferrenberg, D. P. Landau, and Y. Joanna Wong, "Monte Carlo simulations: Hidden errors from "good" random number generators," Phys. Rev. Lett. **69**, 3382 (1992).

Alan M. Ferrenberg and Robert H. Swendsen, "New Monte Carlo technique for studying phase transitions," Phys. Rev. Lett. **61**, 2635 (1988); "Optimized Monte Carlo data analysis," Phys. Rev. Lett. **63**, 1195 (1989); "Optimized Monte Carlo data analysis," Computers in Physics **3** 5, 101 (1989). The second and third papers discuss using the multiple histogram method with data from simulations at more than one temperature.

P. Fratzl and O. Penrose, "Kinetics of spinodal decomposition in the Ising model with vacancy diffusion," Phys. Rev. B **50**, 3477–3480 (1994).

Daan Frenkel and Berend Smit, *Understanding Molecular Simulation*, 2nd ed. (Academic Press, 2002).

Harvey Gould and W. Klein, "Spinodal effects in systems with long-range interactions," Physica D **66**, 61–70 (1993). This paper discusses nucleation in the Ising model and Lennard–Jones systems.

Harvey Gould and Jan Tobochnik, "Overcoming critical slowing down," Computers in Physics **3** (4), 82 (1989).

James E. Gubernatis, *The Monte Carlo Method in the Physical Sciences* (AIP Press, 2004). June 2003 was the 50th anniversary of the Metropolis, Rosenbluth, Rosenbluth, Teller, and Teller publication of what is now called the Metropolis algorithm. This algorithm established the Monte Carlo method in physics and other fields and lead to the development of other Monte Carlo algorithms. Six of the papers in the proceedings of the conference give historical perspectives.

Hong Guo, Martin Zuckermann, R. Harris, and Martin Grant, "A fast algorithm for simulated annealing," Physica Scripta **T38**, 40–44 (1991).

Gary S. Grest, Michael P. Anderson, and David J. Srolovitz, "Domain-growth kinetics for the Q-state Potts model in two and three dimensions," Phys. Rev. B **38**, 4752–4760 (1988).

R. Harris, "Demons at work," Computers in Physics **4** (3), 314 (1990).

S. Istrail, "Statistical mechanics, three-dimensionality and NP-completeness: I. Universality of intractability of the partition functions of the Ising model across non-planar lattices," Proceedings of the 32nd ACM Symposium on the Theory of Computing, ACM Press, pp. 87–96, Portland, Oregon, May 21–23, 2000. This paper shows that it is impossible to obtain an analytic solution for the three-dimensional Ising model.

J. Kertész, J. Cserti and J. Szép, "Monte Carlo simulation programs for microcomputer," Eur. J. Phys. **6**, 232–237 (1985).

S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," Science **220**, 671–680 (1983). See also, S. Kirkpatrick and G. Toulouse, "Configuration space analysis of traveling salesman problems," J. Physique **46**, 1277–1292 (1985).

J. M. Kosterlitz and D. J. Thouless, "Ordering, metastability and phase transitions in two-dimensional systems," J. Phys. C **6**, 1181–1203 (1973); J. M. Kosterlitz, "The critical properties of the two-dimensional xy model," J. Phys. C **7**, 1046–1060 (1974).

D. P. Landau, Shan-Ho Tsai, and M. Exler, "A new approach to Monte Carlo simulations in statistical physics: Wang–Landau sampling," Am. J. Phys. **72**, 1294–1302 (2004).

D. P. Landau, "Finite-size behavior of the Ising square lattice," Phys. Rev. B **13**, 2997–3011 (1976). A clearly written paper on a finite-size scaling analysis of Monte Carlo data. See also D. P. Landau, "Finite-size behavior of the simple-cubic Ising lattice," Phys. Rev. B **14**, 255–262 (1976).

D. P. Landau and R. Alben, "Monte Carlo calculations as an aid in teaching statistical mechanics," Am. J. Phys. **41**, 394–400 (1973).

David Landau and Kurt Binder, *A Guide to Monte Carlo Simulations in Statistical Physics*, 2nd ed. (Cambridge University Press, 2005).

Jooyoung Lee and J. M. Kosterlitz, "New numerical method to study phase transitions," Phys. Rev. Lett. **65**, 137 (1990); *ibid.*, "Finite-size scaling and Monte Carlo simulations of first-order phase transitions," Phys. Rev. B **43**, 3265–3277 (1991).

Jooyoung Lee and Katherine J. Strandburg, "First-order melting transition of the hard-disk system," Phys. Rev. B **46**, 11190–11193 (1992).

Jiwen Liu and Erik Luijten, "Rejection-free geometric cluster algorithm for complex fluids," Phys. Rev. Lett. **92** 035504 (2004) and ibid., Phys. Rev. E **71**, 066701-1–12 (2005).

J. Machta, Y. S. Choi, A. Lucke, T. Schweizer, and L. Chayes, "Invaded cluster algorithm for Potts models," Phys. Rev. E 54, 1332–1345 (1996).

S. S. Mak, "The analytical demon of the Ising model," Phys. Lett. A **196**, 318 (1995).

J. Marro and R. Toral, "Microscopic observations on a kinetic Ising model," Am. J. Phys. **54**, 1114–1121 (1986).

N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller, "Equation of state calculations by fast computing machines," J. Chem. Phys. **21**, 1087–1092 (1953).

A. Alan Middleton, "Improved extremal optimization for the Ising spin glass," Phys. Rev. E **69**, 055701-1–4 (2004). The extremal optimization algorithm, which was inspired by the Bak–Sneppen algorithm for evolution (see Problem 14.12), preferentially flips spins that are "unfit." The adaptive algorithm proposed in this paper is an example of an heuristic that finds exact ground states efficiently for systems with frozen-in disorder.

M. E. J. Newman and G. T. Barkema, *Monte Carlo Methods in Statistical Physics* (Oxford University Press, 1999).

M. A. Novotny, "A new approach to an old algorithm for the simulation of Ising-like systems," Computers in Physics **9** (1), 46 (1995). The n-fold way algorithm is discussed. Also, see M. A. Novotny, "A tutorial on advanced dynamic Monte Carlo methods for systems with discrete state spaces," in *Annual Reviews of Computational Physics IX*, edited by Dietrich Stauffer (World Scientific, 2001), pp. 153–210.

Ole G. Mouritsen, *Computer Studies of Phase Transitions and Critical Phenomena* (Springer–Verlag, 1984).

E. P. Münger and M. A. Novotny, "Reweighting in Monte Carlo and Monte Carlo renormalization-group studies," Phys. Rev. B **43**, 5773–5783 (1991). The authors discuss the histogram method and combine it with renormalization group calculations.

Michael Plischke and Birger Bergersen, *Equilibrium Statistical Physics*, 3rd ed. (Prentice Hall, 2005). A graduate level text that discusses some contemporary topics in statistical physics, many of which have been influenced by computer simulations.

William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery, *Numerical Recipes*, 2nd ed. (Cambridge University Press, 1992). A Fortran program for the traveling salesman problem is given in Section 10.9.

Stephen H. Shenker and Jan Tobochnik, "Monte Carlo renormalization-group analysis of the classical Heisenberg model in two dimensions," Phys. Rev. B **22**, 4462–472 (1980).

Amihai Silverman and Joan Adler, "Animated simulated annealing," Computers in Physics **6**, 277 (1992). The authors describe a simulation of the annealing process to obtain a defect free single crystal of a model material.

H. Eugene Stanley, *Introduction to Phase Transitions and Critical Phenomena* (Oxford University Press, 1971). See Appendix B for the exact solution of the zero-field Ising model for a two-dimensional lattice.

Jan Tobochnik and G. V. Chester, "Monte Carlo study of the planar model," Phys. Rev. B **20**, 3761–3769 (1979).

Jan Tobochnik, Harvey Gould, and Jon Machta, "Understanding the temperature and the chemical potential through computer simulations," Am. J. Phys. **73** (8), 708–716 (2005). This paper extends the demon algorithm to compute the chemical potential.

Simon Trebst, David A. Huse, and Matthias Troyer, "Optimizing the ensemble for equilibration in broad-histogram Monte Carlo simulations," Phys. Rev. E **70**, 046701-1–5 (2004). The adaptive algorithm presented in this paper overcomes critical slowing down and improves upon the Wang–Landau algorithm and is another example of the flexibility of Monte Carlo algorithms.

I. Vattulainen, T. Ala–Nissila, and K. Kankaala, "Physical tests for random numbers in simulations," Phys. Rev. Lett. **73**, 2513 (1994).

B. Widom, "Some topics in the theory of fluids," J. Chem. Phys. **39**, 2808–2812 (1963). This paper discusses the insertion method for calculating the chemical potential.

Chapter 16

Quantum Systems

We discuss numerical solutions of the time-independent and time-dependent Schrödinger equation and describe several Monte Carlo methods for estimating the ground state of quantum systems.

16.1 Introduction

So far we have simulated the microscopic behavior of physical systems using Monte Carlo methods and molecular dynamics. In the latter method, the classical trajectory (the position and momentum) of each particle is calculated as a function of time. However, in quantum systems the position and momentum of a particle cannot be specified simultaneously. Because the description of microscopic particles is intrinsically quantum mechanical, we cannot directly *simulate* their trajectories on a computer (see Feynman).

Quantum mechanics does allow us to *analyze* probabilities, although there are difficulties associated with such an analysis. Consider a simple probabilistic system described by the one-dimensional diffusion equation (see Section 7.2)

$$\frac{\partial P(x,t)}{\partial t} = D\frac{\partial^2 P(x,t)}{\partial x^2}, \tag{16.1}$$

where $P(x,t)$ is the probability density of a particle being at position x at time t. One way to convert (16.1) to a difference equation and obtain a numerical solution for $P(x,t)$ is to make x and t discrete variables. Suppose we choose a mesh size for x such that the probability is given at p values of x. If we choose p to be order 10^3, a straightforward calculation of $P(x,t)$ would require approximately 10^3 data points for each value of t. In contrast, the corresponding calculation of the dynamics of a single particle based on Newton's second law would require one data point.

The limitations of the direct computational approach become even more apparent if there are many degrees of freedom. For example, for N particles in one dimension, we would have to calculate the probability $P(x_1, x_2, \ldots, x_N, t)$, where x_i is the position of particle i. Because we need to choose a mesh of p points for each x_i, we need to specify N^p values at each time t. For the same level of precision, p will be proportional to the length of the system (for particles confined to one dimension). Consequently, the calculation time and memory requirements grow exponentially with the length of the system. For example, for 10 particles on a mesh of 100 points, we would need to store 10^{100} numbers to represent P, which is already much more than any computer today can store. In two and three dimensions the growth is even faster.

Although the direct computational approach is limited to systems with only a few degrees of freedom, the simplicity of this approach will aid our understanding of the behavior of quantum systems. After a summary of the general features of quantum mechanical systems in Section 16.2, we consider this approach to solving the time-independent Schrödinger equation in Sections 16.3 and 16.4. In Section 16.5, we use a half-step algorithm to generate wave packet solutions to the time-dependent Schrödinger equation.

Because we have already learned that the diffusion equation (16.1) can be formulated as a random walk problem, it might not surprise you that Schrödinger's equation can be analyzed in a similar way. Monte Carlo methods are introduced in Section 16.7 to obtain variational solutions of the ground state. We introduce quantum Monte Carlo methods in Section 16.8 and discuss more sophisticated quantum Monte Carlo methods in Sections 16.9 and 16.10.

16.2 Review of Quantum Theory

For simplicity, we consider a one-dimensional, nonrelativistic quantum system consisting of one particle. The state of the system is completely characterized by the position space wave function $\Psi(x,t)$, which is interpreted as a probability amplitude. The probability $P(x,t)\,\Delta x$ of the particle being in a "volume" element Δx centered about the position x at time t is equal to

$$P(x,t)\,\Delta x = |\Psi(x,t)|^2 \Delta x, \tag{16.2}$$

where $|\Psi(x,t)|^2 = \Psi(x,t)\Psi^*(x,t)$, and $\Psi^*(x,t)$ is the complex conjugate of $\Psi(x,t)$. This interpretation of $\Psi(x,t)$ requires the use of normalized wave functions such that

$$\int_{-\infty}^{\infty} \Psi^*(x,t)\Psi(x,t)\,dx = 1. \tag{16.3}$$

If the particle is subjected to the influence of a potential energy function $V(x,t)$, the evolution of $\Psi(x,t)$ is given by the time-dependent Schrödinger equation

$$i\hbar\frac{\partial\Psi(x,t)}{\partial t} = -\frac{\hbar^2}{2m}\frac{\partial^2\Psi(x,t)}{\partial x^2} + V(x,t)\Psi(x,t), \tag{16.4}$$

where m is the mass of the particle, and $\hbar$ is Planck's constant divided by 2π.

Physically measurable quantities, such as the momentum, have corresponding operators. The expectation or average value of an observable A is given by

$$\langle A\rangle = \int \Psi^*(x,t)\hat{A}\Psi(x,t)\,dx, \tag{16.5}$$

where $\hat{A}$ is the operator corresponding to the measurable quantity A. For example, the momentum operator corresponding to the linear momentum p is $\hat{p} = -i\hbar\partial/\partial x$ in position space.

If the potential energy function is independent of time, we can obtain solutions of (16.4) of the form

$$\Psi(x,t) = \phi(x)\,e^{-iEt/\hbar}. \tag{16.6}$$

A particle in the state (16.6) has a well-defined energy E. If we substitute (16.6) into (16.4), we obtain the time-independent Schrödinger equation

$$-\frac{\hbar^2}{2m}\frac{d^2\phi(x)}{dx^2} + V(x)\,\phi(x) = E\,\phi(x). \tag{16.7}$$

Note that $\phi(x)$ is an *eigenstate* of the Hamiltonian operator

$$\hat{H} = -\frac{\hbar^2}{2m}\frac{\partial^2}{\partial x^2} + V(x) \tag{16.8}$$

with the *eigenvalue* E. That is,

$$\hat{H}\,\phi(x) = E\,\phi(x). \tag{16.9}$$

In general, there are many eigenstates ϕ_n, each with eigenvalues E_n that satisfy (16.9) and the boundary conditions imposed on the eigenstates by physical considerations.

The general form of $\Psi(x,t)$ can be expressed as a superposition of the eigenstates of the operator corresponding to any physical observable. For example, if $\hat{H}$ is independent of time, we can write

$$\Psi(x,t) = \sum_n c_n\,\phi_n(x)\,e^{-iE_n t/\hbar}, \tag{16.10}$$

where Σ represents a sum over the discrete states and an integral over the continuum states. The coefficients c_n in (16.10) can be determined from the value of $\Psi(x,t)$ at any time t. For example, if we know $\Psi(x,t=0)$, we can use the orthonormality property of the eigenstates of any physical operator to obtain

$$c_n = \int \phi_n^*(x)\Psi(x,0)\,dx. \tag{16.11}$$

The coefficient c_n can be interpreted as the probability amplitude of a measurement of the total energy yielding a particular value E_n.

There are three steps needed to solve (16.7) numerically. The first is to integrate (16.7) for any given value of the energy E in a way similar to the approach we have used for numerically solving other ordinary differential equations. This approach will usually not satisfy the boundary conditions. The second step is to find the particular values of E that lead to solutions that satisfy the boundary conditions. Finally, we need to normalize the eigenstate wave function using (16.3) so that we can interpret the eigenstate as a probability amplitude.

We first discuss the solution of (16.7) without imposing any boundary conditions by treating the solution to (16.7) as an initial value problem for the wave function and its derivative at some value of x for a given value of E. We will use these solutions to develop our intuition about the behavior of one-dimensional solutions to the Schrödinger equation.

To use an ODE solver, we express the wave function rate in terms of the independent variable x:

$$\frac{d\phi}{dx} = \phi' \tag{16.12a}$$

$$\frac{d\phi'}{dx} = -\frac{2m}{\hbar^2}[E - V(x)]\phi \tag{16.12b}$$

$$\frac{dx}{dx} = 1. \tag{16.12c}$$

Because the time-independent Schrödinger equation is a second-order differential equation, two initial conditions must be specified to obtain a solution. For simplicity, we first assume that the wave function is zero at the starting point, xmin, and the derivative is nonzero. We also assume that the range of values of x is finite and divide this range into intervals of width Δx. We initially consider potential energy functions $V(x)$ such that $V(x) = 0$ for $x < 0$; $V(x)$ changes abruptly at $x = 0$ to V_0, the value of the stepHeight parameter. An implementation of the numerical solution of (16.12) is shown in Listing 16.1.

Listing 16.1: The Schroedinger class models the one-dimensional time-independent Schrödinger equation.

```
package org.opensourcephysics.sip.ch16;
import org.opensourcephysics.numerics.*;

public class Schroedinger implements ODE {
   double energy = 0;
   double[] phi;
   double[] x;
   double xmin, xmax;                 // range of values of x
   double[] state = new double[3]; // state = phi, dphi/dx, x
   ODESolver solver = new RK45MultiStep(this);
   double stepHeight = 0;
   int numberOfPoints;

   public void initialize() {
      phi = new double[numberOfPoints];
      x = new double[numberOfPoints];
      double dx = (xmax-xmin)/(numberOfPoints-1);
      solver.setStepSize(dx);
   }

   void solve() {
      for(int i = 0;i<numberOfPoints;i++) { // zeros wavefunction
         phi[i] = 0;
      }
      state[0] = 0;     // initial phi
      state[1] = 1.0;   // nonzero initial dphi/dx
      state[2] = xmin;  // initial value of x
      for(int i = 0;i<numberOfPoints;i++) {
         phi[i] = state[0];              // stores wavefunction
         x[i] = state[2];
         solver.step();                  // steps Schroedinger equation
         if(Math.abs(state[0])>1.0e9) { // checks for diverging solution
            break;                       // leave the loop
         }
      }
   }

   public double[] getState() {
      return state;
   }

   public void getRate(double[] state, double[] rate) {
      rate[0] = state[1];
      rate[1] = 2.0*(-energy+evaluatePotential(state[2]))*state[0];
      rate[2] = 1.0;
   }

   public double evaluatePotential(double x) { // potential is nonzero for x > 0
      if(x<0) {
         return 0;
      } else {
         return stepHeight;
      }
   }
}
```

The `solve` method initializes the wave function and position arrays and sets the initial value of $d\phi/dx$ to an arbitrary nonzero value of unity. A loop is then used to compute values of ϕ until the solution diverges or until $x \geq$ `xmax`.

`SchroedingerApp` in Listing 16.2 produces a graphical view of $\phi(x)$. We will use this program in Problem 16.1 to study the behavior of the solution as we vary the height of the potential step.

Listing 16.2: `SchroedingerApp` solves the one-dimensional time-independent Schrödinger equation for a given energy.

```
package org.opensourcephysics.sip.ch16;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.display.*;
import org.opensourcephysics.frames.*;

public class SchroedingerApp extends AbstractCalculation {
   PlotFrame frame = new PlotFrame("x", "phi", "Wave function");
   Schroedinger schroedinger = new Schroedinger();

   public SchroedingerApp() {
      frame.setConnected(0, true);
      frame.setMarkerShape(0, Dataset.NO_MARKER);
   }

   public void calculate() {
      schroedinger.xmin = control.getDouble("xmin");
      schroedinger.xmax = control.getDouble("xmax");
      schroedinger.stepHeight =
         control.getDouble("step height at x = 0");
      schroedinger.numberOfPoints = control.getInt("number of points");
      schroedinger.energy = control.getDouble("energy");
      schroedinger.initialize();
      schroedinger.solve();
      frame.append(0, schroedinger.x, schroedinger.phi);
   }

   public void reset() {
      control.setValue("xmin", -5);
      control.setValue("xmax", 5);
      control.setValue("step height at x = 0", 1);
      control.setValue("number of points", 500);
      control.setValue("energy", 1);
   }
   public static void main(String[] args) {
      CalculationControl.createApp(new SchroedingerApp(), args);
   }
}
```

Problem 16.1. Numerical solution of the time-independent Schrödinger equation

(a) Sketch your guess for $\phi(x)$ for a potential step height of $V_0 = 3$ and energies $E = 1, 2, 3, 4,$ and 5.

(b) Choose `xmin = -10` and `xmax = 10`, and run `SchroedingerApp` with the parameters given in part (a). How well do your predictions match the numerical solution? Is there any discontinuity in ϕ or in the derivative $d\phi/dx$ at $x = 0$? Describe the wave function for both $x < 0$ and $x > 0$. Why does the wave function have a larger oscillatory amplitude when $x > 0$ than when $x < 0$ if the energy is greater than the potential step height?

(c) Describe the behavior of the wave function as the energy approaches the potential step height. Consider E in the range 2.5 to 3.5 in steps of 0.1.

(d) Repeat part (b) with the initial condition $\phi = 1$ and $d\phi/dx = 0$. Describe the differences, if any, in $\phi(x)$. □

Problem 16.1 demonstrates that the nature of the solution of (16.7) changes dramatically depending on the relative values of the energy E and the potential energy. If E is greater than V_0, the wave function is oscillatory whereas, if E is less than or equal to V_0, the wave function grows exponentially. The differential equation solver may fail if the difference between the potential energy and E is too large. There is also an exponentially decaying solution in the region where $E < V_0$, but this solution is difficult to detect.

Problem 16.2. Analytic solutions of the time-independent Schrödinger equation

(a) Find the analytic solution to (16.7) for the step potential for the cases: $E > V_0$, $E < V_0$, and $E = V_0$. We will use units such that $m = \hbar = 1$ in all the problems in this chapter.

(b) Run `SchroedingerApp` for the three cases to obtain the numerical solution of (16.7). When the numercial solution shows spatial oscillations in a region of space, estimate the wavelength of the oscillations and compare your numerical solution to the analytic results. When the numerical solution shows exponential decay as a function of position, estimate the decay rate and compare your numerical solution with the analytic solution. □

The solutions that we have obtained so far do not satisfy any condition other than that they solve (16.12). We have plotted only a portion of the wave function, and the solutions can be extended by increasing the number of points and the range of x over which the computation is performed. Physically, these solutions are unrealistic because they cannot be normalized over all of space. The normalization problem can be solved by using a linear combination of energy eigenstates (16.10) with different values of E. This combination is called a *wavepacket*.

Although we used a fourth-order algorithm in Listing 16.1, simpler algorithms can be used. Recall that the solution of (16.7) with $V(x) = 0$ can be expressed as a linear combination of sine and cosine functions. The oscillatory nature of this solution leads us to expect that the Euler–Cromer algorithm introduced in Chapter 3 will yield satisfactory results.

16.3 Bound State Solutions

We first consider potentials for which a particle is confined to a specific region of space. Such a potential is known as the infinite square well and is described by

$$V(x) = \begin{cases} 0 & \text{for } |x| \leq a \\ \infty & \text{for } |x| > a. \end{cases} \tag{16.13}$$

For this potential, an acceptable solution of (16.7) must vanish at the boundaries of the well. We will find that the eigenstates $\phi_n(x)$ can satisfy these boundary conditions only for specific values of the energy E_n.

Problem 16.3. The infinite square well

(a) Show analytically that the energy eigenvalues of the infinite square well are given by $E_n = n^2\pi^2\hbar^2/8ma^2$, where n is a positive integer. Also show that the normalized eigenstates have the form

$$\phi_n(x) = \frac{1}{\sqrt{a}}\cos\frac{n\pi x}{2a} \qquad n = 1, 3, \ldots \qquad \text{(even parity)} \tag{16.14a}$$

$$\phi_n(x) = \frac{1}{\sqrt{a}}\sin\frac{n\pi x}{2a} \qquad n = 2, 4, \ldots \qquad \text{(odd parity).} \tag{16.14b}$$

What is the parity of the ground state solution?

(b) We can solve (16.7) numerically for the infinite square well by setting `stepHeight` = 0. `xmin` = $-a$, and `xmax` = $+a$ in `SchroedingerApp` and requiring that $\phi(x = +a) = 0$. What is the condition for $\phi(x = -a)$ in the program? Choose $a = 1$ and calculate the first four energy eigenvalues exactly using `SchroedingerApp`. Do the numerical and analytic solutions match? Do the solutions satisfy the boundary conditions exactly? Are your numerical solutions normalized? □

Problem 16.4. Bound state solutions of the time-independent Schrödinger equation

(a) Consider the potential energy function defined by

$$V(x) = \begin{cases} 0 & \text{for } -a \leq x \leq 0 \\ V_0 & \text{for } 0 < x \leq a \\ \infty & \text{for } |x| > a. \end{cases} \tag{16.15}$$

As for the infinite square well, the eigenfunction is confined between infinite potential barriers at $x = \pm a$. In addition, there is a step potential at $x = 0$. Choose $a = 5$ and $V_0 = 1$ and run `SchroedingerApp` with an energy of $E = 0.15$. Repeat with an energy of $E = 0.16$. Why can you conclude that an energy eigenvalue is bracketed by these two values?

(b) Choose a strategy for determining the value of E such that the boundary conditions at $x = +a$ are satisfied. Determine the energy eigenvalue to four decimal places. Does your answer depend on the number of points at which the wave function is computed?

(c) Repeat the above procedure starting with energy values of 0.58 and 0.59 and find the energy eigenvalue of the second bound state. □

If you were persistent in doing all of Problem 16.4, you would have discovered two energy eigenvalues, 0.1505 and 0.5857.

The procedure we used is known as the *shooting* algorithm. The allowed eigenvalues are imposed by the requirement that $\phi_n(x) \to 0$ at the boundaries. Although the shooting algorithm usually yields an eigenvalue solution, we often wish to find specific eigenvalues, such as the eigenvalue $E = 1.1195$ corresponding to the third excited state for the potential in (16.15). Because the energy of a wave function increases as the wavelength decreases, we can order the energy eigenvalues by counting the number of times the corresponding eigenstate crosses the x-axis, that is, by the number of nodes. The ground state eigenstate has no nodes. Why? Why can we order the eigenvalues by the number of nodes? The number of nodes can be used to narrow the energy bracket in the shooting algorithm. For example, if we are searching for the third energy eigenvalue and we observe 5 nodes, then the energy is too large. To find a specific quantum state, we automate the shooting method as follows:

1. Choose a value of the energy E and count the number of nodes.
2. Increase E and repeat step 1 until the number of nodes is equal to the desired number.
3. Decrease E and repeat step 1 until the number of nodes is one less than the desired number. The desired value of the energy eigenvalue is now bracketed. We can further narrow the energy by doing the following:
4. Set the energy to the bracket midpoint.
5. Initialize $\phi(x)$ at the left boundary and iterate $\phi(x)$ toward increasing x until ϕ diverges or until the right boundary is reached.
6. If the quantum number is even (odd) and the last value of $\phi(x)$ in step 4 is negative (positive), then the trial value of E is too large.
7. If the quantum number is even (odd) and the last value of $\phi(x)$ in step 4 is positive (negative), then the trial value of E is too small.
8. Repeat steps 2–7 until the wave function satisfies the right-hand boundary condition to an acceptable tolerance. This procedure is known as a binary search because every repetition decreases the energy bracket by a factor of two.

Problem 16.5 asks you to write a program that finds specific eigenvalues using this procedure.

Problem 16.5. Shooting algorithm

(a) Modify `SchroedingerApp` to find the eigenvalue associated with a given number of nodes. How is the number of nodes related to the quantum number? Test your program for the infinite square well. What is the value of Δx needed to determine E_1 to two decimal places? three decimal places?

(b) Add a method to normalize ϕ. Normalize and display the first five eigenstates.

(c) Find the first five eigenstates and eigenvalues for the potential in (16.15) with $a = 1$ and $V - 0 = 1$.

(d) Does your result for E_1 depend on the starting value of $d\phi/dx$? □

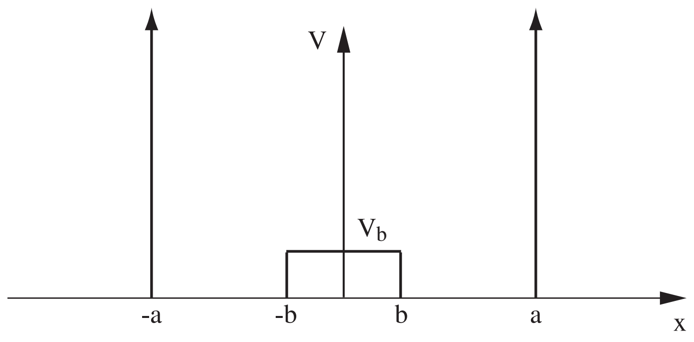


Figure 16.1: An infinite square well with a potential bump of height V_b in the middle.

Problem 16.6. Perturbation of the infinite square well

(a) Determine the effect of a small perturbation on the eigenstates and eigenvalues of the infinite square well. Place a small rectangular bump of half-width b and height V_b symmetrically about $x = 0$ (see Fig. 16.1). Choose $b \ll a$ and determine how the ground state energy and eigenstate change with V_b and b. What is the relative change in the ground state energy for $V_b = 10$, $b = 0.1$ and $V_b = 20$, $b = 0.1$ with $a = 1$? Let ϕ_0 denote the ground state eigenstate for $b = 0$ and let ϕ_b denote the ground state eigenstate for $b \neq 0$. Compute the value of the overlap integral

$$\int_0^a \phi_b(x)\phi_0(x)\,dx. \tag{16.16}$$

This integral would be unity if the perturbation was not present (and the eigenstate was properly normalized). How is the change in the overlap integral related to the relative change in the energy eigenvalue?

(b) Compute the ground state energy for $V_b = 20$ and $b = 0.05$. How does the value of E_1 compare to that found in part (a) for $V_b = 10$ and $b = 0.1$? □

Because numerical solutions to the Schrödinger equation grow exponentially if $V(x) - E > 0$, it may not be possible to obtain a numerical solution for $\phi(x)$ that satisfies the boundary conditions if $V(x) - E$ is large over an extended region of space. The reason is that energy can be specified and ϕ can be computed only to finite accuracy. Problem 16.7 shows that we can sometimes solve this problem using simpler boundary conditions if the potential is symmetric. In this case,

$$V(x) = V(-x), \tag{16.17}$$

and $\phi(x)$ can be chosen to have definite parity. For even parity solutions, $\phi(-x) = \phi(x)$; odd parity solutions satisfy $\phi(-x) = -\phi(x)$. The definite parity of $\phi(x)$ allows us to specify either ϕ or ϕ' at $x = 0$. Hence, the parity of ϕ determines one of the boundary conditions. For simplicity, choose $\phi(0) = 1$ and $\phi'(0) = 0$ for even parity solutions and $\phi(0) = 0$ and $\phi'(0) = 1$ for odd parity solutions.

Problem 16.7. Symmetric potentials

(a) Modify `Schroedinger` to make use of symmetric potential boundary conditions for the harmonic oscillator:

$$V(x) = \frac{1}{2}x^2. \tag{16.18}$$

Start the solution at $x = 0$ using appropriate conditions for even and odd quantum numbers and find the first four energy eigenvalues such that the wave function approaches zero for large values of x. Because the computed $\phi(x)$ will diverge for sufficiently large x, we seek values of the energy such that a small decrease in E causes the wave function to diverge in one direction, and a small increase causes the wave function to diverge in the opposite direction. Initially choose `xmax` = 5 so that the classically forbidden region is sufficiently large so that $\phi(x)$ can decay to zero for the first few eigenstates. Increase `xmax` if necessary for the higher energy eigenvalues. Is there any pattern in the values of the energy eignevalues you found?

(b) Repeat part (a) for the linear potential $V(x) = |x|$. Describe the differences between your results for this potential and for the harmonic oscillator potential. The quantum mechanical treatment of the linear potential can be used to model the energy spectrum of a bound quark-antiquark system known as quarkonium.

(c) Obtain a numerical solution of the anharmonic oscillator $V(x) = \frac{1}{2}x^2 + bx^4$. In this case there are no analytic solutions, and numerical solutions are necessary for large values of b. How do the ground state energy and eigenstate depend on b for small b? □

Problem 16.8. Finite square well

The finite square well potential is given by

$$V(x) = \begin{cases} 0 & \text{for } |x| \leq a \\ V_0 & \text{for } |x| > a. \end{cases} \tag{16.19}$$

The input parameters are the well depth, V_0, and the half-width of the well, a.

(a) Choose $V_0 = 10$ and $a = 1$. How do you expect the value of the ground state energy to compare to its corresponding value for the infinite square well? Compute the ground state eigenvalue and eigenstate by determining a value of E such that $\phi(x)$ has no nodes and is approximately zero for large x. (See Problem (16.7a) for the procedure for finding the eigenvalues.)

(b) Because the well depth is finite, $\phi(x)$ is nonzero in the classically forbidden region for which $E < V_0$ and $x > |a|$. Define the penetration distance as the distance from $x = a$ to a point where ϕ is $\sim 1/e \approx 0.37$ of its value at $x = a$. Determine the qualitative dependence of the penetration distance on the magnitude of V_0.

(c) What is the total number of bound excited states? Why is the total number of bound states finite? □

As we have found, it is difficult to find bound state solutions of the time-independent Schrödinger equation because the exponential solution allows numerical errors to dominate when $V(x) - E > 0$ is large. Because we want to easily generate eigenstates in subsequent sections, we have written a general-purpose eigenstate solver that examines the maxima and minima of the solution as well as the nodes to determine the eigenstate's quantum number. The code for the `Eigenstate` class is in the ch16 package. The `EigenstateApp` target class shows how the `Eigenstate` class is used.

Listing 16.3: The `EigenstateApp` program tests the `Eigenstate` class.

```
package org.opensourcephysics.sip.ch16;
```

```
import org.opensourcephysics.frames.PlotFrame;
import org.opensourcephysics.numerics.Function;

public class EigenstateApp {
   public static void main(String[] args) {
      PlotFrame drawingFrame =
         new PlotFrame("x", "|phi|", "eigenstate");
      int numberOfPoints = 300;
      double
         xmin = -5, xmax = +5;
      Eigenstate eigenstate =
         new Eigenstate(new Potential(), numberOfPoints, xmin, xmax);
      int n = 3; // quantum number
      double[] phi = eigenstate.getEigenstate(n);
      double[] x = eigenstate.getXCoordinates();
      if(eigenstate.getErrorCode()==Eigenstate.NO_ERROR) {
         drawingFrame.setMessage("energy = "+eigenstate.energy);
      } else {
         drawingFrame.setMessage("eigenvalue did not converge");
      }
      drawingFrame.append(0, x, phi);
      drawingFrame.setVisible(true);
      drawingFrame.setDefaultCloseOperation(
         javax.swing.JFrame.EXIT_ON_CLOSE);
   }
}
class Potential implements Function {
   public double evaluate(double x) {
      return(x*x)/2;
   }
}
```

The `getEigenstate` method in the `Eigenstate` class computes the eigenstate for the specified quantum number and returns a zeroed wave function if the algorithm does not converge. We test the validity of the `Eigenstate` class in Problem 16.9.

Problem 16.9. The `Eigenstate` class

(a) Examine the code of the `Eigenstate` class. What "trick" is used to handle the divergence in the forbidden region of deep wells?

(b) Write a class that displays the eigenstates of the simple harmonic oscillator using the `Calculation` interface. Include input parameters that allow the user to vary the principal quantum number and the number of points.

(c) Use a spatial grid of 300 points with $-5 < x < 5$ and compare the known analytic solution for the simple harmonic oscillator eigenstates to the numerical solution for the lowest three energy eigenstates. What is the largest energy eigenvalue that can be computed to an accuracy of 1%? What causes the decreasing accuracy for larger quantum numbers? What if the domain is increased to $-50 < x < 50$?

(d) Describe the conditions under which the `Eigenstate` class fails and demonstrate this failure. Improve the `Eigenstate` class to handle at least one failure mode. □

16.4 Time Development of Eigenstate Superpositions

If the Hamiltonian is independent of time, the time development of the wave function $\Psi(x,t)$ can be expressed as a linear superposition of energy eigenstates $\phi_n(x)$ with eigenvalue E_n:

$$\Psi(x,t) = \sum_n c_n \phi_n(x) e^{-iE_n t/\hbar}. \tag{16.20}$$

To understand the time dependence of $\Psi(x,t)$, we begin by studying superpositions of analytic solutions. The static getEigenstate method in the BoxEigenstate class generates these solutions for the infinite square well.

Listing 16.4: The BoxEigenstate class generates analytic stationary state solutions for the infinite square well.

```java
package org.opensourcephysics.sip.ch16;
public class BoxEigenstate {
   static double a = 1; // length of box

   private BoxEigenstate() {
      // prohibit instantiation because all methods are static
   }

   static double[] getEigenstate(int n, int numberOfPoints) {
      double[] phi = new double[numberOfPoints];
      n++; // quantum number
      double norm = Math.sqrt(2/a);
      for(int i = 0;i<numberOfPoints;i++) {
         phi[i] = norm*Math.sin((n*Math.PI*i)/(numberOfPoints-1));
      }
      return phi;
   }

   static double getEigenvalue(int n) {
      n++;
      return(n*n*Math.PI*Math.PI)/2/a/a; // hbar = 1, mass = 1
   }
}
```

To visualize the evolution of $\Psi(x,t)$ in (16.20), we define a class that stores the energy eigenstates $\phi_n(x)$ the real and imaginary parts of the expansion coefficients c_n and the eigenvalues E_n. As the system evolves, the eigenstates are added together as in (16.20) using the expansion coefficients. The BoxSuperposition class shown in Listing 16.5 creates such a wave function for the infinite square well. Later we will modify this class to study other potentials.

Listing 16.5: The BoxSuperposition class models the time dependence of the wave function of an infinite square well using a superposition of eigenstates.

```java
package org.opensourcephysics.sip.ch16;
public class BoxSuperposition {
   double[] realCoef;
   double[] imagCoef;
   double[][] states;     // eigenfunctions
   double[] eigenvalues; // eigenvalues
   double[] x, realPsi, imagPsi;
   double[] zeroArray;

   public BoxSuperposition(int numberOfPoints, double[] realCoef,
        double[] imagCoef) {
      if(realCoef.length!=imagCoef.length) {
         throw new IllegalArgumentException("Real and imaginary
              coefficients must have equal number of elements.");
      }
      this.realCoef = realCoef;
      this.imagCoef = imagCoef;
      int nstates = realCoef.length;
      // delay allocation of arrays for eigenstates
      states = new double[nstates][];     // eigenfunctions
      eigenvalues = new double[nstates]; // eigenvalues
      realPsi = new double[numberOfPoints];
      imagPsi = new double[numberOfPoints];
      zeroArray = new double[numberOfPoints];
      x = new double[numberOfPoints];
      double dx = BoxEigenstate.a/(numberOfPoints-1);
      double xo = 0;
      for(int j = 0, n = numberOfPoints;j<n;j++) {
         x[j] = xo;
         xo += dx;
      }
      for(int n = 0;n<nstates;n++) {
         states[n] = BoxEigenstate.getEigenstate(n, numberOfPoints);
         eigenvalues[n] = BoxEigenstate.getEigenvalue(n);
      }
      update(0); // compute the superpositon at t = 0
   }

   void update(double time) {
      // set real and imaginary parts of wave function to zero
      System.arraycopy(zeroArray, 0, realPsi, 0, realPsi.length);
      System.arraycopy(zeroArray, 0, imagPsi, 0, imagPsi.length);
      for(int i = 0, nstates = realCoef.length;i<nstates;i++) {
         double[] phi = states[i];
         double re = realCoef[i];
         double im = imagCoef[i];
         double sin = Math.sin(time*eigenvalues[i]);
         double cos = Math.cos(time*eigenvalues[i]);
         for(int j = 1, n = phi.length-1;j<n;j++) {
            realPsi[j] += (re*cos-im*sin)*phi[j];
            imagPsi[j] += (im*cos+re*sin)*phi[j];
         }
      }
   }
}
```

The BoxSuperpositionApp class in Listing 16.6 implements the eigenstate superposition and displays the wave function by extending the AbstractAnimation class and implementing the doStep method.

Listing 16.6: BoxSuperpositionApp shows the evolution of a particle in a box.

```java
package org.opensourcephysics.sip.ch16;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.frames.ComplexPlotFrame;

public class BoxSuperpositionApp extends AbstractSimulation {
   ComplexPlotFrame psiFrame = new ComplexPlotFrame("x", "|Psi|",
       "Time dependent wave function");
   BoxSuperposition superposition;
   double time, dt;

   public BoxSuperpositionApp() {
      psiFrame.limitAutoscaleY(-1, 1);
   }

   public void initialize() {
      time = 0;
      psiFrame.setMessage("t = "+decimalFormat.format(time));
      dt = control.getDouble("dt");
      double[] re = (double[]) control.getObject("real coef");
      double[] im = (double[]) control.getObject("imag coef");
      int numberOfPoints = control.getInt("number of points");
      superposition = new BoxSuperposition(numberOfPoints, re, im);
      psiFrame.append(superposition.x, superposition.realPsi,
           superposition.imagPsi);
   }

   public void doStep() {
      time += dt;
      superposition.update(time);
      psiFrame.clearData();
      psiFrame.append(superposition.x, superposition.realPsi,
          superposition.imagPsi);
      psiFrame.setMessage("t = "+decimalFormat.format(time));
   }

   public void reset() {
      control.setValue("dt", 0.005);
      control.setValue("real coef", new double[] {0.707, 0, 0.707});
      control.setValue("imag coef", new double[] {0, 0, 0});
      control.setValue("number of points", 50);
      initialize();
   }

   public static void main(String[] args) {
      SimulationControl.createApp(new BoxSuperpositionApp());
   }
}
```

Because wave functions have real and imaginary components, the BoxSuperpositionApp class uses a ComplexPlotFrame for plotting. The ComplexPlotFrame renders data using an envelope whose height is proportional to the magnitude, and the region between the envelope is colored from red to blue to show the phase. A more traditional plotting style showing the real and imaginary parts of the wave function is available from the frame's Tools menu. (Also see Appendix 16A.) We use `BoxSuperpositionApp` to study the periodicity of the wave function in Problems 16.10 and 16.11.

Problem 16.10. Time-dependent wave function for the infinite square well

(a) Add a second visualization to the `BoxSuperpositionApp` class that displays the probability density $\Psi(x,t)$.

(b) Change the coefficient array so that the particle is in the ground state. Show that the wave function changes in time, but that the probability density does not. At what times does the ground state wave function return to its initial condition? Find the corresponding times for the first and second excited states.

(c) Choose the coefficient array so that the particle is in a 50:50 superposition of the ground state and the first excited state. At what times does the wave function return to its initial condition? After what time does the probability density return to its initial condition?

(d) Change the coefficient array so that the particle is in a 50:50 superposition of the first and second excited states. After what time does the wave function return to its initial condition? After what time does the probability density return to its initial condition?

(e) Will the initial wave function always revive, that is, return to its initial condition? Explain. □

Problem 16.11. Time-dependent wave function for the simple harmonic oscillator

(a) Modify `BoxSuperpositionApp` and `BoxSuperposition` to superimpose the eigenstates of the simple harmonic oscillator using the `Eigenstate` class to compute the eigenstates. What are the period of the ground state and the first excited state wave functions and the probability density?

(b) Change the coefficient array so that the particle is in a 50:50 superposition of the ground state and the first excited state. At what times does the wave function return to its initial condition? At what times does the probability density return to its initial condition? Compare these times with the period of the classical oscillator.

(c) Repeat part (b) for a 50:50 superposition of the first and second excited states. □

Problem 16.12. Linear potential

Does the linear potential $V(x) = |x|$ exhibit periodicity if the particle is in a superposition state? Test your hypothesis using numerical solutions to the Schrödinger equation. □

As we have seen, the evolution of an arbitrary wave function can be found by expanding the initial state in terms of the energy eigenstates. From the orthogonality property of eigenstates, it is easy to show that

$$c_n = \int_{-\infty}^{\infty} \phi_n^*(x)\Psi(x,0)\,dx. \tag{16.21}$$

This operation is known as a projection of Ψ onto ϕ_n.

Problem 16.13. Projections

(a) Add a `projection` method to the `BoxSuperpositionApp` class using the signature:

```
double[] projection(int n, double[] realPhi, double[] imagPhi)
```

The projection method's arguments are the quantum number, the real component of the wave function, and the imaginary component of the wave function. The method returns a two-component array containing the real and imaginary parts of the projection of the wave function on the *n*th eigenstate.

(b) Test your projection method by projecting an eigenstate onto another eigenstate. That is, verify the orthogonality condition

$$\delta_{nm} = \int_{-\infty}^{\infty} \phi_m(x)\phi_n(x)\,dx. \tag{16.22}$$

(c) Compute the expansion coefficients for a particle in a box using the following initial Gaussian wave function:

$$\Psi(x,0) = e^{-64x^2}. \tag{16.23}$$

Assume a box width $a = 1$. Plot the amplitude of the resulting coefficients as a function of the quantum number n. How does the shape of this plot depend on the width of the Gaussian wave function?

(d) Use the coefficients from part (c) to determine the evolution of the wave function. Does the wave function remain real? Does the initial state revive?

(e) Repeat parts (c) and (d) using the initial wave function

$$\Psi(x,0) = \begin{cases} 2 & |x| \le 1/8 \\ 0, & |x| > 1/8. \end{cases} \tag{16.24}$$

□

Problem 16.14. Coherent states

Because the energy eigenvalues of the simple harmonic oscillator are equally spaced, there exist wave functions known as *coherent states* whose probability density propagates quasi-classically.

(a) Include a sufficient number of expansion coefficients for $V(x) = 10x^2$ to model an initial Gaussian wave function centered at the origin:

$$\Psi(x,0) = e^{-16x^2}. \tag{16.25}$$

Describe the evolution.

(b) Repeat part (a) with

$$\Psi(x,0) = e^{-16(x-2)^2}. \tag{16.26}$$

(c) Show that the wave functions in parts (a) and (b) change their width but not their Gaussian envelope. Construct a wave function with the following expansion coefficients and observe its behavior

$$c_n^2 = \frac{\langle n\rangle^n}{n!} e^{-\langle n\rangle}. \tag{16.27}$$

The expectation of the number of quanta $\langle n\rangle$ is given by

$$\langle n\rangle = \langle E\rangle - \frac{1}{2}\hbar\omega, \tag{16.28}$$

where $\langle E\rangle$ is the energy expectation value of the coherent state. □

The expansion of an arbitrary wave function in terms of a set of eigenstates is closely related to Fourier analysis. Because the eigenstates of a particle in a box are sinusoidal functions, we could have used the fast Fourier transform algorithm (FFT) to compute the projection coefficients. Because these coefficients are calculated only once in Problem 16.14, evaluating (16.21) directly is reasonable. We will use the FFT to study wave functions in momentum space and to implement the operator splitting method for time evolution in Section 16.6.

16.5 The Time-Dependent Schrödinger Equation

Although the numerical solution of the time-independent Schrödinger equation (16.7) is straightforward for one particle, the numerical solution of the time-dependent Schrödinger equation (16.4) is not as simple. A naive approach to its numerical solution can be formulated by introducing a grid for the time coordinate and a grid for the spatial coordinate. We use the notation $t_n = t_0 + n\Delta t$, $x_s = x_0 + s\Delta x$, and $\Psi(x_s, t_n)$. The idea is to relate $\Psi(x_s, t_{n+1})$ to the value of $\Psi(x_s, t_n)$ for each value of x_s. An example of an algorithm that solves the Schrödinger-like equation $\partial\Psi/\partial t = \partial^2\Psi/\partial x^2$ to first order in Δt is given by

$$\frac{1}{\Delta t}\Big[\Psi(x_s, t_{n+1}) - \Psi(x_s, t_n)\Big] = \frac{1}{(\Delta x)^2}\Big[\Psi(x_{s+1}, t_n) - 2\Psi(x_s, t_n) + \Psi(x_{s-1}, t_n)\Big]. \tag{16.29}$$

The right-hand side of (16.29) represents a finite difference approximation to the second derivative of Ψ with respect to x. Equation (16.29) is an example of an *explicit* scheme, because given Ψ at time t_n, we can compute Ψ at time t_{n+1}. Unfortunately, this explicit approach leads to unstable solutions; that is, the numerical value of Ψ diverges from the exact solution as Ψ evolves in time.

One way to avoid the instability is to retain the same form as (16.29) but to evaluate the spatial derivative on the right side of (16.29) at time t_{n+1} rather than time t_n:

$$\frac{1}{\Delta t}\Big[\Psi(x_s, t_{n+1}) - \Psi(x_s, t_n)\Big] = \frac{1}{(\Delta x)^2}\Big[\Psi(x_{s+1}, t_{n+1}) - 2\Psi(x_s, t_{n+1}) + \Psi(x_{s-1}, t_{n+1})\Big]. \tag{16.30}$$

Equation (16.30) is an *implicit* method because the unknown function $\Psi(x_s, t_{n+1})$ appears on both sides. To obtain $\Psi(x_s, t_{n+1})$, it is necessary to solve a set of linear equations at each time step. More details of this approach and the demonstration that (16.30) leads to stable solutions can be found in the references.

Visscher and others have suggested an alternative approach in which the real and imaginary parts of Ψ are treated separately and defined at different times. The algorithm ensures that the total probability remains constant. If we let

$$\Psi(x,t) = \mathrm{R}(x,t) + i\,\mathrm{I}(x,t), \tag{16.31}$$

then Schrödinger's equation $i\partial\Psi(x,t)/\partial t = \hat{H}\Psi(x,t)$ becomes ($\hbar = 1$ as usual)

$$\frac{\partial \mathrm{R}(x,t)}{\partial t} = \hat{H}\,\mathrm{I}(x,t) \tag{16.32a}$$

$$\frac{\partial \mathrm{I}(x,t)}{\partial t} = -\hat{H}\,\mathrm{R}(x,t). \tag{16.32b}$$

A stable method of numerically solving (16.32) is to use a form of the half-step method (see Appendix 3A). The resulting difference equations are

$$\mathrm{R}(x,t+\Delta t) = \mathrm{R}(x,t) + \hat{H}\,\mathrm{I}\Big(x,t+\frac{1}{2}\Delta t\Big)\Delta t \tag{16.33a}$$

$$\mathrm{I}\Big(x,t+\frac{3}{2}\Delta t\Big) = \mathrm{I}\Big(x,t+\frac{1}{2}\Delta t\Big) - \hat{H}\,\mathrm{R}(x,t)\,\Delta t, \tag{16.33b}$$

where the initial values are given by $\mathrm{R}(x,0)$ and $\mathrm{I}(x,\frac{1}{2}\Delta t)$. Visscher has shown that this algorithm is stable if

$$\frac{-2\hbar}{\Delta t} \le V \le \frac{2\hbar}{\Delta t} - \frac{2\hbar^2}{(m\Delta x)^2}, \tag{16.34}$$

where the inequality (16.34) holds for all values of the potential V.

The appropriate definition of the probability density $P(x,t) = \mathrm{R}(x,t)^2 + \mathrm{I}(x,t)^2$ is not obvious because R and I are not defined at the same time. The following choice conserves the total probability:

$$P(x,t) = \mathrm{R}(x,t)^2 + \mathrm{I}\Big(x,t+\frac{1}{2}\Delta t\Big)\mathrm{I}\Big(x,t-\frac{1}{2}\Delta t\Big) \tag{16.35a}$$

$$P\Big(x,t+\frac{1}{2}\Delta t\Big) = \mathrm{R}(t+\Delta t)\,\mathrm{R}(x,t) + \mathrm{I}\Big(x,t+\frac{1}{2}\Delta t\Big)^2. \tag{16.35b}$$

An implementation of (16.33) is given in the `TDHalfStep` class in Listing 16.7. The real part of the wave function is first updated for all positions, and then the imaginary part is updated using the new values of the real part.

Listing 16.7: The `TDHalfStep` class solves the one-dimensional time-dependent Schrödinger equation.

```
package org.opensourcephysics.sip.ch16;
public class TDHalfStep {
   double[] x, realPsi, imagPsi, potential;
   double dx, dx2;
   double dt = 0.001;

   public TDHalfStep(GaussianPacket packet, int numberOfPoints,
        double xmin, double xmax) {
      realPsi = new double[numberOfPoints];
      imagPsi = new double[numberOfPoints];
      potential = new double[numberOfPoints];
      x = new double[numberOfPoints];
      dx = (xmax-xmin)/(numberOfPoints-1);
      dx2 = dx*dx;
      double x0 = xmin;
      for(int i = 0, n = realPsi.length;i<n;i++) {
         x[i] = x0;
         potential[i] = getV(x0);
         realPsi[i] = packet.getReal(x0);
         imagPsi[i] = packet.getImaginary(x0);
         x0 += dx;
      }
      dt = getMaxDt();
      // advances the imaginary part by 1/2 step at start
      for(int i = 1, n = realPsi.length-1;i<n;i++) {
         // deltaRe = change in real part of psi in 1/2 step
         double deltaRe = potential[i]*realPsi[i]-0.5*(realPsi[i+1]-
             2*realPsi[i]+realPsi[i-1])/dx2;
         imagPsi[i] -= deltaRe*dt/2;
      }
   }

   double getMaxDt() {
      double dt = Double.MAX_VALUE;
      for(int i = 0, n = potential.length;i<n;i++) {
         if(potential[i]<0) {
            dt = Math.min(dt, -2/potential[i]);
         }
         double a = potential[i]+2/dx2;
         if(a>0) {
            dt = Math.min(dt, 2/a);
         }
      }
      return dt;
   }

   double step() {
      for(int i = 1, n = imagPsi.length-1;i<n;i++) {
         double imH = potential[i]*imagPsi[i]-0.5*(imagPsi[i+1]
            -2*imagPsi[i]+imagPsi[i-1])/dx2;
         realPsi[i] += imH*dt;
      }
      for(int i = 1, n = realPsi.length-1;i<n;i++) {
         double reH = potential[i]*realPsi[i]-0.5*(realPsi[i+1]
            -2*realPsi[i]+realPsi[i-1])/dx2;
         imagPsi[i] -= reH*dt;
      }
      return dt;
   }

   public double getV(double x) {
      return 0; // change this statement to model other potentials
   }
}
```

Before we can use the `TDHalfStep` class, we need to choose an initial wave function. A convenient form is the Gaussian wave packet with a width w centered about x_0 given by

$$\Psi(x,0) = \left(\frac{1}{2\pi w^2}\right)^{1/4} e^{ik_0(x-x_0)}\, e^{-(x-x_0)^2/4w^2}. \qquad (16.36)$$

The expectation value of the initial velocity of the wave packet is $\langle v\rangle = p_0/m = \hbar k_0/m$. Note that the wave function has a nonzero momentum expectation value, which is known as a *momentum boost*. An implementation of (16.36) is shown in the GaussianPacket class. The constructor is passed the width, center, and momentum of the packet. Real and imaginary values can then be calculated at any x to fill the wave function arrays.

Listing 16.8: The GaussianPacket class creates a wave function with a Gaussian probability distribution and a momentum boost.

```java
package org.opensourcephysics.sip.ch16;
public class GaussianPacket {
   double w, x0, p0;
   double w42;
   double norm;

   public GaussianPacket(double width, double center, double momentum) {
      w = width;
      w42 = 4*w*w;
      x0 = center;
      p0 = momentum;
      norm = Math.pow(2*Math.PI*w*w, -0.25);
   }

   public double getReal(double x) {
      return norm*Math.exp(-(x-x0)*(x-x0)/w42)*Math.cos(p0*(x-x0));
   }

   public double getImaginary(double x) {
      return norm*Math.exp(-(x-x0)*(x-x0)/w42)*Math.sin(p0*(x-x0));
   }
}
```

To start the half-step algorithm, we need the value of $\mathrm{I}(x, t = \frac{1}{2}\Delta t)$ and $\mathrm{R}(x, t = 0)$. To obtain $\mathrm{I}(x, t = \frac{1}{2}\Delta t)$, we use the real component of the wave function to perform a half step.

$$\mathrm{I}\Big(x, t + \frac{1}{2}\Delta t\Big) = \mathrm{I}(x,t) - \hat{H}\,\mathrm{R}(x,t)\,\frac{\Delta t}{2}. \tag{16.37}$$

The normalization factor must be computed after we correct the initial wave function using (16.37). For completeness, we list the TDHalfStepApp target class.

Listing 16.9: The TDHalfStepApp class solves the time-independent Schrödinger equation and displays the wave function.

```java
package org.opensourcephysics.sip.ch16;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.frames.ComplexPlotFrame;

public class TDHalfStepApp extends AbstractSimulation {
   ComplexPlotFrame psiFrame = new ComplexPlotFrame("x", "|Psi|",
        "Wave function");
   TDHalfStep wavefunction;
   double time;

   public TDHalfStepApp() {
      // do not autoscale within this y-range
      psiFrame.limitAutoscaleY(-1, 1);
   }

   public void initialize() {
      time = 0;
      psiFrame.setMessage("t="+0);
      double xmin = control.getDouble("xmin");
      double xmax = control.getDouble("xmax");
      int numberOfPoints = control.getInt("number of points");
      double width = control.getDouble("packet width");
      double x0 = control.getDouble("packet offset");
      double momentum = control.getDouble("packet momentum");
      GaussianPacket packet = new GaussianPacket(width, x0, momentum);
      wavefunction = new TDHalfStep(packet, numberOfPoints, xmin, xmax);
      psiFrame.clearData();     // removes old data
      psiFrame.append(wavefunction.x, wavefunction.realPsi,
         wavefunction.imagPsi);
   }

   public void doStep() {
      time += wavefunction.step();
      psiFrame.clearData();
      psiFrame.append(wavefunction.x, wavefunction.realPsi,
          wavefunction.imagPsi);
      psiFrame.setMessage("t="+decimalFormat.format(time));
   }

   public void reset() {
      control.setValue("xmin", -20);
      control.setValue("xmax", 20);
      control.setValue("number of points", 500);
      control.setValue("packet width", 1);
      control.setValue("packet offset", -15);
      control.setValue("packet momentum", 2);
      // multiple computations per animation step
      setStepsPerDisplay(10);
      enableStepsPerDisplay(true);
      initialize();
   }

   public static void main(String[] args) {
      SimulationControl.createApp(new TDHalfStepApp());
   }
}
```

Problem 16.15. Evolution of a wave packet

(a) Add an array to TDHalfStepApp that saves the imaginary part of the wave function at the previous time step so that the probability density can be computed using (16.35). Show that the probability is conserved.

(b) Use TDHalfStepApp to follow the motion of a wave packet in a potential-free region. Let $x_0 = -15$, $k_0 = 2$, $w = 1$, $\mathtt{dx} = 0.4$, and $\mathtt{dt} = 0.1$. Suitable values for the minimum and maximum values of x on the grid are `xmin` = −20 and `xmax` = 20. What is the shape of the wave packet at different times? Does the shape of the wave packet depend on your choice of the parameters k_0 and w?

(c) Modify `TDHalfStepApp` so that the quantities $x_0(t)$ and $w(t)$, the position and width of the wave packet as a function of time, can be measured directly. What is a reasonable definition of $w(t)$? What is the qualitative dependence of x_0 and w on t? How do your results change if the initial width of the packet is reduced by a factor of four? □

Problem 16.16. Evolution of wave packet incident on a potential step

(a) Use `TDHalfStepApp` with a step potential beginning at $x = 0$ with height $V_0 = 2$. Choose $x_0 = -10$, $k_0 = 2$, $w = 1$, `dx` = 0.4, `dt` = 0.1, `xmin` = −20, and `xmax` = 20. Describe the motion of the wave packet. Does the shape of the wave packet remain a Gaussian for all t? What happens to the wave packet at $x = 0$? Determine the height and width of the reflected and transmitted wave packets, the time t_i for the incident wave to reach the barrier at $x = 0$, and the time t_r for the reflected wave to return to $x = x_0$. Is $t_r = t_i$? If these times are not equal, explain the reason for the difference.

(b) Repeat the analysis in part (a) for a step potential of height $V_0 = 10$. Is $t_r \approx t_i$ in this case?

(c) What is the motion of a classical particle with a kinetic energy corresponding to the central wave vector $k = k_0$? □

Problem 16.17. Scattering of a wave packet from a potential barrier

(a) Consider a potential barrier of the form

$$V(x) = \begin{cases} 0 & (x < 0) \\ V_0 & (0 \le x \le a) \\ 0 & (x > a). \end{cases} \tag{16.38}$$

Generate a series of snapshots that show the wave packet approaching the barrier and then interacting with it to generate reflected and transmitted packets. Choose $V_0 = 2$ and $a = 1$ and consider the behavior of the wave packet for $k_0 = 1$, 1.5, 2, and 3. Does the width of the packet increase with time? How does the width depend on k_0? For what values of k_0 is the motion of the packet in qualitative agreement with the motion of a corresponding classical particle?

(b) Consider a square well with $V_0 = -2$ and consider the same questions as in part (a). □

Problem 16.18. Evolution of two wave packets

Modify `GaussianPacket` in Listing 16.8 to include two wave packets with identical widths and speeds, with the sign of k_0 chosen so that the two wave packets approach each other. Choose their respective values of x_0 so that the two packets are initially well separated. Let $V = 0$ and describe what happens when you determine their time dependence. Do the packets influence each other? What do your results imply about the existence of a superposition principle? □

16.6 Fourier Transformations and Momentum Space

The position space wave function, $\Psi(x,t)$, is only one of many possible representations of a quantum mechanical state. A quantum system also is completely characterized by the momentum space wave function, $\Phi(p,t)$. The probability $P(p,t)\,\Delta p$ of the particle being in a "volume" element Δp centered about the momentum p at time t is equal to

$$P(p,t)\,\Delta p = |\Phi(p,t)|^2\,\Delta p. \tag{16.39}$$

Because either a position space or a momentum space representation provides a complete description of the system, it is possible to transform the wave function from one space to another as:

$$\Phi(p,t) = \frac{1}{\sqrt{2\pi\hbar}}\int_{-\infty}^{\infty}\Psi(x,t)e^{-ipx/\hbar}\,dx \tag{16.40}$$

$$\Psi(x,t) = \frac{1}{\sqrt{2\pi\hbar}}\int_{-\infty}^{\infty}\Phi(p,t)e^{ipx/\hbar}\,dp. \tag{16.41}$$

The momentum and position space transformations, (16.40) and (16.41), are Fourier integrals. Because a computer stores a wave function on a finite grid, these transformations simplify to the familiar Fourier series (see Section 9.3):

$$\Phi_m = \sum_{n=-N/2}^{N/2}\Psi_n e^{-ip_m x_n/\hbar}, \tag{16.42}$$

$$\Psi_n = \frac{1}{N}\sum_{m=-N/2}^{N/2}\Phi_m e^{ip_m x_n/\hbar}, \tag{16.43}$$

where $\Phi_m = \Phi(p_m)$ and $\Psi_n = \Psi(x_n)$. We have not explicitly shown the time dependence in (16.42) and (16.43).

We now use the FFTApp program introduced in Section 9.3 to transform a wave function between position and momentum space. Note that the wavenumber $2\pi/\lambda$ (or $2\pi/T$ in the time domain) in classical physics has the same numerical value as momentum in quantum mechanics $p = h/\lambda = 2\pi\hbar/\lambda$ in units such that $\hbar = 1$. Consequently, we can use the getWrappedOmega and getNaturalOmega methods in the FFT class to generate arrays containing momentum values for a transformed position space wave function.

The FFTApp program in Listing 9.7 transforms N complex data points using an input array that has length $2N$. The real part of the jth data point is stored in array element $2j$ and the imaginary part is stored in element $2j+1$. The FFT class transforms this array and maintains the same ordering of real and imaginary parts. However, the momenta (wavenumbers) are in warp-around order starting with the zero momentum coefficients in the first two elements and switching to negative momenta halfway through the array. The toNaturalOrder class sorts the array in order of increasing momentum. We use the FFTApp class in Problem 16.19.

Problem 16.19. Transforming to momentum space

(a) The FFTApp class initializes the wave function grid using the following complex exponential:

$$\Psi_n = \Psi(n\Delta x) = e^{in\Delta x} = \cos n\Delta x + i\sin n\Delta x. \tag{16.44}$$

Use FFTApp to show that a complex exponential has a definite momentum if the grid contains an integer number of wavelengths. In other words, show that there is only one nonzero Fourier component.

(b) How small a wavelength (or how large a momentum) can be modeled if the spatial grid has N points and extends over a distance L?

(c) Where do the maximum, zero, and minimum values of the momentum occur in wrap-around order? □

After the transformation, the momentum space wave function is stored in an array. The array elements can be assigned a momentum value using the de Broglie relation $p = h/\lambda$. The longest wavelength that can exist on the grid is equal to the grid dimension $L = (N-1)\Delta x$, and this wave has a momentum of

$$p_0 = \frac{h}{L}. \tag{16.45}$$

Points on the momentum grid have momentum values with integer multiples of p_0.

Problem 16.20. Momentum visualization

Add a ComplexPlotFrame to the FFTApp program to show the momentum space wave function of a position space Gaussian wave packet. Add a user interface to control the width of the Gaussian wave packet and verify the Heisenberg uncertainty relation $\Delta x \Delta p \geq \hbar/2$. Shift the center of the position space wave packet and explain the change in the resulting momentum space wave function. □

Problem 16.21. Momentum time evolution

Modify TDHalfStepApp so that it displays the momentum space wave function in addition to the position space wave function. Describe the momentum space evolution of a Gaussian packet for the infinite square well and a simple harmonic oscillator potential. What evidence of classical-like behavior do you observe? □

The FFT can be used to implement a fast and accurate method for solving Schrödinger's equation. We start by writing (16.4) in operator notation as

$$i\hbar \frac{\partial \Psi(x,t)}{\partial t} = \hat{H}\Psi(x,t) = (\hat{T} + \hat{V})\Psi(x,t), \tag{16.46}$$

where $\hat{H}$, $\hat{T}$, and $\hat{V}$ are the Hamiltonian, kinetic energy, and potential energy operators, respectively. The formal solution to (16.46) is

$$\Psi(x,t) = e^{-i\hat{H}(t-t_0)/\hbar}\Psi(x,t_0) = e^{-i(\hat{T}+\hat{V})(t-t_0)/\hbar}\Psi(x,t_0). \tag{16.47}$$

The time evolution operator $\hat{U}$ is defined as

$$\hat{U} = e^{-i\hat{H}(t-t_0)/\hbar} = e^{-i(\hat{T}+\hat{V})(t-t_0)/\hbar}. \tag{16.48}$$

It might be tempting to express the time evolution operator as

$$\hat{U} = e^{-i\hat{T}\Delta t/\hbar} e^{-i\hat{V}\Delta t/\hbar}, \tag{16.49}$$

but (16.49) is valid only for $\Delta t \equiv t - t_0 << 1$, because $\hat{T}$ and $\hat{V}$ do not commute. A more accurate approximation (accurate to second order in Δt) is obtained by using the following symmetric decomposition:

$$\hat{U} = e^{-i\hat{V}\Delta t/2\hbar} e^{-i\hat{T}\Delta t/\hbar} e^{-i\hat{V}\Delta t/2\hbar}. \tag{16.50}$$

The key to using (16.50) to solve (16.46) is to use the position space wave function when applying $e^{-i\hat{V}\Delta t/2\hbar}$ and to use the momentum space wave function when applying $e^{-i\hat{T}\Delta t/2\hbar}$. In position space, the potential energy operator is equivalent to simply multiplying by the potential energy function. That is, the effect of the first and last terms in (16.50) is to multiply points on the position grid by a phase factor that is proportional to the potential energy:

$$\tilde{\Psi}_j = e^{-iV(x_j)\Delta t/2\hbar} \Psi_j. \tag{16.51}$$

Because the kinetic energy operator in position space involves partial derivatives, it is convenient to transform both the operator and the wave function to momentum space. In momentum space the kinetic energy operator is equivalent to multiplying by the kinetic energy $p^2/2m$. The middle term in (16.50) operates by multiplying points on the momentum grid by a phase factor that is proportional to the kinetic energy:

$$\tilde{\Phi}_j = e^{-ip_j^2\Delta t/2m} \Phi_j. \tag{16.52}$$

The split-operator algorithm jumps back and forth between position and momentum space to propagate the wave function. The algorithm starts in position space where each grid value $\Psi_j = \Psi(x_j, t)$ is multiplied by (16.51). The wave function is then transformed to momentum space where every momentum value Φ_j is multiplied by (16.52). It is then transformed back to position space where (16.51) is applied a second time. A single time step can therefore be written as

$$\Psi(x, t + \Delta t) = e^{-iV(x)\Delta t/2\hbar} F^{-1}\Big[e^{-ip^2\Delta t/2m} F[e^{-iV(x)\Delta t/2\hbar}\Psi(x,t)]\Big], \tag{16.53}$$

where F is the Fourier transform to momentum space and F^{-1} is its inverse.

Problem 16.22. Split-operator algorithm

(a) Write a program to implement the split-operator algorithm. It is necessary to evaluate the exponential phase factors only once when implementing the split-operator algorithm. Store the complex exponentials in arrays that match the x-values on the spatial grid and the p-values on the momentum grid. Use wrap-around order when storing the momentum phase factors because the `FFT` class inverse transformation assumes that data are in wrap-around order. You can use the `getWrappedOmega` method in the FFT to obtain the momenta in this ordering.

(b) Compare the evolution of a Gaussian wave packet using the split-operator and half-step algorithms using identical grids. How does the finite grid size affect each algorithm?

(c) Compare the computation speed of the split-operator and half-step algorithms using a Gaussian wave packet in a square well. Disable plotting and other nonessential computation when comparing the speeds. □

Problem 16.23. Split-operator accuracy

The split-operator and half-step algorithms fail if the time step is too large. Use both algorithms to evolve a simple harmonic oscillator coherent state (see Problem 16.14). Describe the error that occurs if the time step becomes too large. □

16.7 Variational Methods

One way of obtaining a good approximation of the ground state energy is to use a variational method. This approach has numerous applications in chemistry, atomic and molecular physics, nuclear physics, and condensed matter physics. Consider a system whose Hamiltonian operator $\hat{H}$ is given by (16.8). According to the variational principle, the expectation value of the Hamiltonian for an arbitrary trial wave function Ψ is greater than or equal to the ground state energy E_0. That is,

$$\langle H \rangle = E[\Psi] = \frac{\int \Psi^*(x)\hat{H}\Psi(x)\,dx}{\int \Psi^*(x)\Psi(x)\,dx} \geq E_0, \tag{16.54}$$

where E_0 is the exact ground state energy of the system. We assume that the wave function is continuous and bounded. The inequality (16.54) reduces to an equality only if Ψ is an eigenstate of $\hat{H}$ with the eigenvalue E_0. For bound states, Ψ may be assumed to be real without loss of generality so that $\Psi^* = \Psi$ and thus $|\Psi(x)|^2 = \Psi(x)^2$. This assumption implies that we do not need to store two values representing the real and imaginary parts of Ψ.

The inequality (16.54) is the basis of the variational method. The procedure is to choose a physically reasonable form for the trial wave function $\Psi(x)$ that depends on one or more parameters. The expectation value $E[\Psi]$ is computed, and the parameters are varied until a minimum of $E[\Psi]$ is obtained. This value of $E[\Psi]$ is an upper bound to the true ground state energy. Often forms of Ψ are chosen so that the integrals in (16.54) can be done analytically. To avoid this restriction we can use numerical integration methods.

In most applications of the variational method the integrals in (16.54) are multidimensional and Monte Carlo integration methods are essential. For this reason we will use Monte Carlo integration in the following, even though we will consider only one- and two-body problems. Because it is inefficient to simply choose points at random to compute $E[\Psi]$, we rewrite (16.54) in a form that allows us to use importance sampling. We write

$$E[\Psi] = \frac{\int \Psi(x)^2 E_L(x)\,dx}{\int \Psi(x)^2\,dx}, \tag{16.55}$$

where E_L is the *local energy*

$$E_L(x) = \frac{\hat{H}\Psi(x)}{\Psi(x)}, \tag{16.56}$$

which can be calculated analytically using the trial wave function. The form of (16.55) is that of a weighted average with the weight equal to the normalized probability density $\Psi(x)^2/\int \Psi(x)^2\,dx$. As discussed in Section 11.6, we can sample values of x using the distribution $\Psi(x)^2$ so that the Monte Carlo estimate of $E[\Psi]$ is given by the sum

$$E[\Psi] = \lim_{n\to\infty} \frac{1}{n}\sum_{i=1}^{n} E_L(x_i), \tag{16.57}$$

where n is the number of times that x is sampled from Ψ^2. How can we sample from Ψ^2? In general, it is not possible to use the inverse transform method (see Section 11.5) to generate a nonuniform distribution. A convenient alternative is the Metropolis method which has the advantage that only an unnormalized Ψ^2 is needed for the proposed move.

Problem 16.24. Ground state energy of several one-dimensional systems

(a) It is useful to test the variational method on an exactly solvable problem. Consider the one-dimensional harmonic oscillator with $V(x) = x^2/2$. Choose the trial wave function to be $\Psi(x) \propto e^{-\lambda x^2}$, with λ the variational parameter. Generate values of x chosen from a normalized $\Psi^2(x)$ using the inverse transform method and verify that $\lambda = 1/2$ yields the smallest upper bound by considering $\lambda = 1/2$ and four other values of λ near 1/2. Another way to generate a Gaussian distribution is to use the Box–Muller method discussed in Section 11.5.

(b) Repeat part (a) using the Metropolis method to generate x distributed according to $\Psi(x)^2 \propto e^{-2\lambda x^2}$ and evaluate (16.57). As discussed in Section 11.7, the Metropolis method can be summarized by the following steps:

(i) Choose a trial position $x_{\text{trial}} = x_n + \delta_n$, where δ_n is a uniform random number in the interval $[-\delta, \delta]$.

(ii) Compute $w = p(x_{\text{trial}})/p(x_n)$, where in this case $p(x) = e^{-2\lambda x^2}$.

(iii) If $w \geq 1$, accept the change and let $x_{n+1} = x_{\text{trial}}$.

(iv) If $w < 1$, generate a random number r and let $x_{n+1} = x_{\text{trial}}$ if $r \leq w$.

(v) If the trial change is not accepted, then let $x_{n+1} = x_n$.

Remember that it is necessary to wait for equilibrium (convergence to the distribution Ψ^2) before computing the average value of E_L. Look for a systematic trend in $\langle E_L \rangle$ over the course of the random walk. Choose a step size δ that gives a reasonable value for the acceptance ratio. How many trials are necessary to obtain $\langle E_L \rangle$ to within 1% accuracy compared to the exact analytic result?

(c) Instead of finding the minimum of $\langle E_L \rangle$ as a function of the various variational parameters, minimize the quantity

$$\sigma_L^2 = \langle E_L^2 \rangle - \langle E_L \rangle^2. \tag{16.58}$$

Verify that the exact minimum value of $\sigma_L^2[\Psi]$ is zero, whereas the exact minimum value of $E_L[\Psi]$ is unknown in general.

(d) Consider the anharmonic potential $V(x) = \frac{1}{2}x^2 + bx^4$. Plot $V(x)$ as a function of x for $b = 1/8$. Use first-order perturbation theory to calculate the lowest order change in the ground state energy due to the x^4 term. Then choose a reasonable form for your trial wave function and use your Monte Carlo program to estimate the ground state energy. How does your result compare with first-order perturbation theory?

(e) Consider the anharmonic potential of part (d) with $b = -1/8$. Plot $V(x)$. Use first-order perturbation theory to calculate the lowest order change in the ground state energy due to the x^4 term and then use your program to estimate E_0. Do your Monte Carlo estimates for the ground state energy have a lower bound? Why or why not?

(f) Modify your program so that it can be applied to the ground state of the hydrogen atom. In this case we have $V(r) = -e^2/r$, where e is the magnitude of the charge on the electron. The element of integration dx in (16.55) is replaced by $4\pi r^2\,dr$. Choose $\Psi \propto e^{-r/a}$, where a is the variational parameter. Measure lengths in terms of the Bohr radius $\hbar^2/me^2$ and energy in terms of the Rydberg $me^4/2\hbar^2$. In these units $\mu = e^2 = \hbar = 1$. Find the optimal value of a. What is the corresponding energy?

(g) Consider the Yukawa or screened Coulomb potential for which $V(r) = -e^2 e^{-\alpha r}/r$, where $\alpha > 0$. In this case the ground state and wave function can only be obtained numerically. For $\alpha = 0.5$ and $\alpha = 1.0$ the most accurate numerical estimates of E_0 are -0.14808 and -0.01016, respectively. What is a good choice for the form of the trial wave function? How close can you come to these estimates? □

Problem 16.25. Variational estimate of the ground state of Helium

Helium has long served as a testing ground for atomic trial wave functions. Consider the ground state of the helium atom with the interaction

$$V(r_1, r_2) = -2e^2\left(\frac{1}{r_1} + \frac{1}{r_2}\right) + \frac{e^2}{r_{12}}, \tag{16.59}$$

where r_{12} is the separation between the two electrons. Assume that the nucleus is fixed and ignore relativistic effects. Choose $\Psi(\mathbf{r}_1, \mathbf{r}_2) = Ae^{-Z_{\rm eff}(r_1+r_2)/a_0}$, where $Z_{\rm eff}$ is a variational parameter. Estimate the upper bound to the ground state energy based on this functional form of Ψ. □

Our discussion of variational Monte Carlo methods has been only introductory in nature. One important application of variational Monte Carlo methods is to optimize a given trial wave function which is then used to "guide" the Monte Carlo methods discussed in Sections 16.8 and 16.9.

16.8 Random Walk Solutions of the Schrödinger Equation

We now introduce a Monte Carlo approach based on expressing the Schrödinger equation in imaginary time. This approach follows that of Anderson (see references). We will then discuss several other *quantum Monte Carlo* methods. We will see that although the systems of interest are quantum mechanical, we can convert them to systems for which we can use classical Monte Carlo methods.

To understand how we can interpret the Schrödinger equation in terms of a random walk in imaginary time, we substitute $\tau = it/\hbar$ into the time-dependent Schrödinger equation for a free particle and write (in one dimension)

$$\frac{\partial \Psi(x,\tau)}{\partial \tau} = \frac{\hbar^2}{2m}\frac{\partial^2 \Psi(x,\tau)}{\partial x^2}. \tag{16.60}$$

Note that (16.60) is identical in form to the diffusion equation (16.1). Hence, we can interpret the wave function Ψ as a probability density with a diffusion constant $D = \hbar^2/2m$.

From our discussion in Chapter 7, we know that we can use the formal similarity between the diffusion equation and the imaginary-time free particle Schrödinger equation to solve the latter by replacing it by an equivalent random walk problem. To understand how we can interpret the role of the potential energy term in the context of random walks, we write Schrödinger's equation in imaginary time as

$$\frac{\partial \Psi(x,\tau)}{\partial \tau} = \frac{\hbar^2}{2m}\frac{\partial^2 \Psi(x,\tau)}{\partial x^2} - V(x)\Psi(x,\tau). \tag{16.61}$$

If we were to ignore the first term (the diffusion term) on the right side of (16.61), the result would be a first-order differential equation corresponding to a decay or growth process depending on the sign of V. We can obtain the solution to this first-order equation by replacing it by a random decay or growth process, for example, radioactive decay. These considerations suggest that we can interpret (16.61) as a combination of diffusion and branching processes. In the latter, the number of walkers increases or decreases at a point x depending on the sign of $V(x)$. The walkers do not interact with each other because the Schrödinger equation (16.61) is linear in Ψ. Note that it is $\Psi \Delta x$ and *not* $\Psi^2 \Delta x$ that corresponds to the probability distribution of the random walkers. This probabilistic interpretation requires that Ψ be nonnegative and real.

We now use this probabilistic interpretation of (16.61) to develop an algorithm for determining the ground state wave function and energy. The general solution of Schrödinger's equation can be written for imaginary time τ as [see (16.10)]

$$\Psi(x,\tau) = \sum_n c_n\,\phi_n(x)\,e^{-E_n\tau}. \tag{16.62}$$

For sufficiently large τ, the dominant term in the sum in (16.62) comes from the term representing the eigenvalue of lowest energy. Hence, we have

$$\Psi(x,\tau\to\infty) = c_0\,\phi_0(x)\,e^{-E_0\tau}. \tag{16.63}$$

From (16.63) we see that the spatial dependence of $\Psi(x,\tau\to\infty)$ is proportional to the ground state eigenstate $\phi_0(x)$. If $E_0 > 0$, we also see that $\Psi(x,\tau)$ and hence the population of walkers will eventually decay to zero unless $E_0 = 0$. This problem can be avoided by measuring E_0 from an arbitrary reference energy $V_{\rm ref}$, which is adjusted so that an approximate steady state distribution of random walkers is obtained.

Although we could attempt to fit the τ-dependence of the computed probability distribution of the random walkers to (16.63) and thereby extract E_0, it is more convenient to compute E_0 directly from the relation

$$E_0 = \langle V\rangle = \frac{\sum n_i V(x_i)}{\sum n_i}, \tag{16.64}$$

where n_i is the number of walkers at x_i at time τ. An estimate for E_0 can be found by averaging the sum in (16.64) for several values of τ once a steady state distribution of random walkers has been reached. To derive (16.64), we rewrite (16.61) and (16.63) by explicitly introducing the reference potential $V_{\rm ref}$:

$$\frac{\partial\Psi(x,\tau)}{\partial\tau} = \frac{\hbar^2}{2m}\frac{\partial^2\Psi(x,\tau)}{\partial x^2} - \big[V(x) - V_{\rm ref}\big]\Psi(x,\tau), \tag{16.65}$$

and

$$\Psi(x,\tau) \approx c_0\phi_0(x)\,e^{-(E_0 - V_{\rm ref})\tau}. \tag{16.66}$$

We first integrate (16.65) with respect to x. Because $\partial\Psi(x,\tau)/\partial x$ vanishes in the limit $|x|\to\infty$, $\int(\partial^2\Psi/\partial x^2)\,dx = 0$, and hence

$$\int \frac{\partial\Psi(x,\tau)}{\partial\tau}\,dx = -\int V(x)\,\Psi(x,\tau)\,dx + V_{\rm ref}\int \Psi(x,\tau)\,dx. \tag{16.67}$$

If we differentiate (16.66) with respect to τ, we obtain the relation

$$\frac{\partial\Psi(x,\tau)}{\partial\tau} = (V_{\rm ref} - E_0)\Psi(x,\tau). \tag{16.68}$$

We then substitute (16.68) for $\partial\Psi/\partial\tau$ into (16.67) and find

$$\int (V_{\rm ref} - E_0)\Psi(x,\tau)\,dx = -\int V(x)\Psi(x,\tau)\,dx + V_{\rm ref}\int \Psi(x,\tau)\,dx. \tag{16.69}$$

If we cancel the terms proportional to V_{ref} in (16.69), we find that

$$E_0 \int \Psi(x,\tau)\,dx = \int V(x),\Psi(x,\tau)\,dx, \tag{16.70}$$

or

$$E_0 = \frac{\int V(x)\Psi(x,\tau)\,dx}{\int \Psi(x,\tau)\,dx}. \tag{16.71}$$

The desired result (16.64) follows by making the connection between $\Psi(x)\Delta x$ and the density of walkers between x and $x+\Delta x$.

Although the derivation of (16.64) is somewhat involved, the random walk algorithm is straightforward. A simple implementation of the algorithm is as follows:

1. Place a total of N_0 walkers at the initial set of positions x_i, where the x_i need not be on a grid.

2. Compute the reference energy $V_{\text{ref}} = \sum_i V_i/N_0$.

3. Randomly move the first walker to the right or left by a fixed step length Δs. The step length Δs is related to the time step $\Delta\tau$ by $(\Delta s)^2 = 2D\Delta\tau$. ($D = 1/2$ in units such that $\hbar = m = 1$.)

4. Compute $\Delta V = V(x) - V_{\text{ref}}$ and a random number r in the unit interval. If $\Delta V > 0$ and $r < \Delta V\Delta\tau$, then remove the walker. If $\Delta V < 0$ and $r < -\Delta V\Delta\tau$, then add another walker at x. Otherwise, just leave the walker at x. This procedure is accurate only in the limit of $\Delta\tau << 1$. A more accurate procedure consists of computing $P_b = e^{-\Delta V\Delta\tau} - 1 = n + f$, where n is the integer part of P_b, and f is the fractional part. We then make n copies of the walker, and if $f > r$, we make one more copy.

5. Repeat steps 3 and 4 for each of the N_0 walkers and compute the mean potential energy (16.71) and the actual number of random walkers. The new reference potential is given by

$$V_{\text{ref}} = \langle V\rangle - \frac{a}{N_0\Delta\tau}(N - N_0), \tag{16.72}$$

   where N is the new number of random walkers, and $\langle V\rangle$ is their mean potential energy. The average of V is an estimate of the ground state energy. The parameter a is adjusted so that the number of random walkers N remains approximately constant.

6. Repeat steps 3–5 until the estimates of the ground state energy $\langle V\rangle$ have reached a steady state value with only random fluctuations. Average $\langle V\rangle$ over many Monte Carlo steps to compute the ground state energy. Do a similar calculation to estimate the distribution of random walkers.

The `QMWalk` class implements this algorithm for the harmonic oscillator potential. Initially, the walkers are randomly distributed within a distance `initialWidth` of the origin. The program also estimates the ground state wave function by accumulating the spatial distribution of the walkers at discrete intervals of position. The input parameters are the desired number of walkers N_0, the number of position intervals to accumulate data for the ground state wave function `numberOfBins`, and the step size `ds`. We also use `ds` for the interval size in the wave function computation. The program computes the current number of walkers, the estimate of the ground state energy, and the value of V_{ref}. The unnormalized ground state wave function is also plotted.

Listing 16.10: The `QMWalk` class calculates the ground state of the simple harmonic oscillator using the random walk Monte Carlo algorithm.

```
package org.opensourcephysics.sip.ch16;
public class QMWalk {
   int numberOfBins = 1000; // for wave function
   double[] x;              // positions of walkers
   double[] phi0;           // estimate of ground state wave function
   double[] xv;             // x values for computing phi0
   int N0;                  // desired number of walkers
   int N;                   // actual number of walkers
   double ds;               // step size
   double dt;               // time interval
   double vave = 0;         // mean potential
   double vref = 0;         // reference potential
   double eAccum = 0;       // accumulation of energy values
   double xmin;             // minimum x
   int mcs;

   public void initialize() {
      N0 = N;
      x = new double[2*numberOfBins];
      phi0 = new double[numberOfBins];
      xv = new double[numberOfBins];
      // minimum location for computing phi0
      xmin = -ds*numberOfBins/2.0;
      double binEdge = xmin;
      for(int i = 0;i<numberOfBins;i++) {
         xv[i] = binEdge;
         binEdge += ds;
      }
      // initial width for location of walkers
      double initialWidth = 1;
      for(int i = 0;i<N;i++) {
         // initial random location of walkers
         x[i] = (2*Math.random()-1)*initialWidth;
         vref += potential(x[i]);
      }
      vave = 0;
      vref = 0;
      eAccum = 0;
      mcs = 0;
      dt = ds*ds;
   }

   void walk() {
      double vsum = 0;
      for(int i = N-1;i>=0;i--) {
         if(Math.random()<0.5) { // move walker
            x[i] += ds;
         } else {
            x[i] -= ds;
         }
         double pot = potential(x[i]);
         double dv = pot-vref;
         vsum += pot;
         if(dv<0) {                  // decide to add or delete walker
            if(N==0||(Math.random()<-dv*dt)&&(N<x.length)) {
               x[N] = x[i];          // new walker at the current location
               vsum += pot;          // add energy of new walker
               N++;
            }
         } else {
            if((Math.random()<dv*dt)&&(N>0)) {
               N--;
               // relabel last walker to deleted walker index
               x[i] = x[N];
               vsum -= pot;          // subtract energy of deleted walker
            }
         }
      }
      vave = (N==0) ? 0 // if no walkers poential = 0
                    : vsum/N;
      vref = vave-(N-N0)/N0/dt;
      mcs++;
   }

   void doMCS() {
      walk();
      eAccum += vave;
      for(int i = 0;i<N;i++) {
         int bin = (int) Math.floor((x[i]-xmin)/ds); // bin index
         if(bin>=0&&bin<numberOfBins) {
            phi0[bin]++;
         }
      }
   }

   void resetData() {
      for(int i = 0;i<numberOfBins;i++) {
         phi0[i] = 0;
      }
      eAccum = 0;
      mcs = 0;
   }

   public double potential(double x) {
      return 0.5*x*x;
   }
}
```

Listing 16.11: The QMWalkApp class computes and displays the result of a random walk Monte Carlo calculation.

```
package org.opensourcephysics.sip.ch16;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.frames.PlotFrame;
```

```
public class QMWalkApp extends AbstractSimulation {
   PlotFrame phiFrame = new PlotFrame("x", "Phi_0", "Phi_0(x)");
   QMWalk qmwalk = new QMWalk();

   public void initialize() {
      qmwalk.N = control.getInt("initial number of walkers");
      qmwalk.ds = control.getDouble("step size ds");
      qmwalk.numberOfBins =
           control.getInt("number of bins for wavefunction");
      qmwalk.initialize();
   }

   public void doStep() {
      qmwalk.doMCS();
      phiFrame.clearData();
      phiFrame.append(0, qmwalk.xv, qmwalk.phi0);
      phiFrame.setMessage("E = "+decimalFormat.format(
         qmwalk.eAccum/qmwalk.mcs)+" N = "+qmwalk.N);
   }

   public void reset() {
      control.setValue("initial number of walkers", 50);
      control.setValue("step size ds", 0.1);
      control.setValue("number of bins for wavefunction", 100);
      enableStepsPerDisplay(true);
   }

   public void resetData() {
      qmwalk.resetData();
      phiFrame.clearData();
      phiFrame.repaint();
   }

   public static void main(String[] args) {
      SimulationControl control =
         SimulationControl.createApp(new QMWalkApp());
      control.addButton("resetData", "Reset Data");
   }
}
```

Problem 16.26. Ground state of the harmonic and anharmonic oscillators

(a) Use `QMWalk` and `QMWalkApp` to estimate the ground state energy E_0 and the corresponding eigenstate for $V(x) = x^2/2$. Choose the desired number of walkers $N_0 = 50$, the step length $\text{ds} = 0.1$, and `numberOfBins` = 100. Place the walkers at random within the range $-1 \le x \le 1$. Compare your Monte Carlo estimate for E_0 to the exact result $E_0 = 0.5$.

(b) Reset your data averages after the averages seemed to have converged and compute the averages again. How many Monte Carlo steps per walker are needed for 1% accuracy in E_0? Plot the probability distribution of the random walkers and compare it to the exact result for the ground state wave function.

(c) Modify `QMWalk` so that more than one copy of the walker can be created at each step (see step 4 on page 691). How much better does the algorithm work now? Can you use a larger step size or fewer Monte Carlo steps to obtain the same accuracy?

(d) Obtain a numerical solution of the anharmonic oscillator with

$$V(x) = \frac{1}{2}x^2 + bx^3. \tag{16.73}$$

Consider $b = 0.1$, 0.2, and 0.5. A calculation of the effect of the x^3 term is necessary for the study of the anharmonicity of the vibrations of a physical system, for example, the vibrational spectrum of diatomic molecules. □

Problem 16.27. Ground state of a square well

(a) Modify `QMWalkApp` to find the ground state energy and wave function for the finite square-well potential (16.13) with $a = 1$ and $V_0 = 5$. Choose $N_0 = 100$, `ds` $= 0.1$, and `numberOfBins` $=$ 100. Place the walkers at random within the range $-1.5 \leq x \leq 1.5$.

(b) Increase V_0 and find the ground state energy as a function of V_0. Use your results to estimate the limiting value of the ground state energy for $V_0 \to \infty$. □

Problem 16.28. Ground state of a cylindrical box

Compute the ground state energy and wave function of the circular potential

$$V(r) = \begin{cases} 0 & (r \leq 1) \\ -V_0, & (r > 1), \end{cases} \tag{16.74}$$

where $r^2 = x^2 + y^2$. Modify `QMWalkApp` by using Cartesian coordinates in two dimensions. For example, add an array to store the positions of the y-coordinates of the walkers. What happens if you begin with an initial distribution of walkers that is not cylindrically symmetric? □

16.9 Diffusion Quantum Monte Carlo

We now discuss an improvement of the random walk algorithm known as *diffusion quantum Monte Carlo*. Although some parts of the discussion might be difficult to follow initially, the algorithm is straightforward. Your understanding of the method will be enhanced by writing a program to implement the algorithm and then reading the following derivation again.

To provide some background, we introduce the concept of a Green's function or propagator defined by

$$\Psi(x,\tau) = \int G(x,x',\tau)\Psi(x',0)\,dx'. \tag{16.75}$$

From the form of (16.75) we see that $G(x,x',\tau)$ "propagates" the wave function from time zero to time τ. If we operate on both sides of (16.75) with first $(\partial/\partial\tau)$ and then with $(H - V_{\rm ref})$, we can verify that G satisfies the equation

$$\frac{\partial G}{\partial \tau} = -(\hat{H} - V_{\rm ref})G, \tag{16.76}$$

which is the same form as the imaginary-time Schrödinger equation (16.65). It is easy to verify that $G(x,x',\tau) = G(x',x,\tau)$. A formal solution of (16.76) is

$$G(\tau) = e^{-(\hat{H}-V_{\text{ref}})\tau}, \tag{16.77}$$

where the meaning of the exponential of an operator is given by its Taylor series expansion.

The difficulty with (16.77) is that the kinetic and potential energy operators $\hat{T}$ and $\hat{V}$ in $\hat{H}$ do not commute. For this reason, if we want to write the exponential in (16.77) as a product of two exponentials, we can only approximate the exponential for short times $\Delta\tau$. To first order in $\Delta\tau$ (higher-order terms involve the commutator of $\hat{V}$ and $\hat{H}$), we have

$$G(\Delta\tau) \approx G_{\text{branch}} G_{\text{diffusion}} \tag{16.78}$$

$$= e^{-(V-V_{\text{ref}})\Delta\tau} e^{-\hat{T}\Delta\tau}, \tag{16.79}$$

where $G_{\text{diffusion}} \equiv e^{-\hat{T}\Delta\tau}$ and $G_{\text{branch}} \equiv e^{-(\hat{V}-V_{\text{ref}})\Delta\tau}$ correspond to the two random processes: diffusion and branching. From (16.76) we see that $G_{\text{diffusion}}$ and G_{branch} satisfy the differential equations:

$$\frac{\partial G_{\text{diffusion}}}{\partial\tau} = -\hat{T} G_{\text{diffusion}} = \frac{\hbar^2}{2m}\frac{\partial^2 G_{\text{diffusion}}}{\partial x^2} \tag{16.80}$$

$$\frac{\partial G_{\text{branch}}}{\partial\tau} = (V_{\text{ref}} - \hat{V}) G_{\text{branch}}. \tag{16.81}$$

The solutions to (16.79)–(16.81) that are symmetric in x and x' are

$$G_{\text{diffusion}}(x,x',\Delta\tau) = (4\pi D\Delta\tau)^{-1/2} e^{-(x-x')^2/4D\Delta\tau}, \tag{16.82}$$

with $D \equiv \hbar^2/2m$, and

$$G_{\text{branch}}(x,x',\Delta\tau) = e^{-(\frac{1}{2}[V(x)+V(x')]-V_{\text{ref}})\Delta\tau}. \tag{16.83}$$

From the form of (16.82) and (16.83), we can see that the diffusion quantum Monte Carlo method is similar to the random walk algorithm discussed in Section 16.8. An implementation of the diffusion quantum Monte Carlo method in one dimension can be summarized as follows:

1. Begin with a set of N_0 random walkers. There is no lattice so the positions of the walkers are continuous. It is advantageous to choose the walkers so that they are in regions of space where the wave function is known to be large.

2. Choose one of the walkers and displace it from x to x'. The new position is chosen from a Gaussian distribution with a variance $2D\Delta\tau$ and zero mean. This change corresponds to the diffusion process given by (16.82).

3. Weight the configuration x' by

$$w(x \to x',\Delta\tau) = e^{-(\frac{1}{2}[V(x)+V(x')]-V_{\text{ref}})\Delta\tau}. \tag{16.84}$$

   One way to do this weighting is to generate duplicate random walkers at x'. For example, if $w \approx 2$, we would have two walkers at x' where previously there had been one. To implement this weighting (branching) correctly, we must make an integer number of copies that is equal on the average to the number w. A simple way to do so is to take the integer part of $w + r$, where r is a uniform random number in the unit interval. The number of copies can be any nonnegative integer including zero. The latter value corresponds to a removal of a walker.

4. Repeat steps 2 and 3 for all members of the ensemble, thereby creating a new ensemble at a later time $\Delta\tau$. One iteration of the ensemble is equivalent to performing the integration
$$\Psi(x,\tau) = \int G(x,x',\Delta\tau)\,\Psi(x',\tau-\Delta\tau)\,dx'. \tag{16.85}$$

5. The quantity of interest $\Psi(x,\tau)$ will be independent of the original ensemble $\Psi(x,0)$ if a sufficient number of Monte Carlo steps are taken. As before, we must ensure that $N(\tau)$, the number of walkers at time τ, is kept close to the desired number N_0.

Now we can understand how the simple random walk algorithm discussed in Section 16.8 is an approximation to the diffusion quantum MC algorithm. First, the Gaussian distribution gives the exact distribution for the displacement of a random walker in a time $\Delta\tau$, in contrast to the fixed step size in the simple random walk algorithm which gives the average displacement of a walker. Hence, there are no systematic errors due to the finite step size. Second, if we expand the exponential in (16.83) to first order in $\Delta\tau$ and set $V(x) = V(x')$, we obtain the branching rule used previously. (We use the fact that the uniform distribution r is the same as the distribution $1-r$.) However, the diffusion quantum MC algorithm is not exact because the branching is independent of the position reached by diffusion, which is only true in the limit $\Delta\tau \to 0$. This limitation is remedied in the Green's function Monte Carlo method where a short time approximation is not made (see the articles on Green's function Monte Carlo in the references).

One limitation of the two random walk methods we have discussed is that they can become very inefficient. This inefficiency is due in part to the branching process. If the potential becomes large and negative (as it is for the Coulomb potential when an electron approaches a nucleus), the number of copies of a walker will become very large. It is possible to improve the efficiency of these algorithms by introducing an importance sampling method. The idea is to use an initial guess $\Psi_T(x)$ for the wave function to guide the walkers toward the more important regions of $V(x)$. To implement this idea, we introduce the function $f(x,\tau) = \Psi(x,\tau)\Psi_T(x)$. If we calculate the quantity $\partial f/\partial t - D\,\partial^2 f/\partial x^2$ and use (16.65), we can show that $f(x,\tau)$ satisfies the differential equation
$$\frac{\partial f}{\partial \tau} = D\frac{\partial^2 f}{\partial x^2} - D\frac{\partial\big[fF(x)\big]}{\partial x} - [E_L(x) - V_{\text{ref}}]f, \tag{16.86}$$
where
$$F(x) = \frac{2}{\Psi_T}\frac{\partial \Psi_T}{\partial x}, \tag{16.87}$$
and the local energy $E_L(x)$ is given by
$$E_L(x) = \frac{\hat{H}\partial_T}{\Psi_T} = V(x) - \frac{D}{\Psi_T}\partial^2\Psi_T\partial x^2. \tag{16.88}$$
The term in (16.86) containing F corresponds to a drift in the walkers away from regions where $|\Psi_T|^2$ is small (see Problem 7.43).

To incorporate the drift term into $G_{\text{diffusion}}$, we replace $(x-x')^2$ in (16.82) by the term $\big(x - x' - D\Delta\tau F(x')\big)^2$ so that the diffusion propagator becomes
$$G_{\text{diffusion}}(x,x',\Delta\tau) = (4\pi D\Delta\tau)^{-1/2} e^{-(x-x'-D\Delta\tau F(x'))^2/4D\Delta\tau}. \tag{16.89}$$

However, this replacement destroys the symmetry between x and x'. To restore it, we use the Metropolis algorithm for accepting the new position of a walker. The acceptance probability p is given by

$$p = \frac{|\Psi_T(x')|^2\, G_{\text{diffusion}}(x, x', \Delta\tau)}{|\Psi_T(x)|^2\, G_{\text{diffusion}}(x', x, \Delta\tau)}. \tag{16.90}$$

If $p > 1$, we accept the move; otherwise, we accept the move if $r \leq p$. The branching step is achieved by using (16.83) with $V(x) + V(x')$ replaced by $E_L(x) + E_L(x')$ and $\Delta\tau$ replaced by an effective time step. The reason for the use of an effective time step in (16.83) is that some diffusion steps are rejected. The effective time step to be used in (16.83) is found by multiplying $\Delta\tau$ by the average acceptance probability. It can be shown (see Hammond et al.) that the mean value of the local energy is an unbiased estimator of the ground state energy.

Another possible improvement is to periodically replace branching (which changes the number of walkers) with a weighting of the walkers. At each weighting step, each walker is weighted by G_{branch}, and the total number of walkers remains constant. After n steps, the kth walker receives a weight $W_k = \Pi_{i=1}^{n} G_{\text{branch}}^{(i,k)}$, where $G_{\text{branch}}^{(i,k)}$ is the branching factor of the kth walker at the ith time step. The contribution to any average quantity of the kth walker is weighted by W_k.

Problem 16.29. Diffusion Quantum Monte Carlo

(a) Modify QMWalkApp to implement the diffusion quantum Monte Carlo method for the systems considered in Problems 16.26 and 16.27. Begin with $N_0 = 100$ walkers and $\Delta\tau = 0.01$. Use at least three values of $\Delta\tau$ and extrapolate your results to $\Delta\tau \to 0$. Reasonable results can be obtained by adjusting the reference energy every 20 Monte Carlo steps with $a = 0.1$.

(b) Write a program to apply the diffusion quantum Monte Carlo method to the hydrogen atom. In this case a configuration is represented by three coordinates.

(c)* Modify your program to include weights in addition to changing walker populations. Redo part (a) and compare your results. □

***Problem 16.30.** Importance sampling

(a) Derive the partial differential equation (16.86) for $f(x, \tau)$.

(b) Modify QMWalkApp to implement the diffusion quantum Monte Carlo method with importance sampling. Consider the harmonic oscillator problem with the trial wave function $\Psi_T = e^{-\lambda x^2}$. Compute the statistical error associated with the ground state energy as a function of λ. How much variance reduction can you achieve relative to the naive diffusion quantum Monte Carlo method? Then consider another form of Ψ_T that does not have a form identical to the exact ground state. Try the hydrogen atom with $\Psi_T = e^{-\lambda r}$. □

16.10 Path Integral Quantum Monte Carlo

The Monte Carlo methods we have discussed so far are primarily useful for estimating the ground state energy and wave function, although it is also possible to find the first few excited states with some effort. In this section we discuss a Monte Carlo method that is of particular interest for computing the thermal properties of quantum systems.

We recall (see Section 7.10) that classical mechanics can be formulated in terms of the principle of least action. That is, given two points in space-time, a classical particle chooses the path that minimizes the action given by

$$S = \int_{x_0,0}^{x,t} L\,dt. \tag{16.91}$$

The Lagrangian L is given by $L = T - V$. Quantum mechanics also can be formulated in terms of the action (cf. Feynman and Hibbs). The result of this *path integral* formalism is that the real-time propagator G can be expressed as

$$G(x, x_0, t) = A \sum_{\text{paths}} e^{iS/\hbar}, \tag{16.92}$$

where A is a normalization factor. The sum in (16.92) is over all paths between $(x_0, 0)$ and (x, t), not just the path that minimizes the classical action. The presence of the imaginary number i in (16.92) leads to interference effects. As before, the propagator $G(x, x_0, t)$ can be interpreted as the probability amplitude for a particle to be at x at time t given that it was at x_0 at time zero. G satisfies the equation [see (16.75)]

$$\Psi(x,t) = \int G(x, x_0, t)\Psi(x_0, 0)\,dx_0 \qquad (t > 0). \tag{16.93}$$

Because G satisfies the same differential equation as Ψ in both x and x_0, G can be expressed as

$$G(x, x_0, t) = \sum_n \phi_n(x)\phi_n(x_0)e^{-iE_n t/\hbar}, \tag{16.94}$$

where the ϕ_n are the eigenstates of H. For simplicity, we set $\hbar = 1$ in the following. As before, we substitute $\tau = it$ into (16.94) and obtain

$$G(x, x_0, \tau) = \sum_n \phi_n(x)\phi_n(x_0)\,e^{-\tau E_n}. \tag{16.95}$$

We first consider the ground state. In the limit $\tau \to \infty$, we have

$$G(x, x, \tau) \to \phi_0(x)^2\,e^{-\tau E_0} \qquad (\tau \to \infty). \tag{16.96}$$

From the form of (16.96) and (16.92), we see that we need to compute G and hence S to estimate the properties of the ground state.

To compute S, we convert the integral in (16.91) to a sum. The Lagrangian for a single particle of unit mass in terms of τ becomes

$$L = -\frac{1}{2}\left(\frac{dx}{d\tau}\right)^2 - V(x) = -E. \tag{16.97}$$

We divide the imaginary time interval τ into N equal steps of size $\Delta\tau$ and write E as

$$E(x_j, \tau_j) = \frac{1}{2}\frac{(x_{j+1} - x_j)^2}{(\Delta\tau)^2} + V(x_j), \tag{16.98}$$

where $\tau_j = j\Delta\tau$, and x_j is the corresponding displacement. The action becomes

$$S = -i\Delta\tau \sum_{j=0}^{N-1} E(x_j, \tau_j) = -i\Delta\tau\Big[\sum_{j=0}^{N-1} \frac{1}{2}\frac{(x_{j+1} - x_j)^2}{(\Delta\tau)^2} + V(x_j)\Big], \tag{16.99}$$

and the probability amplitude for the path becomes

$$e^{iS} = e^{\Delta\tau\left[\sum_{j=0}^{N-1} \frac{1}{2}(x_{j+1}-x_j)^2/(\Delta\tau)^2+V(x_j)\right]}. \tag{16.100}$$

Hence, the propagator $G(x, x_0, N\Delta\tau)$ can be expressed as

$$G(x, x_0, N\Delta\tau) = A\int dx_1 \cdots dx_{N-1}\, e^{\Delta\tau\left[\sum_{j=0}^{N-1} \frac{1}{2}(x_{j+1}-x_j)^2/(\Delta\tau)^2+V(x_j)\right]}, \tag{16.101}$$

where $x \equiv x_N$ and A is an unimportant constant.

From (16.101) we see that $G(x, x_0, N\Delta\tau)$ has been expressed as a multidimensional integral with the displacement variable x_j associated with the time τ_j. The sequence $x_0, x_1, \ldots, x_N$ defines a possible path, and the integral in (16.101) is over all paths. Because the quantity of interest is $G(x, x, N\Delta\tau)$ [see (16.96)], we adopt the periodic boundary condition $x_N = x_0$. The choice of x in the argument of G is arbitrary for finding the ground state energy, and the use of the periodic boundary conditions implies that no point in the closed path is unique. It is thus possible (and convenient) to rewrite (16.101) by letting the sum over j go from 1 to N:

$$G(x_0, x_0, N\Delta\tau) = A\int dx_1 \cdots dx_{N-1}\, e^{-\Delta\tau\left[\sum_{j=1}^{N} \frac{1}{2}(x_j-x_{j-1})^2/(\Delta\tau)^2+V(x_j)\right]}, \tag{16.102}$$

where we have written x_0 instead of x because the x_j that is not integrated over is $x_N = x_0$.

The result of this analysis is to convert a quantum mechanical problem for a single particle into a statistical mechanics problem for N "atoms" on a ring connected by nearest neighbor "springs" with spring constant $1/(\Delta\tau)^2$. The label j denotes the order of the atoms in the ring.

Note that the form of (16.102) is similar to the form of the Boltzmann distribution. Because the partition function for a single quantum mechanical particle contains terms of the form $e^{-\beta E_n}$, and (16.95) contains terms proportional to $e^{-\tau E_n}$, we make the correspondence $\beta = \tau = N\Delta\tau$. We shall see in the following how we can use this identity to simulate a quantum system at a finite temperature.

We can use the Metropolis algorithm to simulate the motion of N "atoms" on a ring. Of course, these atoms are a product of our analysis just as were the random walkers we introduced in diffusion Monte Carlo and should not be confused with real particles. A possible path integral algorithm can be summarized as follows:

1. Choose N and $\Delta\tau$ such that $N\Delta\tau >> 1$ (the zero temperature limit). Also choose δ, the maximum trial change in the displacement of an atom, and `mcs`, the total number of Monte Carlo steps per atom.

2. Choose an initial configuration for the displacements x_j that is close to the approximate shape of the ground state probability amplitude.

3. Choose an atom j at random and a trial displacement $x_j' \to x_j' + (2r-1)\delta$, where r is a uniform random number in the unit interval. Compute the change ΔE in the energy E, where ΔE is given by

$$\begin{aligned}\Delta E \;=\; & \frac{1}{2}\left[\frac{x_{j+1}'-x_j'}{\Delta\tau}\right]^2 + \frac{1}{2}\left[\frac{x_j'-x_{j-1}'}{\Delta\tau}\right]^2 + V(x_j') \\ & - \frac{1}{2}\left[\frac{x_{j+1}-x_j}{\Delta\tau}\right]^2 - \frac{1}{2}\left[\frac{x_j-x_{j-1}}{\Delta\tau}\right]^2 - V(x_j).\end{aligned} \tag{16.103}$$

If $\Delta E < 0$, accept the change; otherwise, compute the probability $p = e^{-\Delta\tau\Delta E}$ and a random number r in the unit interval. If $r \le p$, then accept the move; otherwise reject the trial move.

4. Divide the possible x values into equal size bins of width Δx. Update $P(x)$; that is, let $P(x = x_j) \to P(x = x_j) + 1$, where x is the displacement of the atom chosen in step 3 after step 3 is completed. Do this update even if the trial move was rejected.

5. Repeat steps 3 and 4 until a sufficient number of Monte Carlo steps per atom has been obtained. (Do not take data until the memory of the initial path is lost and the system has reached "equilibrium.")

Normalize the probability density $P(x)$ by dividing by the product of N and `mcs`. The ground state energy E_0 is given by

$$E_0 = \sum_x P(x)[T(x) + V(x)], \tag{16.104}$$

where $T(x)$ is the kinetic energy as determined from the virial theorem

$$\Big\langle 2T(x) \Big\rangle = \Big\langle x\frac{dV}{dx} \Big\rangle, \tag{16.105}$$

which is discussed in many texts (see Griffiths for example). It is also possible to compute T from averages over $(x_j - x_{j-1})^2$, but the virial theorem yields a smaller variance. The ground state wave function $\phi(x)$ is obtained from the normalized probability $P(x)\Delta x$ by dividing by Δx and taking the square root.

We can also find the thermodynamic properties of a particle that is connected to a heat bath at temperature $T = 1/\beta$ by not taking the $\beta = N\Delta\tau \to \infty$ limit. To obtain the ground state, which corresponds to the zero temperature limit ($\beta >> 1$), we have to make $N\Delta\tau$ as large as possible. However, we need $\Delta\tau$ to be as small as possible to approximate the continuum time limit. Hence, to obtain the ground state we need a large number of time intervals N. For the finite temperature simulation, we can use smaller values of N for the same level of accuracy as the zero temperature simulation.

The path integral method is very flexible and can be generalized to higher dimensions and many mutually interacting particles. For three dimensions, x_j is replaced by the three-dimensional displacement $\mathbf{r}_j$. Each real particle is represented by a ring of N "atoms" with a spring-like potential connecting each atom within a ring. Each atom in each ring also interacts with the atoms in the other rings through an interparticle potential. If the quantum system is a fluid where indistinguishability is important, then we must consider the effect of exchange by treating the quantum system as a classical polymer system where the "atoms" represent the monomers of a polymer, and where polymers can split up and reform. Chandler and Wolynes discuss how the quantum mechanical effects due to exchanging identical particles can be associated with the chemical equilibrium of the polymers. They also discuss Bose condensation using path integral techniques.

Problem 16.31. Path integral calculation

(a) Write a program to implement the path integral algorithm for the one-dimensional harmonic oscillator potential with $V(x) = x^2/2$. Use the structure of your Monte Carlo Lennard–Jones program from Chapter 15 as a guide.

(b) Let $N\Delta\tau = 15$ and consider $N = 10$, 20, 40, and 80. Equilibrate for at least 2000 Monte Carlo steps per atom and average over at least 5000 mcs. Compare your results with the exact result for the ground state energy given by $E_0 = 0.5$. Estimate the equilibration time for your calculation. What is a good initial configuration? Improve your results by using larger values of $N\Delta\tau$.

(c) Find the mean energy $\langle E\rangle$ of the harmonic oscillator at the temperature T determined by $\beta = N\Delta\tau$. Find $\langle E\rangle$ for $\beta = 1$, 2, and 3 and compare it with the exact result $\langle E\rangle = \frac{1}{2}\coth(\beta/2)$.

(d) Repeat the above calculations for the Morse potential $V(x) = 2(1-e^{-x})^2$. □

16.11 Projects

Many of the techniques described in this chapter can be extended to two-dimensional quantum systems. The `Complex2DFrame` tool in the frames package is designed to show two-dimensional complex scalar fields such as quantum wave functions. Listing 16.13 in Appendix A shows how this class is used to show a two-dimensional Gaussian wave packet with a momentum boost.

Project 16.32. Separable systems in two dimensions

The shooting method is inappropriate for the calculation of eigenstates and eigenvalues in two or more dimensions with arbitrary potential energy functions $V(\mathbf{r})$. However, the special case of separable potentials can be reduced to several one-dimensional problems that can be solved using the numerical methods described in this chapter. Many molecular modeling programs use the Hartree–Fock self-consistent field approximation to model nonseparable systems as a set of one-dimensional problems. Recently, there has been significant progress motivated by a molecular dynamics algorithm developed by Car and Parrinello.

Write a two-dimensional eigenstate class `Eigenstate2d` that calculates eigenstates and eigenvalues for a separable potential of the form

$$V(x,y) = V_1(x) + V_2(y). \tag{16.106}$$

Test this class using the known analytic solutions for the two-dimensional rectangular box and two-dimensional harmonic oscillator. Use this class to model the evolution of superposition states. Under what conditions are there wave function revivals? □

Project 16.33. Excited state wave functions using quantum Monte Carlo

Quantum Monte Carlo methods can be extended to compute the excited state wave functions using a Gram–Schmidt procedure to insure that each excited state is orthogonal to all lower lying states (see Roy et al.). A quantum Monte Carlo method is used to compute the ground state wave function. A trial wave function for the first exited state is then selected and the ground state component is subtracted from the trial wave function. This subtraction is repeated after every iteration of the Monte Carlo algorithm. Because excited states decay with a time constant $e^{-(E_j-E_0)}$, the lowest remaining excited state dominates the remaining wave function. After the first excited state is obtained, the second excited state is computed by subtracting both known states from the trial wave function. This process is repeated to obtain additional wave functions.

Implement this procedure to find the first few excited state wave functions for the one-dimensional harmonic oscillator. Then consider the one-dimensional double-well oscillator

$$V(x) = -\frac{1}{2}kx^2 + a_3x^3 + a_4x^4, \tag{16.107}$$

with $k = 40$, $a_3 = 1$, and $a_4 = 1$. □

Project 16.34. Quantum Monte Carlo in two dimensions

The procedure described in Project 16.33 can be used to compute two-dimensional wave functions (see Roy et al.).

(a) Test your program using a separable two-dimensional double-well potential.

(b) Find the first few excited states for the two-dimensional double-well potential

$$V(x,y) = -\frac{1}{2}k_x x^2 - \frac{1}{2}k_y y^2 + \frac{1}{2}(a_{xx}x^4 + 2a_{xy}x^2y^2 + a_{yy}y^4), \tag{16.108}$$

with $k_x = k_y = 20$ and $a_{xx} = a_{yy} = a_{xy} = 5$. Repeat with $k_x = k_y = 20$ and $a_{xx} = a_{yy} = a_{xy} = 1$. □

Project 16.35. Evolution of a wave packet in two dimensions

Both the half-step and split-operator algorithms can be extended to model the evolution of two-dimensional systems with arbitrary potentials $V(x,y)$. (See *Numerical Recipes* for how the FFT algorithm is extended to more dimensions.) Implement either algorithm and model a wave packet scattering from a central barrier and a wave packet passing through a double slit.

A clever way to insure stability in the half-step algorithm is to use a boolean array to tag grid locations where the solution becomes unstable and to set the wave function to zero at these grid points.

```
double minV = -2/dt;
double maxVx = 2/dt-2/(dx*dx);
double maxVy = 2/dt-2/(dy*dy);
double maxV = Math.min(maxVx,maxVy);
for(int i = 0, n = potential.length; i <= n; i++) {
   for(int j = 0, m = potential[0].length; j <= m; j++) {
      if (potential[i][j] >= minV && potential[i][j] <= maxV) // stable
         stable[i][j] = true;  // stable
      else
         stable[i][j] = false; // unstable, set wave function to zero
       }
    }
}
```

Project 16.36. Two-particle system

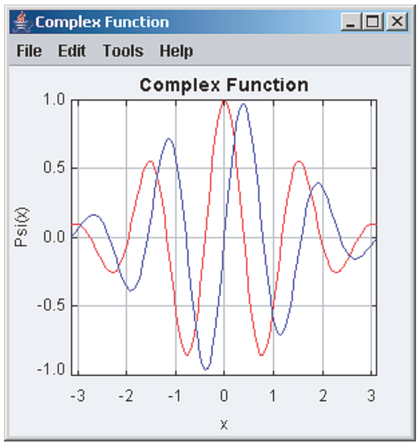
Rubin Landau has studied the time dependence of two particles interacting in one dimension with a potential that depends on their relative separation:

$$V(x_1,x_2) = V_0 e^{-(x_1-x_2)^2/2\alpha^2}. \tag{16.109}$$

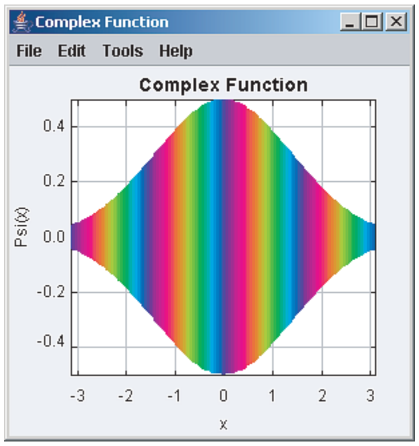
Model a scattering experiment for particles having momentum p_1 and p_2 by assuming the following (unnormalized) initial wave function:

$$\Psi(x_1,x_2) = e^{ip_1x_1}\, e^{-(x_1-a)^2/4w^2}\, e^{ip_2x_2}\, e^{-(x_2-a)^2/4w^2}, \tag{16.110}$$

where $2a$ is the separation and w is the variance in each particle's position. Do the particles bounce off each other when the interaction is repulsive? What happens when the interaction is attractive?



(a) Real and imaginary.



(b) Amplitude and phase.

Figure 16.2: Two representations of complex wave functions. (The actual output is in color.)

Appendix 16A: Visualizing Complex Functions

Complex functions are essential in quantum mechanics and the frames package contains classes for displaying and analyzing these functions. Listing 16.12 uses a ComplexPlotFrame to display a one-dimensional wave function.

Listing 16.12: The ComplexPlotFrameApp class displays a one-dimensional Gaussian wave packet with a momentum boost.

```
package org.opensourcephysics.sip.ch16;
import org.opensourcephysics.frames.ComplexPlotFrame;

public class ComplexPlotFrameApp {
   public static void main(String[] args) {
      ComplexPlotFrame frame =
          new ComplexPlotFrame("x", "Psi(x)", "Complex function");
      int n = 128;
      double
         xmin = -Math.PI, xmax = Math.PI;
      double
         x = xmin, dx = (xmax-xmin)/n;
      double[] xdata = new double[n];
      // real and imaginary values alternate
      double[] zdata = new double[2*n];
      // test function is e^(-x*x/4)e^(i*mode*x) for x=[-pi,pi)
      int mode = 4;
      for(int i = 0;i<n;i++) {
         double a = Math.exp(-x*x/4);
         zdata[2*i] = a*Math.cos(mode*x);
         zdata[2*i+1] = a*Math.sin(mode*x);
         xdata[i] = x;
         x += dx;
      }
      frame.append(xdata, zdata);
      frame.setVisible(true);
      frame.setDefaultCloseOperation(javax.swing.JFrame.EXIT_ON_CLOSE);
   }
}
```

Figure 16.2 shows two representations of a quantum wave function. The real and imaginary representation displays the real and imaginary parts of the wave function $\Psi(x)$ by drawing two curves. In the amplitude and phase representation the vertical height represents the wave function magnitude and the color indicates phase. Note that the complex phase is oscillating, indicating that the wave function has a nonzero momentum expectation value, which is known as a *momentum boost*.

Wave function visualizations can be selected at runtime using the `Tools` menu, or they can be selected programmatically using convert methods such as `convertToPostView` and `convertToReImView`. The `Tools` menu also allows the user to select a table view to examine the data being used to draw the wave function and to display a phase legend that shows the color to phase relation.

A `Complex2DFrame` displays a two-dimensional complex scalar field such as a two-dimensional wave function. We instantiate a `Complex2DFrame` and then pass to it a multidimensional array containing the field's real and imaginary components. Listing 16.13 shows how this class is used to show a two-dimensional Gaussian wave packet with a momentum boost.

Listing 16.13: The `Complex2DFrameApp` program displays a two-dimensional Gaussian wave packet with a momentum boost.

```
package org.opensourcephysics.sip.ch16;
import org.opensourcephysics.frames.Complex2DFrame;

public class Complex2DFrameApp {
   public static void main(String[] args) {
      Complex2DFrame frame =
         new Complex2DFrame("x", "y", "Complex field");
      frame.setPreferredMinMax(-1.5, 1.5, -1.5, 1.5);
      double[][][] field = new double[2][32][32];
      frame.setAll(field);
      for(int i = 0, nx = field[0].length;i<nx;i++) {
         double x = frame.indexToX(i);
         for(int j = 0, ny = field[0][0].length;j<ny;j++) {
            double y = frame.indexToY(j);
            double a = Math.exp(-4*(x*x+y*y));
            field[0][i][j] = a*Math.cos(5*x); // real component
            field[1][i][j] = a*Math.sin(5*x); // complex component
         }
      }
      frame.setAll(field);
      frame.setVisible(true);
      frame.setDefaultCloseOperation(javax.swing.JFrame.EXIT_ON_CLOSE);
   }
}
```

The complex field is computed on a n-row by m-column grid and stored in an array with dimensions $2 \times m \times n$. The default visualization uses a grid in which every cell is colored using brightness to show the complex number's magnitude and color to show phase. Other visualizations can be programmed or selected at runtime using the menu.

References and Suggestions for Further Reading

The ALPS project, `<http://alps.comp-phys.org/>`, has open source simulation programs for strongly correlated quantum mechanical systems and C++ libraries for simplifying the development of such code. Although most of the code is beyond the level of this text, this open source project is another example of software for use in both research and education.

J. B. Anderson, "A random walk simulation of the Schrödinger equation: H_3^+," J. Chem. Phys. **63**, 1499–1503 (1975); "Quantum chemistry by random walk. H ^2P, H_3^+ $D_{3h}\,^1A'_1$, H_2 $^3\Sigma_u^+$, H_4 $^1\Sigma_g^+$, Be ^1S," J. Chem. Phys. **65**, 4121–4127 (1976); "Quantum chemistry by random walk: Higher accuracy," J. Chem. Phys. **73**, 3897–3899 (1980). These papers describe the random walk method, extensions for improved accuracy, and applications to simple molecules.

G. Baym, *Lectures on Quantum Mechanics*, Westview Press (1990). A discussion of the Schrödinger equation in imaginary time is given in Chapter 3.

M. A. Belloni, W. Christian, and A. Cox, *Physlet Quantum Physics*, Prentice Hall (2006) This book contains interactive exercises using Java applets to visualize quantum phenomena.

H. A. Bethe, *Intermediate Quantum Mechanics*, Westview Press (1997). Applications of quantum mechanics to atomic systems are discussed.

Jay S. Bolemon, "Computer solutions to a realistic 'one-dimensional' Schrödinger equation," Am. J. Phys. **40**, 1511 (1972).

Siegmund Brandt and Hans Dieter Dahmen, *The Picture Book of Quantum Mechanics*, third edition, Springer-Verlag (2001); Siegmund Brandt, Hans Dieter Dahmen, and Tilo Stroh, *Interactive Quantum Mechanics*, Springer-Verlag (2003).These books show computer generated pictures of quantum wave functions in different contexts.

R. Car and M. Parrinelli, "Unified approach for molecular dynamics and density-functional theory," Phys. Rev. Lett. **55**, 2471 (1985).

David M. Ceperley, "Path integrals in the theory of condensed helium," Rev. Mod. Phys. **67**, 279–355 (1995).

David M. Ceperley and Berni J. Alder, "Quantum Monte Carlo," Science **231**, 555 (1986). A survey of some of the applications of quantum Monte Carlo methods to physics and chemistry.

David Chandler and Peter G. Wolynes, "Exploiting the isomorphism between quantum theory and classical statistical mechanics of polyatomic fluids," J. Chem. Phys. **74**, 4078–4095 (1981). The authors use path integral techniques to look at multiparticle quantum systems.

D. F. Coker and R. O. Watts, "Quantum simulation of systems with nodal surfaces," Mol. Phys. **58**, 1113–1123 (1986).

Jim Doll and David L. Freeman, "Monte Carlo methods in chemistry," Computing in Science and Engineering **1** (1), 22–32 (1994).

Robert M. Eisberg and Robert Resnick, *Quantum Physics*, second edition, John Wiley & Sons (1985). See Appendix G for a discussion of the numerical solution of Schrödinger's equation.

R. P. Feynman, "Simulating physics with computers," Int. J. Theor. Phys. **21**, 467–488 (1982). A provocative discussion of the intrinsic difficulties of simulating quantum systems. See also R. P. Feynman, *Feynman Lectures on Computation*, Westview Press (1996).

Richard P. Feynman and A. R. Hibbs, *Quantum Mechanics and Path Integrals*, McGraw–Hill (1965).

David J. Griffiths, *Introduction to Quantum Mechanics*, second edition, Prentice Hall (2005). An excellent undergraduate text that discusses the virial theorem in several problems.

B. L. Hammond, W. A. Lester Jr., and P. J. Reynolds, *Monte Carlo Methods in Ab Initio Quantum Chemistry*, World Scientific (1994). An excellent book on quantum Monte Carlo methods.

Steven E. Koonin and Dawn C. Meredith, *Computational Physics*, Addison–Wesley (1990). Solutions of the time-dependent Schrödinger equation are discussed in the context of parabolic partial differential equations in Chapter 7. Chapter 8 discusses Green's function Monte Carlo methods.

Rubin Landau, "Two-particle Schrödinger equation animations of wavepacket-wavepacket scattering," Am. J. Phys. **68** (12), 1113–1119 (2000).

Michel Le Bellac, Fabrice Mortessagne, and G. George Batrouni, *Equilibrium and Non-Equilibrium Statistical Thermodynamics*, Cambridge University Presss (2004). Chapter 7 of this graduate level text discusses the world line algorithm for bosons and fermions on a lattice.

M. A. Lee and K. E. Schmidt, "Green's function Monte Carlo," Computers in Physics **6** (2), 192 (1992). A short and clear explanation of Green's function Monte Carlo.

P. K. MacKeown, "Evaluation of Feynman path integrals by Monte Carlo methods," Am. J. Phys. **53**, 880—885 (1985). The author discusses projects suitable for an advanced undergraduate course. Also see P. K. MacKeown and D. J. Newman, *Computational Techniques in Physics*, Adam Hilger (1987).

Jean Potvin, "Computational quantum field theory. Part II: Lattice gauge theory," Computers in Physics **8**, 170 (1994); and "Computational quantum field theory," Computers in Physics **7**, 149 (1993).

William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery, *Numerical Recipes*, second edition, Cambridge University Press (1992). The numerical solution of the time-dependent Schrödinger equation is discussed in Chapter 19.

Peter J. Reynolds, David M. Ceperley, Berni J. Alder, and William A. Lester Jr., "Fixed-node quantum Monte Carlo for molecules," J. Chem. Phys. **77**, 5593–5603 (1982). This paper describes a random walk algorithm for use in molecular applications including importance sampling and the treatment of Fermi statistics.

P. J. Reynolds, J. Tobochnik, and H. Gould, "Diffusion quantum Monte Carlo," Computers in Physics **4** (6), 882 (1990).

U. Rothlisberger, "15 Years of Car–Parrinello simulations in physics, chemistry and biology," in *Computational Chemistry: Reviews of Current Trends*, edited by Jerzy Leszczynski, World Scientific (2001), Vol. 6.

Amlan K. Roy, Neetu Gupta, and B. M. Deb, "Time-dependent quantum mechanical calculation of ground and excited states of anharmonic and double-well oscillators," Phys. Rev A **65**, 012109-1–7 (2001).

Amlan K. Roy, Ajit J. Thakkar, and B. M. Deb, "Low-lying states of two-dimensional double-well potentials," J. Phys. A **38**, 2189–2199 (2005).

K. E. Schmidt, Parhat Niyaz, A. Vaught, and Michael A. Lee, "Green's function Monte Carlo method with exact imaginary-time propagation," Phys. Rev. E **71**, 016707-1–17 (2005).

Bernd Thaller, *Visual Quantum Mechanics: Selected Topics with Computer-Generated Animations of Quantum-Mechanical Phenomena*, Telos (2000); Bernd Thaller, *Advanced Visual Quantum Mechanics*, Springer (2005).

J. Tobochnik, H. Gould, and K. Mulder, "An introduction to quantum Monte Carlo," Computers in Physics **4** (4), 431 (1990). An explanation of the path integral method applied to one particle.

P. B. Visscher, "A fast explicit algorithm for the time-dependent Schrödinger equation," Computers in Physics **5** (6), 596 (1991).

Chapter 17

Visualization and Rigid Body Dynamics

We study affine transformations in order to visualize objects in three dimensions. We then solve Euler's equation of motion for rigid body dynamics using the quaternion representation of rotations.

17.1 Two-Dimensional Transformations

Physicists frequently use transformations to convert from one system of coordinates to another. A very common transformation is an *affine transformation*, which has the ability to rotate, scale, stretch, skew, and translate an object. Such a transformation maps straight lines to straight lines. They are often represented using matrices and are manipulated using the tools of linear algebra such as matrix multiplication and matrix inversion. The Java 2D API defines a set of classes designed to create high quality graphics using image composition, image processing, antialiasing, and text layout. Because linear algebra and affine transformations are used extensively in imaging and drawing APIs, we begin our study of two- and three-dimensional visualization techniques by studying the properties of transformations.

It is straightforward to rotate a point (x,y) about the origin by an angle θ (see Figure 17.2) or scale the distance from the origin by (s_x, s_y) using matrices:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \tag{17.1}$$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}. \tag{17.2}$$

Performing several transformations corresponds to multiplying matrices. However, the translation of the point (x,y) by (d_x, d_y) is treated as an addition and not as a multiplication and must be written differently:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} d_x \\ d_y \end{bmatrix} + \begin{bmatrix} x \\ y \end{bmatrix}. \tag{17.3}$$

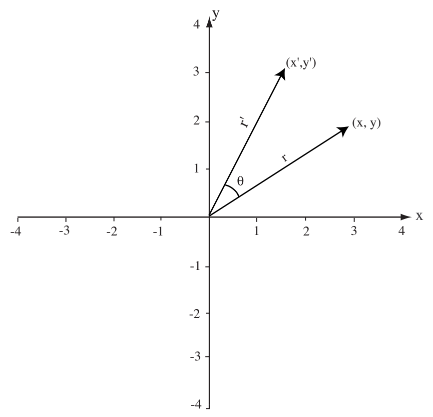


Figure 17.1: A two-dimensional rotation of a point (x,y) produces a point with new coordinates (x',y') as computed according to (17.1).

This inconsistency in the type of mathematical operation is easily overcome if points are expressed in terms of *homogeneous coordinates* by adding a third coordinate w. Homogeneous coordinates are used extensively in computer graphics to treat all transformations consistently. Instead of representing a point in two dimensions by a pair of numbers (x,y), each point is represented by a triple (x,y,w). Because two homogeneous coordinates represent the same point if one is a multiple of the other, we usually *homogenize* the point by dividing the x-y coordinates by w and write the coordinates in the form $(x,y,1)$. (The w coordinate can be used to add perspective (see Foley et al.).) By using homogeneous coordinates, an arbitrary affine transformation can be written as

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}. \tag{17.4}$$

A translation, for example, can be expressed as:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & d_x \\ 0 & 1 & d_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}. \tag{17.5}$$

Exercise 17.1. Homogeneous coordinates

(a) How are the rotation and scaling transformations expressed in matrix notation using homogeneous coordinates? Sketch the transformation matrices for a 30° clockwise rotation and for a scaling along the x-axis by a factor of two and then write the transformation matrices. Do these matrices commute?

(b) Describe the effect of the affine transformation

$$\begin{bmatrix} 1 & 0.2 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{17.6}$$

□

Exercise 17.1 shows that a coordinate transformation can be broken into parts using a *block matrix* format:

$$\begin{bmatrix} \mathcal{A} & \mathbf{d}^T \\ \mathbf{0} & 1 \end{bmatrix}. \tag{17.7}$$

We will use boldface for row vectors such as **0** and **d** and calligraphic symbols to represent matrices. The translation vector **d** is transposed to convert it to a column vector. The upper left-hand submatrix $\mathcal{A}$ produces rotation and scaling while the vector $\mathbf{d} = [d_x, d_y]$ produces translation.

Homogeneous coordinates have another advantage because they can be used to distinguish between points and vectors. Unlike points, vectors should remain invariant under translation. To transform vectors using the same transformation matrices that we use to transform points, we set w to zero, thereby removing the effect of the last column. Note that the difference between two homogeneous points produces a w equal to zero. The elimination of the effect of translation makes sense because the difference between two points is a displacement vector, and vectors are defined in terms of their components, not their location.

The `AffineTransform` class in the `java.awt.geom` package defines two-dimensional affine transformations. Instances of this class are constructed as

```java
AffineTransform at = new AffineTransform(double m00, double m10,
      double m01,double m11,double m02, double m12);
```

The methods in this class encapsulate most of the matrix arithmetic that is required for two-dimensional visualization. For example, there are methods to calculate a transformation's inverse and to combine transformations using the rules of matrix multiplication. There are also static methods for constructing pure rotations, scalings, and translations that require only one or two parameters.

```java
double theta = Math.PI/6;
AffineTransform at = AffineTransform.getRotateInstance(theta);
```

A method such as `getRotateInstance` is known as a *convenience method* because it simplifies a complicated API.

The `AffineTransform` class can transform geometric objects, images, and even text. The following code fragment shows how this class is used to rotate a point and a rectangle.

```java
Point2D.Double pt = Point2D.Double(2.0,3.0);
pt = AffineTransform.getRotateInstance(Math.PI/3).transform(pt,null);
Shape shape = new Rectangle2D.Double(50,50,100,150);
shape = AffineTransform.getRotateInstance(Math.PI/3).createTransformedShape(shape);
```

The `Affine2DApp` class in Listing 17.1 demonstrates affine transformations by applying them to a rectangle.

Listing 17.1: The `Affine2DApp` class.

```java
package org.opensourcephysics.sip.ch17;
import java.awt.*;
import java.awt.geom.*;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.display.*;
import org.opensourcephysics.frames.*;

public class Affine2DApp extends AbstractCalculation {
    DisplayFrame frame = new DisplayFrame("2D Affine transformation");
    RectShape rect = new RectShape();

    public void calculate() {
       // allocate 3 rows but not the row elements
       double[][] matrix = new double[3][];
       // set the first row of the matrix
       matrix[0] = (double[]) control.getObject("row 0");
       // set the second row
       matrix[1] = (double[]) control.getObject("row 1");
       // set the third row
       matrix[2] = (double[]) control.getObject("row 2");
       rect.transform(matrix);
    }

    public void reset() {
       control.clearMessages();
       control.setValue("row 0", new double[] {1, 0, 0});
       control.setValue("row 1", new double[] {0, 1, 0});
       control.setValue("row 2", new double[] {0, 0, 1});
       rect = new RectShape();
       frame.clearDrawables();
       frame.addDrawable(rect);
       calculate();
    }

    public static void main(String[] args) {
       CalculationControl.createApp(new Affine2DApp());
    }

    class RectShape implements Drawable { // inner class
       Shape shape = new Rectangle2D.Double(50, 50, 100, 100);

       public void draw(DrawingPanel panel, Graphics g) {
          Graphics2D g2 = ((Graphics2D) g);
          g2.setPaint(Color.BLUE);
          g2.fill(shape);
          g2.setPaint(Color.RED);
          g2.draw(shape);
       }

       public void transform(double[][] mat) {
          shape = (new AffineTransform(mat[0][0], mat[1][0], mat[0][1],
             mat[1][1], mat[0][2],mat[1][2])).createTransformedShape(shape);
       }
    }
 }
```

Exercise 17.2. Two-dimensional affine transformations

(a) Enter an affine transformation for a 30° clockwise rotation into `Affine2DApp`. About what point does the rectangle rotate? Why?

(b) Add and test a convenience method named `translate` to the `RectShape` class that takes two parameters (d_x, d_y). Add a custom button to invoke this method.

(c) Add and test a convenience method named `rotate` to the `RectShape` class that takes a θ parameter.

(d) An object can be rotated about its center by first translating the object to the center of rotation, performing the rotation, and then translating the object back to its original position. Implement a method that performs a rotation about the center of the rectangle by invoking a sequence of `translate` and `rotate` methods.

(e) Affine transformations have the property that transformed parallel lines remain parallel. Demonstrate that this property is plausible by transforming a rectangle using arbitrary values for the transformation matrix. □

To facilitate the creation of simple geometric shapes using world coordinates, the Open Source Physics library defines the `DrawableShape` and `InteractiveShape` classes in the display package. These classes define convenience methods to create common drawable shapes whose (x, y) location is their geometric center. These shapes can later be transformed without having to instantiate new objects. The following code fragment shows how these classes are used.

```
// circle of radius 3 in world units located at (-1,2)
DrawableShape circle = InteractiveShape.createCircle(-1,2,3);
circle.transform(AffineTransform.getShearInstance(1,2));
frame.addDrawable(circle);
// rectangle of width 2 and height 1 centered at (3,4)
InteractiveShape rect = InteractiveShape.createRectangle(3,4,2,1);
rect.transform(new AffineTransform(2,1,0,1,0,0));
frame.addDrawable(rect);
```

Because `DrawableShape` and `InteractiveShape` classes are written using the Java 2D API, the objects that they define are fundamentally different from the objects that use the `awt` API introduced in Section 3.3 because Java 2D shapes can be transformed. In addition, the Java 2D API is not restricted to pixel coordinates nor is it restricted to solid single-pixel lines.[1]

Exercise 17.3. Open Source Physics shape classes

The Open Source Physics shape classes can be manipulated using a wide variety of linear algebra-based tools. Modify the `Affine2DApp` program so that it instantiates and transforms a rectangular `DrawableShape` into a trapezoid. Test your program by repeating Exercise 17.2. □

17.2 Three-Dimensional Transformations

There are several available APIs for three-dimensional visualizations using Java. Although Sun has developed the Java 3D package, this package is currently not included in the standard Java runtime environment. The `gl4java` and `jogl` libraries are also popular because they are based on the Open GL language. Because 3D graphics libraries are in a state of active development and because we want a three-dimensional visualization framework designed for physics simulations, we have developed a three-dimensional visualization framework that relies only on

[1]See Chapter 4 in the *Open Source Physics: A Users Guide with Examples* for a more complete discussion of the `DrawableShape` and `InteractiveShape` classes.

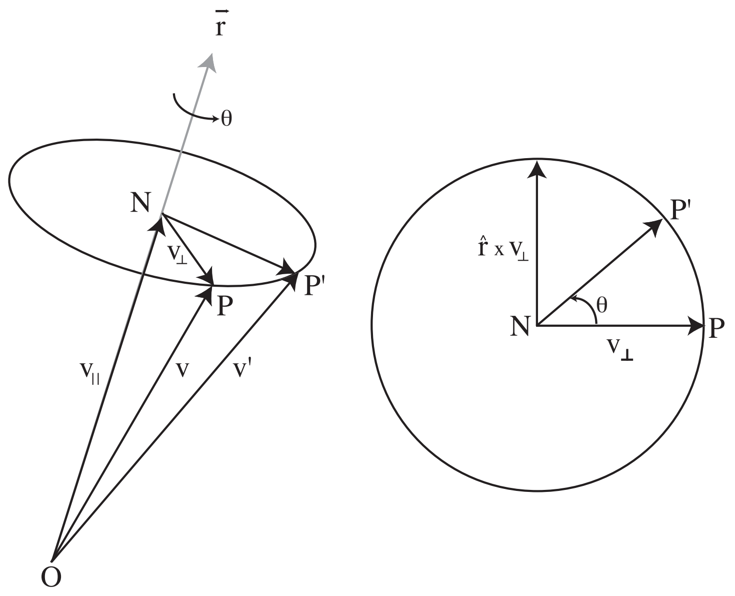


Figure 17.2: A rotation of the vector **v** about an axis $\hat{\mathbf{r}}$ to produce the vector $\mathbf{v}'$ can be decomposed into parallel $\mathbf{v}_{\|}$ and perpendicular $\mathbf{v}_{\perp}$ components.

the standard Java API. This section describes the mathematics that forms the basis of all three-dimensional libraries.

The simplest rotation is a rotation about one of the coordinate axes with the center of rotation at the origin. This transformation can be written using a 3×3 matrix acting on the coordinates (x, y, z). For example, a rotation about the z-axis can be written as

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \mathcal{R}_z \begin{bmatrix} x \\ y \\ z \end{bmatrix}. \tag{17.8}$$

The extension of homogeneous coordinates to three dimensions is straightforward. We add a w-coordinate to the spatial coordinates to create a homogenous point $(x, y, z, 1)$. This point is transformed as

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} m_{00} & m_{01} & m_{02} & m_{03} \\ m_{10} & m_{11} & m_{12} & m_{13} \\ m_{20} & m_{21} & m_{22} & m_{23} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} \mathcal{R}_z & \mathbf{d}^T \\ \mathbf{0} & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}. \tag{17.9}$$

Although rotations about one of the coordinate axes are easy to derive and can be combined using the rules of linear algebra to produce an arbitrary orientation, the general case of rotation about the origin by an angle θ around an arbitrary axis $\hat{\mathbf{r}}$ can be constructed directly. The strategy is to decompose the vector **v** into components that are parallel and perpendicular to the direction $\hat{\mathbf{r}}$ as shown in Figure 17.2. The parallel part $\mathbf{v}_{\|}$ does not change, while the perpendicular part $\mathbf{v}_{\perp}$ is a two-dimensional rotation in a plane perpendicular to $\hat{\mathbf{r}}$. The parallel part is the projection of $\hat{\mathbf{r}}$ onto **v**,

$$\mathbf{v}_{\|} = (\mathbf{v} \cdot \hat{\mathbf{r}})\hat{\mathbf{r}}, \tag{17.10}$$

and the perpendicular part is what remains of $\hat{\mathbf{v}}$ after we subtract the parallel part:

$$\mathbf{v}_\perp = \mathbf{v} - (\mathbf{v}\cdot\hat{\mathbf{r}})\hat{\mathbf{r}}. \tag{17.11}$$

To calculate the rotation of $\mathbf{v}_\perp$, we need two perpendicular basis vectors in the plane of rotation. If we use $\mathbf{v}_\perp$ as the first basis vector, then we can take the cross product with $\hat{\mathbf{r}}$ to produce a vector $\mathbf{w}$ that is guaranteed to be perpendicular to $\mathbf{v}_\perp$ and $\hat{\mathbf{r}}$:

$$\mathbf{w} = \hat{\mathbf{r}}\times\mathbf{v}_\perp = \hat{\mathbf{r}}\times\mathbf{v}. \tag{17.12}$$

The rotation of $\mathbf{v}_\perp$ is now calculated in terms of this new basis:

$$\mathbf{v}' = \mathcal{R}(\mathbf{v}_\perp) = \cos\theta\,\mathbf{v}_\perp + \sin\theta\,\mathbf{w}. \tag{17.13}$$

The final result is the sum of this rotated vector and the parallel part that does not change:

$$\begin{aligned}\mathcal{R}(\mathbf{v}) &= \mathcal{R}(\mathbf{v}_\perp) + \mathbf{v}_\| && (17.14a)\\ &= \cos\theta\,\mathbf{v}_\perp + \sin\theta\,\mathbf{w} + \mathbf{v}_\| && (17.14b)\\ &= \cos\theta\,[\mathbf{v} - (\mathbf{v}\cdot\hat{\mathbf{r}})\hat{\mathbf{r}}] + \sin\theta\,(\hat{\mathbf{r}}\times\mathbf{v}) + (\mathbf{v}\cdot\hat{\mathbf{r}})\hat{\mathbf{r}} && (17.14c)\end{aligned}$$

or

$$\mathcal{R}(\mathbf{v}) = [1-\cos\theta](\mathbf{v}\cdot\hat{\mathbf{r}})\hat{\mathbf{r}} + \sin\theta\,(\hat{\mathbf{r}}\times\mathbf{v}) + \cos\theta\,\mathbf{v}. \tag{17.15}$$

Equation (17.15) is known as the *Rodrigues formula* and provides a way of constructing rotation matrices in terms of the direction of the axis of rotation $\hat{\mathbf{r}} = (r_x, r_y, r_z)$, the cosine of the rotation angle $c = \cos\theta$, and the sine of the rotation angle $s = \sin\theta$. If we expand the vector products in (17.15), we obtain the matrix:

$$\mathcal{R} = \begin{bmatrix} tr_xr_x + c & tr_xr_y - sr_z & tr_xr_z + sr_y \\ tr_xr_y + sr_z & tr_yr_y + c & tr_yr_z - sr_x \\ tr_xr_z - sr_y & tr_yr_z + sr_x & tr_zr_z + c \end{bmatrix}, \tag{17.16}$$

where $t = 1 - \cos\theta$. Homogeneous coordinates are transformed using

$$\begin{bmatrix} \mathcal{R} & \mathbf{0}^T \\ \mathbf{0} & 1 \end{bmatrix}, \tag{17.17}$$

where the $\mathcal{R}$ submatrix is given in (17.16).

The `Rotation3D` class constructor (see Listing 17.2) computes the rotation matrix. The `direct` method uses this matrix to transform a point. Note that the point passed to this method as an argument is copied into a temporary vector and that the point's coordinates are then changed. You will define an `inverse` method that reverses this operation in Exercise 17.5.

Listing 17.2: The `Rotation3D` class implements three-dimensional rotations using a matrix representation.

```
package org.opensourcephysics.sip.ch17;
public class Rotation3D {
   // transformation matrix
   private double[][] mat = new double[4][4];

   public Rotation3D(double theta, double[] axis) {
      double norm = Math.sqrt(axis[0]*axis[0]+axis[1]*axis[1]
            +axis[2]*axis[2]);
      double
         x = axis[0]/norm, y = axis[1]/norm, z = axis[2]/norm;
      double
         c = Math.cos(theta), s = Math.sin(theta);
      double t = 1-c;
      // matrix elements not listed are zero
      mat[0][0] = t*x*x+c;
      mat[0][1] = t*x*y-s*z;
      mat[0][2] = t*x*y+s*y;
      mat[1][0] = t*x*y+s*z;
      mat[1][1] = t*y*y+c;
      mat[1][2] = t*y*z-s*x;
      mat[2][0] = t*x*z-s*y;
      mat[2][1] = t*y*z+s*x;
      mat[2][2] = t*z*z+c;
      mat[3][3] = 1;
   }

   public void direct(double[] point) {
      int n = point.length;
      double[] pt = new double[n];
      System.arraycopy(point, 0, pt, 0, n);
      for(int i = 0;i<n;i++) {
         point[i] = 0;
         for(int j = 0;j<n;j++) {
            point[i] += mat[i][j]*pt[j];
         }
      }
   }
}
```

Exercise 17.4. Rodrigues formula

Show that a rotation about the z-axis is consistent with (17.15) and (17.16). That is, define the direction of rotation to be $\hat{\mathbf{r}} = (0,0,1)$ and show that both formulas give the same result and that this result is consistent with a two-dimensional rotation in the x-y plane. Write a test program to test the `Rotation3D` class. □

Exercise 17.5. Inverse matrix

What is the inverse matrix of (17.16)? Hint: What happens physically if you change the sign of the rotation angle. Is the matrix orthogonal? Add an inverse method to `Rotation3D` and write a test program to test the `inverse` method. Show that the original vector is recovered if the `inverse` and `direct` methods are applied in succession. □

A projection transforms an object in a coordinate system of dimension d into another object in a coordinate system less than d. The simplest projection is an *orthographic parallel projection*, which maps an object onto a plane perpendicular to a coordinate axis. For example, if we choose to project along the z-axis, the point (x,y,z) is mapped to the point (x,y) by dropping the third coordinate. A line is projected by projecting the endpoints and then connecting the projected values. A sphere with radius R is displayed by projecting the center and drawing a circle with the sphere's radius. Listing 17.3 uses these simple projections to visualize the structure of the methane CH_4 molecule.

Listing 17.3: The Methane class implements a visualization of the methane molecule CH_4.

```
package org.opensourcephysics.sip.ch17;
import java.awt.*;
import org.opensourcephysics.display.*;

public class Methane implements Drawable {
   static final double cos30 = Math.cos(Math.PI/6);
   static final double sin30 = Math.sin(Math.PI/6);
   static final double h = Math.sqrt(1.0-4.0*cos30*cos30/9.0);
   double[][] atoms = new double[5][];
   Circle circle = new Circle();

   public Methane() {
      // atom locations in 3D homogeneous coordinates
      // C atom at origin
      atoms[0] = new double[] {0, 0, 0, 1};
      // H atom on z axis
      atoms[1] = new double[] {0, 0, 0.75*h, 1};
       // H atom
      atoms[2] = new double[] {2.0*cos30/3.0, 0, -0.25*h, 1};
      // H atom
      atoms[3] = new double[] {-cos30/3.0, sin30, -0.25*h, 1};
      // H atom
      atoms[4] = new double[] {-cos30/3.0, -sin30, -0.25*h, 1};
   }

   void transform(Rotation3D t) {
      for(int i = 0, n = atoms.length;i<n;i++) {
         t.direct(atoms[i]);
      }
   }

   public void draw(DrawingPanel panel, Graphics g) {
      g.setColor(Color.black);
      int x0 = panel.xToPix(0);
      int y0 = panel.yToPix(0);
      for(int i = 0, n = atoms.length;i<n;i++) {
         int xpix = panel.xToPix(atoms[i][0]);
         int ypix = panel.yToPix(atoms[i][1]);
         g.drawLine(x0, y0, xpix, ypix);
      }
      for(int i = 0, n = atoms.length;i<n;i++) {
         circle.setXY(atoms[i][0], atoms[i][1]);
         circle.draw(panel, g);
      }
   }
}
```

The carbon and hydrogen atom coordinates are given in the Methane constructor. These coordinates are stored in a multidimensional array so that we can loop over the atoms in the draw method. The carbon atom (the center of symmetry) is placed at the origin and a hydrogen atom is placed above it along the z-axis. The three remaining hydrogen atoms are placed so as to form a tetrahedron with an H-H separation equal to unity. This orientation can be changed using the `transform` method to rotate the coordinates. Note that the `draw` method draws lines from the origin to each hydrogen atom's coordinates and then draws circles at these coordinates.

Exercise 17.6. Methane

Determine the angle between two hydrogen bonds using the data in the `Methane` class. □

The `MethaneApp` program uses the `Rotation3D` and `Methane` classes by rotating the methane molecule about the origin in response to mouse actions. The `handleMouseAction` method stores the current mouse position when the mouse is clicked. A `Rotation3D` object is created when the mouse is dragged; the drag distance determines the angle of rotation. If the mouse is dragged vertically, the molecule is rotated about the y-axis, and if the mouse is dragged horizontally, the molecule is rotated about the z-axis. The `direct` transform is then applied to every atom in the methane molecule.

Listing 17.4: The `MethaneApp` class instantiates a `Methane` object and rotates this object using mouse actions.

```
package org.opensourcephysics.sip.ch17;
import java.awt.event.MouseEvent;
import javax.swing.JFrame;
import org.opensourcephysics.frames.*;
import org.opensourcephysics.display.InteractiveMouseHandler;
import org.opensourcephysics.display.InteractivePanel;

public class MethaneApp implements InteractiveMouseHandler {
   DisplayFrame frame = new DisplayFrame("Methane");
   Methane methane = new Methane();
   double mouseX = 0, mouseY = 0;

   public MethaneApp() {
      frame.addDrawable(methane);
      frame.setPreferredMinMax(-1, 1, -1, 1);
      frame.setInteractiveMouseHandler(this);
      frame.setVisible(true);
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
   }

   public void handleMouseAction(InteractivePanel panel, MouseEvent evt) {
      switch(panel.getMouseAction()) {
      case InteractivePanel.MOUSE_DRAGGED :
         double dx = panel.getMouseX()-mouseX;
         double dy = panel.getMouseY()-mouseY;
         Rotation3D rotation =
           new Rotation3D(Math.sqrt(dx*dx+dy*dy), new double[] {dy, 0, dx});
         methane.transform(rotation);
         mouseX += dx;
         mouseY += dy;
         panel.repaint();
         break;
      case InteractivePanel.MOUSE_PRESSED :
         mouseX = panel.getMouseX();
         mouseY = panel.getMouseY();
         break;
      }
```

```
    }

    public static void main(String[] args) {
      new MethaneApp();
    }
}
```

Exercise 17.7. Methane rotation

Modify the methane program by replacing the `direct` transformation with the `inverse` transformation. Run and compare this program to the original program. How did this change affect the mouse actions? Why? □

Exercise 17.8. Hidden surface removal

`MethaneApp` draws atoms using the same color which hides the fact that the drawing is not correct. Modify the `Methane` class so that the carbon atom is drawn as a green circle and run the `MethaneApp` program again. Notice that the carbon atom is hidden by the hydrogen atoms that are behind the carbon atom. Rewrite the `Methane` class so that the drawing order is determined by the atom's z coordinate. Ordering along the line of sight is often used in graphics programs for hidden line and hidden surface removal. □

17.3 The Three-Dimensional Open Source Physics Library

The `display3d` package contains a three-dimensional drawing framework that makes it easy to create simple visualizations. In the spirit of object oriented programming, we will not study the Open Source Physics 3D implementation in detail but will describe only its API. The `Box3DApp` class creates a simple three-dimensional visualization. Run the program and drag the mouse within the panel. We need not concern ourselves with details such as hidden line removal, perspective, or rotation.

The Open Source Physics 3D API is defined in the `org.opensourcephysics.display3d` package. This API can be be implemented using any 3D library, and we have done so in the `simple3d` package using only standard Java. We use the `simple3d` package in this book because programs written using this package will run on any Java enabled computer. If truly high resolution rendering is required, it is best to import an OSP 3D implementation that uses an add-on library such as Java for Open GL (JOGL) or Java 3D. These libraries must be installed, but they support hardware accelerated rendering using Open GL graphics language drivers. A JOGL implementation of Open Source Physics 3D is being developed. All that is required to use this implementation is to change the import statement to use the `jogl` package.

Listing 17.5 demonstrates that it is just as easy to create a 3D visualization as it is to create a 2D visualization.

Listing 17.5: The `Box3DApp` class creates a box within a `Display3DFrame`

```
package org.opensourcephysics.sip.ch17;
import java.awt.*;
import org.opensourcephysics.frames.*;
import org.opensourcephysics.display3d.simple3d.*;

public class Box3DApp {
  public static void main(String[] args) {
      // creates a drawing frame and a drawing panel
      Display3DFrame frame = new Display3DFrame("3D demo");
      frame.setPreferredMinMax(-10, 10, -10, 10, -10, 10);
      frame.setDecorationType(VisualizationHints.DECORATION_AXES);
      // use shading when rotating
      frame.setAllowQuickRedraw(false);
      Element block = new ElementBox();
      block.setXYZ(0, 0, 0);
      block.setSizeXYZ(6, 6, 3);
      block.getStyle().setFillColor(Color.RED);
      // divides block into subblocks
      block.getStyle().setResolution(new Resolution(6, 6, 3));
      frame.addElement(block);
      frame.setVisible(true);
      frame.setDefaultCloseOperation(javax.swing.JFrame.EXIT_ON_CLOSE);
   }
}
```

Drawable objects in the `simple3d` package implement the `Element` interface. The `Box3DApp` program demonstrates that this interface mimics simple geometric objects in the physical world. Elements have position and size properties and contain `Style` and `Resolution` objects to control their visual representation.

The `Display3DFrame` class behaves very much like its two-dimensional `DisplayFrame` counterpart. Some methods such as `setPreferredMinMax` have been extended by adding parameters for the third dimension. New methods such as `getStyle` and `setDecorationType` enable the programmer to control the three-dimensional viewing perspective and axis types, respectively. Three-dimensional elements, such as `ElementBox`, `ElementCylinder`, and `ElementCircle`, are added to a container using the `addElement` method. As in the two-dimensional case, the high level `Display3DFrame` is composed of lower level objects such as `DrawingPanel3D` which fill the frame's viewing area. The program must repaint the frame for property changes to take effect.

Elements generate interaction events and these events can have one or more interaction targets. Interaction events are action events that are generated when an object is accessed using the mouse. Interaction targets receive and process these events. Listing 17.6 creates a particle and an arrow and responds to interaction events when these objects are moved. You can move an interactive element by dragging it. You can also generate and process interaction events at an arbitrary location within the viewing space, but you might wish to first click the ⟨alt⟩ (⟨option⟩ on Mac OS X) key to disable the viewing space rotation. Clicking the ⟨shift⟩ key while dragging enables you to zoom in and out. Clicking the ⟨control⟩ key while dragging enables you to translate.

Listing 17.6: The `Interaction3DApp` class demonstrates how to respond to interaction events.

```
package org.opensourcephysics.sip.ch17;
import org.opensourcephysics.display3d.simple3d.*;
import org.opensourcephysics.frames.*;
import org.opensourcephysics.display3d.core.interaction.*;

public class Interaction3DApp implements InteractionListener {
   Interaction3DApp() {
      Display3DFrame frame = new Display3DFrame("3D interactions");
      frame.setPreferredMinMax(-2.5, 2.5, -2.5, 2.5, -2.5, 2.5);
      // accepts interactions from the frame's 3D drawing panel
      frame.addInteractionListener(this);
      Element particle = new ElementCircle();
      particle.setSizeXYZ(1, 1, 1);
      // enables interactions that change positions
      particle.getInteractionTarget(Element.TARGET_POSITION).
         setEnabled(true);
       // accepts interactions from the particle
      particle.addInteractionListener(this);
      frame.addElement(particle);// adds particle to panel
      ElementArrow arrow = new ElementArrow();
      // enables interactions that change the size
      arrow.getInteractionTarget(Element.TARGET_SIZE)
            .setEnabled(true);
      // accepts interactions from the arrow
      arrow.addInteractionListener(this);
      frame.addElement(arrow); // adds the arrow to the panel
      // enables interactions with the 3D Frame
      frame.enableInteraction(true);
      // accepts interactions from the frame
      frame.addInteractionListener(this);
      frame.setDefaultCloseOperation(javax.swing.JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }

   public void interactionPerformed(InteractionEvent _evt) {
      Object source = _evt.getSource();
      if(_evt.getID()==InteractionEvent.MOUSE_PRESSED) {
         System.out.println("Mouse clicked");
      }
      if(source instanceof ElementCircle) {
         System.out.println("A particle has been hit");
      }
   }

   static public void main(String args[]) {
      new Interaction3DApp();
   }
}
```

Elements can be grouped together and manipulated as a single object by creating geometric shapes such as spheres, boxes, and arrows and adding them to a Group. An easy way to create an Element consisting of many geometric shapes is to subclass Group and instantiate the other Element objects in the constructor and add them to the Group. Listing 17.7 shows an example of such a composite. The entire group acts like a single Element that can be translated and rotated. The (x,y,z) parameters passed to the setXYZ method of an object placed within a group are relative to the group's position. The (x,y,z) parameters passed to an element's setSizeXYZ method are along the group's axes even if the group has been rotated.

Listing 17.7: The Barbell3D class creates a compound object by instantiating simpler shapes and adding them to a Group.

```
package org.opensourcephysics.sip.ch17;
import org.opensourcephysics.display3d.simple3d.*;
```

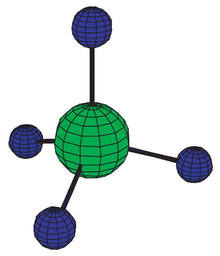


Figure 17.3: A visualization of the methane molecule using the Open Source Physics `simple3d` package.

```
public class Barbell3D extends Group {
   public Barbell3D() {
      ElementCylinder bar = new ElementCylinder();
      bar.setXYZ(0, 0, 5);
      bar.setSizeXYZ(0.2, 0.2, 10);
      addElement(bar);
      Element sphere = new ElementSphere();
      sphere.setXYZ(0, 0, -5);
      sphere.setSizeXYZ(4, 4, 4);
      addElement(sphere);
      sphere = new ElementSphere();
      sphere.setXYZ(0, 0, 5);
      sphere.setSizeXYZ(4, 4, 4);
      addElement(sphere);
   }
}
```

Exercise 17.9. Group test

Write a test program that instantiates and displays a `Barbell`. Describe the change in rendering while dragging within the view. □

The code package for this chapter includes an Open Source Physics 3D version of the methane molecule (see Figure 17.3) but is not listed here because of its length. As in the previous example, a `Group` is used to define a `Methane` class. This model is instantiated and added to a `Display3DFrame` in the `Methane3DApp` class.

17.4 Dynamics of a Rigid Body

The dynamical behavior of a rigid body is determined by

$$\frac{d\mathbf{P}}{dt} = \mathbf{F} \tag{17.18a}$$

$$\frac{d\mathbf{L}}{dt} = \mathbf{N}, \tag{17.18b}$$

where the rate of change of the total linear momentum **P** and total angular momentum **L** about a point O is determined by the total force **F** on the body and the total torque **N** about O. These

Body	**Axis**	**Moment of Inertia**
ellipsoid, axes $(2a, 2b, 2c)$	axes a	$I = m(b^2 + c^2)/5$
	axes b	$I = m(a^2 + c^2)/5$
	axes c	$I = m(a^2 + b^2)/5$
parallelopiped, sides (a, b, c)	perpendicular to (a, b)	$I = m(a^2 + b^2)/12$
	perpendicular to (b, c)	$I = m(b^2 + c^2)/12$
	perpendicular to (c, a)	$I = m(c^2 + a^2)/12$
sphere, radius r	any diameter	$I = m(2/5)r^2$
thin rod, length l	normal to length at center	$I = ml^2/12$
thin circular sheet, radius r	normal to sheet at center	$I = mr^2/2$
	any diameter	$I = mr^2/4$
thin rectangular sheet, sides (a, b)	normal to sheet at center	$I = m(a^2 + b^2)/2$
	parallel to a at center	$I = mb^2/12$
	parallel to b at center	$I = ma^2/12$

Table 17.1: The moment of inertia about the center of mass of various geometric shapes. The mass m is assumed to be uniformly distributed.

momenta are expressed in terms of the translational $\mathbf{V}$ and rotational velocity $\boldsymbol{\omega}$ by

$$\mathbf{P} = M\mathbf{V} \tag{17.19a}$$

$$\mathbf{L} = \mathcal{I}\boldsymbol{\omega}, \tag{17.19b}$$

where M is the mass, and $\mathcal{I}$ is the moment of inertia tensor. For an unconstrained body, the point O is usually taken to be the center of mass. For a constrained body, such as a spinning top, the point O is usually taken to be the point of support.

Although the translational and rotational equations of motion appear similar, the fact that the inertia tensor is not always constant with respect to axes fixed in space complicates the analysis. To use a constant inertia tensor, we must describe the motion using a noninertial reference frame known as the *body frame* that is fixed in the body. Because we are free to orient the axes within the body, we choose axes for which the moment of inertia tensor is diagonal. These axes are referred to as the body's principal axes and are easy to determine for symmetrical objects. The diagonal elements of the moment of inertia tensor are calculated using the volume integrals

$$I_1 = \int_V \rho(y^2 + z^2)\,dV \tag{17.20a}$$

$$I_2 = \int_V \rho(z^2 + x^2)\,dV \tag{17.20b}$$

$$I_3 = \int_V \rho(x^2 + y^2)\,dV, \tag{17.20c}$$

where ρ is the mass density. The off-diagonal elements are zero in the principal axis coordinate system. The moments of inertia for various simple geometrical objects are shown in Table 17.1.

Given that the general relation between the derivative of a vector $\mathbf{A}$ in the space frame to the derivative in the body frame is (see Goldstein)

$$\left(\frac{d\mathbf{A}}{dt}\right)_{\text{space}} = \left(\frac{d\mathbf{A}}{dt}\right)_{\text{body}} + \boldsymbol{\omega} \times \mathbf{A}, \tag{17.21}$$

it is easy to show that the rotational equation of motion in the body frame can be written as

$$\frac{d\mathbf{L}}{dt} + \boldsymbol{\omega} \times (\mathcal{I}\boldsymbol{\omega}) = \mathbf{N}. \tag{17.22}$$

Equation (17.22) is Euler's equation for the motion of a rigid body. Because the moment of inertia tensor $\mathcal{I}$ is diagonal in the body frame,(17.22) may be written in component form as

$$I_1\dot{\omega}_1 + (I_3 - I_2)\omega_3\omega_2 = N_1 \tag{17.23a}$$
$$I_2\dot{\omega}_2 + (I_1 - I_3)\omega_1\omega_3 = N_2 \tag{17.23b}$$
$$I_3\dot{\omega}_3 + (I_2 - I_1)\omega_2\omega_1 = N_3. \tag{17.23c}$$

Open source physics elements support the concept of a body frame by providing `toBodyFrame` and `toSpaceFrame` methods in the `Element` class. These methods are used in Listing 17.8 to show the torque on a rectangular sheet rotating on a fixed shaft with uniform angular velocity ω (see Figure 17.4). The mass can be tilted on the shaft to produce an out-of-balance configuration that causes the system to shake unless a torque is applied to the shaft. The program displays the angular momentum vector and the torque vector using color-coded arrows.

Listing 17.8: A mass rotating on a fixed shaft with uniform angular velocity ω.

```
package org.opensourcephysics.sip.ch17;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.display3d.simple3d.*;
import org.opensourcephysics.frames.Display3DFrame;
import org.opensourcephysics.numerics.*;

public class TorqueApp extends AbstractSimulation {
   Display3DFrame frame = new Display3DFrame("Rotation test");
   Element body = new ElementBox();            // shows rigid body
   Element shaft = new ElementCylinder();      // shows shaft
   Element arrowOmega = new ElementArrow();    // shows angular velocity of shaft
   Element arrowL = new ElementArrow();        // shows angular momentum of body
   Element arrowTorque = new ElementArrow();   // shows torque on shaft
   // contains shaft and arrowOmega
   Group shaftGroup = new Group();
   // contains body, arrowL, and arrowTorque
   Group bodyGroup = new Group();
   double theta = 0, omega = 0.1, dt = 0.1;
   // principal moments of inertia
   double Ixx = 1.0, Iyy = 1.0, Izz = 2.0;

   public TorqueApp() {
      frame.setDecorationType(VisualizationHints.DECORATION_AXES);
      body.setSizeXYZ(1.0, 1.0, 0.1); // thin rectangle
      shaft.setSizeXYZ(0.1, 0.1, 0.8);
      arrowL.getStyle().setLineColor(java.awt.Color.MAGENTA);
      arrowTorque.getStyle().setLineColor(java.awt.Color.CYAN);
      bodyGroup.addElement(body);
      bodyGroup.addElement(arrowTorque);
      bodyGroup.addElement(arrowL);
      shaftGroup.addElement(bodyGroup);
      shaftGroup.addElement(arrowOmega);
      shaftGroup.addElement(shaft);
      frame.addElement(shaftGroup);
   }

   void computeVectors() {
      // convert omega to body frame
      double[] omega =
         body.toBodyFrame(new double[] {0, 0, this.omega});
      // L in body frame
      double[] angularMomentum =
         new double[] {omega[0]*Ixx, omega[1]*Iyy, omega[2]*Izz};
      // torque is computed in body frame
      double[] torque = VectorMath.cross3D(omega, angularMomentum);
      arrowL.setSizeXYZ(angularMomentum);
      arrowTorque.setSizeXYZ(torque);
      // position torque arrow at tip of angular momentum
      arrowTorque.setXYZ(angularMomentum);
   }

   public void initialize() {
      omega = control.getDouble("omega");
      arrowOmega.setSizeXYZ(0, 0, omega);
      double tilt = control.getDouble("tilt");
      double cos = Math.cos(tilt/2), sin = Math.sin(tilt/2);
      Transformation rotation = new Quaternion(cos, sin, 0, 0);
      bodyGroup.setTransformation(rotation);
      computeVectors();
   }

   public void reset() {
      control.setValue("omega", "pi/4");
      control.setValue("tilt", "pi/5");
   }

   protected void doStep() {
      theta += omega*dt;
      double cos = Math.cos(theta/2), sin = Math.sin(theta/2);
      shaftGroup.setTransformation(new Quaternion(cos, 0, 0, sin));
      computeVectors();
   }

   public static void main(String[] args) {
      SimulationControl.createApp(new TorqueApp());
   }
}
```

TorqueApp instantiates a shaft group and a body group in its constructor. The shaft group contains visual representations of the shaft, the angular velocity vector, and the body group. The body group contains representations of a thin rectangular sheet, the angular velocity, and the torque. The body group is rotated about the x-axis in the `initialize` method and the shaft group is rotated about the z-axis in the `doStep` method. Because the body group is within the shaft group, the body group also rotates. The `computeVectors` method displays the angular momentum and torque vectors by computing their values in the rotating (body) frame of reference. The torque is computed using a cross produce of the body frame angular velocity and the body

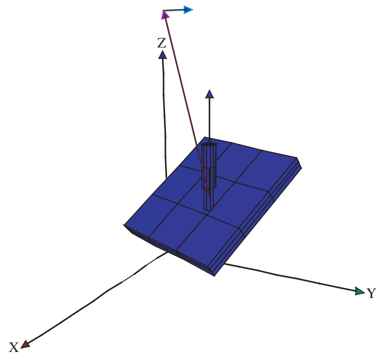


Figure 17.4: TorqueApp shows the torque on a rotating shaft with an attached rectangular sheet.

frame angular momentum according to (17.21).

A complete description of a rigid body requires three angular orientation variables as well as three angular velocity variables. The orientation is often given in terms of the Euler angles ψ, θ, and ϕ. These angles specify the orientation as a series of three independent rotations about prechosen axes and must be applied in exactly the order given because the rotation matrices do not commute. To use these angles as differential equation state variables, we must be able to calculate their rate as a function of the body frame angular velocity in (17.23). The expression for the angular velocity in the body frame in terms of the Euler angles is (see Goldstein)

$$\begin{bmatrix}\omega_1\\ \omega_2\\ \omega_3\end{bmatrix} = \begin{bmatrix}\sin\theta\sin\psi & \cos\psi & 0\\ \sin\theta\cos\psi & -\sin\psi & 0\\ \cos\theta & 0 & 1\end{bmatrix}\begin{bmatrix}\dot\phi\\ \dot\theta\\ \dot\psi\end{bmatrix}. \tag{17.24}$$

Unfortunately, the matrix in (17.24) is singular when $\sin\theta = 0$.

Because we must invert this matrix to solve for the rate of the Euler angles, the Euler angle rate equation becomes unstable when θ approaches 0 or π. A better approach for numerical work is to abandon Euler angles and to use *quaternions*.

17.5 Quaternion Arithmetic

The rotation of an arbitrary two-dimensional vector $\mathbf{A} = a_x\hat{x} + a_y\hat{y}$ by an angle θ can be reformulated using complex numbers as

$$a' = ae^{i\theta} = e^{i\theta/2}ae^{i\theta/2}, \tag{17.25}$$

where the vector $\mathbf{A}$ is expressed as a complex number $a = a_x + ia_y$. The real and imaginary components correspond to the vector components. This idea can be extended to three dimensions using quaternions. A quaternion can be represented in terms of real and *hypercomplex* numbers i, j, and k as

$$\hat{q} = q_0 + iq_1 + jq_2 + kq_3 = (q_0, q_1, q_2, q_3), \tag{17.26}$$

where the hypercomplex numbers obey Hamilton's rules

$$i^2 = j^2 = k^2 = ijk = -1. \tag{17.27}$$

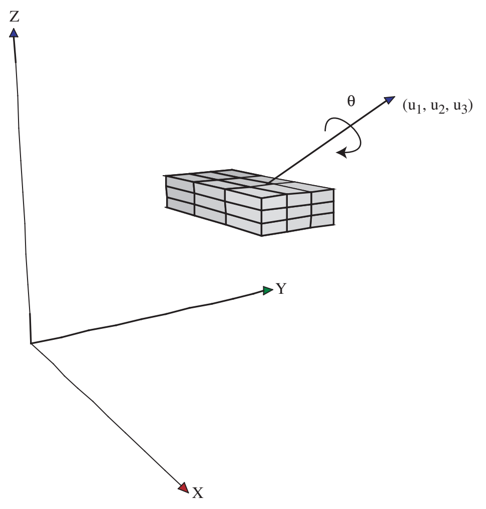


Figure 17.5: The most general change of a rigid body with one point fixed is a rotation about a fixed axis with direction cosines (u_1, u_2, u_3) by the angle θ.

Similar to imaginary numbers, the quaternion conjugate is defined as

$$\hat{q}^* = q_0 - iq_1 - jq_2 - kq_3 = (q_0, -q_1, -q_2, -q_3). \tag{17.28}$$

Unlike imaginary numbers, quaternion multiplication does not commute and obeys the rules

$$ij = k, \quad jk = i, \quad ki = j, \quad ji = -k, \quad kj = -1, \quad ik = -j. \tag{17.29}$$

Although it would be a mistake to identify hypercomplex numbers with unit vectors in three-dimensional space (just as it would be a mistake to identify the imaginary number i with the y direction in a two-dimensional space), it is convenient to think of a quaternion as the sum of a scalar q_0 and a vector $\mathbf{q}$

$$\hat{q} = q_0 + \mathbf{q} = (q_0, \mathbf{q}). \tag{17.30}$$

The quaternion is said to be *pure* if the scalar part is zero.

By using the above definitions, the product of two quaternions $\hat{p}$ and $\hat{q}$ can be shown to be

$$\hat{p}\hat{q} = (p_0, \mathbf{p})(q_0, \mathbf{q}) = q_0 p_0 - \mathbf{q} \cdot \mathbf{p} + q_0 \mathbf{p} + p_0 \mathbf{q} + \mathbf{p} \times \mathbf{q}. \tag{17.31}$$

Note that except for the additional cross product term, quaternion multiplication is similar to complex multiplication, $(a_0, a_1)(b_0, b_1) = (a_0 b_0 - a_1 b_1, a_1 b_0 + b_1 a_0)$. The norm (length) of a quaternion is defined to be $|\hat{q}\hat{q}^*| = q_0 q_0 + q_1 q_1 + q_2 q_2 + q_3 q_3$.

Exercise 17.10. Quaternion multiplication

Show that Hamilton's rules for hypercomplex numbers in (17.27) lead to (17.31). □

Euler has shown that the most general change of a rigid body with one point fixed is a rotation about a fixed axis as shown in Figure 17.5. Quaternions provide an elegant representation of both the rotation angle and the axis orientation. It can be shown (see Shoemake) that a rotation through an angle θ about an axis with direction cosines (u_1, u_2, u_3) can be represented as a unit quaternion with components

$$\hat{q} = \cos\frac{\theta}{2} + (iu_1 + ju_2 + ku_3)\sin\frac{\theta}{2}. \tag{17.32}$$

To stress the analogy with the complex exponential $e^{i\theta} = \cos\theta + i\sin\theta$, some textbooks represent this rotation through θ about the axes $\mathbf{u}$ using exponential notation $\hat{q} = e^{\mathbf{u}\theta/2}$.

Given a unit quaternion $\hat{q}$ that represents a rotation, how do we apply this rotation to an arbitrary vector **A**? If we define a pure quaternion $\hat{a} = (0, a_x, a_y, a_z)$, it can be shown that

$$\hat{a}' = \hat{q}\hat{a}\hat{q}^*, \tag{17.33}$$

where the resulting quaternion $\hat{a}' = (0, a_x', a_y', a_z')$ contains the components of the rotated vector $\mathbf{A}'$ (see Rapaport). Note the similarity to (17.25) if the quaternion is represented using exponential notation.

Exercise 17.11. Quaternion rotation

(a) Use the properties of the direction cosines to show that (17.32) defines a quaternion of unit length.

(b) Show that the length of the vector **A** does not change when using (17.33). □

Bodies can be oriented using any representation of a rotation including quaternions, Euler angles, and rotation matrices. Because the quaternion representation does not involve trigonometric functions, it is very efficient for computing rigid body dynamics. To make it easy to create other representations of transformations, we define a `Transformation` interface in the `numerics` package. A concrete implementation of the interfaces based on rotation matrices is described in Appendix A. The quaternion implementation is similar and is available in the `numerics` package. Listing 17.9 shows how the quaternion implementation is used to rotate a `BoxWithArrows`. Because `BoxWithArrows` is a subclass of `Group` containing a box and three arrows, it is not listed.

Listing 17.9: A class that tests the quaternion representation of rotations.

```java
package org.opensourcephysics.sip.ch17;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.numerics.Quaternion;
import org.opensourcephysics.frames.*;
import org.opensourcephysics.display3d.simple3d.VisualizationHints;

public class QuaternionApp extends AbstractCalculation {
   Display3DFrame frame = new Display3DFrame("Quaternion rotations");
   Quaternion transformation = new Quaternion(1, 0, 0, 0);
   BoxWithArrows box = new BoxWithArrows();

   public QuaternionApp() {
      frame.setDecorationType(VisualizationHints.DECORATION_AXES);
      // scene is simple, so draw it properly when rotating
      frame.setAllowQuickRedraw(false);
      frame.setPreferredMinMax(-6, 6, -6, 6, -6, 6);
      box.setTransformation(transformation);
      frame.addElement(box);
   }

   public void calculate() {
      double q0 = control.getDouble("q0");
      double q1 = control.getDouble("q1");
      double q2 = control.getDouble("q2");
      double q3 = control.getDouble("q3");
      transformation.setCoordinates(q0, q1, q2, q3);
      box.setTransformation(transformation);
   }

   public void reset() {
      control.clearMessages();
      control.setValue("q0", 1);
      control.setValue("q1", 0); // initial orientation is along x axis
      control.setValue("q2", 0);
      control.setValue("q3", 0);
      calculate();
   }

   public static void main(String[] args) {
      CalculationControl.createApp(new QuaternionApp());
   }
}
```

Exercise 17.12. Quaternion representation of rotations

(a) Compile and run the QuaternionApp target class. Find and then test a quaternion that orients the long side of the box at 45° in the xy plane. Repeat for the yz plane.

(b) Use a quaternion to orient the long axis of the box along an axis in the $(1,2,1)$ direction.

(c) What happens if the quaternion does not have unit norm? □

17.6 Quaternion equations of motion

Because we will often transform torque and other vectors to and from the rotating object's body frame, we need the transformation matrix from the space frame to the body frame using quaternions. We represent a rotation using a unit quaternion (q_0, q_1, q_2, q_3), carry out (17.33) using (17.31), and express the result in matrix form. The resulting rotation matrix is

$$\mathcal{R} = 2\begin{bmatrix} \frac{1}{2} - q_2^2 - q_3^2 & q_1q_2 + q_0q_3 & q_1q_3 - q_0q_2 \\ q_1q_2 - q_0q_3 & \frac{1}{2} - q_1^2 - q_3^2 & q_2q_3 + q_0q_1 \\ q_1q_3 + q_0q_2 & q_2q_3 - q_0q_1 & \frac{1}{2} - q_1^2 - q_2^2 \end{bmatrix}. \tag{17.34}$$

This matrix is equal to the rotation matrix derived from the Rodrigues formula (17.15).

The angular velocity in the body frame can be written as $\dot{\hat{q}}(t) = \frac{1}{2}\hat{\omega}(t)\hat{q}(t)$, where $\hat{\omega}$ is a pure quaternion $(0, \omega_1, \omega_2, \omega_3)$. Our discussion follows the derivation in Rapaport. The time dependence of a vector $\mathbf{r}$ can be expressed as a transformation of its initial value $\mathbf{r}_0$ as

$$\hat{r}(t) = \hat{q}(t)\hat{r}_0\hat{q}^*(t). \tag{17.35}$$

If we differentiate $\hat{r}(t)$ with respect to time, we have

$$\dot{\hat{r}} = \dot{\hat{q}}\hat{r}_0\hat{q}^* + \hat{q}\hat{r}_0\dot{\hat{q}}^*, \tag{17.36}$$

where we have dropped the explicit time dependence of $\hat{q}$. We substitute $\hat{r}_0 = \hat{q}^*\hat{r}\hat{q}$ and obtain

$$\dot{\hat{r}} = \dot{\hat{q}}\hat{q}^*\hat{r} + \hat{r}\hat{q}\dot{\hat{q}}^* = \dot{\hat{q}}\hat{q}^*\hat{r} - \hat{r}\dot{\hat{q}}\hat{q}^*, \tag{17.37}$$

where we have used the fact $\hat{q}\hat{q}^* = 1$. The only part of the $\hat{r}$ and $\dot{\hat{q}}\hat{q}^*$ product that does not commute is the vector cross product. The scalar part commutes and is zero. If we denote the pure (vector) part of the quaternion $\dot{\hat{q}}\hat{q}^*$ by $\mathbf{u}$, we find

$$\dot{\mathbf{r}} = \mathbf{u}\times\mathbf{r} - \mathbf{r}\times\mathbf{u} = 2\mathbf{u}\times\mathbf{r}. \tag{17.38}$$

Because $\dot{\mathbf{r}} = \boldsymbol{\omega}\times\mathbf{r}$ for rotational motion, we obtain

$$\boldsymbol{\omega} = 2\mathbf{u}. \tag{17.39}$$

Equation (17.39) can be expressed using components by writing $\boldsymbol{\omega}$ as

$$\begin{bmatrix}\omega_1\\ \omega_2\\ \omega_3\\ 0\end{bmatrix} = 2\mathcal{W}\begin{bmatrix}\dot{q_0}\\ \dot{q_1}\\ \dot{q_2}\\ \dot{q_3}\end{bmatrix}, \tag{17.40}$$

where

$$\mathcal{W} = \begin{bmatrix}-q_1 & q_0 & q_3 & -q_2\\ -q_2 & -q_3 & q_0 & q_1\\ -q_3 & q_2 & -q_1 & q_0\\ q_0 & q_1 & q_2 & q_3\end{bmatrix}. \tag{17.41}$$

Because $\mathcal{W}$ is orthogonal, the transpose of (17.41) is its inverse $\mathcal{W}^T\mathcal{W} = \mathbf{1}$, and

$$\begin{bmatrix}\dot{q_0}\\ \dot{q_1}\\ \dot{q_2}\\ \dot{q_3}\end{bmatrix} = \frac{1}{2}\mathcal{W}^T\begin{bmatrix}\omega_1\\ \omega_2\\ \omega_3\\ 0\end{bmatrix}. \tag{17.42}$$

Because the quaternion derivatives depend on both $\hat{q}$ and $\boldsymbol{\omega}$ even in the simplest Euler-like ODE algorithm, solving (17.42) is a two-step process. A modified leap-frog algorithm that advances the angular momentum in the space frame at every time step

$$\mathbf{L}(t+\Delta t/2) = \mathbf{L}(t-\Delta t/2) + \mathbf{N}(t)\Delta t \tag{17.43}$$

and then transforms the angular momentum to the body frame to advance the quaternion

$$\hat{q}(t+\Delta t) = \hat{q}(t) + \dot{\hat{q}}(t)\Delta t \tag{17.44}$$

may be applied to simple systems (see Allen and Tildesley). We use this algorithm to study the dynamics of a spinning plate.

Richard Feynman (see references) noticed that the wobble and spin of a Cornell cafeteria plate as it was tossed into the air was in a 2:1 ratio. Although this result can be derived analytically,[2] we will simulate the dynamics using (17.44) to discover the simple dynamics and to test the numerical algorithm.

`FeynmanPlate` in Listing 17.10 and `FeynmanPlateView` in Listing 17.11 simulate the torque-free rotation of a body about its center of mass. Because `FeynmanPlateApp` is similar to other target programs, we do not list it here.

[2]See Tuleja et al. for a simple and elegant proof of the wobble-to-spin ratio using basic mechanics and symmetry arguments.

Listing 17.10: The FeynmanPlate class implements the dynamics of a rotating plate.

```
package org.opensourcephysics.sip.ch17;
import org.opensourcephysics.numerics.*;

public class FeynmanPlate {
   Quaternion toBody = new Quaternion(1, 0, 0, 0);
   // spaceview displays the plate as seen from the laboratory

   FeynmanPlateView spaceView = new FeynmanPlateView(this);
   double[] spaceL = new double[3]; // space frame angular momentum
   double I1 = 1, I2 = 1, I3 = 1;   // default moments of inertia
   double wx = 0, wy = 0, wz = 0;   // angular velocity in the body frame
   double q0, q1, q2, q3;           // quaternion components
   double dt = 0.1;

   void setOrientation(double[] q) {
      double norm = Math.sqrt(q[0]*q[0]+q[1]*q[1]+q[2]*q[2]+q[3]*q[3]);
      q0 = q[0]/norm;
      q1 = q[1]/norm;
      q2 = q[2]/norm;
      q3 = q[3]/norm;
      toBody.setCoordinates(q0, q1, q2, q3);
      spaceView.initialize();
   }

   Transformation getTransformation() {
      toBody.setCoordinates(q0, q1, q2, q3);
      return toBody;
   }

   void setInertia(double I1, double I2, double I3) {
      setOrientation(new double[] {1, 0, 0, 0}); // default orientation
      this.I1 = I1;
      this.I2 = I2;
      this.I3 = I3;
      spaceView.initialize();
   }

   void computeOmegaBody() {
      double[] bodyL = (double[]) spaceL.clone();
      toBody.inverse(bodyL);
      wx = bodyL[0]/I1;
      wy = bodyL[1]/I2;
      wz = bodyL[2]/I3;
   }

   public void advanceTime() {
      computeOmegaBody();
      // compute quaternion rate of change
      double q0dot = 0.5*(-q1*wx-q2*wy-q3*wz); // dq0/dt
      double q1dot = 0.5*(q0*wx-q3*wy+q2*wz);  // dq1/dt
      double q2dot = 0.5*(q3*wx+q0*wy-q1*wz);  // dq2/dt
      double q3dot = 0.5*(-q2*wx+q1*wy+q0*wz); // dq3/dt
      // update quaternion
      q0 += q0dot*dt;
      q1 += q1dot*dt;
      q2 += q2dot*dt;
      q3 += q3dot*dt;
      // normalize to eliminate drift
      double norm = 1/Math.sqrt(q0*q0+q1*q1+q2*q2+q3*q3);
      q0 *= norm;
      q1 *= norm;
      q2 *= norm;
      q3 *= norm;
      toBody.setCoordinates(q0, q1, q2, q3);
      spaceView.update();
   }
}
```

FeynmanPlate contains a transformation QuaternionRotation that keeps track of a spinning plate using quaternions. The physics and the algorithm are contained in the advanceTime method. The first step is to compute $\boldsymbol{\omega}$ in the body frame using $\boldsymbol{\omega} = \mathcal{I}^{-1}\mathbf{L}$. This calculation is simple because the angular momentum in the space frame $\mathbf{L}$ is constant and the inverse of a diagonal matrix is also diagonal with diagonal elements equal to the reciprocal of the original elements. The second step is to compute $\dot{\hat{q}}$ using (17.42) and advance the quaternion components. Because most numerical algorithms are subject to drift, we renormalize the quaternion before updating the view as seen from the inertial laboratory (space) reference frame.

Listing 17.11: The FeynmanPlateView class shows a spinning plate as seen from the space frame.

```
package org.opensourcephysics.sip.ch17;
import org.opensourcephysics.display3d.simple3d.*;
import org.opensourcephysics.frames.*;
import org.opensourcephysics.numerics.VectorMath;

public class FeynmanPlateView {
   Element plate = new ElementCylinder();
   Element omega = new ElementArrow();
   Element angularMomentum = new ElementArrow();
   Element bodyX = new ElementArrow(); // x axis in body frame
   Element bodyY = new ElementArrow(); // y axis in body frame
   ElementTrail trailX = new ElementTrail();
   ElementTrail trailY = new ElementTrail();
   // note that OSP does Greek symbols
   ElementText labelOmega = new ElementText("$\\omega$");
   Group bodyGroup = new Group();
   Display3DFrame frame = new Display3DFrame("Space View");
   FeynmanPlate rigidBody;

   public FeynmanPlateView(FeynmanPlate rigidBody) {
      this.rigidBody = rigidBody; // save a reference to the model
      plate.setSizeXYZ(2, 2, 0.1);
      bodyX.setSizeXYZ(1.4, 0, 0);
      bodyY.setSizeXYZ(0, 1.4, 0);
      frame.setPreferredMinMax(-2, 2, -2, 2, -2, 2);
      frame.setSize(600, 600);
      plate.getStyle().setFillColor(java.awt.Color.LIGHT_GRAY);
      frame.setDecorationType(VisualizationHints.DECORATION_NONE);
      //frame.setDecorationType(VisualizationHints.DECORATION_AXES);
      omega.getStyle().setFillColor(java.awt.Color.YELLOW);
      angularMomentum.getStyle().setFillColor(java.awt.Color.BLUE);
      bodyX.getStyle().setResolution(new Resolution(10));
      bodyY.getStyle().setResolution(new Resolution(10));
      bodyX.getStyle().setFillColor(java.awt.Color.RED);
      bodyY.getStyle().setFillColor(java.awt.Color.GREEN);
      trailX.getStyle().setLineColor(java.awt.Color.RED);
      trailY.getStyle().setLineColor(java.awt.Color.GREEN);
      bodyGroup.addElement(plate);
      bodyGroup.addElement(omega);
      bodyGroup.addElement(labelOmega);
      bodyGroup.addElement(bodyX);
      bodyGroup.addElement(bodyY);
      // coordinates in space
      frame.addElement(trailX);
      frame.addElement(trailY);
      frame.addElement(angularMomentum);
      frame.addElement(bodyGroup);
      // scene is simple, so draw it properly when rotating
      frame.setAllowQuickRedraw(false);
      bodyGroup.setTransformation(rigidBody.getTransformation());
   }

   void initialize() {
      trailX.clear();
      trailY.clear();
      update();
   }

   void update() {
      bodyGroup.setTransformation(rigidBody.getTransformation());
      double[] vec =
         new double[] {rigidBody.wx, rigidBody.wy, rigidBody.wz};
      double norm = Math.sqrt(vec[0]*vec[0]+vec[1]*vec[1]+vec[2]*vec[2]);
      norm = Math.max(norm, 1.0e-6);
      double s = VectorMath.magnitude(rigidBody.spaceL)/norm;
      // scale omega to same length as angular momentum
      omega.setSizeXYZ(s*vec[0], s*vec[1], s*vec[2]);
      labelOmega.setXYZ(s*vec[0], s*vec[1], s*vec[2]);
      angularMomentum.setSizeXYZ(rigidBody.spaceL);
      double[] tipX = new double[] {bodyX.getSizeX(), 0, 0};
      bodyGroup.toSpaceFrame(tipX);
      trailX.addPoint(tipX[0], tipX[1], tipX[2]);
      double[] tipY = new double[] {0, bodyY.getSizeY(), 0};
      bodyGroup.toSpaceFrame(tipY);
      trailY.addPoint(tipY[0], tipY[1], tipY[2]);
   }
}
```

`FeynmanPlateView` shows a spinning plate as seen from the space frame. The elements needed to visualize the plate as seen in the laboratory (space) frame are instantiated in the constructor. The plate is represented by an `ElementCylinder` object and the x- and y-body axes are `ElementArrow` objects. Additional arrows are used to show the angular momentum and angular velocity. As the body frame axes rotate, we add points to trail objects to display the trajectories of the body axis arrows. These trails show the wobble. The plate, angular velocity arrow, and axes arrows are placed in a group so that their orientation can be set with a single transformation. The angular momentum arrow is added to the 3D frame because it is constant in the space frame. Because the trails are also added to the 3D frame, we will need to transform the tips of the body axes vectors into the space frame in order to add points to the trails.

The `update` method in the `FeynmanPlateView` is invoked after every time step. The method begins by setting the body group's transformation using the transformation computed in `FeynmanPlate`. The body frame's angular velocity components are then scaled so that the angular velocity arrow has the same length as the angular momentum arrow. (Relative arrow length has little meaning because angular velocity has dimension $1/L$ and angular momentum has dimension ML^2/T.) Note that $\boldsymbol{\omega}$ is not transformed separately because this arrow is in the body group and this group has been transformed. However, the coordinates of the tips of the body-axes arrows are transformed into the space frame because the trails are not part of the body group.

Problem 17.13. Simulation of a wobbling plate

(a) Run the target class `FeynmanPlateApp`. Does your simulation confirm Feynman's observation of the wobbling plate in the Cornell cafeteria? Are the discrepancies between Feynman's description and the simulation due to the fact that the wobble is not small or due to numerical inaccuracies? Justify your answer.

(b) How well does the algorithm conserve energy and angular momentum?

(c) Will a rectangular food tray wobble the same way as a plate?

(d) Does the plate wobble if it spun about an axis close to a diameter rather than close to a line perpendicular to the flat side of the disk? □

Problem 17.14. Torque-free rotation about a principal axis

(a) Add a second visualization showing the motion as viewed in the noninertial body frame.

(b) If the angular velocity vector coincides with a body's principal axis, the angular momentum and the angular velocity coincide. The body should then rotate steadily about the corresponding principal axis because the net torque is zero and the angular momentum is constant. Does the simulation show this result for all three axes if the moments of inertia are unequal? Perturb the angular velocity. Are rotations about the three principal axes stable or unstable? Check each axis. Repeat this simulation with a different set of moments of inertia. □

Problem 17.15. Torque-free rotation of a symmetrical body

(a) Add a plot showing the time dependence of the angular velocity components $\omega_1(t)$, $\omega_2(t)$, and $\omega_3(t)$ to the `FeynmanPlateApp` class.

(b) Verify that $\omega_3(t)$ is constant if $I_1 = I_2 = I_s$.

(c) Verify that $\omega_1(t)$ and $\omega_2(t)$ exhibit an out-of-phase sinusoidal dependence if $I_1 = I_2$.

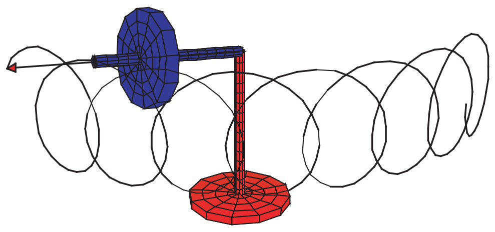


Figure 17.6: The dynamics of a spinning top are computed using a `RigidBodyModel` and displayed using a `SpinningTopSpaceView`.

(d) If α denotes the angle between the body-frame $\hat{3}$-axis and the axis of rotation, verify that

$$\Omega = \left(\frac{I_3}{I_s} - 1\right)\omega\cos\alpha, \tag{17.45}$$

where Ω is the angular velocity of the $\boldsymbol{\omega}$ vector's precession about the body frame's $\hat{3}$-axis. □

Problem 17.16. Free rotation of a thin cylindrical rod

Model the free rotation of a thin cylindrical rod. Show that $\boldsymbol{\omega}$ precesses about the axis of the rod. Why does $\boldsymbol{\omega}$ precess? Why does $\mathbf{L}$ not precess? How is the frequency of precession related to the moment of inertia of the rod? □

17.7 Rigid Body Model

As seen in Problem 17.13, the simple rigid body algorithm presented in Section 17.6 does a good job of conserving energy and angular momentum but should be improved if our goal is to compute accurate trajectories over long times. To use a higher-order differential equation solver, we need to obtain an expression for the second derivative of the quaternion by eliminating the angular velocities from Euler's equation of motion (17.23). The resulting quaternion acceleration is the rate for the quaternion velocity in a differential equation solver.

If we start with

$$\frac{d}{dt}\hat{q}^*\dot{\hat{q}} = \dot{\hat{q}}^*\dot{\hat{q}} + \hat{q}^*\ddot{\hat{q}}, \tag{17.46}$$

multiply by $\hat{q}$, and rearrange the terms, we obtain

$$\ddot{\hat{q}} = \hat{q}\left(\frac{d}{dt}\hat{q}^*\dot{\hat{q}} - \dot{\hat{q}}^*\dot{\hat{q}}\right). \tag{17.47}$$

Some messy algebra shows that (17.47) can be written in matrix form as

$$\begin{bmatrix}\ddot{q}_0\\ \ddot{q}_1\\ \ddot{q}_2\\ \ddot{q}_3\end{bmatrix} = \frac{1}{2}\mathcal{W}^T\begin{bmatrix}\dot{\omega}_1\\ \dot{\omega}_2\\ \dot{\omega}_3\\ -2\Sigma\dot{q}_m^2\end{bmatrix}. \tag{17.48}$$

The dynamical behavior of a rigid body can now be expressed in terms of a quaternion state vector $(q_0, \dot{q}_0, q_1, \dot{q}_1, q_2, \dot{q}_2, q_3, \dot{q}_3, t)$. The following steps summarize the algorithm for computing the rate for this state:

1. Use the state vector to compute the angular velocity $(\omega_1, \omega_2, \omega_3)$ in the body frame using (17.40).
2. Project the external torques onto the body frame and compute $\dot{\omega}$ using Euler's equation (17.23).
3. Compute the quaternion acceleration using (17.48).

These steps are implemented in the RigidBody class shown in Listing 17.12.

Listing 17.12: The RigidBody class solves Euler's equation of motion for a rotating rigid body using quaternions.

```
package org.opensourcephysics.sip.ch17;
import org.opensourcephysics.numerics.*;

public class RigidBody implements ODE {
   Quaternion rotation = new Quaternion(1, 0, 0, 0);
   double[] state = new double[9];
   ODESolver solver = new RK45MultiStep(this);
   double[] omegaBody = new double[3];        // body frame omega
   // body frame angular momentum
   double[] angularMomentumBody = new double[3];
   // principal moments of inertia
   double I1 = 1, I2 = 1, I3 = 1;
   // angular acceleration in the body frame
   double wxdot = 0, wydot = 0, wzdot = 0;
   // torque in the body frame
   double t1 = 0, t2 = 0, t3 = 0;

   void setOrientation(double[] q) {
      double norm = Math.sqrt(q[0]*q[0]+q[1]*q[1]+q[2]*q[2]+q[3]*q[3]);
      state[0] = q[0]/norm;
      state[2] = q[1]/norm;
      state[4] = q[2]/norm;
      state[6] = q[3]/norm;
      rotation.setCoordinates(state[0], state[2], state[4], state[6]);
   }

   Transformation getTransformation() {
      rotation.setCoordinates(state[0], state[2], state[4], state[6]);
      return rotation;
   }

   void setBodyFrameOmega(double[] omega) {
      // use components for clarity
      double q0 = state[0], q1 = state[2], q2 = state[4], q3 = state[6];
      double wx = omega[0];
      double wy = omega[1];
      double wz = omega[2];
      state[1] = 0.5*(-q1*wx-q2*wy-q3*wz); // dq0/dt
      state[3] = 0.5*(q0*wx-q3*wy+q2*wz);  // dq1/dt
      state[5] = 0.5*(q3*wx+q0*wy-q1*wz);  // dq2/dt
      state[7] = 0.5*(-q2*wx+q1*wy+q0*wz); // dq3/dt
      updateVectors();
```

```
}

public double[] getBodyFrameOmega() {
   return omegaBody;
}

public double[] getBodyFrameAngularMomentum() {
   return angularMomentumBody;
}

void updateVectors() {
   double q0 = state[0], q1 = state[2], q2 = state[4], q3 = state[6];
   omegaBody[0] = 2*(-q1*state[1]+q0*state[3]+q3*state[5]-q2*state[7]);
   omegaBody[1] = 2*(-q2*state[1]-q3*state[3]+q0*state[5]+q1*state[7]);
   omegaBody[2] = 2*(-q3*state[1]+q2*state[3]-q1*state[5]+q0*state[7]);
   angularMomentumBody[0] = I1*omegaBody[0];
   angularMomentumBody[1] = I2*omegaBody[1];
   angularMomentumBody[2] = I3*omegaBody[2];
}

public void advanceTime() {
   solver.step();
   double norm =
     1/Math.sqrt(state[0]*state[0]+state[2]*state[2]+state[4]*state[4]
        +state[6]*state[6]);
   state[0] *= norm;
   state[2] *= norm;
   state[4] *= norm;
   state[6] *= norm;
   updateVectors();
}

public double[] getState() {
   return state;
}

public void getRate(double[] state, double[] rate) {
   computeBodyFrameAcceleration(state);
   double sum = 0;
   for(int i = 1;i<9;i += 2) { // sum the q dot values
      sum += state[i]*state[i];
   }
   sum = -2.0*sum;
   // use q components for clarity
   double q0 = state[0], q1 = state[2], q2 = state[4], q3 = state[6];
   rate[0] = state[1];
   rate[1] = 0.5*(-q1*wxdot-q2*wydot-q3*wzdot+q0*sum);
   rate[2] = state[3];
   rate[3] = 0.5*(q0*wxdot-q3*wydot+q2*wzdot+q1*sum);
   rate[4] = state[5];
   rate[5] = 0.5*(q3*wxdot+q0*wydot-q1*wzdot+q2*sum);
   rate[6] = state[7];
   rate[7] = 0.5*(-q2*wxdot+q1*wydot+q0*wzdot+q3*sum);
```

```
      rate[8] = 1.0; // time rate
   }

   void computeBodyFrameTorque(double[] state) {
      t1 = t2 = t3 = 0;
   }

   void computeBodyFrameAcceleration(double[] state) {
      // use components for clarity
      double q0 = state[0], q1 = state[2], q2 = state[4], q3 = state[6];
      double wx = 2*(-q1*state[1]+q0*state[3]+q3*state[5]-q2*state[7]);
      double wy = 2*(-q2*state[1]-q3*state[3]+q0*state[5]+q1*state[7]);
      double wz = 2*(-q3*state[1]+q2*state[3]-q1*state[5]+q0*state[7]);
      computeBodyFrameTorque(state);
      wxdot = (t1-(I3-I2)*wz*wy)/I1; // Euler's equations of motion
      wydot = (t2-(I1-I3)*wx*wz)/I2;
      wzdot = (t3-(I2-I1)*wy*wx)/I3;
   }
}
```

The RigidBody class makes use of the RigidBodyUtil utility class in the ch17 package to handle quaternion normalization and transformations between space and body frames. The RigidBodyUtil class is not listed. The static spaceToBody method multiples a given vector by (17.34) and the static bodyToSpace method multiples the given vector by the inverse.

FreeRotationApp and FreeRotationView use a subclass of RigidBody to display the dynamics of free rotation. The FreeRotation subclass does a number of simple housekeeping chores such as computing the principal moments using the formulas for an ellipsoid in Table 17.1. FreeRotationApp animates torque-free rotation by extending AbstractSimulation and implementing the doStep method. These classes are not listed because they are similar to other classes we have studied.

FreeRotationSpaceView shown in Listing 17.13 displays the body, the angular momentum vector, and the angular velocity vector in a Display3DFrame. The body is represented using an ellipsoid whose principal axes are $(2a, 2b, 2c)$ and the orientation of the body is computed in RigidBody. The angular momentum and angular velocity vectors are retrieved from RigidBody and used to set the direction of the corresponding arrows. The path of the angular velocity arrow through space is recorded by adding points to an ElementTrail object.

Exercise 17.17. Rigid body model

Study the FreeRotationSpaceView and how it is used by FreeRotationApp. Adapt the FeynmanPlateView so that it too uses the RigidBody class to compute the dynamics. Do you obtain the same results as in Problem 17.13? □

Listing 17.13: The FreeRotationSpaceView class displays the rotation of a rigid body as seen in the space frame.

```
package org.opensourcephysics.sip.ch17;
import org.opensourcephysics.display3d.simple3d.*;
import org.opensourcephysics.frames.*;

public class FreeRotationSpaceView {
   Element ellipsoid = new ElementEllipsoid();
   Element omega = new ElementArrow();
   Element angularMomentum = new ElementArrow();
   ElementTrail omegaTrace = new ElementTrail();
   Display3DFrame frame = new Display3DFrame("Space view");
   FreeRotation rigidBody;
   double scale = 1;

   public FreeRotationSpaceView(FreeRotation _rigidBody) {
      this.rigidBody = _rigidBody;
      frame.setSize(600, 600);
      frame.setDecorationType(VisualizationHints.DECORATION_AXES);
      omega.getStyle().setFillColor(java.awt.Color.RED);
      omegaTrace.getStyle().setLineColor(java.awt.Color.RED);
      angularMomentum.getStyle().setFillColor(java.awt.Color.GREEN);
      ellipsoid.setTransformation(rigidBody.getTransformation());
      frame.addElement(ellipsoid);
      frame.addElement(omega);
      frame.addElement(angularMomentum);
      frame.addElement(omegaTrace);
   }

   void initialize(double a, double b, double c) {
      ellipsoid.setSizeXYZ(2*a, 2*b, 2*c);
      // bounding dimension for space frame
      scale = Math.max(Math.max(3*a, 3*b), 3*c);
      frame.setPreferredMinMax(-scale, scale, -scale, scale,
         -scale, scale);
      omegaTrace.clear();
      update();
   }

   void update() {
      ellipsoid.setTransformation(rigidBody.getTransformation());
      double[] vec = ellipsoid.toSpaceFrame(rigidBody.getBodyFrameOmega());
      double norm = Math.sqrt(vec[0]*vec[0]+vec[1]*vec[1]+vec[2]*vec[2]);
      norm = Math.max(norm, 1.0e-6);
      double s = 0.75*scale/norm;
      omega.setSizeXYZ(s*vec[0], s*vec[1], s*vec[2]);
      omegaTrace.addPoint(s*vec[0], s*vec[1], s*vec[2]);
      vec = ellipsoid.toSpaceFrame(
         rigidBody.getBodyFrameAngularMomentum());
      norm = Math.sqrt(vec[0]*vec[0]+vec[1]*vec[1]+vec[2]*vec[2]);
      norm = Math.max(norm, 1.0e-6);
      s = 0.75*scale/norm;
      angularMomentum.setSizeXYZ(s*vec[0], s*vec[1], s*vec[2]);
      frame.repaint();
   }
}
```

17.8 Motion of a spinning top

Object oriented programming makes it easy to add a torque to the `RigidBody` class. All that needs to be done is to extend `RigidBody` and override the `computeBodyFrameTorque` method. A spinning top shown in Figure 17.6 is a rigid body that rotates about a pivot. Because the pivot point is not the center of mass, there is an external torque due to the weight of the top. We must compute the torque's vector components in the body frame to use Euler's equation of motion.

The SpinningTop shown in Listing 17.14 models a symmetric body rotating about the axis of symmetry. The axis of symmetry is taken to be the $\hat{3}$-axis. If we assume that the center of mass lies a unit distance from the pivot along the $\hat{3}$-axis, then the torque is computed by taking the cross project after projecting the force into the body frame $\tau = \hat{3} \times F_{\text{body}}$. Listing 17.14 assumes that the center of mass is one unit from the pivot and is acted on by an external force in the z-direction $(0,0,1)$.

Listing 17.14: The SpinningTop class models a symmetric body rotating about the axis of symmetry.

```
package org.opensourcephysics.sip.ch17;
public class SpinningTop extends RigidBody {
   SpinningTopSpaceView spaceView = new SpinningTopSpaceView(this);

   void setInertia(double Is, double Iz) {
      I1 = Is;
      I2 = Is;
      I3 = Iz;
      // orient top along y axis
      setOrientation(new double[] {1/Math.sqrt(2),
         1/Math.sqrt(2), 0, 0});
      spaceView.initialize();
   }

   public void advanceTime() {
      super.advanceTime();
      spaceView.update();
   }

   void setBodyFrameOmega(double[] omega) {
      super.setBodyFrameOmega(omega);
      spaceView.initialize();
   }

   void computeBodyFrameTorque(double[] state) {
      // external force in space frame
      double[] vec = new double[] {0, 0, 1};
      RigidBodyUtil.spaceToBody(state, vec);
      t1 = -vec[1]; // torque components declared in RigidBody
      t2 = vec[0];
      t3 = 0;
   }
}
```

The visualization of a spinning top can take many forms. Listing 17.15 presets the spinning top model as a table-top gyroscope using cylinders and arrows. The shaft of the gyroscope is the body's axis of symmetry (the body frame's $\hat{3}$-axis) and draws a trace showing the gyroscope's precession. The spinning about the $\hat{3}$-axis) might appear to be incorrect if the animation step is too large. This aliasing effect is often seen in movies showing carriage wheel spokes rotating backward to the direction of travel.

Listing 17.15: The SpinningTopSpaceView class models a symmetric body rotating about the axis of symmetry.

```
package org.opensourcephysics.sip.ch17;
import org.opensourcephysics.frames.*;
import org.opensourcephysics.display3d.simple3d.*;
import org.opensourcephysics.numerics.Transformation;

public class SpinningTopSpaceView {
   Group topGroup = new Group();
   Element shaft = new ElementCylinder();
   Element disk = new ElementCylinder();
   Element base = new ElementCylinder();
   Element post = new ElementCylinder();
   Element orientation = new ElementArrow();
   ElementTrail orientationTrace = new ElementTrail();
   Display3DFrame frame = new Display3DFrame("Space View");
   SpinningTop rigidBody;

   public SpinningTopSpaceView(SpinningTop rigidBody) {
      this.rigidBody = rigidBody;
      frame.setSize(600, 600);
      // panel.setDisplayMode(VisualizationHints.DISPLAY_NO_PERSPECTIVE);
      double d = 4;
      frame.setPreferredMinMax(-d, d, -d, d, -d, d);
      frame.setDecorationType(VisualizationHints.DECORATION_AXES);
      orientation.getStyle().setFillColor(java.awt.Color.RED);
      orientationTrace.getStyle().setLineColor(java.awt.Color.BLACK);
      base.setSizeXYZ(2, 2, 0.15);
      base.getStyle().setResolution(new Resolution(4, 12, 1));
      base.getStyle().setFillColor(java.awt.Color.RED);
      base.setZ(-3);
      post.setSizeXYZ(0.2, 0.2, 3);
      post.getStyle().setResolution(new Resolution(2, 10, 15));
      post.setZ(-1.5); // shift by half the length
      post.getStyle().setFillColor(java.awt.Color.RED);
      shaft.setSizeXYZ(0.2, 0.2, 3);
      shaft.setXYZ(0, 0, 1.5);
      shaft.getStyle().setResolution(new Resolution(1, 10, 15));
      disk.setSizeXYZ(1.75, 1.75, 0.25);
      disk.setXYZ(0, 0, 2.0);
      disk.getStyle().setResolution(new Resolution(4, 12, 1));
      topGroup.addElement(shaft);
      topGroup.addElement(disk);
      topGroup.setTransformation(rigidBody.getTransformation());
      frame.addElement(base);
      frame.addElement(post);
      frame.addElement(orientation);
      frame.addElement(orientationTrace);
      frame.addElement(topGroup);
   }

   void initialize() {
      // dimension of ellipsoid is inverse to inertia
      double dx = 1/Math.sqrt(rigidBody.I1);
      double dy = 1/Math.sqrt(rigidBody.I2);
      double dz = 1/Math.sqrt(rigidBody.I3);
      // bounding dimension
      double scale = Math.max(Math.max(4*dx, 4*dy), 4*dz);
      frame.setPreferredMinMax(-scale, scale, -scale, scale,
         -scale, scale);
      orientationTrace.clear();
      update();
   }

   void update() {
      Transformation transformation = rigidBody.getTransformation();
      topGroup.setTransformation(transformation);
      double s = 1.5*shaft.getSizeZ();
      double[] vec = topGroup.toSpaceFrame(new double[] {0, 0, 1});
      orientation.setSizeXYZ(s*vec[0], s*vec[1], s*vec[2]);
      orientationTrace.addPoint(s*vec[0], s*vec[1], s*vec[2]);
      frame.render();
   }
}
```

Problem 17.18. Uniform precession

If the angular velocity about the axis of symmetry is large, there are two possible rates of steady precession. These precession rates $\dot{\phi}$ are approximately

$$\dot{\phi} \approx \frac{I_3}{I_s}\frac{w_3}{\cos\theta_0} \tag{17.49a}$$

and

$$\dot{\phi} \approx \frac{mgl}{I_3\omega_3}, \tag{17.49b}$$

where θ_0 is the angle between the vertical and the axis of gyroscope, and mgl is the weight times the distance from the pivot to the center of mass. Demonstrate both precession rates. □

17.9 Projects

Project 17.19. Rotating reference frames

Model the projectile motion of a ball thrown into the air as seen from a rotating platform. Do so by solving the equations of motion in an inertial reference frame and transforming the trajectory to the noninertial rotating frame. □

Project 17.20. Falling box

Do a two-dimensional simulation of a rotating and falling box hitting and rebounding from a floor. If you are bold, do the simulation in three dimensions by adding translational center of mass coordinates to the quaternion state vector. Does the average kinetic energy of translation equal the average energy of rotation over many bounces? □

Appendix 17A: Matrix Transformations

Transformations, such as rotations, can be implemented using matrices, Euler angles, or analytic functions. All that is required is that a transformation provide a rule for mapping points in a domain to points in a range. Some, but not all, transformations have an inverse that reverses this operation. These abstract concepts are used to define the `Transformation` interface in the numerics package. This interface contains the `direct` and `inverse` methods for transforming points. The `clone` method creates a new transformation that is an exact duplicate of the existing transformation. Objects that can make copies of themselves implement the `clone` interface and are said to be *cloneable*.

```
package org.opensourcephysics.numerics;

public interface Transformation extends Cloneable {
   public void direct (double[] point);
   public void inverse (double[] point) throws
      UnsupportedOperationException;
   public Object clone();
}
```

The `Affine3DMatrix` class shown in Listing 17.16 implements the `Transformation` interface. Because rotations are very common, the class implements the `Rotation` convenience method to create a matrix using (17.16). The `direct` and `inverse` methods use the matrix and the inverse matrix to transform a point, respectively. Because a program might not need the inverse (which might not exist) and because an inverse is expensive to compute, we do not compute this matrix until it is needed, in which case the inverse is computed only once. Note that the inverse is calculated using a numerical method known as *lower upper (LU) matrix decomposition* (see Press et al.). The name LU decomposition comes from the realization that a nonsingular square matrix $\mathcal{A}$ can be decomposed into a product of two matrices $\mathcal{L}$ and $\mathcal{U}$ whose components below and above the diagonal are zero, respectively:

$$\mathcal{A} = \mathcal{L}\mathcal{U}. \tag{17.50}$$

We will not describe this technique, but you are encouraged to test that the method in the numerics package works.

Listing 17.16: The `Affine3DMatrix` class implements the `Transformation` interface using a matrix representation of the affine transformations.

```
package org.opensourcephysics.sip.ch17;
import org.opensourcephysics.numerics.*;

public class Affine3DMatrix implements Transformation {
// transformation matrix
   private double[][] matrix = new double[4][4];
   // inverse transformation matrix if it exists
   private double[][] inverse = null;
   // true if inverse has been computed
   private boolean inverted = false;

   public Affine3DMatrix(double[][] matrix) {
      if(matrix==null) { // identity matrix
         this.matrix[0][0] = this.matrix[1][1] = this.matrix[2][2] =
            this.matrix[3][3] = 1;
         return;
      }
      for(int i = 0;i<matrix.length;i++) { // loop over the rows
         System.arraycopy(matrix[i], 0, this.matrix, 0,
            matrix[i].length);
      }
   }

   public static Affine3DMatrix Rotation(double theta, double[] axis) {
      Affine3DMatrix at = new Affine3DMatrix(null);
      double[][] atMatrix = at.matrix;
      double norm = Math.sqrt(axis[0]*axis[0]+axis[1]*axis[1]
         +axis[2]*axis[2]);
      double x = axis[0]/norm, y = axis[1]/norm;
      double z = axis[2]/norm;
      double c = Math.cos(theta), s = Math.sin(theta);
      double t = 1-c;
      // matrix elements not listed are zero
      atMatrix[0][0] = t*x*x+c;
      atMatrix[0][1] = t*x*y-s*z;
      atMatrix[0][2] = t*x*y+s*y;
      atMatrix[1][0] = t*x*y+s*z;
      atMatrix[1][1] = t*y*y+c;
      atMatrix[1][2] = t*y*z-s*x;
      atMatrix[2][0] = t*x*z-s*y;
      atMatrix[2][1] = t*y*z+s*x;
      atMatrix[2][2] = t*z*z+c;
      atMatrix[3][3] = 1;
      return at;
   }

   public static Affine3DMatrix Translation(double dx, double dy, double dz) {
      Affine3DMatrix at = new Affine3DMatrix(null);
      double[][] m = at.matrix;
      // matrix elements not listed are zero
      m[0][0] = 1;
      m[0][3] = dx;
      m[1][1] = 1;
      m[1][3] = dy;
      m[2][2] = 1;
      m[2][3] = dz;
      m[3][3] = 1;
      return at;
   }

   public Object clone() {
      return new Affine3DMatrix(matrix);
   }

   public double[] direct(double[] point) {
      int n = point.length;
      double[] tempPoint = new double[n];
      System.arraycopy(point, 0, tempPoint, 0, n);
      for(int i = 0;i<n;i++) {
         point[i] = 0;
         for(int j = 0;j<n;j++) {
            point[i] += matrix[i][j]*tempPoint[j];
         }
      }
      return point;
   }

   public double[] inverse(double[] point) throws
        UnsupportedOperationException {
      if(!inverted) {
         calcInverse(); // computes inverse using LU decompostion
      }
      if(inverse==null) { // inverse does not exist
         throw new UnsupportedOperationException("inverse matrix does
            not exist.");
      }
      int n = point.length;
      double[] pt = new double[n];
      System.arraycopy(point, 0, pt, 0, n);
      for(int i = 0;i<n;i++) {
         point[i] = 0;
         for(int j = 0;j<n;j++) {
            point[i] += inverse[i][j]*pt[j];
         }
      }
      return point;
   }

   // calculates inverse using Lower-Upper decomposition
   private void calcInverse() {
      LUPDecomposition lupd = new LUPDecomposition(matrix);
      inverse = lupd.inverseMatrixComponents();
      // signal that the inverse computation has been performed
      inverted = true;
   }
}
```

Exercise 17.21. Transformation interface

Write a simple program to test the `Affine3DMatrix` class. Show that the direct and inverse transformations reverse the mappings. □

Appendix 17B: Conversions

Physicists use Euler angles because they are useful for analytically solving Euler's rigid body equations of motion in a small number of special cases. Because the quaternion representation is unfamiliar and is not taught in standard texts, we present conversion formulas between quaternions, rotation matrices, and Euler angles. See Shoemake for a more complete discussion of these conversions.

Quaternion to matrix. For a quaternion $q = (q_0, q_1, q_2, q_3)$ that satisfies the normalization condition $1 = q_0^2 + q_1^2 + q_2^2 + q_3^2$, the rotation matrix is

$$\mathcal{R} = 2\begin{bmatrix} \frac{1}{2} - q_2^2 - q_3^2 & q_1q_2 + q_0q_3 & q_1q_3 - q_0q_2 \\ q_1q_2 - q_0q_3 & \frac{1}{2} - q_1^2 - q_3^2 & q_2q_3 + q_0q_1 \\ q_1q_3 + q_0q_2 & q_2q_3 - q_0q_1 & \frac{1}{2} - q_1^2 - q_2^2 \end{bmatrix}. \tag{17.51}$$

Matrix to quaternion. The quaternion components can be computed using linear combinations of the rotation matrix elements of the (3×3) matrix $\mathcal{R} = [r_{i,j}]_{3\times 3}$. To avoid the pitfall of dividing by a small number ϵ (the machine precision), we compute quaternion components using if statements:

- Compute $w^2 = (1 + r_{0,0} + r_{1,1} + r_{2,2})/4$. If $w^2 > \epsilon$, then

$$q_0 = \sqrt{w^2} \tag{17.52a}$$
$$q_1 = (r_{1,2} - r_{2,1})/4q_0 \tag{17.52b}$$
$$q_2 = (r_{2,0} - r_{0,2})/4q_0 \tag{17.52c}$$
$$q_3 = (r_{0,1} - r_{1,0})/4q_0, \tag{17.52d}$$

  else compute $x^2 = -1/2(r_{1,1} + r_{2,2})$.

- If $x^2 > \epsilon$, then

$$q_0 = 0 \tag{17.53a}$$
$$q_1 = \sqrt{x^2} \tag{17.53b}$$
$$q_2 = r_{0,1}/2q_1 \tag{17.53c}$$
$$q_3 = r_{0,2}/2q_1, \tag{17.53d}$$

  else compute $y^2 = 1/[2(1 - r_{2,2})]$.

- If $y^2 > \epsilon$, then

$$q_0 = 0 \tag{17.54a}$$
$$q_1 = 0 \tag{17.54b}$$
$$q_2 = \sqrt{y^2} \tag{17.54c}$$
$$q_3 = r_{1,2}/2q_2, \tag{17.54d}$$

  else compute

$$q_0 = 0 \tag{17.55a}$$
$$q_1 = 0 \tag{17.55b}$$
$$q_2 = 0 \tag{17.55c}$$
$$q_3 = 1. \tag{17.55d}$$

Euler angles to matrix. Euler angles are generally described in physics texts (see Goldstein) as a group of three rotations about a set of body frame axes. An object is created with the body frame aligned with the space frame. The first rotation is about the body frame's $\hat{3}$-axis by an angle ϕ; the second rotation is about the new x-axis by an angle θ, and the third rotation is about the new z-axis by an angle ϕ. Other definitions of Euler angles are possible. For example, the Java 3D API defines Euler angles as three rotations about a set of axes fixed in space. All possible positions of an object can be represented using either of these conventions.

The first rotation is about the z-axis and is given by

$$\mathcal{A}(\phi) = \begin{bmatrix} \cos\phi & \sin\phi & 0 \\ -\sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{17.56}$$

The second rotation is about the new x-axis and is given by

$$\mathcal{B}(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & \sin\theta \\ 0 & -\sin\theta & \cos\theta \end{bmatrix}. \tag{17.57}$$

The last rotation is again about the new z-axis

$$\mathcal{C}(\psi) = \begin{bmatrix} \cos\psi & \sin\psi & 0 \\ -\sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{17.58}$$

The application of the three Euler rotation matrices $C(\psi)$, $B(\theta)$, $A(\phi)$ in this order produces the transformation

$$\mathcal{R}(\psi,\theta,\phi) = \begin{bmatrix} \cos\psi\cos\phi - \cos\theta\sin\phi\sin\psi & \cos\psi\sin\phi + \cos\theta\cos\phi\sin\psi & \sin\psi\sin\theta \\ -\sin\psi\cos\phi - \cos\theta\sin\phi\cos\psi & -\sin\psi\sin\phi + \cos\theta\cos\phi\cos\psi & \cos\psi\sin\theta \\ \sin\theta\sin\phi & -\sin\theta\cos\phi & \cos\theta \end{bmatrix}. \tag{17.59}$$

Euler angles to quaternion. There are many possible conventions for the Euler angles. We again use the definition found in Goldstein:

$$q_0 = \cos\theta/2\cos\frac{1}{2}(\phi+\psi) \tag{17.60a}$$

$$q_1 = \sin\theta/2\cos\frac{1}{2}(\phi-\psi) \tag{17.60b}$$

$$q_2 = \sin\theta/2\sin\frac{1}{2}(\phi-\psi) \tag{17.60c}$$

$$q_3 = \sin\theta/2\cos\frac{1}{2}(\phi+\psi). \tag{17.60d}$$

Matrix to Euler Angles. The conversion from matrix elements to Euler angles is ill-defined because inverse trigonometric functions do not uniquely specify the resulting quadrant. From (17.59) we see that $\cos\theta = r_{2,2}$. We then use $\sin\theta = \sqrt{1-\cos^2\theta}$ to compute $\sin\theta$ to within a sign. As in the matrix to quaternion conversion, we again use if statements to avoid dividing a number less than the machine precision ϵ:

If $|r_{2,2}| > \epsilon$, then

$$\cos\theta = r_{2,2} \tag{17.61a}$$
$$\sin\theta = \sqrt{1 - \cos^2\theta} \tag{17.61b}$$
$$\cos\phi == r_{1,0}/\sin\theta \tag{17.61c}$$
$$\sin\phi = -r_{2,0}/\sin\theta \tag{17.61d}$$
$$\cos\psi = r_{1,2}/\sin\theta \tag{17.61e}$$
$$\sin\psi = r_{0,2}/\sin\theta, \tag{17.61f}$$

else

$$\cos\theta = 0 \tag{17.62a}$$
$$\sin\theta = 1 \tag{17.62b}$$
$$\cos\phi = r_{1,0} \tag{17.62c}$$
$$\sin\phi = -r_{2,0} \tag{17.62d}$$
$$\cos\psi = 1 \tag{17.62e}$$
$$\sin\psi = 0. \tag{17.62f}$$

Quaternions to Euler Angles. Convert the quaternion to a rotation matrix and then convert the matrix to Euler angles. This conversion is easy if these two conversion algorithms have already been programmed.

References and Suggestions for Further Reading

M. P. Allen and D. J. Tildesley, *Computer Simulation of Liquids* (Oxford University Press, 1987).

Didier H. Besset, *Object-Oriented Implementation of Numerical Methods* (Morgan Kaufmann, 2001).

David H. Eberly, *Game Physics* (Morgan Kaufmann, 2004).

Denis J. Evans, "On the representation of orientation space," Mol. Phys. **34** (2) 317–325 (1977).

R. P. Feynman, *Surely You are Joking, Mr. Feynman!*, W. W. Norton (1985). See pp. 157–158 for a discussion of the motion of the rotating plate. Feynman had fun doing physics.

James D. Foley, Andries van Dam, Steven Feiner, and John Hughes, *Computer Graphics: Principles and Practice*, 2nd ed. (Addison–Wesley, 1990).

Herbert Goldstein, Charles P. Poole, and John L. Safko, *Classical Mechanics*, 3rd ed. (Addison–Wesley, 2002). Chapter 4 discusses the kinematics of rigid body motion.

William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery, *Numerical Recipes*, 2nd ed. (Cambridge University Press, 1992).

Jerry Marion and Stephen Thornton, *Classical Dynamics*, 5th ed. (Brooks/Cole, 2003).

D. C. Rapaport, "Molecular dynamics simulation using quaternions," J. Comp. Phys. **60**, 306–314 (1985).

D. C. Rapaport, *The Art of Molecular Dynamics Simulation*, 2nd ed. (Cambridge University Press, 2004). Chapter 8 discusses the molecular dynamics of rigid molecules.

Philip J. Schneider and David H. Eberly, *Geometric Tools for Computer Graphics* (Morgan Kaufmann, 2003).

Ken Shoemake,"Animating rotation with quaternion curves," ACM Transactions in Graphics **19** (3), 256–276 (1994).

Keith R. Symon, *Mechanics* (Addison–Wesley, 1971).

Slavomir Tuleja, Boris Gazovic, and Alexander Tomori, and Jozef Hanc, "Feynman's wobbling plate," Am. J. Phys. **75**, 240–244 (2007).

John R. Taylor, *Classical Mechanics* (University Science Books, 2005).

James M. Van Verth and Lars M. Bishop, *Essential Mathematics for Games and Interactive Applications* (Morgan Kaufmann, 2004).

Chapter 18

Seeing in Special and General Relativity

We compute how objects appear at relativistic speeds and in the vicinity of a large spherically symmetric mass.

18.1 Special Relativity

How do objects appear at relativistic speeds? The Lorentz–Fitzgerald length contraction in the direction of motion is not the only effect that needs to be considered when determining the apparent shape of an object. A single observer forms an image of an object by collecting light emitted from the entire object. When an observer *sees* the object, the observer does not see its current position nor its true shape but sees each part of the object where it was when the light was emitted. This position is known as the *retarded position*. Because of the finite speed of light, we must calculate when and where along the object's trajectory each light ray originated to determine the image formed on the observer's retina.

The relative velocity of an object with respect to a single observer defines a direction, which we take to be the direction of the x-axis in the observer's frame of reference S. Let S' be the rest frame of the object and v be the velocity of S with respect to S', such that the origins coincide at $t = t' = 0$. The Lorentz transformation connecting S and S' is

$$x' = \frac{1}{\gamma}(x - vt) \tag{18.1a}$$

$$y' = y \tag{18.1b}$$

$$z' = z \tag{18.1c}$$

$$t' = \gamma(t - vx/c^2), \tag{18.1d}$$

where $\gamma = 1/\sqrt{1 - v^2/c^2} = 1/\sqrt{1 - \beta^2}$ and $\beta = v/c$. In the rest frame of the object, the spatial separation between two points on the object is

$$d' = \sqrt{(x_2' - x_1')^2 + (y_2' - y_1')^2 + (z_2' - z_1')^2}. \tag{18.2}$$

In the rest frame of the observer, the separation is

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}, \tag{18.3}$$

where

$$x_2 - x_1 = \gamma(x_2' - x_1') \tag{18.4a}$$
$$y_2 - y_1 = y_2' - y_1' \tag{18.4b}$$
$$z_2 - z_1 = z_2' - z_1'. \tag{18.4c}$$

The change in the x separation is known as the Lorentz-Fitzgerald contraction. If we know the shape of the object in the rest frame, we can determine the shape in the observer's frame by applying an affine transformation (see Chapter 17) that rescales the object's x dimension by γ. Listing 18.1 shows how this transformation is done using a two-dimensional wire frame model of a ring. The ContractedRing class defines a ring of unit radius in the object's rest frame using an array of Point2D objects. Point2D represents a location and can be transformed because it is part of the standard Java 2D API. These points are transformed into the observer's frame and the transformed shape is drawn by connecting the points using line segments.

Listing 18.1: The ContractedRing class implements the Lorentz–Fitzgerald contraction of a ring moving in the x direction.

```java
package org.opensourcephysics.sip.ch18;
import java.awt.*;
import java.awt.geom.*;
import org.opensourcephysics.display.*;

public class ContractedRing implements Drawable {
   double vx = 0, time = 0;
   Point2D[] labPoints, pixPoints;

   public ContractedRing(double x0, double y0, double vx,
      int numberOfPoints) {
      labPoints = new Point2D[numberOfPoints];
      pixPoints = new Point2D[numberOfPoints];
      double
         theta = 0, dtheta = 2*Math.PI/(numberOfPoints-1);
      // unit radius circle
      for(int i = 0;i<numberOfPoints;i++) {
         double x = Math.cos(theta); // x coordinate
         double y = Math.sin(theta); // y coordinate
         labPoints[i] = new Point2D.Double(x, y);
         theta += dtheta;
      }
      this.vx = vx;
      // Lorentz-Fitzgerald contraction
      AffineTransform at =
         AffineTransform.getScaleInstance(Math.sqrt(1-vx*vx), 1);
      at.transform(labPoints, 0, labPoints, 0, labPoints.length);
      // translate to initial position
      at = AffineTransform.getTranslateInstance(x0, y0);
      at.transform(labPoints, 0, labPoints, 0, labPoints.length);
   }

   public void setTime(double t) {
      double dt = t-time;
      // convert position to position at new time
```

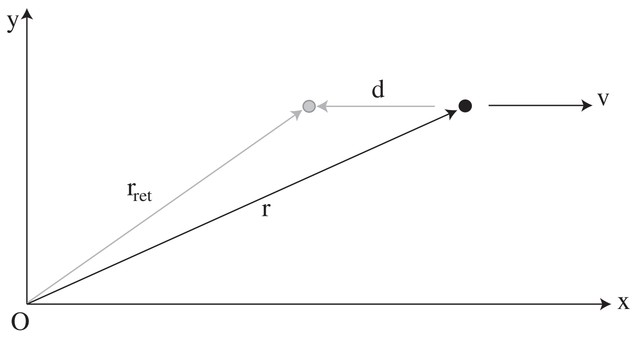


Figure 18.1: The geometry used to derive the retarded position and time for an observer at the origin. The observed point is moving with constant speed v in the x direction.

```
      AffineTransform at = AffineTransform.getTranslateInstance(vx*dt, 0);
      at.transform(labPoints, 0, labPoints, 0, labPoints.length);
      time = t;
   }

   void drawShape(DrawingPanel panel, Graphics2D g2) {
      // convert from lab coordinates to pixels
      AffineTransform at = panel.getPixelTransform();
      at.transform(labPoints, 0, pixPoints, 0, labPoints.length);
      g2.setColor(Color.RED);
      for(int i = 1, n = labPoints.length;i<n;i++) {
         g2.draw(new Line2D.Double(pixPoints[i-1], pixPoints[i]));
      }
   }

   public void draw(DrawingPanel panel, Graphics g) {
      Graphics2D g2 = (Graphics2D) g;
      drawShape(panel, g2);
   }
}
```

Exercise 18.1. Lorentz–Fitzgerald contraction

Write a test program that instantiates and displays a `ContractedRing` object. Measure the dimensions of the on-screen object to verify that the Lorentz–Fitzgerald contraction is computed correctly. □

We now introduce retardation effects. Let $\mathbf{r} = (x, y)$ be the current location of an arbitrary point on the object as shown in Figure 18.1. Because of the finite speed of light, an observer at the origin sees a point moving with a speed v in the x direction not at its current location, but at the previous location

$$\mathbf{r}_{\text{ret}} = (x - \delta, y). \tag{18.5}$$

The x-coordinate is *retarded* by $\delta = v\tau$, where τ is the travel time of light from $\mathbf{r}_{\text{ret}}$ to the observer. The distance from the retarded position to the observer is

$$r_{\text{ret}} = \sqrt{(x - \delta)^2 + y^2}. \tag{18.6}$$

We use the speed of light to convert distance to light travel time ($r_{\text{ret}} = c\tau$), substitute for δ, and obtain

$$c\tau = \sqrt{(x - v\tau)^2 + y^2}. \tag{18.7}$$

If we square both sides and solve for τ, we find

$$\tau = \frac{-x\beta + \sqrt{x^2\beta^2 + (1-\beta^2)(x^2+y^2)}}{c(1-\beta^2)}, \tag{18.8}$$

where we have chosen the positive square root to make the travel time τ positive.

We can incorporate the time delay (18.8) into a program to obtain the image seen by a stationary observer. We subclass `ContractedRing` and add methods to compute and draw the retarded positions of the points on the ring. Retarded points are stored in an array and the retarded positions are computed by solving (18.8). The class `ObservedRing` is shown in Listing 18.2.

Listing 18.2: The `ObservedRing` class models the appearance of a ring traveling in the x direction at relativistic speeds.

```
package org.opensourcephysics.sip.ch18;
import java.awt.*;
import java.awt.geom.*;
import org.opensourcephysics.display.*;

public class ObservedRing extends ContractedRing {
   Point2D[] retardPts;

   public ObservedRing(double x0, double y0, double vx, int numPts) {
      super(x0, y0, vx, numPts); // x would change to numberOfPoints
      retardPts = new Point2D[numPts];
      for(int i = 0;i<numPts;i++) {
         retardPts[i] = new Point2D.Double();
      }
   }

   void setRetardedPts() {
      double oneOverGammaSquared = (1-vx*vx);
      for(int i = 0, n = labPoints.length;i<n;i++) {
         double x = labPoints[i].getX();
         double y = labPoints[i].getY();
         double tau =
            (-vx*x+Math.sqrt(x*x*vx*vx+oneOverGammaSquared*(x*x+y*y)))
                    /oneOverGammaSquared;
         retardPts[i].setLocation(x-vx*tau, y);
      }
   }

   void drawObservedShape(DrawingPanel panel, Graphics2D g2) {
      setRetardedPts();
      // converts from view to pixel coordinates
      AffineTransform at = panel.getPixelTransform();
      at.transform(retardPts, 0, pixPoints, 0, retardPts.length);
      g2.setColor(Color.BLACK);
      for(int i = 1, n = retardPts.length;i<n;i++) {
         g2.draw(new Line2D.Double(pixPoints[i-1], pixPoints[i]));
      }
   }

   public void draw(DrawingPanel panel, Graphics g) {
      Graphics2D g2 = (Graphics2D) g;
      drawShape(panel, g2);
      drawObservedShape(panel, g2);
   }
}
```

Exercise 18.2. Relativistic ring

Write a test program that instantiates and displays the apparent shape of a rapidly moving ring. Explain the sharp convex point when the front edge of the ring touches (reaches) the observer. Does the ring ever appear to be concave? Why?

Exercise 18.3. Relativistic ruler

Modify the `ObservedRing` class to display a long narrow rectangle. Can an observer see the Lorentz–Fitzgerald contraction if this "ruler" is moving along the x-axis? Click-drag within the display to measure the apparent length of the ruler at various positions. Explain the meaning of the term *observer* in relativity. Some authors (see Taylor and Wheeler) prefer the term *bookkeeper*. Why might this term be better? □

What an observer sees is quite different from what is given by the Lorentz contraction. What makes Einstein's special theory of relativity profound is not the appearance, but rather that length really does contract and time really does slow down.

Exercise 18.4. Relativistic square

Write a target class that instantiates and displays the apparent shape of a moving square whose trajectory is $(x_0 - vt, b)$ past a stationary observer at $(0,0)$. Is the apparent shape still a square? Explain why the observer can see the square's hidden side. This effect is known as *Terrell rotation*. □

We can rotate the shape seen in the simulation around the x-axis to visualize the apparent shape of a three-dimensional sphere approaching an observer head-on. This case was treated analytically by Suffern, but most other two- and three-dimensional objects cannot be treated analytically and are best visualized using the help of a computer. A complete and accurate visualization must take into account additional physics such as the Doppler effect, aberration, and angular changes in the intensity distribution of the emitted light (see Weisskopf).

18.2 General Relativity

The idea that space is curved was first tested by Gauss who measured the interior angles of a large triangle by placing lanterns on three mountain tops. Although Gauss obtained the Euclidian (flat-space) result of 180°, measurements of stellar positions during the 1919 total solar eclipse by Eddington showed that the sum of the interior angles is not 180°. It is an experimental fact that the universe is non-Euclidian.

The Eddington experiment was remarkable because it confirmed Einstein's general theory of relativity and showed that space and time are not separate entities. We cannot measure space, only distances between events in space using rulers, light beams, and clocks. Furthermore, the separation between events is not the same for different observers unless they incorporate both spatial and temporal separations into their definition of distance. In the absence of gravitational fields, observers moving at constant relative velocity can reconcile (18.2) and (18.3) and obtain the same "distance" only if they agree that the distance between events $\Delta\sigma$ includes time and is measured as

$$(\Delta\sigma)^2 = (\Delta x)^2 + (\Delta y)^2 + (\Delta z)^2 - c^2(\Delta t)^2, \tag{18.9}$$

where c is the speed of light, Δt is the temporal separation, and Δx, Δy, and Δz are the spatial coordinate separations. Equation (18.9) is based on Einstein's special theory of relativity and is known as the Minkowski metric. It follows from Einstein's assumption that Maxwell's equations must be the same for all observers in uniform relative motion and leads naturally to the equivalence of mass and energy embodied in the famous equation $E = mc^2$.

Einstein's great insight that acceleration and gravity are indistinguishable enabled him to incorporate gravity into the spacetime fabric by generalizing (18.9). Imagine an elevator compartment resting on the surface of Earth in which the occupants perform experiments, such as dropping an object or observing a swinging pendulum, that reveal the presence of Earth's gravity. Then the occupants are placed in a compartment far from any gravitational object, and the compartment accelerates at $9.8\,\mathrm{m/s^2}$. According to Einstein, the experimental results must be identical. Furthermore, if the near-Earth elevator cable is cut to produce a freely falling reference frame, then the occupants will be unable to detect the gravitational field. The implication is that we can do away with gravity and regard it as a consequence of an accelerated reference frame in four-dimensional spacetime. It took Einstein ten years to incorporate this equivalence of gravitational forces and accelerated motion to the special theory of relativity to produce the general theory.

Einstein's general theory of relativity produces ten simultaneous coupled nonlinear partial differential equations. Calculations using this theory are truly daunting and require sophisticated mathematical techniques such as tensor analysis and Riemannian geometry. All forms of energy gravitate (attract), and nonlinearities arise because a body's gravitational field is itself a form of energy and therefore gravitates. Few analytic results are known. Two of the most important are the Schwarzschild and Kerr metrics in the vicinity of a spherically symmetric mass. Except for very special cases or very weak fields, the dynamical equations derived from these metrics and the principle that the path taken by an objects is a maximum as measured by a watch carried with the object (maximum aging) must be solved numerically to predict how particles move and how they appear when seen by an observer.

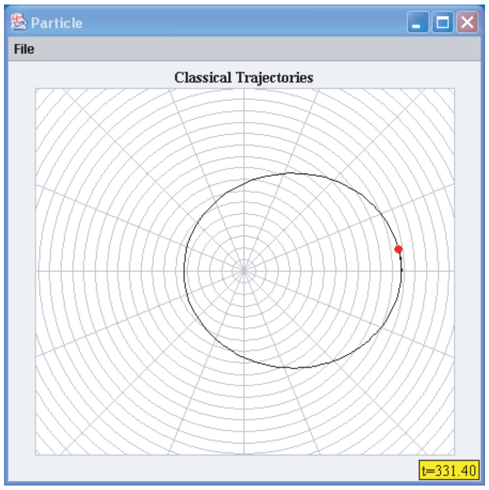
18.3 Dynamics in Polar Coordinates

General relativistic trajectories of particles and light in the vicinity of spherically symmetric gravitational fields are conveniently described using polar coordinates. We therefore reformulate the classical two-body problem (see Chapter 5) using polar coordinates. If the motion is confined to a plane, rectangular coordinates (x, y) and polar coordinates (r, ϕ) are related by

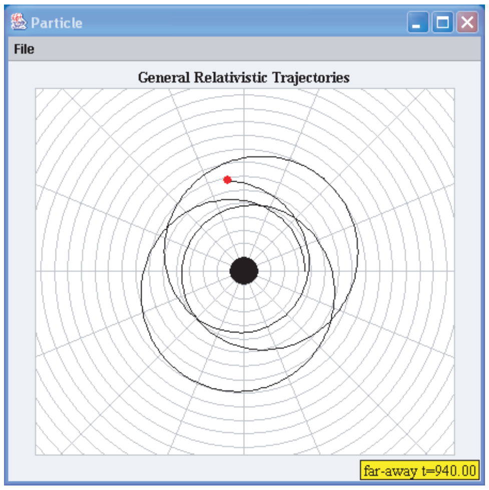
$$x = r\cos\phi, \quad y = r\sin\phi \tag{18.10}$$

and

$$r = \sqrt{x^2 + y^2}, \quad \phi = \arctan\frac{y}{x}. \tag{18.11}$$



(a) Classical



(b) General relativistic

Figure 18.2: Comparison of classical and general relativistic particle trajectories in the vicinity of a spherically symmetric mass.

The radial velocity is given by

$$\dot{r} = \frac{dr}{dt} = \frac{\mathbf{r}\cdot\mathbf{v}}{r}, \tag{18.12}$$

and the angular velocity is given by

$$\dot{\phi} = \frac{d\phi}{dt} = \frac{L}{mr^2}, \tag{18.13}$$

where L is the magnitude of the conserved angular momentum $\mathbf{L} = \mathbf{r} \times \mathbf{p}$.

To construct the appropriate differential equations, the radial and angular accelerations can be obtained by differentiating (18.12) and (18.13) with respect to time. Another approach is to use the Lagrangian

$$\mathcal{L} = \frac{1}{2}\left[\dot{r}^2 + r^2\dot{\phi}^2\right] + \frac{GM}{r}, \tag{18.14}$$

and apply Lagrange's equations of motion:

$$\frac{d}{dt}\frac{\partial\mathcal{L}}{\partial\dot{r}} - \frac{\partial\mathcal{L}}{\partial r} = 0, \quad \frac{d}{dt}\frac{\partial\mathcal{L}}{\partial\dot{\phi}} - \frac{\partial\mathcal{L}}{\partial\phi} = 0. \tag{18.15}$$

If we do the differentiation, we obtain the following rate equations for the polar state vector

$(r, \dot{r}, \phi : \dot{\phi}, t)$.

$$\frac{dr}{dt} = \dot{r} \tag{18.16a}$$

$$\frac{d\dot{r}}{dt} = r\dot{\phi}^2 - \frac{GM}{r^2} \tag{18.16b}$$

$$\frac{d\phi}{dt} = \dot{\phi} \tag{18.16c}$$

$$\frac{d\dot{\phi}}{dt} = -\frac{2}{r}\dot{\phi}\dot{r} \tag{18.16d}$$

$$\frac{dt}{dt} = 1. \tag{18.16e}$$

Exercise 18.5. Angular momentum

Show that (18.16) leads to the polar coordinate expression for conservation of angular momentum

$$r\ddot{\phi} + 2\dot{\phi}\dot{r} = 0. \tag{18.17}$$

□

Exercise 18.6. Classical trajectories

Modify the PlanetApp program introduced in Chapter 5 so that the classical trajectory of a particle is calculated using polar variables rather than cartesian variables. Use (18.10)–(18.13) to determine the initial state and compare your results with those of the PlanetApp program. □

The Open Source Physics plotting panel contains an axis object that displays a cartesian coordinate grid by default. This grid can be replaced by a polar grid by using the method setPolar (see Figure 18.2):

```
plotFrame.setPolar("Trajectory",1.0);
```

The first parameter is the plot's title and the second parameter is the radial grid separation. The new plotting panel also displays the polar coordinates in the bottom left when the mouse is clicked or dragged.

Exercise 18.7. Polar coordinates

Modify Exercise 18.6 so that polar coordinate values are displayed when the mouse is click-dragged within the display. □

18.4 Black Holes and Schwarzschild Coordinates

Our goal is to compute the worldlines (trajectories in spacetime) of particles and light in the vicinity of spherically symmetric gravitational objects. Readers should consult the classic text *Exploring Black Holes* by Edwin Taylor and John Wheeler for a more complete discussion of the physics near objects that have undergone gravitational collapse. Such an object is known as a *black hole* because light cannot escape from its vicinity. We will calculate the general relativistic trajectories of particles and light near a spherically symmetric gravitational mass. Because physical space is non-Euclidian, a two-dimensional plot of these trajectories will be distorted. Unlike classical orbits, the general relativistic orbits appear very different when seen by a viewer in the real world. We must calculate the trajectories of multiple light rays to construct the view as seen by a single observer.

Because time is incorporated as a fourth dimension and because space is curved, a general relativistic coordinate system centered on a spherically symmetric mass is more complicated than a three-dimensional Euclidean coordinate system. The azimuthal angle ϕ can still be defined as the ratio of the arc length to the circumference on an imaginary circle because the spherically symmetric gravitational mass M is located at the origin. However, the radial coordinate is not defined as the physical distance from the center. Rather, it is calculated using a path that circumnavigates the central mass:

$$r = \text{circumference}/(2\pi). \tag{18.18}$$

The time coordinate is defined using a wristwatch located far from the center of attraction. Note the *nonlocal* character of the (r, ϕ, t) spacetime coordinates. The wristwatch worn by the surveyor circumnavigating the mass used to measure r is not the time used to record events at that value of r.

This (r, ϕ, t) spacetime coordinate system is known as *Schwarzschild coordinates* and is a universal bookkeeping device that enables us to translate observations from one reference frame to another. The Schwarzschild coordinates give rise to a metric, known as the Schwarzschild metric, that enables us to calculate the four-dimensional distance between adjacent spacetime events. This metric is given by

$$d\sigma^2 = -d\tau^2 = -\left(1 - \frac{2M}{r}\right) dt^2 + \left(1 - \frac{2M}{r}\right)^{-1} dr^2 + r^2\, d\phi^2, \tag{18.19}$$

where t, r, and ϕ refer to the faraway time, the radial coordinate, and the azimuthal coordinate, respectively. Because we associate distance with a positive number, it is common to use $d\sigma^2$ when the right-hand side of (18.19) is positive and to use $d\tau^2$ when the right-hand side of (18.19) is negative. As in special relativity, these two forms are referred to as the space-like form and the time-like form of the metric, respectively. Note that both time and distance have units of length in (18.19). The speed of light c is the conversion factor

$$t_{\text{meters}} = c t_{\text{seconds}}. \tag{18.20}$$

Mass also has units of meters, and the conversion factor to kilograms is given in terms of the speed of light and Newton's gravitational constant, G:

$$M = \frac{G}{c^2} M_{\text{kg}}. \tag{18.21}$$

If we freeze time so that $dt = 0$, then the Schwarzschild metric predicts that two simultaneous events far from the central mass are separated by the Euclidian metric in polar coordinates,

$$d\sigma^2 = dr^2 + r^2\, d\phi^2. \tag{18.22}$$

Strange things happen if two events are near the gravitational mass. The separation (known as the proper distance) between two adjacent events becomes

$$d\sigma^2 = \left(1 - \frac{2M}{r}\right)^{-1} dr^2 + r^2\, d\phi^2. \tag{18.23}$$

The proper distance $d\sigma$ is the distance measured by a surveyor placing meter sticks in space between two locations. This distance is clearly greater than the result predicted by (18.22) when the events occur at different r-coordinates. In fact, the rate of change of the proper length with respect to the r-coordinate becomes infinite as we approach what is known as the event horizon $r = 2M$. The distance around a gravitational mass has no such singularity, which is why this distance is used to define the r-coordinate. (The singularity at the event horizon is an artifact of the Schwarzschild coordinate system. An object falling into a black hole passes through the event horizon without incident and is crushed only at $r = 0$.)

Exercise 18.8. Measuring distance

Although the rate of change of distance with respect to r becomes infinite, the distance from a point outside the event horizon to the event horizon is finite. Write a test program that integrates $d\sigma$ from a point $r = a$ outside the event horizon to a point arbitrarily close to the event horizon. What is the distance from $r = 4$ to $r = 2$ when $M = 1$? □

Exercise 18.9. Event horizon

a) Although general relativity predicts the shape of orbits around any spherically symmetric gravitational object, not all objects have an event horizon. (The objects that do are black holes.) The event horizon assumes that the mass of the entire object is within the horizon, which implies very high mass densities. Calculate the r-coordinate of the event horizon for an object having the mass of the Earth and compare it to the radius of the Earth. Repeat the calculation for the sun.

b) The event horizon for the black hole believed to exist at the center of our galaxy has an event horizon of $r = 7.6 \times 10^9$ m. What is its mass in units of our own Sun (solar mass)? □

The variable t in the Schwarzschild metric is the time as measured by a faraway observer. Time as measured by a local observer is known as the proper time, τ. Observers experience time as measured by their wristwatches and (18.19) shows that the wristwatch time interval $d\tau$ depends on location. A faraway observer who measures the time between two light flashes records a value of dt, while an observer standing next to these flashes measures a time interval $d\tau$ given by

$$d\tau^2 = \left(1 - \frac{2M}{r}\right) dt^2. \tag{18.24}$$

Proper time intervals near a gravitational mass are clearly smaller than faraway time intervals. This result gives rise to the gravitational red shift when applied to light.

Exercise 18.10. Local and faraway time

Estimate the difference due to gravitational effects between local time and faraway time during one hour for an observer on Earth. Does special relativity play a role in an actual measurement? □

18.5 Particle and Light Trajectories

The physics describing the trajectory of a particle in the vicinity of a gravitational mass can be formulated using the principle of stationary aging (see Hanc). This principle states that a particle takes a path through spacetime such that the elapsed time $\delta\tau$ recorded by the wristwatch attached to the particle is a maximum. (In general, $\delta\tau$ is an extrenum so it could also be a minimum or a saddle point.) Because Lagrangian dynamics is based on the principle that the integral of the Lagrangian over time (called the action) also is stationary, we construct a Lagrangian using the Schwarzschild metric:

$$\mathcal{L}(r,\dot{r},\phi,\dot{\phi}) = \left[\left(1-\frac{2M}{r}\right) - \left(1-\frac{2M}{r}\right)^{-1}\dot{r}^2 - r^2\dot{\phi}^2\right]^{1/2}, \tag{18.25}$$

such that

$$\tau = \int_{\text{initial event}}^{\text{final event}} d\tau = \int_{\text{initial event}}^{\text{final event}} \mathcal{L}(r,\dot{r},\phi,\dot{\phi})\,dt. \tag{18.26}$$

Because the Lagrangian in (18.25) satisfies (18.15), we can take the required derivatives and simplify terms to obtain the following system of first-order differential equations:

$$\frac{dr}{dt} = \dot{r} \tag{18.27a}$$

$$\frac{d\dot{r}}{dt} = \frac{4M^3 - 4M^2 r - 4M^2 r^3\dot{\phi}^2 + 4Mr^4\dot{\phi}^2 - r^5\dot{\phi}^2 + r^2(M - 3M\dot{r}^2)}{(2M-r)r^3} \tag{18.27b}$$

$$\frac{d\phi}{dt} = \dot{\phi} \tag{18.27c}$$

$$\frac{d\dot{\phi}}{dt} = \frac{2(-3M+r)\dot{r}\dot{\phi}}{(2M-r)r} \tag{18.27d}$$

$$\frac{dt}{dt} = 1. \tag{18.27e}$$

Note that the independent variable in (18.27) is faraway time. The metric provides an additional differential equation if we wish to track the particle's proper time τ:

$$\frac{d\tau}{dt} = \left[\left(1-\frac{2M}{r}\right) - \left(1-\frac{2M}{r}\right)^{-1}\dot{r}^2 - r^2\dot{\phi}^2\right]^{1/2}. \tag{18.28}$$

Exercise 18.11. General relativistic trajectories

(a) Write a program that plots the general relativistic trajectory of a particle using Schwarzschild coordinates. Verify that circular orbits are obtained for $v = \sqrt{M/r}$ for $r \geq 6M$.

(b) Show that there are no stable circular orbits for $r < 6M$.

(c) Add the differential equation for proper time. What is the proper time for one complete orbit at $r = 6$? This interval is the orbital period as measured by an observer traveling with the particle. Compare this wristwatch orbital period to the faraway orbital period and to the time interval predicted by (18.24). Explain any discrepancies in your numerical values.

(d) Perturb the circular orbit at $r = 9$ by giving the particle an initial tangential velocity of $v = 0.345c$. At what rate does the perihelion of the orbit advance? □

The equations for light can be obtained by adding a constraint to (18.25) using a Lagrange multiplier. The constraint is the condition that the proper time along a light worldline is zero:

$$0 = -\left(1-\frac{2M}{r}\right)dt^2 + \left(1-\frac{2M}{r}\right)^{-1}dr^2 + r^2\,d\phi^2. \tag{18.29}$$

If we add the Lagrange multiplier, do the differentiation, and simplify terms, we obtain rate equations that can be solved using standard numerical techniques. (The use of a computer algebra program would be helpful.)

$$\frac{dr}{dt} = \dot{r} \tag{18.30a}$$
$$\frac{d\dot{r}}{dt} = \frac{-4M^2 + 2Mr + (r-5M)r^3\dot{\phi}^2}{r^3} \tag{18.30b}$$
$$\frac{d\phi}{dt} = \dot{\phi} \tag{18.30c}$$
$$\frac{d\dot{\phi}}{dt} = \frac{2(-3M+r)\dot{r}\dot{\phi}}{(2M-r)r} \tag{18.30d}$$
$$\frac{dt}{dt} = 1. \tag{18.30e}$$

Exercise 18.12. Light trajectories

a) Write a program that plots the general relativistic trajectory of light using Schwarzschild coordinates. Demonstrate the deflection of starlight passing near a gravitational mass by plotting the trajectory of a light ray.

b) Verify that light orbits a black hole at $r = 3$ and $M = 1$.

c) Show that a gravitational mass can act as a lens by plotting the trajectory of two light rays that leave a point source at different angles but later cross. The two light rays should pass on opposite sides of the mass. Do the two light beams always arrive at the crossing point at the same time? □

18.6 Seeing

Because of the nonlinearity of the Schwarzschild metric, simulation plays an essential role. A calculation of a view of the stars in the vicinity of a black hole, for example, would require the solution of the light-ray trajectory for angles within the eye's field of view.

The angles drawn on a Schwarzschild map are not the same as the angles seen by an observer because distances on the map are distorted. A stationary observer at a constant r-value is known as a *shell observer* because he is on a stationary shell at fixed (r, ϕ) coordinates. Launch angles measured by such a shell observer can easily be converted to angles on the Schwarzschild map by taking into account the contraction by $\sqrt{1-2M/r}$ in the radial direction:

$$\tan\theta_{\text{shell}} = \left(1 - \frac{2M}{r}\right)^{1/2} \tan\theta_{\text{schw}}. \tag{18.31}$$

Use this transformation in Exercise 18.13 and Project 18.20.

Exercise 18.13. Knife-edge trajectory

Many important properties of light rays can be expressed in terms of an impact parameter, b defined as

$$b = r\left(1 - \frac{2M}{r}\right)^{-1/2} \sin\theta_{\text{shell}}. \tag{18.32}$$

For example, light that is launched with $b = \sqrt{27}M$ enters an unstable orbit that teeters between an escape to infinity and a plunge into the black hole. This trajectory is known as a *knife-edge* trajectory because the result is very sensitive to the initial conditions and numerical roundoff error and cannot be predicted. What will a shell observer see if he looks into space at an angle that has this impact parameter? □

Problem 18.14. Seeing near a black hole

Imagine a grid of light beacons located far away from a black hole in the $\phi = \pi$ direction. A shell observer at $\phi = 0$ with an arbitrary value of r attempts to view the grid by looking toward the black hole. What will the observer see? One way to answer this question is to assume a reasonable field of view (for example, 180°) for the eye and calculate light rays leaving the eye at equal angular intervals. Compute the light paths and tabulate where the ray crosses the beacon grid as a function of viewing angle. Because it is unlikely that the light rays will intersect a beacon location, use interpolation to determine the angles at which beacons appear. Plot these locations to show the observer's view. □

18.7 General Relativistic Dynamics

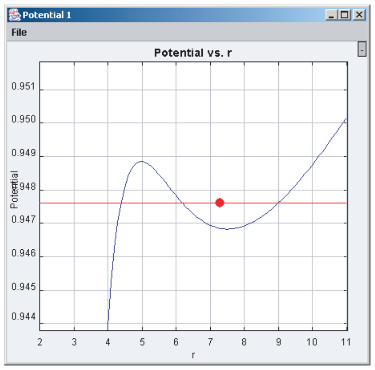


Figure 18.3: The effective potential $V(r)$ of a particle in the vicinity of a black hole.

In general relativity the magnitude of the angular momentum L per unit mass m of a particle is

$$\ell \equiv \frac{L}{m} = r^2 \frac{d\phi}{d\tau}, \tag{18.33}$$

and the energy E per unit mass is

$$e \equiv \frac{E}{m} = \left(1 - \frac{2M}{r}\right)\frac{dt}{d\tau}. \tag{18.34}$$

We can solve (18.33) for $d\phi$ and (18.34) for dt and substitute the result into the time-like form of the metric and obtain a relation for $dr/d\tau$:

$$\left(\frac{dr}{d\tau}\right)^2 = e^2 - \left(1 - \frac{2M}{r}\right)\left[1 + \left(\frac{\ell}{r}\right)^2\right]. \tag{18.35}$$

In analogy with the classical effective potential function for a particle in a gravitational field, we use (18.35) to define a *relativistic effective potential* (see Figure 18.3):

$$\left(\frac{V(r)}{m}\right)^2 = \left(1-\frac{2M}{r}\right)\left[1+\left(\frac{\ell}{r}\right)^2\right]. \tag{18.36}$$

Exercise 18.15. Energy and angular momentum

Show that the energy and angular momentum are conserved for the orbits you observed in Exercise 18.11. □

Exercise 18.16. Effective potential

Add a plot of the effective potential $V(r)$ to your program for Exercise 18.11. Add a horizontal line showing the energy per unit mass and place a red marker on this line showing the particle's radial position. Describe the effective potential and the motion of the marker when the orbit is circular, when the orbit precesses, and when the orbit plunges toward the event horizon. □

18.8 *The Kerr Metric

Because almost all astronomical objects rotate, most black holes likely have angular momentum. The metric for a spinning black hole was derived by Kerr in 1964. For simplicity, we show the metric for particle motion in the equatorial plane. Note that this metric contains a new angular momentum parameter a:

$$d\tau^2 = \left(1-\frac{2M}{r}\right)dt^2 + \frac{4Ma^2}{r}dt\,d\phi - \left(1-\frac{2M}{r}+\frac{a^2}{r^2}\right)^{-1}dr^2 - \left(1+\frac{a^2}{r^2}+\frac{2Ma^2}{r^3}\right)r^2\,d\phi^2. \tag{18.37}$$

Because there are two values at which the coefficient of dr^2 increases without limit, $r_h = M \pm \sqrt{M^2-a^2}$, there are two horizons. We also see that the largest real value of a consistent with real values of r_h is $a = M$. This maximum value of a limits the angular momentum of a black hole. Because we are interested in maximizing the effect of rotation, we simplify (18.37) by letting the angular momentum parameter take on its maximum value. The metric for this extreme Kerr black hole is

$$d\tau^2 = \left(1-\frac{2M}{r}\right)dt^2 + \frac{4Ma}{r}dt\,d\phi - \left(1-\frac{M}{r}\right)^{-2}dr^2 - R^2 d\phi^2, \tag{18.38}$$

where

$$R^2 \equiv r^2 + M^2 + \frac{2M^3}{r}. \tag{18.39}$$

We recast this metric as a Lagrangian and follow the derivation by Hanc and Tuleja and obtain the rate

$$\frac{dr}{dt} = \dot{r} \tag{18.40a}$$

$$\frac{d\dot{r}}{dt} = -\frac{(M-r)^2(M-2M^2\dot{\phi}+M^3\dot{\phi}^2-r^3\dot{\phi}^2)}{r^4} + \frac{2M^3-2M^4\dot{\phi}+3Mr^2-M^2r(1+6r\dot{\phi})}{r^2(M-r)^2}\dot{r}^2 \tag{18.40b}$$

$$\frac{d\phi}{dt} = \dot{\phi} \tag{18.40c}$$

$$\frac{d\dot{\phi}}{dt} = \frac{4M^3\dot{\phi}-2M^4\dot{\phi}^2+6Mr^2\dot{\phi}-2r^3\dot{\phi}-2M^2(1+3r^2\dot{\phi}^2)}{r^2(M-r)^2}\dot{r} \tag{18.40d}$$

$$\frac{dt}{dt} = 1. \tag{18.40e}$$

Problem 18.17. Falling into a spinning black hole

a) Write a program that plots the general relativistic trajectory of a particle near an extreme black hole using (18.40).

b) Follow the trajectory of a particle that starts from rest far from the center of the extreme black hole. Describe the trajectory.

c) A particle is thrown with an angular momentum opposite to the hole's spin starting at $r = 3M$. Write a program to simulate this situation and describe the particle's motion. □

A space ship near a black hole must fire its rockets radially to keep from falling into a black hole. It has an angular momentum appropriate for that radius, so that the remote stars do not move overhead and, therefore, does not fire its rockets tangentially. However, if the space ship moves inward, it must fire its rockets tangentially or it will be swept sideways with respect to the remote stars. (The ship must only fire its rockets while moving inward.) This effect, known as *frame dragging*, occurs near any spinning gravitational object including Earth.

Problem 18.17 shows that frame dragging becomes dramatic as the falling particle approaches the horizon for the extreme black hole, $r_h = M$. (The horizon is where the metric coefficient of dr^2 becomes infinite.) Note that the coefficient of the dt^2 term goes to zero at $r = r_s = 2M$. This value is called the *static limit*. The space between the static limit and the horizon is dragged along in the direction of rotation of the black hole so that an observer cannot remain at a fixed angle no matter how powerful the rockets are.

18.9 Projects

Numerical relativity is still in its infancy but is making progress in simulating astrophysical scenarios such as binary black hole mergers, binary neutron star mergers, and supernova core collapse. A key problem is achieving long-time stability of the numerical solutions. A search on "numerical relativity" will yield many interesting Web sites and entrees to current research.

Project 18.18. Three-dimensional rapidly moving objects

Extend the analysis in Section 18.1 to three-dimensional objects and model their appearance as seen by a single observer at the origin using the transformation and rendering techniques described in Chapter 17. Does a sphere appear to be a sphere even when it passes by an observer? Does a cube appear to be a cube? □

Project 18.19. Light Links

a) Imagine two stationary observers near a black hole wishing to establish a communication link using a laser beam. In what direction should the laser be pointed to establish such a link? Simulate this scenario using two dragable objects on a Schwarzschild map and draw the light ray representing the communication link. Use a root finding algorithm, such as the bisection method introduced in Chapter 6, to determine the proper launch angle. Calculate and display the proper distance along this light path and study how this distance changes as the light path grazes the event horizon.

b) Construct a light triangle connecting three observers. Display the sum of the interior angles as measured by the observers to simulate Gauss's mountain top experiment. □

Project 18.20. Seeing orbits

Viewing an orbit requires that we calculate the particle's trajectory and the trajectory of the light ray from the particle to the viewer. An added complication arises because the light reaching the view is retarded by the travel time. Write a program that shows an orbiting particle as seen by a stationary observer in the equatorial plane by keeping track of both particle and light-link parameters. □

References and Suggestions for Further Reading

Robert J. Deissler, "The appearance, apparent speed, and removal of optical effects for relativistically moving objects," Am. J. Phys. **73** (7), 663–669 (2005).

Jozef Hanc and Edwin Taylor, "From conservation of energy to the principle of least action: A story line," Am. J. Phys. **72** (4), 514–521 (2004).

Charles W. Misner, Kip S. Thorn, and John A. Wheeler, *Gravitation* (W. H. Freeman, 1973).

Kevin G. Suffern, "The apparent shape of a rapidly moving sphere," Am. J. Phys. **56** (8), 729–733 (1988).

James Terrell, " Invisibility of the Lorentz contraction," Phys. Rev. **116**, 1041–1045 (1959). This paper corrected the erroneous belief that had been taught for fifty years that an observer sees the Lorentz contraction when viewing a relativistically moving object.

Kip S. Thorne, *Black Holes and Time Warps* (W. M. Norton and Company, 1994).

Victor Weisskopf, "The visual appearance of rapidly moving objects," Physics Today **13** (9), 24–27 (1960).

Most general relativity texts begin with a treatment of tensor analysis. The following two texts present this material using the four-dimensional spacetime metric.

Edwin F. Taylor and John A. Wheeler, *Exploring Black Holes: An Introduction to General Relativity* (Addison–Wesley Longman, 2000).

James Hartle, *Gravity: An Introduction to General Relativity* (Addison–Wesley, 2003).

The website, `<archive.ncsa.uiuc.edu/Cyberia/NumRel/NumRelHome.html>`, developed by the National Center for Supercomputing Applications, is one of many that discusses Einstein's contributions and recent progress in numerical relativity.

Chapter 19

Epilogue: The Unity of Physics

We emphasize that the methods we have discussed can be applied to a wide variety of natural phenomena and contexts.

19.1 The Unity of Physics

Although we have discussed many topics and applications, we have covered only a small fraction of the possible computer simulations and models of natural phenomena. However, we know that the same algorithms can be applied to many kinds of phenomena. For example, the Monte Carlo methods that we applied to the simulation of classical liquids and to the analysis of quantum mechanical wave functions have been applied to the transport of neutrons and problems in chemical kinetics. Similar Monte Carlo methods are being used to analyze problems in quark confinement. Indeed, the increasing role of the computer in research is strengthening the interconnections of the various subfields of physics and the relation of physics to other disciplines.

We have also emphasized that the computer has helped us think of natural phenomena in new ways that complement traditional methods. For example, consider a predator-prey model of the dynamics of fish (minnows) and sharks. Assume that the birth rate of the fish is independent of the number of sharks, and that each shark kills a number of fish proportional to their number. If we assume that $F(t)$, the number of fish at time t, changes continuously, we can write

$$\frac{dF(t)}{dt} = [b_1 - d_1 S(t)]F(t), \tag{19.1}$$

where $S(t)$ is the number of sharks at time t, and b_1 and d_1 are parameters independent of F and S. To obtain an equation for the rate of change of the number of sharks, we assume that the number of offspring produced by each shark is proportional to the number of fish eaten by the shark. If we also assume that the death rate of the sharks is constant, we can write

$$\frac{dS(t)}{dt} = [b_2 F(t) - d_2]S(t). \tag{19.2}$$

Equations (19.1) and (19.2) are known as the *Lotka–Volterra* equations. They can be analyzed by standard methods and solved numerically using simple algorithms. Why is the dynamical behavior of (19.1) and (19.2) cyclic?

In the Lotka–Volterra model the numbers of predator and prey are assumed to change continuously and their spatial distribution is ignored. We now summarize an alternative model that can be simply expressed as a computer algorithm. The model is a two-dimensional cellular automaton known as *Wa-Tor*.

1. Fish and sharks are placed at random on the sites of a lattice with the desired concentrations. The fish and sharks are assigned random ages.
2. At time (iteration) t, consider each fish sequentially. Determine the number of nearest neighbor sites that are unoccupied at time $t-1$ and move the fish at random to one of the unoccupied sites. If all the nearest neighbor sites are occupied, the fish does not move.
3. If a fish has survived for a time that is equal to a multiple of `fbreed`, the fish has a single offspring. The new fish is placed at the previous position of the parent fish.
4. At time t, consider each shark sequentially. If one or more of the nearest neighbor sites at time $t-1$ is occupied by a fish, the shark moves at random to one of the occupied sites and eats the fish. If not, the shark moves to one of the unoccupied sites at random.
5. If a shark moves `nstarve` times without eating, the shark dies. If a shark survives for a multiple of `sbreed` iterations, the shark has a single offspring. The new shark is placed at the previous position of the parent shark.

What is the dynamical behavior of Wa-Tor? Do the Wa-Tor and the Lotka–Volterra equations exhibit similar behavior? Is the Wa-Tor model more realistic than the Lotka–Volterra equations? Which approach would be easier to explain to a nonexpert? Which approach is more flexible? See the references for suggestions for the numerical values of the parameters.

19.2 Spiral Galaxies

In addition to making it easier to investigate complex nonlinear problems and more realistic systems, the computer has reinforced one of the contemporary themes in physics, the unifying role of collective behavior. Systems composed of many individual constituents can exhibit common properties under certain conditions, even though there might be differences in the nature of the constituents and in their mutual interaction. The behavior of a system near a critical point is probably the best example of collective behavior in a familiar context. In the following, we discuss an example of collective behavior in the context of the structure of spiral galaxies.

The internal structure of a galaxy has traditionally been studied using Newtonian dynamics. This point of view is very useful but is complemented by thinking about the large scale structure of a galaxy using ideas from statistical mechanics. Because we only briefly summarize this alternative point of view here, we encourage you to explore the properties of the percolation-based model of Schulman and Seiden by running `GalaxyApp`, a simple version of their model, which can be downloaded from the ch19 directory.

The basic assumption of the model is that even though a region of the galaxy might have the necessary ingredients for star formation, nothing happens if it is left alone. However, if a shock wave from a supernova passes through the gas, there is a good chance that a star will be formed. The supernova is itself the result of an earlier nearby star formation. The theory of *self-propagating star formation* is based on the importance of this mechanism. Rather than determining which regions have the necessary conditions for star formation, we summarize

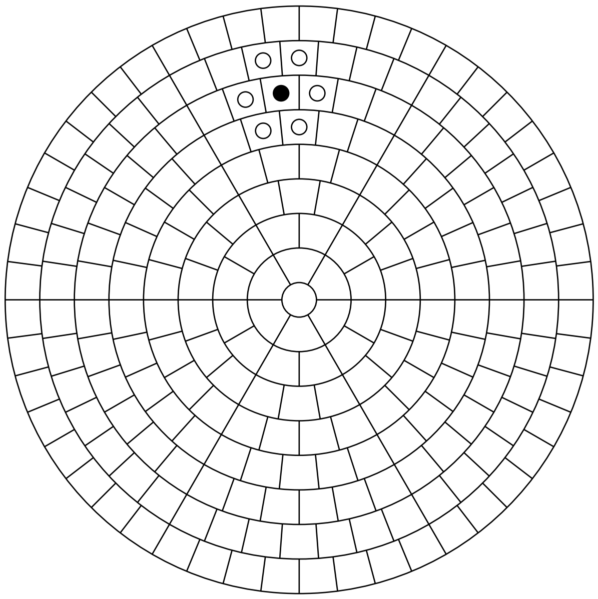


Figure 19.1: The nature of the polar grid used in `GalaxyApp`. Each cell has the same area and six nearest neighbors on the average. The filled circle denotes an active region of star formation. At the next time step, it can induce star formation in cells containing open circles. As time passes, the neighbors in adjacent rings change because of differential rotation.

all the uncertainty and variability in a single parameter p, the probability that a supernova explosion in one region gives rise to star formation in a neighboring region.

The other important observation we need to make about spiral galaxies is that galaxies do not rotate rigidly (with a constant angular velocity), but to a good approximation each region rotates with the same tangential velocity. The properties of random self-propagating star formation and constant tangential velocity are incorporated into `GalaxyApp` as follows. Imagine dividing a galaxy into concentric rings which are divided into cells of equal size (see Figure 19.1). Initially, a small number of cells are activated. Each cell corresponds to a region of space that is the size of a giant molecular cloud and moves with the same tangential velocity v. The angular velocity is given by $\omega = v/r$, where r is the distance of the ring from the center of the galaxy. At each time step, the active cells activate neighboring cells with probability p and then become inactive. Then the rings are rotated, and the process is repeated again in the next time step. At each time step, cells that have been active within the last 15 time steps are displayed as filled boxes, with the size of each box inversely proportional to the time since the cell become active. More details of the simulation are shown in Figure 19.1 and in the program. A typical galaxy generated by `GalaxyApp` is shown in Figure 19.2.

Our brief discussion of galaxies is not meant to convince you that the mechanism proposed by Schulman and Seiden is correct. Rather our purpose is to show how an alternative point of view can suggest new approaches in different fields. The images produced by computer simulations of the galaxy model show unanticipated features and have been the impetus for further studies by astrophysicists and astronomers.

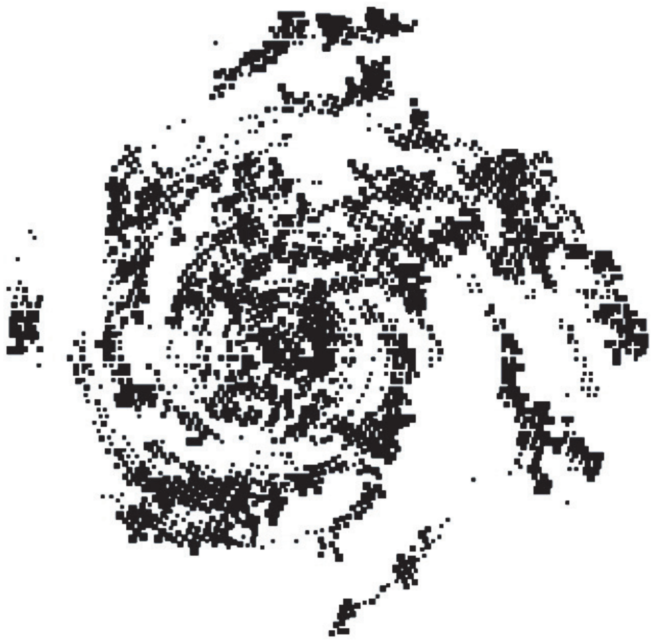


Figure 19.2: A typical structure generated by `GalaxyApp`. The parameters are the number of rings, `nring` = 50, the initial number of active cells, `nactive` = 200, the circular velocity v = 1 (200 km/s), the probability of induced star formation $p = 0.18$, and the time interval `dt` = 10 (10^7 years). The structure shown is at $t = 2720$ with 393 active star clusters. The diameter of the circle representing a star cluster is proportional to the remaining lifetime of the cluster.

19.3 Numbers, Pretty Pictures, and Insight

The power of physics comes in part from its ability to give numerical agreement between theory and experiment. However, numerical agreement has little significance unless this agreement leads to insight into the phenomena of interest. For example, it is possible to design elaborate epicycle models of planetary motion that yield numerical results which are consistent with observations. Nevertheless, we prefer the Copernican approach, not for its impressive numerical success, but because it provided insight and lead to further advances by Kepler and Newton.

Computer simulations raise similar questions. The numbers produced by simulations, which are consistent with experimental observations, and the pictures that are suggestive of physical phenomena are not sufficient to establish the value of a simulation. As an example, let us briefly consider a simulation of river networks. You might have seen aerial photographs of the Earth's topography and the fractal-like drainage patterns formed by many rivers. A variety of random walk models can generate patterns that look remarkably like river networks and even share some of their statistical properties. In these models the path of a walker represents a river, and the branching and intersections of rivers are modeled by the intersection of the paths of many walkers. However, because models do not directly incorporate the important physical processes of erosion and sedimentation, they do not provide much insight.

Leheny has proposed a lattice model whose dynamics reflect actual physical processes. The model consists of first creating a terrain for the network and then defining the network on the terrain. The model can be summarized as follows:

1. The initial terrain is assumed to have a constant slope m. Each site of the lattice is given an initial height, $h(x,y) = my$.
2. Precipitation is placed on a random site of the (square) lattice.
3. Water flows from this site to a nearest neighbor site with a probability proportional to $e^{E\Delta h}$, where Δh is the difference in height between the site and a neighbor, and E is a parameter. If $\Delta h < 0$, then the probability is equal to zero, and the flow will not return to the site previously visited if there is a nonzero probability of flowing to another site.
4. Step 3 is repeated until the water flows to the bottom of the lattice, $y = 0$.
5. Each lattice point that has been visited by the flowing water has its height reduced by a constant amount D. This process represents erosion.
6. Any site at which the height difference Δh with a neighbor exceeds a critical amount M is reduced in height by an amount $\Delta h/S$, where S is another parameter in the model. This process represents mud slides.
7. Steps 2–6 are repeated until you wish to analyze the resulting network. The river network is defined as follows. Every site in the lattice receives one unit of precipitation. Then water flows from a site to the nearest neighbor with the smallest height. Then the water flows to the neighbor of this new site with the smallest height. This flow continues until the flow reaches the bottom of the lattice. This process is repeated for each site, and the number of times that a site receives water is recorded. The river network is defined as the set of all sites that received at least R units of water, where R is another parameter of the model.

The parameters of the model can be related to measurable quantities, and the different steps of the algorithm correspond to real dynamical processes. If you are interested, you will find rich literature on the structure of river networks. An efficient way of understanding this literature is to think in terms of a model such as the one we have presented. Of course, the best way to understand the model is by converting it into a working program.

19.4 Constrained Dynamics

After finishing many of the simulations in the text, you might wonder whether similar techniques can be used for systems in everyday life such as a roller coaster at an amusement park. How would you simulate the motion of such a system? In this case the motion of the roller coaster is constrained to a curved surface. Such a simulation is of general interest and *physically–based modeling* has become an important technique in computer animation and computer graphics (see Pixar and Eberly). The approach we will discuss is also of interest in simulating polymers where it is frequently necessary to introduce geometrical restrictions such as constant bond lengths (see Rapaport). The following considerations remind us that much of computer science is of interest in physics.

We will consider *holonomic constraints* — constraints that can be eliminated by expressing the coordinates in terms of a new set of variables. The new variables, which are fewer than the original ones, are called *generalized coordinates* and implicitly satisfy the constraints. An example of a system with a holonomic constraint is a simple pendulum. The state of the system can be described by the (x,y) coordinates of the swinging mass, along with a constraint that the connecting rod has a fixed length. This constraint is holonomic because we can also describe the system by the variable θ, the angle of the mass from the pivot. The constraint is automatically satisfied in the θ-representation. Other examples of holonomic constraints include particles restricted to motion on a curve and two particles restricted to be a given distance apart.

More generally, suppose the original system coordinates are given by $r = (r_1, r_2, \ldots, r_n)$, where each r_i corresponds, for example, to a Cartesian component of a particle in the system. For example, r_1 might correspond to the horizontal position x and r_2 might correspond to the vertical position y of the first particle. If there are holonomic constraints, not all values of r correspond to valid system configurations. The idea is to introduce a set of generalized coordinates $q = (q_1, q_2, \ldots, q_f)$ and express r as a function of q such that $r(q)$ is always a valid system configuration. The reduction in the configurational dimension $n - f$ equals the number of holonomic constraints.

Our goal is to derive an ordinary differential equation of motion in terms of the generalized coordinates, that is, $\ddot{q} = \ddot{q}(q, \dot{q}, t)$. The standard way to do so is to first determine the Lagrangian, $L = T - U$, where T is the kinetic energy and U the potential energy. The equations of motion, derived in intermediate classical mechanics textbooks, are then given by

$$\frac{\partial L}{\partial q_i} - \frac{d}{dt}\frac{\partial L}{\partial \dot{q}_i} = 0. \tag{19.3}$$

We can convert (19.3) to a set of first-order differential equations in the usual way and solve them numerically. Unfortunately, in many cases, because the numerical solvers are not exact, the constraints may not be satisfied. Various methods have been developed to maintain the constraints. One of the more common methods called *Shake* iteratively enforces each constraint one at a time until all the constraints are satisfied within the desired level of accuracy. In the following, we discuss a more recent technique developed by several computer scientists.

We begin with Newton's equations of motion in Cartesian coordinates, which we express as

$$\mu_i \ddot{r}_i = F_i(r, \dot{r}) = F_i^{(a)} + F_i^{(c)}, \tag{19.4}$$

where F_i is the total force on a component of a particle, and μ_i is the associated mass. For example, for the motion of two particles in two dimensions of mass m_1 and m_2, respectively, we have $F_1 = F_{1x}$, $F_2 = F_{1y}$, $F_3 = F_{2x}$, $F_4 = F_{2y}$, $\mu_1 = \mu_2 = m_1$, and $\mu_3 = \mu_4 = m_2$. The total force is the sum of the applied force $F^{(a)}$ and constraint force $F^{(c)}$. For example, the applied force could be due to gravity, interparticle potentials, and friction. The constraint forces are not initially known but are such that the evolution of r_i satisfies the constraints.

Because forces that do work are best interpreted as applied forces, we assume without loss of generality that the constraint forces do no work. This assumption uniquely specifies $F^{(c)}$ and implies that

$$\sum_i F_i^{(c)} \dot{r}_i = 0, \tag{19.5}$$

if $\dot{r}$ is consistent with the constraints. To express $\dot{r}$ in terms of q and $\dot{q}$, we introduce the Jacobian of $r(q)$, $J_{i,j} = \partial r_i / \partial q_j$. We will see how to compute the matrix J in the example that follows. We use the chain rule to write

$$\dot{r}_i = \sum_j J_{i,j} \dot{q}_j, \tag{19.6}$$

and substitute (19.6) into (19.5) and obtain

$$\sum_{i,j} F_i^{(c)} J_{i,j} \dot{q}_j = 0, \tag{19.7}$$

which is valid for all $\dot{q}$. Because of this independence, the prefactor for each term must be identically zero:

$$\sum_i F_i^{(c)} J_{i,j} = 0. \tag{19.8}$$

Equation (19.8) gives us a way to eliminate the constraint force. That is, we multiply (19.4) by J_{ij} and sum over i. If we use the condition (19.8), the $F^{(c)}$ term will disappear, and we are left with

$$\sum_i J_{i,j}\mu_i q_i \ddot{r}_i = \sum_i J_{i,j} F_i^{(a)}. \tag{19.9}$$

Note that we have eliminated the constraint force from the equation of motion.

Because we want to describe the system by the generalized coordinates, we need to know $\ddot{q}$. Fortunately, this information is embedded in $\ddot{r}$. We differentiate (19.6) with respect to time and switch the summation index from j to k and write

$$\ddot{r}_i = \sum_k (\dot{J}_{i,k}\dot{q}_k + J_{i,k}\ddot{q}_k). \tag{19.10}$$

We still need to determine $\dot{J}_{i,k}(q,\dot{q})$. We again apply the chain rule and write

$$\dot{J}_{i,k} = \sum_l (\partial J_{i,k}/\partial q_l)\dot{q}_l. \tag{19.11}$$

We substitute (19.11) into (19.10), rearrange terms, and obtain our desired result:

$$\sum_{i,k} J_{i,j}\mu_i J_{i,k}\ddot{q}_k = \sum_{i,k}(J_{i,j}F_i^{(a)} - J_{i,j}\mu_i \dot{J}_{i,k}\dot{q}_k), \tag{19.12}$$

or in matrix notation,

$$(J^T M J)\ddot{q} = J^T F^{(a)} - J^T M \dot{J}\dot{q}, \tag{19.13}$$

where $M_{i,j} = \delta_{i,j}\mu_i$, and J^T is the transpose of J. The unknown in (19.13) is $\ddot{q}$, which can be obtained at each time step by inverting the matrix $J^T M J$.

To make the above matrix manipulations more concrete, we consider the motion of two particles of mass m_1 and mass m_2 connected by a spring and constrained to move on the curve given by $y(x) = x^4 - 2x^2$. The equilibrium length of the spring is L_0. We choose q_1 and q_2 to be x_1 and x_2, respectively, and calculate the various matrices in (19.13). In this case the matrix J can be written as

$$J = \begin{pmatrix} \dfrac{\partial r_1}{\partial q_1} & \dfrac{\partial r_1}{\partial q_2} \\ \vdots & \vdots \\ \dfrac{\partial r_4}{\partial q_1} & \dfrac{\partial r_4}{\partial q_2} \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ y_1' & 0 \\ 0 & 1 \\ 0 & y_2' \end{pmatrix}, \tag{19.14}$$

where $y_1' = 4x_1^3 - 4x_1$ and $y_2' = 4x_2^3 - 4x_2$. We also have

$$M = \begin{pmatrix} m_1 & 0 & 0 & 0 \\ 0 & m_1 & 0 & 0 \\ 0 & 0 & m_2 & 0 \\ 0 & 0 & 0 & m_2 \end{pmatrix} \tag{19.15}$$

$$MJ = \begin{pmatrix} m_1 & 0 \\ m_2 y_1' & 0 \\ 0 & m_2 \\ 0 & m_2 y_2' \end{pmatrix} \tag{19.16}$$

$$J^T MJ = \begin{pmatrix} m_1 + m_1 {y_1'}^2 & 0 \\ 0' & m_2 + m_2 {y_2'}^2 \end{pmatrix}. \tag{19.17}$$

Because $J^T MJ$ is a diagonal matrix, it is straightforward to calculate its inverse.

The force due to gravity can be written as

$$F^g = \begin{pmatrix} 0 \\ -m_1 g \\ 0 \\ -m_2 g \end{pmatrix}. \tag{19.18}$$

We write the force due to the spring connecting the two masses as $|F^s| = k(L - L_0)$, where L is the length of the spring and is given by $L^2 = (x_2 - x_1)^2 + (y_2 - y_1)^2$. It can be shown that F^s can be written as

$$F^s = \begin{pmatrix} (x_2 - x_1)k_{\text{eff}} \\ (y_2 - y_1)k_{\text{eff}} \\ -(x_2 - x_1)k_{\text{eff}} \\ -(y_2 - y_1)k_{\text{eff}} \end{pmatrix}, \tag{19.19}$$

where $k_{\text{eff}} = k(1 - L_0/L)$.

The class ConstraintApp solves the equation of motion (19.13) using the Open Source Physics ODE solver in the usual way. To see how straightforward it is to implement the constrained dynamics in this case, we show the getRate method in Listing 19.1.

Listing 19.1: The getRate method.

```
public void getRate(double[] state, double[] rate) {
   // generalized coordinates
   double x1 = state[0];
   double vx1 = state[1];
   double x2 = state[2];
   double vx2 = state[3];
   double y1 = y(x1);
   double y2 = y(x2);
   double yp1 = yp(x1);
   double yp2 = yp(x2);       // first derivative
   double ypp1 = ypp(x1);
   double ypp2 = ypp(x2);      // second derivative
   // displacements
   double Lx = x2-x1, Ly = y2-y1;
   double L = Math.sqrt(Lx*Lx + Ly*Ly);     // length of spring
   // forces. L0 is equilibrium length of spring
   double keff = k*(1-L0/L);            // effective spring constant
   // net applied force on particle 1 in x-direction
   double fx1 =  keff*Lx;
   double fy1 =  keff*Ly-g*m1;
```

```
      double fx2 = -keff*Lx;    // net applied force on particle 2
      double fy2 = -keff*Ly-g*m2;
      // elements of diagonal matrix (J^T M J)^-1:
      double a11 = 1/(m1*(1+yp1*yp1));
      double a22 = 1/(m2*(1+yp2*yp2));  // other matrix elemts are zero
      // elements of vector J^T F^(a):
      double b1 = fx1 + yp1*fy1;
      double b2 = fx2 + yp2*fy2;
      // elements of vector J^T M J du/dt:
      double c1 = m1*yp1*ypp1*vx1*vx1;
      double c2 = m2*yp2*ypp2*vx2*vx2;
      rate[0] = vx1;
      rate[1] = a11*(b1 - c1);
      rate[2] = vx2;
      rate[3] = a22*(b2 - c2);
   }
```

The complete program can be downloaded from the ch19 directory.

19.5 What are Computers Doing to Physics?

There is probably no need to convince you that computers are changing the way we think about the physical world. The question, "How can I formulate this problem for a computer?" has lead to new insights into old problems and is allowing us to consider new problems.

What will be the effect of computers in physics education? The most common use of computers has been to assist students to understand topics that have been in the curriculum for many years. So far the computer has not qualitatively changed the way we learn nor the topics we study. Will computer simulation and numerical analysis make analytic methods less important? Has this happened already? Should calculus retain its traditional importance in the curriculum? Do we understand a natural phenomenon when we are able to construct a computer model that allows us to make predictions that agree with experiment? Is it necessary to obtain at least some analytic results? What do you think should be the role of computers in education?

Computers and the visual images produced by computer models can be very seductive. However, we need to remember that the goal of science is to understand nature. Theory and experiment have been the traditional routes to this end, and computation has become a third and complementary route. Although we have stressed the importance of computation in this text, it is important to stress its complementary role. We must not let the rapid advances of computer technology and the easy availability of information overshadow our ultimate goal of gaining more knowledge and a deeper understanding of natural phenomena.

References and Suggestions for Further Reading

It would be impossible to list even a small subset of references to areas of physics and related disciplines that we have not discussed. Also the development of algorithms and applications in areas we have discussed is evolving rapidly. Many references to other applications and current developments can be found in archival journals. An important site for recent developments is <arxiv.org/>, an e-print service in physics, mathematics, nonlinear science, computer science, and quantitative biology. The magazine, Computing in Science and Engineering,

<cise.aip.org/cise/>, especially the Departments on Education, Scientific Programming, and Computer Simulations, regularly feature articles that are generally accessible. The journal, American Journal of Physics, <scitation.aip.org/ajp/>, regularly has articles on computers and physics. We encourage readers to regularly visit <www.opensourcephysics.org/sip>, where new developments will be listed and discussed, as well as opensourcephysics.org> for updates of the Open Source Physics library.

Several references relevant to this chapter are given in the following.

Eric Bonabeau and Laurent Dagorn, "Possible universality in the size distribution of fish schools," Phys. Rev. E **51**, R5220–R5223 (1995). The authors apply a model originally developed for river networks to the size distribution of schools of fish. See also Kjartan G. Magnüsson, "Can physics save fish stocks?" Physics World **13** (2), 21–22 (2000).

Marek Cieplak, Achille Giacometti, Amos Maritan, Andrea Rinaldo, Ignacio Rodriguez–Iturbe, and Jayanth R. Banavar, "Models of fractal river basins," J. Stat. Phys. **91**, 1–15 (1998), or cond-mat/9803287.

A. K. Dewdney, "Computer Recreations," Sci. Am. **251** (12), 14–22 (1984). A discussion of the Wa-Tor model. Also see R. E. Durrett and S. Levin, "Lessons on pattern formation from planet WATOR," J. Theor. Biology **205**, 201–214 (2000).

David Eberly, *Game Physics* (Morgan Kaufmann, 2004).

Zvonko Fazarinc, Saša Divjak, Dean Korošec, Aleš Holobar, Matjaž Divjak, and Damjan Zazula, "Quest for effective use of computer technology in education: From natural sciences to medicine," Computer Applications in Engineering Education **11** (3), 116–131 (2003). The authors discuss some of the reasons for the relatively low impact of computer technology on university education. Also see Zvonko Fazarinc, "A viewpoint on calculus," Hewlett–Packard Journal **38** (3), 38–40 (1987).

Alexander K. Hartmann and Heiko Rieger, *Optimization Algorithms in Physics* (Wiley–VCH, 2002). A graduate level textbook that illustrates the need for physicists to become familiar with developments in computer science.

Robert L. Leheny, "A simple model for river network evolution," Phys. Rev. E **52**, 5610–5620 (1995) and references therein.

Simon Hubbard, Petro Babak, Sven Th. Sigurdsson, and Kjartan G. Magnüsson, "A model of the formation of fish schools and migrations of fish," Ecological Modelling **174**, 359–374 (2004).

D. C. Rapaport, "Molecular dynamics simulation of polymer helix formation using rigid-link methods," Phys. Rev. E **66**, 011906-1–15 (2002).

Lawrence S. Schulman and Philip E. Seiden, "Percolation and Galaxies," Science **233**, 425 (1986) and Philip E. Seiden and Lawrence S. Schulman, "Percolation model of galactic structure," Adv. Phys. **39**, 1 (1990). See also J. Perdang and A. Lejeune, "Cellular automaton experiments on local galactic structure. I. Model assumptions," Astron. Astrophys. Suppl. Series **119**, 231–248 (1996); A. Lejeune and J. Perdang, " Cellular automaton experiments on local galactic structure. II. Numerical simulations," Astron. Astrophys. Suppl. **Vol. 119**, 249–263 (1996); and D. Cartin and G. Khanna, "A self-regulated model of galactic spiral structure formation," Phys. Rev. E **65**, 016120-1–7 (2002).

Section 19.4 on constrained dynamics is based in part on lectures notes by Andrew Witkin and David Baraff available at <www.pixar.com/companyinfo/research/pbm2001/>.

The website <immsimteam.med.nyu.edu>, Immune System Modeling and Simulation, is an example of the result of the collaborative research of physicists, physicians, and computer scientists to construct a cellular automaton model of the immune system. A version of their model, IMMSIM, can be downloaded from their website.

Index